Algorithmic Tools for Network Analysis

by

Jingbang Chen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Jingbang Chen 2025

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Ioannis Koutis Associate Professor Dept. of Computer Science, New Jersey Institute of Technology
Supervisor(s):	Richard Peng Associate Professor School of Computer Science, Carnegie Mellon University
	Lap Chi Lau Professor Cheriton School of Computer Science, University of Waterloo
Internal Member:	Trevor Brown Associate Professor Cheriton School of Computer Science, University of Waterloo
Internal Member:	Arne Storjohann Associate Professor Cheriton School of Computer Science, University of Waterloo

Internal-External Member: Luke Postle
Professor
Combinatorics and Optimization Dept., University of Waterloo

Other Member(s): J. Ian Munro
Professor Emeritus
Cheriton School of Computer Science, University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis is based on the following publications or unpublished papers. For (2), (3), (4), the first three authors are the primary authors that share equal contributions, listed in a manner that applies in database/data mining conferences. (1) follows the alphabetical order that applies in the theory conference, where everyone shares equal contributions.

1) [50] Ruinian Chang, Jingbang Chen, J. Ian Munro, Richard Peng, Qingyu Shi, and Zeyu Zheng. Sandpile prediction on undirected graphs
In Chapter 2.

This paper proposes sandpile prediction algorithms for trees, paths, cliques, and general graphs. Initially, this project was mainly run by Jingbang and two other high school students Qingyu Shi, and Ruinian Chang were involved. We came up with new algorithms for trees and paths that outperform prior results by reducing several log factors. Jingbang basically wrote every section. In writing, the two high school students have taken charge of some proofs and have written a brief introduction to using splay trees to conduct special queries. Most of their contents in the current version are deferred to the appendix. Zeyu Zheng joined the project later on, Jingbang wrote the initial version of the general graph algorithm and its proof. Zeyu further improved it through some random walk theorems. In our recent version, we tried adding experiments and submitted it to a non-TCS conference, Ruinian took charge of the experiment and Jingbang wrote the whole experiment section in the paper.

2) [152] Qiuyang Mang*, Jingbang Chen*, Hangrui Zhou*, Yu Gao, Yingli Zhou, Richard Peng, Yixiang Fang, and Chenhao Ma. Efficient historical butterfly counting in large temporal bipartite networks via graph structure-aware index. In *Proceedings of the VLDB Endowment, VLDB 2025, September 1-5, 2025, London, United Kindom*, Vol. 18, No. 6 ISSN 2150-8097, 2025
In Chapter 3.

This paper proposes algorithms to solve the historical butterfly problem. The initial algorithm idea was proposed during the discussion between Jingbang, Qiuyang, and Hangrui in February 2024. Hangrui conducted experiments, and Qiuyang wrote the Preliminaries section, the hardness and the Experiments. Jingbang organized weekly discussions and wrote most of the paper (including the Appendix). The power-law model analysis part was initially drafted together by Jingbang, Qiuyang, and Hangrui and eventually finished by Yu Gao.

3) [55] Jingbang Chen*, Weinuo Li*, Yingli Zhou*, Hangrui Zhou, Qiuyang Mang, Can Wang, Yixiang Fang, Chenhao Ma. Scalable Approximate Biclique Counting over Large Bipartite Graphs
In Chapter 3.

This paper proposes approximate algorithms to count the number of bicliques. Weinuo has been advised by Jingbang since 2019. He is in charge of the experiment part. Jingbang wrote mostly everything else, including the introduction, algorithm specification, and math analysis. The algorithm idea was initially proposed during the discussion between Jingbang, Yingli, Qiuyang, and Hangrui in October 2024. Jingbang have organized weekly discussions since then. There were also some preliminary discussions with Yingli and Qiuyang back in 2023.

4) [56] Jingbang Chen*, Qiuyang Mang*, Hangrui Zhou*, Richard Peng, Yu Gao, and Chenhao Ma. Scalable algorithm for finding balanced subgraphs with tolerance in signed networks. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, August 25-29, 2024, Barcelona, Spain*, pages 278–287, 2024
In Chapter 4.

This paper proposes a new model in signed networks for finding balanced subgraphs with tolerance, along with algorithms that outperform prior work. The initial ideas on models and algorithms are from the discussion between Jingbang, Qiuyang and Chenhao in December 2023. Between then and the paper submission in February 2024, Jingbang organized weekly discussions and wrote most of the paper. Hangrui conducted the experiments, and Qiuyang wrote the experimental section.

Abstract

Network analysis is a crucial technique used in various fields such as computer science, telecommunications, transportation, social sciences, and biology. Its importance includes optimizing network performance, understanding social and organizational structures, and detecting fraud or misinformation. In this thesis, we propose algorithmic results on several aspects of network analysis.

The *Abelian sandpile* model is recognized as the first dynamical system discovered exhibiting *self-organized criticality*. We present algorithms that compute the final state of the sandpile instance on various classes of graphs, solving the *sandpile prediction* problem on: (1) general graphs, with further analyses on regular graphs, expander graphs, and hypercubes. (2) trees and paths, surpassing previous methods in time complexity.

To analyze the structure and dynamics of networks, counting motifs is one of the most popular methods, as they are considered the basic construction block of the network. In this thesis, we introduce several tools developed for counting motifs on bipartite networks.

Despite its importance, counting (p, q)-bicliques is very challenging due to its exponential increase with respect to p and q. We present a new sampling-based method that produces a high-accuracy approximate counting of (p, q)-bicliques, with provably error guarantee and unbiasedness.

In another line of work, we consider the temporal bipartite graphs, which edges carry timestamps. To capture the dynamic nature of relationships, we consider counting butterflies ($(2, 2)$-bicliques) in temporal bipartite graphs within specified time windows, called the *historical butterfly counting* problem. We present a hardness result between memory usage and query time for this problem and a new index algorithm that surpasses the hardness when applied to power-law graphs, with outstanding empirical performance.

Lastly, we discuss tools that find the polarized community in the network. A classical model that applies to networks to deal with polarization is the signed graphs, which have positive and negative edges between vertices. A signed graph is balanced if it can be decomposed into two disjoint sets such that positive edges are between vertices in the same set while negative edges are between vertices from different sets. This notion of balance is strict in that no edge can disobey the condition, which seldom appears in reality. To address this phenomenon, we propose a new model for identifying balanced subgraphs with tolerance in signed graphs and a new heuristic algorithm that computes maximal balanced subgraphs under the new tolerance model.

Acknowledgements

My sincere and deepest gratitude goes to my advisor, Richard Peng, for reintroducing me to the world of data structures, enlightening me on how research should be conducted, providing insights when I was lost, and most importantly, showing me how competitive programming can be a part of our lives. My entire Ph.D. life would not have been possible without his support. I am sure his wisdom will continue to help me in the future.

I cannot survive this long journey without my family. My heartfelt thanks go out to my parents, Rongmei Wei and Zongbiao Chen, for their endless emotional and material support. Since childhood, they have always encouraged me to chase my dreams and keep believing in doing the right things. I would also like to thank my coach and undergraduate advisor Can Wang. Thank you for recruiting me 10 years ago and supporting me in every important step of life and study since then.

I want to express my gratitude to my thesis committee, Ioannis Koutis, Richard Peng, Lap Chi Lau, Trevor Brown, Arne Storjohann, Luke Postle, and J. Ian Munro.

I want to thank my academic collaborators over the past years. Thank you to Xinyuan Cao, Ruinian Chang, Li Chen, Chenxin Dai, Jiangqi Dai, Yixiang Fang, Yu Gao, Meng He, Yufan Huang, Weinuo Li, Yang P. Liu, Chenhao Ma, Qiuyang Mang, J. Ian Munro, Richard Peng, Arvind Ramaswami, Qingyu Shi, Alicia Stepin, Can Wang, Runze Wang, Kaiyu Wu, Tingqiang Xu, Daniel J. Zhang, Zeyu Zheng, Hangrui Zhou, and Yingli Zhou.

Pursuing a Ph.D. is a painful path that you must go through alone. Luckily, I never feel lonely because of my friends. I want to list some of them here: Thank you to Yu Jiang (a best friend since over 20 years ago); Yu Zhang, Jiaxin Zhu, Junhao Liao, Xiaoxiao Zou (friends since middle school); Anxiao He, Zicheng Ye, Renzhong Li, Yiwen Jiang, Yubai Xie, Yinuo Lin, Zhiqiang Xiao, Zhicong Lyu, Mingrui Liu, Changdong Li, Weinuo Li, Jiachen Tang, Bo Peng, Qiuyang Mang, Wenhan Huang, Longao Dai (friends since undergraduate); Yu Gao, Xinyuan Cao, Kuan Wang, Cangdi Li, Qiyang Sun, Ziyi Zhang, Tianzhe Wang (friends since in Atlanta); Chris Xu, Christian Kapsales, Zhiang Wu, Shufan Zhang, Zhuo Yang, Ningtong Zhang, Wenjuan Zhou, Hangrui Zhou (friends since in Waterloo). I also like to thank my landlord, Alex Dou, in Waterloo. Special thanks go to Yuan Xu and Jing Li. Lastly, I would like to thank and send my best wishes to Qingyu Shi, the most talented and wonderful person I have ever met in the CP community. I wish you all the best.

Finally, I want to express my gratitude to Genshin Impact and table tennis, a game that launched at the same time as I started my Ph.D. in September 2020, and a sport that I have been addicted to since childhood. They have accompanied me for all highs and lows in these 5 years, especially during the COVID times.

Dedication

This is dedicated to the fools who dream, the hearts that break, and the mess we make. Despite how crazy they may seem, smiling through it, we would do it again.

Table of Contents

Examining Committee Membership ii

Author's Declaration iv

Statement of Contributions v

Abstract vii

Acknowledgements viii

Dedication ix

List of Figures xiv

List of Tables xvii

1 Introduction 1

1.1 Conclusion 4

2 Sandpile Prediction on Undirected Graphs 6

2.1 Introduction 7

2.1.1 Overview 9

2.1.1.1 Sandpile Prediction on General Graphs 9

2.1.1.2 Sandpile Prediction on Structured Graphs 10
2.1.2 Related Work 11
2.2 Preliminaries 12
2.3 Prediction on General Graphs 14
2.3.1 Performance Analysis on General Graphs 15
2.3.2 Performance Analysis on Structured Graphs 15
2.4 Prediction on Structured Graphs 18
2.4.1 Local Behavior on Trees 18
2.4.2 Algorithm on Trees 20
2.4.3 Prediction on Other Structured Graphs 25
2.5 Experiments 25
2.5.1 Experimental Setting 25
2.5.2 Running Time Analysis on General Graphs 27
2.5.3 Running Time Analysis on Trees 27
2.5.4 Logarithm Dependency on General Graphs 28
2.6 Data Structure for the Tree Algorithm 29
2.6.1 Overview 29
2.6.2 Splay Trees 30
2.6.3 Rotation 33
2.6.4 Merging by Small-To-Large Technique 37
2.6.5 Splitting by Undoing Merges 38
2.6.6 Difference Aggregation by Tree Walk 42
2.6.7 Computing Partial Firing Numbers by Pop-Up Mechanism 43
2.6.7.1 DeltaSum Calculation 45
2.6.8 Moment Updating and Reverting 46
2.6.9 Overall Analysis 51
2.7 Sandpile Prediction on Paths 54
2.8 Deferred Proofs 59

3 Motif Counting on Bipartite Graphs **64**
3.1 Approximate Biclique Counting . . . 64
3.1.1 Introduction . . . 65
3.1.1.1 Related Work . . . 67
3.1.1.2 Preliminaries . . . 68
3.1.2 Algorithm . . . 69
3.1.2.1 Coloring . . . 70
3.1.2.2 Pattern Counting . . . 71
3.1.2.3 Approximate Counting via Sampling . . . 74
3.1.3 Experiments . . . 80
3.1.3.1 Experimental Setting . . . 80
3.1.3.2 Overall Comparison Results . . . 81
3.1.3.3 Detailed Analysis . . . 85
3.2 Historical Butterfly Counting . . . 88
3.2.1 Introduction . . . 88
3.2.1.1 Related Work . . . 92
3.2.2 Preliminaries . . . 93
3.2.2.1 Problem Definitions . . . 93
3.2.2.2 Some Key Techniques in Motif Counting . . . 94
3.2.2.3 Baselines . . . 95
3.2.3 Algorithm . . . 96
3.2.3.1 Problem Hardness . . . 96
3.2.3.2 Enumeration-based Index . . . 98
3.2.3.3 Combination-based Index . . . 101
3.2.3.4 Graph Structure-aware Index . . . 103
3.2.3.5 Index Compression . . . 108
3.2.4 Analysis on Power-law Graphs . . . 111

3.2.5 Experiments . . . 120
3.2.5.1 Efficiency . . . 121
3.2.5.2 Comparison with TBC++ . . . 123
3.2.5.3 Space-Time Trade-Offs . . . 124
3.2.5.4 Study on Compressed Indexes . . . 125
3.2.6 Case Study . . . 126
3.2.7 Handling Duplicate Edges . . . 127

4 Balanced Subgraph Finding on Signed Graphs 133
4.1 Introduction . . . 133
4.1.1 Related Work . . . 136
4.2 Problem Specification . . . 137
4.3 Algorithm . . . 141
4.3.1 Relaxation . . . 141
4.3.2 Local Search . . . 142
4.3.3 Region-based Sampling . . . 145
4.4 Experiments . . . 149
4.4.1 Experimental Setting . . . 149
4.4.2 Finding Balanced Subgraphs with Tolerance . . . 151
4.4.3 Finding Strictly Balanced Subgraphs . . . 152
4.4.4 Running Time Analysis . . . 153
4.4.5 Solving the 2PC Problem . . . 154
4.4.6 Statistical Hypothesis Testing . . . 155
4.4.7 Hyperparameter Analysis . . . 157
4.4.8 Stability Study . . . 157

References 159

List of Figures

2.1 A firing operation on a full vertex. The chips transferred to a sink vertex are ignored, and no firing operation can happen on a sink vertex (colored in blue). . . . 12

2.2 Experimental results on general graph datasets. . . . 27

2.3 Experimental results on tree datasets. . . . 28

2.4 Runtime trend for two general graph algorithms as the number of chips increases from 2^2 to 2^{20}. . . . 28

2.5 This figure shows the life cycle of D_u where arrows describe the calling order and blocks contain the current maintained information. Blue ones are called in SolvePartial(u, G, σ') and red ones are in SolveComplete(u, G). . . . 30

2.6 The figure corresponds to a call of Zig(x), which is simply rotating the node x. . . . 35

2.7 The figure corresponds to a call of Zig-Zig(x) 36

2.8 The figure corresponds to a call of Zig-Zag(x) 36

3.1 An illustrative example of two $(3, 2)$-bicliques. . . . 69

3.2 An illustrative example of a $(6, 3)$-broom. . . . 69

3.3 An example of edges' directions in a $(6, 3)$-broom, where edges with different directions are represented by `Up` and `Down` in Dir and colored in red and blue, respectively. . . . 72

3.4 Average runtime of different biclique counting algorithms for all $3 \leq p, q \leq 9$. 82

3.5 Average error of EP/Zz++ and CBS for all $3 \leq p, q \leq 9$. . . . 82

3.6 The heat-map of sampling time of EP/Zz++ and CBS with varying p and q (s). . . . 83

3.7 The heat-map of estimation errors of EP/Zz++ and CBS with varying p and q (%). 84
3.8 Average sampling time of EP/Zz++ and CBS with varying T. 85
3.9 Average error of EP/Zz++ and CBS with varying T. 85
3.10 Results of BS and CBS with varying T. 86
3.11 Average initializing and sampling time of EP/Zz++ and CBS for all $3 \leq p, q \leq 9$ ($T = 10^5$). 87
3.12 Jim Gray's activeness in database community (a) and astronomy (b) community. 89
3.13 Finding the time-window of the closest collaboration. 91
3.14 A temporal bipartite graph and its projected graphs in two time-windows, associated with their butterfly counts. 93
3.15 An example for reducing the set disjointness problem into exact historical butterfly counting in temporal bipartite graphs. 98
3.16 An illustrative example for active timestamps of wedges and the butterfly constructed from them. 98
3.17 An example for transforming historical butterfly counting into 2D-range counting. 101
3.18 An illustrative example for demonstrating `GSI`. 104
3.19 Time cost of 5,000 random queries for historical butterfly counting on datasets, where all algorithms use sequential versions. 121
3.20 Evaluating the parallelization: time cost for 5,000 random queries against the number of threads. 122
3.21 Time cost of 5,000 random queries for historical butterfly counting on different power-law bipartite graphs, where all algorithms use sequential versions. 123
3.22 Comparison with [45] on the TBC task with random δ, where the time of `GSI` includes both index time and query time. 124
3.23 Space-query trade-offs study 125
3.24 Evaluating memory usage and relative error in various compression ratio settings. Note that both our compression algorithms are not affect the query effciency. 126

3.25 Case Study (1): The trend of closeness in global research collaboration. . . 127

4.1 Balanced subgraphs are found in CLOISTER with different tolerance, where solid edges are positive, dashed edges are negative, black edges are balanced, and red edges are imbalanced. 134

4.2 A signed graph (a), its MBS-V and MBS-E (b), its TMBS-V and TMBS-E (c), and its β-TMBS (d) with $\beta = \frac{1}{3}$. 138

4.3 PROBLEM 6: Comparing maximum tolerantly balanced subgraphs in vertex cardinality ($|V|$) across various tolerances ($\beta = 2^{-x/2}$), where the part corresponding to trivial β values for this problem has been shaded. 152

4.4 PROBELM 7: Comparing maximum tolerantly balanced subgraphs in edge cardinality ($|E^+ \cup E^-|$) across various tolerances ($\beta = 2^{-x/2}$), where the part corresponding to trivial β values for this problem has been shaded. 152

4.5 PROBELM 8: Comparing maximum tolerantly balanced subgraphs in TBC ($\hat{\Phi}$) across various tolerances ($\beta = 2^{-x/2}$). 153

4.6 We study the influence of hyperparameters (*i.e.,* Non-incremental probability p, Iteration constant $\mathcal{C}$ and Early stop turn limit T) on CONFLICT dataset with $\beta = 1/4$: For each hyperparameter, we execute the algorithm when setting the hyperparameter as various values while keeping other hyperparameters fixed. The results include the average Tolerant Balance Count (TBC) and running time in 25 rounds. 157

List of Tables

2.1 Algorithmic Result on Various Structured Graphs 10

2.2 The datasets used in our experiments, where $|V|$ and $|E|$ denotes the number of vertices and edges, respectively. 26

3.1 Datasets used in experiments. 80

3.2 The value of $\frac{B^2}{C^2}$ and $\frac{Z^2}{\rho^2}$ with varying p, q (Amazon). 87

3.3 The summary of datasets, including the size of vertices (*i.e.,* $|U|$ and $|L|$) and edges (*i.e.,* $|E|$), butterfly counts (*i.e.,* B), and the γ value of power-law exponent assuming the graph's degree follows a power-law distribution (*i.e.,* $\hat{\gamma}$). For synthetic datasets, we provide the γ value for each side (*i.e.,* γ_1 and γ_2). 120

3.4 The index construction time for `GSI` in Figure 3.19. 122

4.1 Signed network datasets used in experiments, including the number of vertices ($|V|$) and edges ($|E|$), the ratio of negative edges (ρ^-), and the ratio of non-zero elements (δ). 151

4.2 Case study: Largest strictly balanced subgraph (in terms of $|V|$ or $|E|$) found by each method for each dataset ($\beta < \frac{1}{|E^+ \cup E^-|}$), where *NA* denotes the corresponding method cannot finish in 100,000 seconds on the respective dataset. 153

4.3 Mean running time (s) for each method on the various β in Section 4.4.2. . 154

4.4 Optimal subgraphs in the term of 2PC-polarity ($\frac{x^T A x}{x^T x}$) found by each method, associated with running time. 156

4.5 Verifing HYPOTHESIS 4.3.1: For each dataset, we compute the maximum $1-\epsilon$ such that there exists a subset $V' \subseteq V(G_{opt})$ with $\frac{|V'|}{|V(G_{opt})|} \geq \frac{1}{2}$ such that $\Phi(G_x, \beta) \geq (1-\epsilon)\Phi(G_{opt}, \beta), \forall x \in V'$ under various β. 156

4.6 Verifing HYPOTHESIS 4.3.2: For each dataset, we compute max $\frac{|V(G_b)|}{|V(G_a)|}$ for any two starting vertices a, b satisfying $\hat{\Phi}(G_a, \beta, \mathcal{X}_a) \geq \hat{\Phi}(G_b, \beta, \mathcal{X}_b)$ under various β. 156

4.7 Stability study of Tolerant Balance Count (TBC) under $\beta = 1/8$ as example: We conduct 100 rounds of RH and RH (INSERTION ONLY) on each dataset. The results include maximum TBC ($\max \hat{\Phi}$), minimum TBC ($\min \hat{\Phi}$), mean TBC (μ), and the variance (σ^2) across the 100 rounds. Additionally, we present the variance reduction achieved by considering *non-incremental operations*. 158

Chapter 1

Introduction

Network analysis is the study of relationships and interactions within a network that is often represented as a graph consisting of nodes (entities) and edges (connections). By designing algorithmic tools, we can reveal hidden structures and dynamics within complex systems. This helps to understand social structures, identify influential nodes, and optimize system performance. Some concrete applications include detecting fraud and preventing the spread of misinformation. In all, network analysis is a crucial technique used in various fields such as telecommunications [174, 127, 127], transportation [104, 83, 39], social sciences [27, 164], epidemiology [184, 198], cybersecurity [30, 202], and biology [141, 28, 60].

Despite its importance, modern network analysis faces several challenges and difficulties, especially as networks grow in size and complexity:

1. **Scalability**: Large-scale networks, such as those in social networks or telecommunications, can contain millions of nodes and edges, making computation and storage challenging. Many graph algorithms are theoretically computationally intensive, especially for problems like community detection or motif counting in massive graphs. Sometimes, algorithms with good theoretical guarantees could also have empirical performance below our expectations.

2. **Data Uncertainty**: Incomplete, noisy, or inconsistent data can lead to inaccurate or misleading analysis. Meanwhile, using methods with strict mathematical conditions may not find the most desired target because of the noise, which might prevent us from properly identifying, extracting, and characterizing the network in real scenarios.

3. **Dynamic Nature**: Many real-world networks change over time, requiring dynamic analysis methods that can adapt to evolving structures. This nature often requires

the use of dynamic graph algorithms or data structures to accelerate the computation. On the other hand, it is sometimes represented as the temporal setting by considering temporal graphs whose edges carry timestamps.

In this paper, we propose several algorithms for network analysis tasks that handle the aforementioned three types of challenges. These algorithmic tools are equipped with theoretical guarantees that surpass previous results, while obtaining outstanding empirical performance on both real-world and synthetic data.

In Chapter 2, we discuss several results on the *sandpile prediction* problem, addressing and resolving the scalability issue. In recent years, data scientists have extensively employed models and methodologies derived from the physics of complex systems to study the structure of networks. The *Abelian sandpile* model [21] is recognized as the first dynamical system discovered exhibiting *self-organized criticality*. In a network that obtains self-organized criticality, a single small local activity may trigger a significant impact globally. In this model, an initial configuration is set by placing chips on the vertices of the graph. If any vertex x has at least $\texttt{degree}(x)$ chips, it may distribute a single chip to each neighboring vertex. One major property of the sandpile is that any distribution order leads to the same result [34]. We present algorithms that compute the final state of the sandpile instance on various classes of graphs, solving the *sandpile prediction* problem. Our results include the following [50].

- An algorithm that works on general graphs, obtaining a $\log N$ dependency on the number of chips N, improving over the $\texttt{poly}(N)$ dependency in pure simulation. Further analyses are provided for regular graphs, expander graphs, and hypercubes.
- An algorithm for trees that runs in $O(n \log n)$ time, where n is the number of vertices. It can be further modified to an algorithm for paths that runs in $O(n)$ time. These two algorithms surpass previous stat-of-the-art methods with $O(n \log^5 n)$ [178] and $O(n \log n)$ [162] time complexity, respectively.

In Chapter 3, we present results on motif counting in bipartite graphs, addressing both scalability and dynamic nature. Motif counting is a widely used method for analyzing the structure and dynamics of such graphs, as motifs are regarded as fundamental building blocks of networks. For instance, the *butterfly* motif ($(2, 2)$-biclique) captures essential patterns of interaction and has broad applications across domains such as biological ecosystems and social networks. Our focus is on counting (p, q)-bicliques on bipartite networks. Bipartite networks naturally represent relationships between two distinct sets of entities. Counting (p, q)-bicliques serves as a fundamental operator in many applications, including

cohesive subgraph analysis, information aggregation in graph neural networks, and densest subgraph mining. Our results include the following.

- Despite its importance, counting (p, q)-bicliques is very challenging due to its exponential increase with respect to p and q. On the other hand, in many applications of biclique counting, an approximate count is often sufficient. We present a new sampling-based method that produces a high-accuracy approximate counting of (p, q)-bicliques, with provably error guarantee and unbiasedness.
- To capture the dynamic nature, we consider counting butterflies on temporal bipartite graphs within specified time windows, called the *historical butterfly counting* problem. We first present a hardness result between memory usage and query time, showing that $O(n^2/\lambda^2)$ memory is needed for an $O(\lambda)$ runtime, where n denotes the number of vertices. Then, we propose a new index algorithm that surpasses the hardness when applied to power-law graphs, with promising empirical performance.

In Chapter 4, we study the problem of extracting the balance subgraph, mainly resolving the issue of data uncertainty. A significant area of research in network analysis focuses on developing methods to identify polarized communities within networks. A classical framework for modeling polarization is the signed graph, where edges between vertices are labeled as either positive or negative. A signed graph is said to be balanced if its vertex set can be partitioned into two disjoint subsets such that all positive edges lie within each subset and all negative edges span across subsets. This concept has important practical implications. For example, in a social network, a balanced structure implies the presence of two internally cohesive but mutually antagonistic communities. However, this definition of balance is often too rigid for real-world networks, where violations of the strict balance condition—referred to as imbalanced edges—are common. Such imbalances highlight the limitations of the classical model and motivate the need for more flexible approaches to capture community polarization in complex networks. To address this phenomenon, we present the following results [56].

- A new model for identifying balanced subgraphs with tolerance in signed graphs. We define β-tolerance by involving the frustration index in the condition. This admits balanced subgraphs with a certain percentage of imbalanced edges.
- A new heuristic algorithm that computes maximal balanced subgraphs under the new tolerance model. It takes advantage of the newly defined model and empirically manages to extract large polarized communities with faster runtime. It also outperforms prior methods without tolerance allowance.

1.1 Conclusion

All the work in these three directions is, in fact, interconnected, besides serving the purpose of network analysis. One of the most important things is that the algorithms are all proposed and derived in similar ways, with assistance of either general-purpose data structures from literature, or new ones designed for the specific problem:

- In sandpile prediction, the key technical detail is a function that determines the outcome when a certain number of chips is added to a local area (subtree), which we maintain using a splittable binary search tree.
- For motif counting, we reduce problems to 2-D range counting, utilizing prior work on this subproblem as a result.
- For balanced subgraphs, in our local search process, we use a priority queue to effectively maintain the current optimal neighbor solution to search on.

In the network analysis of this era, where we need to deal with extensive and complicated networks, to develop useful tools, we believe it is not enough to be biased towards either pure theory or practice. Motivation and insight from either side are often helpful to the other. For example, we first derived our algorithm in the balanced subgraph finding project and then worked to prove why it is so effective. Ultimately, on the assumptions made in our analysis, we conduct further experiments to validate them using real-world datasets.

There are many future works that can be pursued in these directions, either directly or indirectly. We think it is worthwhile mentioning some of them to give a better insight:

- For the sandpile prediction problem, the ultimate goal is to develop a provably fast algorithm (compared to naive simulation). There are currently two attempts: (1) Reduction to linear inequality equations; (2) Graph decomposition. We managed to formalize it into a series of linear inequality systems, but we are not aware of how to solve it fast. For the latter idea, it is already known that removing a vertex can increase the overall complexity by a log factor, with additional restrictions, which have caused difficulties in using any existing algorithm. Whether such a scheme is applicable remains unknown.
- For the motif counting and balanced subgraph finding, it is helpful to consider the actual dynamic setting, subject to edge/vertex insertions and deletions. We can also try expanding the ideas on bipartite graphs into general graphs, for example, counting the number of k-cliques or finding quasi-cliques.

In all, we propose several results rooted in fundamental algorithm design. The proposed algorithms are equipped and supported with theoretical analysis and empirical performance evaluation. We believe our work is contributing to the community of data mining and databases, as well as bridging the gap between theory and practice.

Chapter 2

Sandpile Prediction on Undirected Graphs

As one of the most famous and fundamental models for self-organized criticality, the *Abelian Sandpile* model is of particular interest, and has been used to model the dynamics of social networks. However, despite a large amount of work on other aspects of sandpiles, there have been limited results in efficiently computing the terminal state, known as the *sandpile prediction* problem. This chapter considers efficiently solving the sandpile prediction problems on both general graphs and graphs with special structures. On general graphs, we propose a simulation-based algorithm, obtaining a $logN$ dependency on the number of chips N, improving over the $mathrmpoly(N)$ dependency in purely simulation-based algorithms. Through the random walk methods, we further provide theoretical analysis for our algorithm on various types of graphs, including regular graphs, expander graphs, and hypercubes. On graphs with special structures, we present tailored algorithms that run within $O(n \log n)$ time on trees and $O(n)$ time on paths, where n is the number of vertices. Our algorithms improve the previous best runtime of $O(n \log^5 n)$ on trees [178] and $O(n \log n)$ on paths [162]. We complement our analysis with experiments on both synthetic and real-world data, demonstrating our algorithms' efficiency and scalability.

The results of this chapter come from the joint work [50] with Ruinian Chang, Ian Munro, Richard Peng, Qingyu Shi, and Zeyu Zheng. Initially, Ruinian, Qingyu, and Jingbang together developed the sketch of the tree algorithm. Specifically, Ruinian developed details of how to use the splay tree correctly. Jingbang wrote the submitted version with assistance from Qingyu, who drafted most of the pseudocode. The general graph part is developed mainly by Jingbang and Zeyu. Jingbang first proposed and wrote the proof on general graphs, and Zeyu wrote the part of the analysis on other classes of graphs.

2.1 Introduction

In recent years, data scientists have been using models and methods from complex systems' physics widely to study network structures. Fruitful results have been developed in understanding critical phenomena in networks from the modern real world, including the internet and social networks. One major reason is that these networks usually obtain a good self-organization property [163]. By fine-tuning the control parameter into its critical value, the dynamic system transitions into a critical phase, where some critical phenomena may occur. Numerous studies have explored social networks' phase transition phenomenon [216, 91, 17], which have potential applications such as event detection on social media [26].

However, as shown by Bak, Tang, and Wiesenfeld in 1987, many complex dynamical systems may self-organize themselves into a critical point regardless of their initial state or controlling parameter fine-tuning [21], which is known as the concept of *self-organized criticality*. For example, on social networks, information spreading is spontaneous. When a rumor spreads among a small number of users without deliberate control, it may cause a massive discussion in the whole community and eventually become a heated topic. That is to say, in such a system obtaining self-organized criticality, a single small local activity may trigger a significant impact globally. This concept is initially referenced when studying many natural phenomena, such as earthquakes, forest fires, and avalanches [20]. It has also been identified and scrutinized across a diverse range of disciplines such as sociology [74, 130], geophysics [195, 196], and neuroscience [141, 28, 60]. Self-organized criticality has also played a significant role in the understanding of economic systems [33, 189], evolutionary biology [173], materials science [179], astrophysics [16], statistical physics [72], and epidemiology [184]. More importantly, many recent fundamental studies have shown the existence of the self-organized criticality state on social networks [149, 210, 13, 151, 215, 233, 200].

The *Abelian sandpile* model, recognized as the first discovered dynamical system exhibiting self-organized criticality, is frequently utilized as a simple yet powerful model for the study of self-organized criticality [120, 213]. For example, for social networks like Twitter, the self-organized criticality can be analyzed similarly to a growing sandpile model [74]. In order to study self-organized criticality, the sandpile model has thus attracted considerable attention [125]. Recent research has delved into various aspects of the model, encompassing topics such as sandpile groups [53, 155, 232, 10], predictability [154, 160], special variants of the model [78, 123, 77, 80], algebraic connections [2], and its impact on real-world scenarios [153].

Dhar [71] offers a generalized interpretation of the Abelian sandpile model on finite

graphs, also known as the chip-firing game on graphs [34]. In this model, an initial configuration is set by placing chips on the vertices of the graph. If any vertex x has at least $\texttt{degree}(x)$ chips, it may distribute a single chip to each neighboring vertex. This distributing process is called a "firing". To avoid infinite loops, sink vertices are added to the graph, which consume all received chips and do not fire. As a result, sandpile instances always terminate after all possible firings [125]. The chip-firing game has some other variants and connections to other theories on dynamical systems. A similar chip-firing game is defined in [23], where it allows vertices with a negative amount of chips (referred to as dollars). In this variant, the goal is to make all vertices have a non-negative number of chips. Winning strategies [23] and its relation to potential theory [24] are discussed. It is also worthwhile to note that we can construct injections mapping the chip-firing game to a special Petri net [133], which is widely used to represent concurrent interactions among different components of a system.

Our focus is on the version introduced by Dhar, and thus, no negative number of chips is allowed. One major property of the sandpile is that any firing order leads to the same result [34]. A series of natural fundamental algorithmic problems thus arise, known as the *Sandpile Prediction*:

- Can we develop efficient algorithms to compute the final state of the sandpile on arbitrary graphs?
- Can we develop faster algorithms for certain classes of graphs?

These prediction problems hold significant importance in the fields of physics [93], computer science [161], and mathematics [32]. Moreover, the sandpile prediction problem has direct connections with practical applications such as load balancing [177] and the derandomization of models like internal diffusion-limited aggregation [73, 132]. Efficient prediction algorithms can also be directly considered as a tool to detect and prevent critical avalanches on the network. For example, consider modeling a social network into a sandpile instance. By testing with different initial configurations, we can find individuals that potentially have a great global impact.

In general, sandpile prediction unfolds into two different lines of research, one focusing on mathematically bounding the number of firings and the other on algorithmically predicting the result faster than mere simulation (*prediction algorithms*). Despite the abundant literature on other aspects of the sandpile model, there have been limited results in developing prediction algorithms. On structured graphs including trees [96] and high dimensional grids [162], the prediction problem has been shown to be P-Complete, which

means it is difficult to develop parallel algorithms. On general graphs, there is no algorithm that works faster than simulation.

In this thesis, we bridge this gap and address the above-mentioned questions affirmatively by developing efficient algorithms for general graphs and various types of structured graphs, improving from the previous best theoretical results. For general graphs, we propose an algorithm that obtains a logarithmic dependency on the number of chips. For structured graphs, including trees and paths, we propose algorithms that work in near-linear time and space. Experimental results demonstrate that all our algorithms work well empirically, outperforming non-trivial baselines.

2.1.1 Overview

In this thesis, our work is divided into two types: solving the sandpile prediction problem on general graphs and structured graphs.

2.1.1.1 Sandpile Prediction on General Graphs

When a sandpile contains sinks, it always terminates [125], and the number of firings can be $\texttt{poly}(N)$ [108] where N denotes the total number of chips. As N gets larger, simulating could be extremely slow. In this thesis, we propose a new simulation-based algorithm. It follows a recursive reduction approach combined with a simple greedy strategy and is very easy to implement. The following theorem provides a theoretical guarantee of its running time.

Theorem 2.1.1 (Sandpile Prediction on General Graphs)**.** *Given a sandpile instance $S(G, \sigma)$ that contains exactly one sink, there is a simulation-based algorithm that runs in $O(Rm^2n \log N)$ time, where m, n denote the number of edges and vertices respectively, N denotes the total number of chips, and R denotes the maximum effective resistance between the sink and any other vertex.*

Theorem 2.1.1 shows a logarithmic dependency on N, which is largely distinct from the $\texttt{poly}(N)$ bound of chips moving [108]. Thus, our algorithm highly reduces the effect when the number of chips becomes extremely large. Moreover, our algorithm has a better performance guarantee if the input graph has special structures. As shown in Table 2.1, we provide several results for our algorithm on several special graph classes.

Table 2.1: Algorithmic Result on Various Structured Graphs

Graph Classes	Running Time	Formal Statement
general	$O(Rm^2 n \log N)$	Theorem 2.1.1
d-regular	$O(n^3(d + \log n) \log N)$	Theorem 2.3.1
expander graph with minimum degree δ	$O(m^2 n \log N/\delta)$	Corollary 2.3.1
d-regular expanders	$O(n^2(d + \log n) \log n \log N)$	Theorem 2.3.2
hypercubes	$O(n^2 \log n \log N)$	Theorem 2.3.3
with maximum degree Δ	$O(\Delta^2 n^3(\Delta + \log n) \log N)$	Corollary 2.3.2
planar	$O(n^4 \log N)$	Corollary 2.3.3

2.1.1.2 Sandpile Prediction on Structured Graphs

In this thesis, we solve the sandpile prediction problem on various structured graphs. Here, we refer to graphs with special structures, such as trees and paths. We believe studying solving sandpile prediction on structured graphs is a necessary step for developing efficient algorithms on arbitrary graphs. Starting from trees and paths, we propose new algorithms that outperform the previous best runtimes. Our algorithm for sandpile prediction on trees achieves a time complexity of $O(n \log n)$ and requires only $O(n)$ memory, where n represents the number of vertices in the tree.

Theorem 2.1.2 (Sandpile Prediction on Trees)**.** *Given a sandpile instance $S(G, \sigma)$ such that G is a tree, there is an algorithm that determines whether S terminates and computes the terminal configuration of S in $O(n \log n)$ time, with $O(n)$ memory.*

Compared to the previous fastest algorithm [178] that runs in $O(n \log^5 n)$ time, our algorithm takes a distinct approach, not relying on the decomposition of trees into paths. Instead, we compute the number of firings that occur at any given vertex u. The terminal configuration can in turn be constructed. When the input graph is a path, we can also slightly modify our algorithm to run in linear time. This improvement surpasses the previous result presented in [162], which required $O(n \log n)$ time to compute the terminal configuration. We also believe our method of using splittable search trees accelerating dynamic programming is of independent interest for developing algorithms on structured graphs. In all, our contributions can be summarized as follows:

- We propose a simulation-based algorithm for sandpile prediction on general graphs, which, to the best of our knowledge, is the first known result to obtaining a logarithmic dependency on the number of chips. (Section 2.3)

- We propose near-linear algorithms for sandpile prediction on trees and paths, significantly improving from previous theoretical state-of-the-art results. (Section 2.4)
- Our general graph algorithm is further analyzed under several special graph classes, using random walk properties. (Section 2.3)
- Extensive experiments on both real-world and synthetic data show favorable performance and scalability. This also demonstrates our algorithms work well practically. (Section 2.5)

2.1.2 Related Work

Bounding the Number of Firings Numerous studies estimate the number of chip firings necessary to arrive at a terminal configuration. This aspect has been examined for various classes of directed graphs with sinks [159]. Eriksson et al. [85] showed that no polynomial bound exists for general directed graphs without sinks. Considering undirected graphs without sinks, Tardos et al. [201] proposed a bound of $\Theta(n^4)$ for the firing number in a graph with n vertices and m edges. An alternative bound was offered by Bjorner et al. [34], suggesting that a maximum of nk/λ_2 firings can occur, where k represents the total number of chips and λ_2 stands for the smallest non-trivial eigenvalue of the graph Laplacian. Holroyd et al. [108] presented an improved bound for sandpiles with sinks, stating that the number of chip movements can be at most $2NmR$, where N is the number of chips, and R is the maximum effective resistance between the sink and any vertex. For an $n \times n$ grid, Babai et al. [19] introduced the concept of the *transience class* to explore the maximum number of chips to be added to a sandpile instance before entering a recurrent state, initially providing an $O(n^{30})$ polynomial bound. Choure et al. [63] enhanced the upper bound to $O(n^7)$ and also proved a lower bound of $\Omega(n^3)$. Durfee et al. [79] used techniques from electrical networks to offer a nearly tight upper bound of $O(n^4 \log^4 n)$ and a lower bound of $\Omega(n^4)$. This work was extended to n^d-sized d-dimensional grids, providing an upper bound of $O(n^{3d-2} \log^{d+2} n)$ and a lower bound of $\Omega(n^{3d-2})$.

Non-Simulation Approaches This line of work is dedicated to calculating the terminal configuration of sandpile instances faster than by simple simulation. Moore and Nilsson [162] proposed an algorithm that solves the prediction on a path of length n in $O(n \log n)$ time. They also provided a parallel algorithm that runs in $O(\log^3 n)$, showing that the sandpile prediction on a path is in $\mathbf{NC}^3$. In contrast, the sandpile prediction problem has been classified as P-Complete for various classes of graphs, including tree structures [96]

and grids with a dimension exceeding three [162]. Thus, an $O(\text{polylog}(n))$ depth parallel algorithm would imply $\mathbf{P} = \mathbf{NC}$. Ramachandran and Schild proposed an algorithm that solves the prediction problem on trees in $O(n \log^5 n)$ time [178].

2.2 Preliminaries

We assume all graphs are undirected, unweighted, and simple (no self-loop or duplicate edge). For a graph G, we use $V(G)$ and $E(G)$ to denote the vertex set and edge set, respectively. For any vertex $v \in V(G)$, we define $\texttt{degree}(v)$ as the number of neighbors, and $\texttt{neighbor}(v)$ as the set of the neighbors of vertex v. $n = |V(G)|$ refers to the number of vertices. Given a graph G, a configuration vector $\sigma \in \mathbb{N}^n$, we define the sandpile instance on them as $S(G, \sigma)$. Configurations represent the number of chips on each vertex. The sandpile model we study contains sink vertices, which refer to vertices that only consume chips but are not allowed to fire.

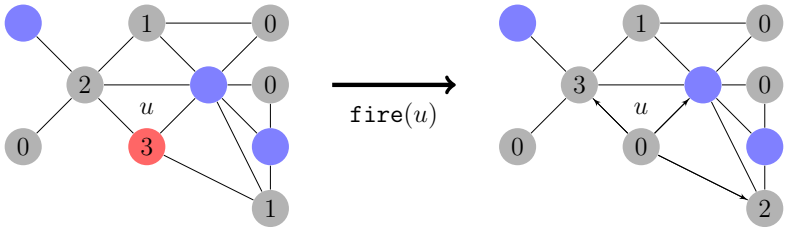


Figure 2.1: A firing operation on a full vertex. The chips transferred to a sink vertex are ignored, and no firing operation can happen on a sink vertex (colored in blue).

Figure 2.1 shows an example of firing on a sandpile instance with sinks. The integer marked on each vertex corresponds to the number of chips on the vertices. Since no firing operation could happen on the sink vertex, we can ignore the number of chips on the sink vertex and only care about the other vertices. Whenever a firing operation happens, the chips transferred on sink vertices can be treated as removed from the graph.

A non-sink vertex v is said to be full if and only if $\sigma_v \geq \texttt{degree}(v)$. A firing operation is defined on any full vertex v, which updates σ in the following way: **(1)** $u = v$: $\sigma'_u = \sigma_u - \texttt{degree}(u)$; **(2)** $u \in \texttt{neighbor}(v)$: $\sigma'_u = \sigma_u + 1$. The configuration σ' obtained by firing any vertex u in a configuration σ, denoted by $\texttt{fire}(\sigma, u)$, is called the successor of σ. For a given sandpile instance $S(G, \sigma)$ with sink s, we call the configuration σ a terminal configuration if and only if for all vertices $u \in V(G) \setminus s$, we have $\sigma_u < \texttt{degree}(u)$. Following from [125], we assume only one sink s is in the graph since we can merge all sinks into

one for input with multiple sinks, which does not affect the result. In solving the sandpile prediction problem, there are two key background theorems:

Theorem 2.2.1 ([125]). *For any sandpile instance $S(G, \sigma)$ with a sink s such that $M \neq \varnothing$ and G is a connected graph, S is always a terminal instance. That is, the terminal configuration always exists.*

Theorem 2.2.2 ([34]). *For any sandpile instance, the terminal configuration and the number of times that each vertex fires are both unique and independent of the order of firings.*

Together they show that any sandpile instance on a connected undirected graph with a sink always terminates, and the number of firings is independent of the order of firings. Thus, for a sandpile instance, we can well-define the firing number, indicating the number of firings performed on each vertex to make the configuration terminal. Formally:

Definition 2.2.1 (Firing number). *Given a terminal sandpile instance $S(G, \sigma)$, consider the process of firing all full vertices until the configuration is terminal. The number of firings performed on each vertex $v \neq s$ is denoted by $\mathbf{c}(v)$.*

The firing operation can be viewed as adding a vector to the configuration vector. We use $F(u)$ to denote the following vector of length n:

$$F(u)_v = \begin{cases} 1 & v \in \texttt{neighbor}(u) \\ -\texttt{degree}(u) & v = u \\ 0 & \text{otherwise} \end{cases}, v \in V(G)$$

Then the configuration obtained by firing vertex u is $\sigma + F(u)$. Note that $F(u)$ is a column vector of G's Laplacian matrix. Because of the commutativity of configuration addition, if we can calculate the firing number $\mathbf{c}(v)$ for each vertex v, then we can easily find that the terminal configuration is

$$\sigma + \sum_{v \in V(G) \setminus \{s\}} \mathbf{c}(v) \cdot F(v) \tag{2.1}$$

Lastly, we define the following problem as the main task to solve throughout the thesis:

Problem 1 (Sandpile Prediction). *Given a sandpile instance $S(G, \sigma)$ and a sink s, compute its terminal configuration.*

2.3 Prediction on General Graphs

In this section, we propose a simulation-based algorithm (Algorithm 1) that works on arbitrary graphs with sinks. We discuss its performance on various types of graphs, including regular graphs, expander graphs, and hypercubes. In the following analysis, we always assume there is only one sink vertex by merging all of the sinks in the graph.

Algorithm 1: SOLVE(G, σ)

```
   /* Phase 1                                                              */
 1 if there exists σ_u ≥ 2degree(u) in σ then
 2     Create empty configurations (G, σ^(1)), (G, σ^(2));
 3     for u in G do
 4         σ_u^(1) ← σ_u  mod 2;
 5         σ_u^(2) ← ⌊σ_u / 2⌋;
 6     (G, σ') = SOLVE(G, σ^(2));
 7     σ = σ^(1) + 2σ';
   /* Phase 2                                                              */
 8 while σ is not a terminal configuration do
 9     u ← arg max_{v≠s} ⌊σ_v / degree(v)⌋;
10     k ← ⌊σ_u / degree(u)⌋;
11     σ_u ← σ_u − k · degree(u);
12     for v ≠ s such that (u, v) ∈ E(G) do
13         σ_v ← σ_v + k;
```

The algorithm has two phases. In the first phase, the algorithm recursively calls itself, reducing the number of chips on each vertex by half, until the number of chips on each vertex is less than twice its degree (Line 1 to Line 5). In the second phase when the number of chips on each vertex is less than twice its degree, the algorithm iteratively fires a non-sink vertex u with the maximum ratio of chips to degree, i.e. $u = \arg\max_{v \neq s} \lfloor \sigma_u / \texttt{degree}(u) \rfloor$ (Line 9). This ratio, denoted as k, is the number of firings that can occur at u with the current number of chips (Line 10). We perform all these firings at once, distributing k chips to each of its neighbors (Line 11 to Line 13). This process continues until no more vertex can fire (Line 8), at which point the configuration is terminal. To maintain the vertex with maximum ratio, we can use heaps or other data structures that support insertions and the FINDMIN operation.

To better evaluate the performance of Algorithm 1, we present theoretical analysis on various types of graphs, along with corresponding experimental results. Throughout the performance analysis of this algorithm, we use N to denote the total number of chips on the non-sink vertices in the initial configuration.

2.3.1 Performance Analysis on General Graphs

To prove Theorem 2.1.1, we use the following result from [108] that gives an upper bound of the firing number.

Lemma 2.3.1 ([108]). *Let G be an m-edge graph in which a special vertex is chosen to be a sink, and the maximum effective resistance between the sink and any other vertex is R. For any chip configuration σ with a total number of N chips on the non-sink vertices, the number of chips moves needed to stabilize σ is bounded by $2mNR$.*

Proof of Theorem 2.1.1. Let $T(N)$ denote the time needed to solve a configuration (G, σ) of N chips in total, and let Δ denote the maximum degree of G. In the first phase of the algorithm, we create two instances in $O(n)$ time and solve a subproblem of at most $N/2$ chips. Hence, we have recurrence relation

$$T(N) = T\left(\frac{N}{2}\right) + O(n) + T_2 \cdot O(\Delta + \log n), \tag{2.2}$$

where T_2 denotes the number of iterations required for the second phase of the algorithm, $O(\Delta + \log n)$ denotes the time required to update the number of chips on a vertex and its neighbors, as well as maintain the heap that tracks the ratio of chips to degree.

Note that in the second phase, the total number of chips is bounded above by

$$\sum_{v \neq s} 2\texttt{degree}(v) \leq 4m$$

, so the number of iterations can be upper bounded by $8m^2R$ by Lemma 2.3.1. Plug in $T_2 = O(m^2R)$ to the recurrence relation Equation (2.2) and relax $O(\Delta + \log n)$ to $O(n)$, we get $T(N) = O(Rm^2n \log N)$. □

2.3.2 Performance Analysis on Structured Graphs

Our proposed algorithm performs better on graphs with some special properties. Below, we present the method we use in performance analysis.

Let t denote the current number of firings and let $\sigma^0, \sigma^1, \ldots, \sigma^t$ be a sequence of chip configurations, where σ^{i+1} is got by firing some non-sink vertex in σ^i, σ^0 is the initial configuration and σ^t is the terminal configuration. Define the weight $w(v)$ of vertex v to be the expected time $\mathbb{E}_v T_s$ of a random walk T_s that starts from v to hit the sink s, and the weight of a chip configuration

$$w(\sigma^i) = \sum_{v \neq s} \sigma^i_v w(v).$$

Conditioning on the first step X_1 of the random walk, we have

$$\Delta w(v) = \mathbb{E}_v(\mathbb{E}_{X_1} T_s - T_s) = -1.$$

This shows that if we fire v in σ^i to get σ^{i+1}, the total weight goes down by $\texttt{degree}(v)$. Hence, if the graph has minimum degree δ, the total number of firings is

$$\sum_{i=0}^{t-1} \frac{w(\sigma^i) - w(\sigma^{i+1})}{\texttt{degree of fired vertex}} \leq \frac{1}{\delta}(w(\sigma^0) - w(\sigma^t)). \tag{2.3}$$

Regular Graphs A d-regular graph is a graph in which every vertex has degree d. Note that in d-regular graphs, the number of edges is $nd/2$, and R can be bounded by n. Plugging these in Theorem 2.1.1 gives us $O(d^2 n^4 \log(N))$. The following theorem shows the algorithm does better than that, especially when d is large.

Theorem 2.3.1. *Given a sandpile instance $S(G, \sigma)$ such that G is d-regular and contains exactly one sink, Algorithm 1 runs in time $O(n^3(d + \log n) \log N)$.*

Proof of Theorem 2.3.1. By d-regularity, the total weight goes down by exactly d each time when firing a vertex. In phase 2 of Algorithm 1, the total number of chips is at most $2dn$. By [148], the weight of each vertex is bounded from above by $2n^2$ for regular graphs. Hence, we have $w(\sigma^0) - w(\sigma^t) \leq 4dn^3$. Substituting this bound and $\delta = d$ into Equation (2.3), the number of iterations in phase 2 is at most $4n^3$. Thus, substituting $T_2 = O(n^3)$ into the recurrence relation in Equation (2.2), we obtain $T(N) = O(n^3(d + \log n) \log N)$. □

Expander Graphs The notion of expanders arises frequently in many areas of math and CS theory. It has wide applications from constructing error-correcting codes [193], designing robust networks [197] to serving as a tool to prove results in complexity theory [5] and number theory [129].

Definition 2.3.1. *An ϵ-vertex-expander is a graph G s.t. every vertex set $X \subset V(G)$ satisfying $|X| \leq n/2$, has $|\texttt{neighbor}(X) - X| \geq \epsilon|X|$.*

A result in [49] gives an upper bound on the maximum effective resistance of ϵ-vertex-expanders.

Lemma 2.3.2 ([49])**.** *A ϵ-vertex-expander G, with minimum degree δ, has maximum effective resistance at most $24/(\epsilon^2(\delta+1))$.*

Hence, we have the following corollary of Theorem 2.1.1.

Corollary 2.3.1. *Algorithm 1 runs in $O(m^2 n \log N/\delta)$ time for ϵ-vertex-expander G with minimum degree δ, where ϵ is a constant and m, n denotes the number of edges and vertices, respectively.*

Oftentimes, expanders are explicitly constructed with the additional property of being regular. For regular ϵ-vertex-expander G, we have the following stronger result:

Theorem 2.3.2. *Let ϵ be a constant. The running time of Algorithm 1 is $O(n^2(d + \log n)\log n \log N)$ for d-regular ϵ-vertex-expander G.*

Proof. By [183], the weight of each vertex is bounded by $cn \log n$ for d-regular ϵ-vertex-expanders, where c is a constant. The rest part of the proof proceeds the same as the proof of Theorem 2.3.1. The running time of phase 2 is bounded by $O(2dn \cdot cn \log n/d) \cdot O(d + \log n)$, so by the reccurence relation Equation (2.2), we obtain $T(N) = O(n^2(d + \log n)\log n \log N)$. □

Other Structured Graphs A d-dimensional hypercube is a graph defined on the vertex set $\{0,1\}^d$, in which two vertices are connected if and only if they are different in exactly one of the d coordinates. We have the following result for hypercube graphs.

Theorem 2.3.3. *Algorithm 1 runs in $O(n^2 \log n \log N)$ time for d-dimensional hypercube.*

Proof. Since d-dimensional hypercube has 2^d vertices, we know $d = \log n$. Note that the d-dimensional hypercube is d-regular, and the total weight goes down by d when firing a vertex. By [188], the weight of each vertex is bounded by cn for some constant c. Plug in these values to the framework of the rest proof of Theorem 2.3.1, we get $T(n) = O(n^2 \log n \log N)$. □

Below we list the running time of Algorithm 1 on graphs having some other properties. These bounds can be obtained by plugging in the corresponding values and Lemma 2.3.1 in Equation (2.2).

Corollary 2.3.2 (of Theorem 2.1.1)**.** *Algorithm 1 runs in* $O(\Delta^2 n^3(\Delta + \log n)\log N)$ *time for graph* G *with maximum degree at most* Δ*.*

Corollary 2.3.3 (of Theorem 2.1.1)**.** *Algorithm 1 runs in* $O(n^4 \log N)$ *time for planar graph* G*.*

2.4 Prediction on Structured Graphs

We provide algorithms for various types of structured graphs: **(1)** Trees; (Section 2.4.1, 2.4.2) **(2)** Paths (Section 2.4.3, Section 2.7).

2.4.1 Local Behavior on Trees

We root the tree at the sink vertex s (below we also noted as r) and further define $\texttt{subtree}(v)$, $\texttt{children}(v)$, $\texttt{parent}(v)$ for each vertex v as the vertex set of its subtree (including v), children and its direct ancestor, respectively. Our algorithm relies on the concept of computing the outcomes after all firings in subtrees have occurred. Thus, it is crucial to establish clear definitions for events and configurations within a subtree. We also need to prove that they are consistent with the global behavior of the entire tree.

Definition 2.4.1 (Local terminal configuration)**.** *Let* $S(G, \sigma)$ *be a sandpile instance. For* $S \subseteq V(G)$*, if all the vertices* $v \in S$ *satisfy* $\sigma_v < \texttt{degree}(v)$*, then* σ *is said to be local terminal in* S*. Specially, if* G *is a tree rooted at* r*. For a vertex* $u \in V(G)$*, if all the vertices* $v \in \texttt{subtree}(u)$ *satisfy* $\sigma_v < \texttt{degree}(v)$*, then* σ *is local terminal in* $\texttt{subtree}(u)$*.*

Theorem 2.2.2, which shows that the terminal configuration is unique for any sandpile instance, can be generalized to any local subset of vertices $S \subseteq V$. Formally we have the following two lemmas:

Lemma 2.4.1 (Unique local terminal configuration)**.** *Let* $S \subseteq V(G)$ *be any subset of vertices. Consider the process that keeps firing all the full vertices in* S *until* σ *is local terminal in* S*. Then:*

1. *Any firing order reaches the same local terminal configuration.*

2. *For each vertex u, any firing order fires u the same number of times.*

Definition 2.4.2 (Local finalize operation)**.** *Let $S(G, \sigma)$ be a sandpile instance where G is a tree rooted at r. For a vertex $u \in V(G)$, let $\texttt{final}(\sigma, u)$ be the configuration obtained by firing all full vertices in the subtree of u until every vertex in $\texttt{subtree}(u)$ is not full. Formally,*

$$\texttt{final}(\sigma, u) = \begin{cases} \sigma & \text{if } \sigma \text{ is local terminal} \\ & \text{in the subtree of } u, \\ \texttt{final}(\texttt{fire}(\sigma, v), u) & \text{if } v \in \texttt{subtree}(u) \\ & \text{and } \sigma_v \geq \texttt{degree}(v). \end{cases}$$

We also define the partial firing numbers, denoted by $\mathbf{c}^{\downarrow}(\sigma, u)$, as the number of the firing operations performed on vertex u to make σ local terminal in the subtree of u. Formally,

$$\mathbf{c}^{\downarrow}(\sigma, u) = \begin{cases} 0 & \text{if } \sigma \text{ is local terminal} \\ & \text{in the subtree of } u, \\ \mathbf{c}^{\downarrow}(\texttt{fire}(\sigma, v), u) + [u = v] & \text{if } v \in \texttt{subtree}(u) \\ & \text{and } \sigma_v \geq \texttt{degree}(v). \end{cases}$$

Here, $[u = v]$ denotes the Iverson bracket.

By Lemma 2.4.1, for any $S \subseteq V(G)$, any firing order in the set S leads to the same local terminal configuration in S, and the value of $\mathbf{c}^{\downarrow}(\sigma, u)$ is also independent of the order of the firings. Thus the definitions in Definition 2.4.2 are well-defined. We use the notation $\mathbf{c}^{\downarrow}(u)$ to denote $\mathbf{c}^{\downarrow}(\sigma, u)$ as we are only considering a single given sandpile instance $S(G, \sigma)$. By Theorem 2.2.2, the terminal configuration is independent of the order of firings. Thus, we have the following lemma

Lemma 2.4.2. *Let σ be a configuration and σ' be a configuration obtained by performing several firing operations in $\texttt{subtree}(u)$ on σ. For any configuration σ^*, $\texttt{final}(\sigma + \sigma^*, u) = \texttt{final}(\sigma' + \sigma^*, u)$.*

Lemma 2.4.3. *Let σ and σ' be any two configurations and $u \in V(G)$ be any vertex. Then $\texttt{final}(\sigma + \sigma', u) = \texttt{final}(\texttt{final}(\sigma, u) + \texttt{final}(\sigma', u), u)$.*

Lemma 2.4.1, 2.4.2 and 2.4.3 are proved in Section 2.8. We use $\texttt{final}(\sigma)$ to refer to $\texttt{final}(\sigma, r)$, and Problem 1 is equivalent to find $\texttt{final}(\sigma)$.

2.4.2 Algorithm on Trees

The main idea of our algorithm is to compute the value of $\mathbf{c}(v)$ for all $v \in V(G)$. After that, we can apply Equation (2.1) to retrieve the terminal configuration. It is difficult to calculate the value of $\mathbf{c}(v)$ directly. However, we can complete the calculation in two phases.

Partial Firing The first phase reduces the configuration σ to a state which is local terminal in all vertices excluding r. In other words, after this round of firings, all the vertices other than r are not full. To do this, we push all the chips to the root by firing from bottom to top, and as a result, we compute $\mathbf{c}^{\downarrow}(v)$ for all vertices $v \in V(G) \setminus \{r\}$. We propose Algorithm 2 to correctly and efficiently finish this phase, which will be discussed later in this section. The following lemma summarizes the process:

Lemma 2.4.4. SolvePartial*$(u,\ G,\ \sigma')$ computes the value of all $\mathbf{c}^{\downarrow}(v)$ for all $v \in \texttt{subtree}(u)$. In particular, it computes the value of $\mathbf{c}^{\downarrow}(v)$ for all $v \in V(G)$ in $O(n \log n)$ time. It also converts the initial configuration σ into another configuration that is local terminal in the subtree of u for any non-root vertex u.*

Assume we are currently visiting vertex u. Since the process is from bottom to top, we further assume the computation has already been done on u's children. The main difficulty of such recursive computation is that after we fire u, chips will be sent to its subtree and might cause further firings in the subtree. What's worse, the firing in the subtree might cause another firing on u if they return enough chips back to u. Such repetition could happen many times before every vertex in $\texttt{subtree}(u)$ becomes not full.

We aim to eliminate these repetitions by maintaining additional information for each child node. This allows us to accurately compute the state after firing vertex u to not full without revisiting the subtree of u. More precisely, for any vertex u, we track how many chips are returned to its parent after adding k chips to u and repeatedly firing all full vertices within $\texttt{subtree}(u)$, until the configuration reaches a local terminal state in $\texttt{subtree}(u)$. The formal definition of such quantity is as follows.

Definition 2.4.3 (Local Upward Contribution)**.** *Let $S(G, \sigma)$ be a sandpile instance where G is a tree rooted at r. For a vertex $u \in V(G)$ $(u \neq r)$ such that σ is local terminal in the subtree of u, the local upward contribution of adding k $(k \geq 0)$ chips to the vertex u is denoted as $\delta(u, k)$, where $\delta(u, k) = \texttt{final}(\sigma + k_u, u)_{\texttt{parent}(u)} - \texttt{final}(\sigma, u)_{\texttt{parent}(u)}$. k_u denotes a vector of all zeros except the value of the u-th term is k.*

Algorithm 2: SolvePartial(u, G, σ')

```
D_u ← ∅;
dfs_order_u ← visit_time;
visit_time ← visit_time + 1;
if children(u) = ∅ then
    σ'_{parent(u)} ← σ'_{parent(u)} + σ'_u;
    c↓(u) ← σ'_u;
    σ'_u ← 0;
    return;
for v ∈ children(u) in arbitrary order I do
    SolvePartial (v, G, σ');
    Merge (u, v);
k ← ComputeC (u);
c↓(u) ← k;
σ'_u ← σ'_u + DeltaSum(u) − c↓(u) · degree(u);
if u is not the root of G then
    σ'_{parent(u)} ← σ'_{parent(u)} + k;
Update (D_u);
```

Then, Lemma 2.4.5 shows that the number of the remaining chips on vertex u after firing u exactly k times and make σ be local terminal in all $\texttt{subtree}(v_i)$ for $v_i \in \texttt{children}(u)$ is exactly $\psi_u(k) \stackrel{\text{def}}{=} \sigma_u - k \cdot \texttt{degree}(u) + \sum_{v \in \texttt{children}(u)} \delta(v, k)$.

Lemma 2.4.5. *Let $u \in V(G)$ and σ be local terminal in the subtree of all its children $v_i \in \texttt{children}(u)$. For any positive integer k, if $\psi_u(k-1) \geq \texttt{degree}(u)$, then: **(1)** It is possible to fire vertex u at least k times without firing any vertex not in $\texttt{subtree}(u)$. **(2)** Assume we fired vertex u exactly k times, and fired all full vertices in $\texttt{subtree}(v_i)$ for all $v_i \in \texttt{children}(u)$, while not firing any vertex outside $\texttt{subtree}(u)$. Then the number of chips at vertex u is exactly $\psi_u(k)$.*

We further show $\delta(u, k)$ has monotonicity:

Lemma 2.4.6. *For any vertex $u \in V(G) \notin r$ and integer $k \geq 0$, $\delta(u, k) \leq \delta(u, k+1) \leq \delta(u, k) + 1$.*

Lemma 2.4.7. *Let k be the smallest non-negative integer such that $\psi_u(k) < \texttt{degree}(u)$. Then $\mathbf{c}^{\downarrow}(u) = k$.*

By Lemma 2.4.7, our task is to find the smallest integer k such that $\psi_u(k) < \texttt{degree}(u)$ on a monotonically non-increasing function ψ_u. This integer k is exactly the value of $\mathbf{c}^{\downarrow}(u)$. We use a data structure D_u to maintain the following value: **(1)** The value of $\sum_{v \in \texttt{children}(u)} \delta(v, k)$ for a given vertex v and integer k. **(2)** The smallest integer k such that $\psi_u(k) < \texttt{degree}(u)$.

Lemma 2.4.8. *For any vertex $u \in V(G)$ the $\psi_u(k)$ is monotonically non-increasing. In other words, $\psi_u(k) \geq \psi_u(k+1)$ for all $k \in \mathbb{N}$.*

Since the function $\psi_u(k)$ is monotonically non-increasing by Lemma 2.4.8, the value of $\mathbf{c}^{\downarrow}(u)$ can be found by performing a binary search procedure. While doing binary search, we need to check if a specific value k_0 meets the inequality

$$\sigma_u - k \cdot \texttt{degree}(u) + \sum_{v \in \texttt{children}(u)} \delta(v, k) < \texttt{degree}(u),$$

which means that we need to find the value of $\sum_{v \in \texttt{children}(u)} \delta(v, k)$ efficiently.

Proofs of Lemma 2.4.5, 2.4.6, 2.4.7 and 2.4.8 can be found in Section 2.8.

Data Structure In all, to speed up the above searching procedure, as well as other computations related to δ, we design a tailored Splay tree [194] based data structure D_u for each vertex u. Recall that $\delta(u, k)$ indicates the number of chips returning to u after putting k chips on it. k (Definition 2.4.3). We cannot afford recalculation into the subtree. Instead, we prove that the number of key pairs satisfying $\delta(u, k) = \delta(u, k-1)$ is limited (Section 2.6.1). Therefore, we use a data structure to maintain the key-pairs sequence for each vertex. We further prove that computing $\delta(u, k)$ can be transformed into counting legit key pairs (Lemma 2.6.1, 2.6.5). Every vertex's key-pair sequence can be obtained by merging children's and updating the value altogether (Section 2.6.4), which can be done by the lazy propagation trick. We choose to use the splay tree to guarantee merging complexity. To sum up, D_u supports the following queries:

- ComputeC(u): return $\mathbf{c}^{\downarrow}(u)$. (Section 2.6.7)
- DeltaSum(u): return $\sum_{v \in \texttt{children}(u)} \delta(v, \mathbf{c}^{\downarrow}(u))$, where k is the returned value of ComputeC(u) ($k = \mathbf{c}^{\downarrow}(u)$). (Section 2.6.7.1)
- DeltaQuery(u, k): return $\delta(u, k)$. (Section 2.6.6)

- MERGE(u, v): merge D_v to D_u. (Section 2.6.4)
- SPLIT(u, v): split D_v from D_u. The data structure D_v becomes the one before the procedure MERGE(u, v) being called. (Section 2.6.5)
- UPDATE(u): Update D_u to match u. (Section 2.6.8)
- REVERT(u): revert the data structure D_u to the one before the procedure COMPUTEC(u) called. (Section 2.6.8)

Now, we can complete the binary search procedure by traversing over D_u while maintaining all necessary information, rather than re-calculating $\sum_{v\in\texttt{children}(u)}\delta(v,k)$ in each iteration. Theorem 2.6.3 ensures that the data structure costs $O(n\log n)$ time in total in the whole procedure of our algorithm to handle all the requests.

Throughout the algorithm, we maintain two global arrays `num` and `dfs_order`, and two global variables `visit_time` and r. These are used in both phases and data structures. r denotes the root and `visit_time` is set to keep track of the visit order in SOLVEPARTIAL, which is stored in $\texttt{dfs_order}_u$ when visiting u.

If u is a leaf, since $\texttt{children}(u)=\varnothing$, firing vertex u is equivalent to moving a chip from vertex u to $\texttt{parent}(u)$. So, we have $\mathbf{c}^{\downarrow}(u)=\sigma_u$, and the algorithm updates $\sigma'_{\texttt{parent}(u)}$ and σ'_u correspondingly (Line 4 to Line 8). Otherwise, the procedure initializes the data structure D_u (Line 1). It maintains the visit order for each vertex (Line 2 to Line 3). This ensures that the earlier the vertex is accessed, the smaller `dfs_order` value is given. After that, we merge all the children of u together to get D_u (Line 9 to Line 11; here, $\mathcal{I}$ is an arbitrary order of merging the children). By Lemma 2.4.7, the variable k computed in Line 12 is exactly the value of $\mathbf{c}^{\downarrow}(u)$. Then, we compute the number of the remaining chips on u after finishing all firings in $\texttt{subtree}(u)$ (Line 14). By Lemma 2.4.5, the number of the chips on vertex u is changed to $\psi_u(k)$. The value of $\sum_{v\in\texttt{children}(u)}\delta(v,k)$ can be computed by DELTASUM(u, k). Finally, we update the value of $\sigma'_{\texttt{parent}(u)}$ (Line 15 to Line 16) and the data structure D_u (Line 17), so that D_u has the full information in $\texttt{subtree}(u)$. Note that we do not update the value of σ'_v for $v\in\texttt{children}(u)$ explicitly, since the number of chips moved from v to u is already calculated as $\delta(v,k)$ (Line 14). After visiting vertex u, we no longer use the value of σ'_v for all $v\in\texttt{subtree}(u)$. Thus, the value of σ'_v for $v\in\texttt{subtree}(u)$ can be ignored.

Complete Firing After calculating $\mathbf{c}^{\downarrow}(u)$ for all $u\in V(G)$, we recover all the $\mathbf{c}(u)$ from the top to the bottom. This process is based on the relationship between firing numbers and partial firing numbers described in Lemma 2.4.9 (Proof can be found in Section 2.8).

Algorithm 3: SolveComplete(u, G)

```
1 c(u) ← 0;
2 if u ≠ r then
3     c(u) ← c↓(u) + DeltaQuery(u, c(parent(u)));
4 Revert (u);
5 for v ∈ children(u) in the reversed order of 𝓘 do
      ; ▷ Iterate the children of u in the reversed order
6     Split (u, v);
7     SolveComplete (v, G);
```

Lemma 2.4.9. *For each vertex $u \in V(G)$ such that $u \neq r$, $\mathbf{c}(u) = \mathbf{c}^{\downarrow}(u) + \delta(u, \mathbf{c}(\texttt{parent}(u)))$. Specially, for the root vertex, $\mathbf{c}(r) = 0$,*

Note that if the tree contains no sink (Same as in [178]), we instead let $\mathbf{c}(r) = \mathbf{c}^{\downarrow}(r)$. By Lemma 2.4.9, as long as we maintained $c(u)$ and the value of $\delta(u, i)$ for all required i correctly, we can recursively recover all the values of $c(v)$ for $v \in \texttt{subtree}(u)$. Specifically, we subsequently traverse the tree and compute the firing number $\mathbf{c}(u)$ based on the results obtained from $\texttt{parent}(u)$. We also need to restore information for vertices in $\texttt{subtree}(u)$ before visiting them, which is done by reverting and splitting operations on D_u.

In Algorithm 2 we use Merge(D_u, D_v) to make D_u store the information of $\texttt{subtree}(u)$. Now we need to revert all these merges. For each $u \in V(G)$, we revert the changes that were made in Update(D_u). After that for each child $v \in \texttt{children}(u)$ we recover the structure of D_v by splitting D_u. Proved in Section 2.6.9, the values of $\mathbf{c}(u)$ for all $u \in V(G)$ are computed correctly after calling SolveComplete(r, G). The following theorem summarizes the whole phase.

Lemma 2.4.10. SolveComplete*(u, G) computes the value of all $\mathbf{c}(v)$ for all $v \in \texttt{subtree}(u)$, based on the value of $\mathbf{c}^{\downarrow}(u)$ found in the* SolvePartial *part. In particular, it can compute the value of $\mathbf{c}(v)$ for all $v \in V(G)$ in $O(n \log n)$ time.*

In Algorithm 3, we first handle the case where u is the root, so we have $\mathbf{c}(u) = 0$. If u is not the root, then we calculate the value of $\mathbf{c}(u)$ (Line 3) by Lemma 2.4.9. Afterward, we revert D_u to allow us to apply splitting (Line 4). Finally, we recursively process all the children of u in the reversed order of $\mathcal{I}$ (Line 5 to Line 7), where $\mathcal{I}$ is the same as the one in Algorithm 2. This ensures the algorithm reverts all the merging in the correct order.

Overall Analysis Finally, we present the main structure of our proposed algorithm as follows: We root the tree at s and initialize σ' as the input configuration. In addition, we set a global variable `visit_time` to 1. After that, we call SolvePartial to compute all $\mathbf{c}^{\downarrow}$ values, the partial firing numbers when we only consider the final state of the subtree. Subsequently, we further call SolveComplete to compute all $\mathbf{c}$ values, which can be converted to the final terminal configuration by Equation (2.1). The performance of our algorithm depends on the data structure we use throughout Algorithm 2 and Algorithm 3. We show that by using splittable binary search trees, we can implement a data structure that supports all operations in $O(n \log n)$ time in total, with $O(n)$ memory. The implementation details are discussed in Section 2.6. As a result, we prove Theorem 2.1.2 in Section 2.6.9 in the end.

2.4.3 Prediction on Other Structured Graphs

Paths can be considered a special variant of trees, and our algorithm successfully demonstrates the unification of these two graph structures. Based on the *Dynamic Optimality Conjecture* [194] of the Splay tree, we conjecture that our algorithm on trees achieves a linear runtime if the input graph is a path. As a result, we have successfully modified our algorithm to leverage the *Dynamic Finger Theorem* [65, 64] instead, leading to a provable linear runtime in Theorem 2.7.1. Details are analyzed in Section 2.7.

2.5 Experiments

In this section, we address the following research questions to evaluate various important aspects of our algorithm: What are our method's and baselines' performances when the input is general graphs **(RQ1)**? What are our method's and baselines' performances when the input is trees **(RQ2)**? Can the logarithm dependency proven in Section 2.3 be observed on experimental results **(RQ3)**?

2.5.1 Experimental Setting

Approaches We summarize each method in our experiment as follows: **(1)** FastTree: Our proposed algorithm on trees in Section 2.4. **(2)** GreedyGeneral: Our proposed algorithm on general graphs in Section 2.3. **(3)** RandomGeneral: A simulation method with the following strategy: We first find a full vertex u. Let k and d be its number of

Table 2.2: The datasets used in our experiments, where $|V|$ and $|E|$ denotes the number of vertices and edges, respectively.

Dataset		Type	$\lvert V\rvert$	$\lvert E\rvert$
IP03	Internet	Tree	149,637	149,636
IP13	Internet	Tree	499,887	499,886
IP23	Internet	Tree	1,038,657	1,038,656
Bitcoin (BC)	Financial	General	5,881	21,492
Elections (EL)	Political	General	7,115	100,693
Twitter (TW)	Social	General	10,884	251,406
RT1 $\sim$ RT8	Synthetic	Tree	$10^2 \sim 10^7$	$10^2 \sim 10^7$
RG1 $\sim$ RG4	Synthetic	General	$10^2 \sim 10^3$	$10^2 \sim 10^4$

chips and degree respectively. We fire u for $\lfloor k/d \rfloor$ times. Repeat this process until there is no full vertex. Note that, we do not compare our method with [178], which not only has a significantly worse theoretical complexity but is also impractical to implement.

Datasets We evaluate approaches on both real-world datasets and synthetic datasets, where further information on all datasets we used can be found in Table 2.2.

- **Real-world datasets** Our real-world data is primarily sourced from the OPTE project[1], which provides graphs representing network addresses and communications from 1997 to 2023. Specifically, we selected three datasets from 2003, 2013, and 2023, computed a spanning tree for each, and uniformly randomly assigned chips between 0 and 100 to each vertex. These datasets are referred to as IP03, IP13, and IP23. For experiments on general graphs, we selected three publicly available real-world networks[2][3]: Bitcoin (BC), Elections (EL), and Twitter (TW), randomly assigning chips between 0 and 1000 to each vertex.
- **Synthetic datasets** For synthetic tree datasets, we uniformly sampled trees[4] of sizes 10^2, 5×10^2, 10^3, 10^4, 5×10^4, 10^5, 5×10^5, and 10^6, labeled as RT1 to RT8. Similarly, we assigned chips between 0 and 100 to each vertex. Additionally, we used the Erdős–Rényi model [84] with $p = 0.05$ to generate general graphs with 10^2, 3×10^2, 5×10^2, and 10^3 vertices, assigning chips between 0 and 2^{20} to each vertex. We label them as RG1 to RG4.

[1] https://www.opte.org/
[2] https://konect.cc
[3] https://snap.stanford.edu
[4] https://networkx.org/

Environments All experiments are conducted on a Windows 11 workstation equipped with Intel(R) Core(TM) Ultra7-155H.

2.5.2 Running Time Analysis on General Graphs

This experiment is designed to study the efficiency of our methods on general graphs (**RQ1**). We compare our GreedyGeneral with the mere simulation algorithm (Random General). As shown in Figure 2.2, our algorithm can compute results for real-world graphs up to 2.5×10^5 edges while the baseline cannot compute results within 10,000 seconds. To the best of our knowledge, this is the first practical sandpile prediction algorithm that can produce results in a reasonable time for real-world datasets. We can also observe that our algorithm performs better in our synthetic datasets (RG1 $\sim$ RG4), while the runtime increases as the size of graphs gets larger.

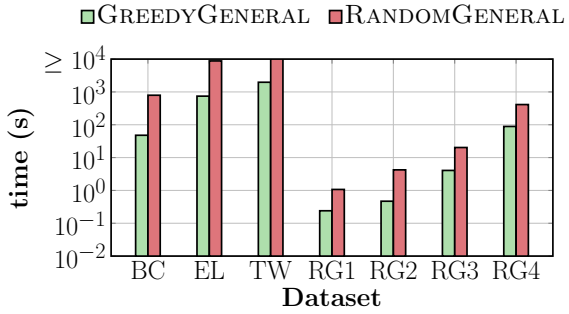


Figure 2.2: Experimental results on general graph datasets.

2.5.3 Running Time Analysis on Trees

This experiment is designed to study the efficiency of our methods on trees (**RQ2**). We mainly compare our algorithm (FastTree) and the mere simulation algorithm (Random General). As shown in Figure 2.3, our algorithm can produce results within 1 second for either real-world or synthetic datasets up to 10^7 vertices. On the other hand, the baseline cannot produce results within 10,000 seconds for trees with 5×10^4 vertices. This shows the necessity of developing efficient algorithms for graphs of special classes.

We further evaluate the scalability of our algorithms with the synthetic datasets. We can observe that as the tree size gets larger, the runtime increases moderately, showing good scalability in large datasets. In all, the empirical results demonstrate the efficiency and scalability of our method, matching its near-linear complexity.

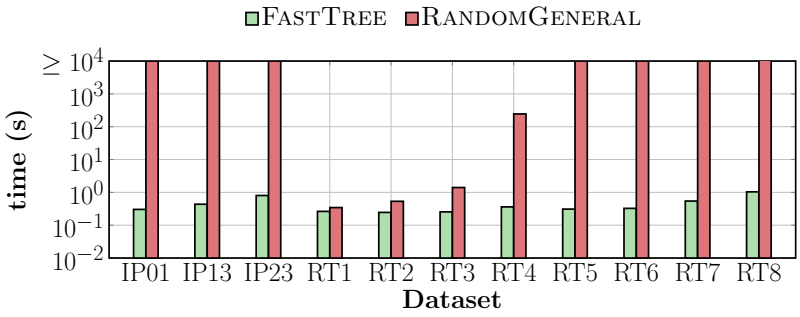
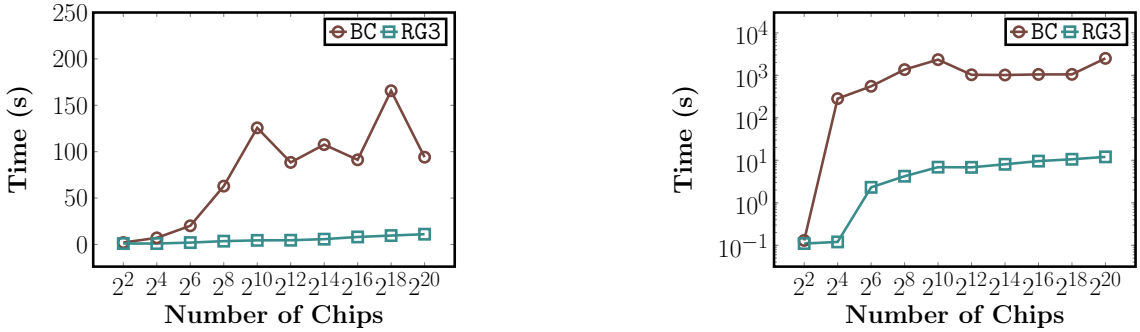


Figure 2.3: Experimental results on tree datasets.

2.5.4 Logarithm Dependency on General Graphs

This experiment aims to verify the logarithm dependency proven in Section 2.3 (**RQ3**), which is one of the main contributions of our work. We select one real-world dataset Bitcoin (BC) and one synthetic dataset RG3. Then, by considering k from 2 to 20, we uniformly randomly assign 0 to 2^k chips onto each vertex and execute our algorithm GreedyGeneral as well as RandomGeneral on it. Specifically, we take the average of the runtime from 5 such independent experiments for each k and report them in Figure 2.4. As shown in Figure 2.4, as k increases, our algorithm's runtime doesn't increase significantly. Instead, the baseline is affected greatly. Compared to the $k = 2$ case, its runtime increases from $10^3\times$ to $10^5\times$ for larger k when in the real-world dataset. This empirical result aligns with our theoretical analysis, in which the number of chips N is within a logarithm term. It also shows our algorithm's effectiveness.



(a) GreedyGeneral (b) RandomGeneral

Figure 2.4: Runtime trend for two general graph algorithms as the number of chips increases from 2^2 to 2^{20}.

2.6 Data Structure for the Tree Algorithm

In this section, we discuss the implementation details of the data structure used in the algorithm on trees.

To describe how we maintain the data structure, we will first introduce the concept of *key pairs*.

2.6.1 Overview

Definition 2.6.1 (Key Pairs)**.** *A pair (u, k) such that $u \in V(G)$ and $k \in \mathbb{N}_+$ is said to be a key pair if and only if $\delta(u, k) = \delta(u, k-1)$.*

By Lemma 2.4.6, the value of $\delta(u, k) - \delta(u, k-1)$ will be either 0 or 1. If, for a given vertex $u \in V(G)$ and integer k, we can find the number of key pairs (u, k') such that $k' \leq k$, denoted as C, then we can calculate the value of $\delta(u, k)$ which would be exactly $k - C$. Formally:

Lemma 2.6.1. *Let u be an arbitrary vertex $u \in V(G)$ and k be a non-negative integer. Then*

$$\delta(u, k) = k - \sum_{i=1}^{k} [\delta(u, i) = \delta(u, i-1)]$$

Proof. We can prove the lemma by induction.

First, the lemma is trivial for $k = 0$. For a positive integer $k \in \mathbb{N}_+$, we have $\delta(u, k-1) = (k-1) - \sum_{i=1}^{k-1} [\delta(u, i) = \delta(u, i-1)]$ by inductive hypothesis. Then $\delta(u, k) = \delta(u, k-1) + (1 - [\delta(u, k) = \delta(u, k-1)]$ since $\delta(u, k) - \delta(u, k-1)$ could be either 0 or 1. By substituting $\delta(u, k-1) = (k-1) - \sum_{i=1}^{k-1} [\delta(u, i) = \delta(u, i-1)]$, we have $\delta(u, k) = (k-1) - \sum_{i=1}^{k-1} [\delta(u, i) = \delta(u, i-1)] + 1 - [\delta(u, k) = \delta(u, k-1)] = k - \sum_{i=1}^{k} [\delta(u, i) = \delta(u, i-1)]$. □

To design our data structure, for each D_u ($u \in G$) we will use a splay tree to maintain the key pairs. Each node x in the splay tree D_u represents a key pair (u, k). Let $\texttt{moment}_x$ denote the value k for the node x. For any two different nodes $x_1, x_2 \in D_u$, we define $x_1 < x_2$ if and only if $\texttt{moment}_{x_1} < \texttt{moment}_{x_2}$. This is obviously a well-defined partial order.

Therefore, we design each D_u ($u \in V(G)$) to be a structure that maintains the orders of key pair nodes implemented by a splay tree. In addition, we maintain some arrays of

length n to store necessary information for algorithms as well as operations on $\mathcal{D}$. This part takes $O(n)$ memory to store. Each node x on D_u is mapped to the exact position on these arrays, which means the corresponding information can be accessed in $O(1)$ while accessing node x.

- `moment`: $\texttt{moment}_x$ denotes the value of k where (u, k) is the key pair corresponding to the node x.
- `timestamp`: $\texttt{timestamp}_x$ denotes the value of $\texttt{dfs_order}_u$, where the tree vertex u satisfying node x was first added to D_u.
- `timemin` and `timemax`: $\texttt{timemin}_x$ and $\texttt{timemax}_x$ denotes the minimum and the maximum value of $\texttt{timestamp}_{x'}$, where $x' \in \texttt{subtreeT}(x)$.
- `a`, `b`: two supportive integer arrays to store tags for the lazy propagation.

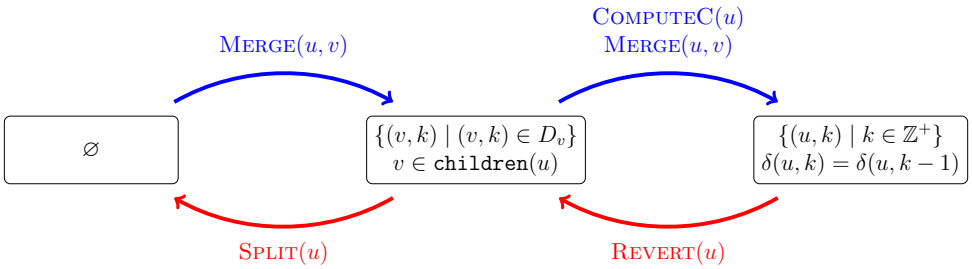


Figure 2.5: This figure shows the life cycle of D_u where arrows describe the calling order and blocks contain the current maintained information. Blue ones are called in SolvePartial(u, G, σ') and red ones are in SolveComplete(u, G).

2.6.2 Splay Trees

A splay tree supports accessing, inserting and deleting a node in an amortized $O(\log n)$ time [194]. We define the following basic interfaces for our splay tree:

- NewNode(k, t): Create a new node x. It will also create a mapping to $\texttt{timestamp}_x$, $\texttt{timemax}_x$, $\texttt{timemin}_x$, $\texttt{a}_x$ and $\texttt{b}_x$. Then it initializes
  - $\texttt{moment}_x \leftarrow k$

- $\texttt{timestamp}_x \leftarrow t, \texttt{timemin}_x \leftarrow t, \texttt{timemax}_x \leftarrow t$
- $\texttt{a}_x \leftarrow 0, \texttt{b}_x \leftarrow 0$

- $\textsc{Insert}_T(x)$: Insert the node x into the splay tree T.
- $\textsc{Delete}_T(x)$: Remove the node x from the splay tree T. Note that after removing the node x, we do not delete the information in $\texttt{timestamp}_x, \texttt{timemax}_x, \texttt{timemin}_x, \texttt{a}_x$ and $\texttt{b}_x$.

We give some notations on any splay tree T:

- $\texttt{subtreeT}(x)$ denotes the union of nodes where node x is on the path to the splay tree root.
- $\texttt{size}(x)$ denotes the number of nodes in $\texttt{subtreeT}(x)$; Specially, $\texttt{size}(T)$ denotes the number of nodes in the splay tree T.
- $x \in T$ denotes that T contains the node x;
- For any node $x \in T$, $\texttt{parentT}(x)$, $\texttt{left}(x)$ and $\texttt{right}(x)$ denotes the father, left child and right child of node x on T respectively. If not exists, the value will be `nil` by default. Furthermore, define $\texttt{childrenT}(x)$ as the set of the children for the node x. Note that `nil` is not considered as an element of $\texttt{childrenT}(x)$.
- $\texttt{root}(T)$ denotes the root node of T. If T is empty, $\texttt{root}(T)$ will be `nil`.
- $\textsc{Splay}(x)$ denotes the operation to make node x as the root by a series of rotations.
- $\texttt{rank}_T(x)$ denotes the rank of the node x in the splay tree T. Here, the rank of a node is defined as the number of nodes that precede it while performing an in-order walk.
- $\texttt{pred}_T(x)$ and $\texttt{succ}_T(x)$ denotes the predecessor and successor of the node x, respectively. Formally, if $\texttt{rank}_T(x) = k$, then $\texttt{pred}_T(x)$ is the node of the rank $k-1$ and $\texttt{succ}_T(x)$ is the node of the rank $k+1$. Specially, if there is no such node, then the predecessor (or the successor) of the node x will be considered as `nil`.

On a splay tree, we define $\textsc{FindMin}(u)$ as a subroutine to find the node x with the minimum rank on D_u. This subroutine is described in Algorithm 4. It is known that such a process runs in $O(\log n)$ amortized time.

During the splay tree maintenance in our algorithm, we need to perform the following modification to update the information:

Algorithm 4: FindMin(u)

```
x ← root(D_u);
while left(x) ≠ nil do
    x ← left(x);
Splay(x);
return x;
```

- For a given node $x \in D_u$ and two parameters a and b. Increase the value of $\texttt{moment}_y$ by $\mathsf{rank}_{\texttt{subtreeT}(x)}(y) \cdot a + b$ for all $y \in \texttt{subtreeT}(x)$.

However, it is not efficient to perform such a change for a whole $\texttt{subtreeT}(x)$ every time. Here we will use the classic *lazy propagation* trick. In general, we defer the modification on the node to the time we actually visit it. Since we always visit $\texttt{parentT}(x)$ before visiting any node x, we execute the modification on x whenever we visit $\texttt{parentT}(x)$, clearing the *lazy tag* on $\texttt{parentT}(x)$ afterward. Here we use IncTime(x, a, b) to denote a modification for node x with two parameter a and b. Specifically, for each node $x \in D_u$, we maintain two lazy tags $\texttt{a}_x$ and $\texttt{b}_x$ indicating "it should perform a IncTime(y, $\texttt{a}_x$, $\texttt{b}_x$) operation for any child $y \in \texttt{childrenT}(x)$ of the node x".

Algorithm 5: IncTime(x, a, b)

```
moment_x ← moment_x + (size(left(x)) + 1) · a + b;
a_x ← a_x + a;
b_x ← b_x + b;
```

Remark 2.6.1. *The reason we can use lazy propagation here is because the* IncTime*(x, a, b) operation follows the associative law.*

Note that although the splay tree will change forms, as long as we push down lazy tags before changing, the correctness is guaranteed. We discuss the details of external information maintenance while node rotations occur based on this in Section 2.6.3.

Furthermore, the *Dynamic Finger Theorem*, described in Theorem 2.6.1, gives us a better bound for a sequence of the access operations. This theorem is vital to our time complexity analysis.

Theorem 2.6.1 (Dynamic Finger Theorem [65, 64])**.** *Let T be a splay tree with n nodes. Consider a sequence of accesses in splay tree T (denoted as $a_1, a_2, \cdots, a_m$ and assume a_0*

is the root of the splay). Then the cost of the access sequence is bounded by

$$O\left(m+n+\sum_{i=1}^{m}\log(1+d(a_{j-1},a_j))\right),$$

where $d(x,y)$ denotes the difference between the ranks of the node x and y.

With such dynamic finger property, when merging multiple splay trees with the small-to-large trick, we can reach a total time complexity of $O(n\log n)$ where n denotes the total number of nodes. Conversely, we can split the final splay trees back in the same complexity, which can be regarded as undoing the merging. In our tree algorithm, we design two interfaces MERGE and SPLIT to support merging and splitting splay trees. Specifically, we have the following two lemmas:

Lemma 2.6.2. MERGE*$(u,\ v)$ will merge all nodes from D_v into D_u. Note that there are no node in D_v after merging. During the execution of Algorithm 17, all* MERGE *operations take $O(n\log n)$ time in total.*

Lemma 2.6.3. *If the current D_u contains all key pairs from D_u and D_v before calling* MERGE(u,v) *and no key pair from $D_{v'}$ exists if v' is after v in $\mathcal{I}$,* SPLIT*$(u,\ v)$ will extract nodes to D_v from D_u, reverting D_u, D_v from the corresponding call of* MERGE(u,v)*. After* SPLIT(u,v)*, no key pair from $D_{v'}$ exists if v' is no earlier than v in $\mathcal{I}$. During the execution of Algorithm 17, all* SPLIT *operations take $O(n\log n)$ time in total.*

2.6.3 Rotation

The fundamental operation of the splay tree is SPLAY. A SPLAY(x) operation is called whenever we access any node x. By [194] splaying the node after we access it will give us amortized $O(\log n)$ time complexity for inserting, deleting and searching. In SPLAY(x), we will make node x to the root of the splay tree while maintaining the in-order traverse of the tree unchanged. The way to do this is by performing a series of *splay steps* [194]. Each splay step might contain one or two rotations of x and its current `parentT`(x), which moves x closer to the root while the overall in-order sequence remains unchanged. More precisely, let x be an arbitrary non-root node in D_u and $y=$ `parentT`(x). a single ROTATE(x) will make x become the father of y and y become one of the two children of x.

We analyze here that as long as we call PUSHUP and PUSHDOWN at the proper timing during each splay step, we are able to maintain `timemin`, `timemax` correctly and the lazy propagation mechanism is still correct. Specifically, when a rotation happens, it might cause:

- $\texttt{timemin}_x$ and $\texttt{timemax}_x$ should be recalculated base on the current $\texttt{subtreeT}$.
- $\texttt{a}_x$ and $\texttt{b}_x$ might take effect on the wrong set of nodes.

We design PushDown(x) described in Algorithm 6 to modify $\texttt{left}(x)$ and $\texttt{right}(x)$'s information according to $\texttt{a}_x$ and $\texttt{b}_x$. After that, we clear up these two values. In this way, we ensure that these lazy tags are always taking effect on the correct set of nodes.

Algorithm 6: PushDown(u)

```
if left(u) ≠ nil then
    IncTime(left(u), a_u, b_u);
if right(u) ≠ nil then
    IncTime(right(u), a_u, b_u + (size(left(u)) + 1) · a_u);
```

After x is involved in a rotation, it is likely that $\texttt{left}(x)$ and $\texttt{right}(x)$ will change, thus $\texttt{timemin}$ and $\texttt{timemax}$ will need to be recalculated since they should denote the minimum and maximum $\texttt{timestamp}$ value of the current subtree of x.

We design PushUp(x) described in Algorithm 7 to recalculate $\texttt{timemin}_x$ and $\texttt{timemax}_x$ based on the information of its children for any node $x \in D_u$. To prove the correctness, we concentrate on the change of x's corresponding $\texttt{left}(x)$, $\texttt{right}(x)$, assuming $\texttt{left}(x)$ and $\texttt{right}(x)$'s subtree structure remains unchanged, thus having the correct value of $\texttt{timemin}$ and $\texttt{timemax}$.

Algorithm 7: PushUp(x)

```
timemin_x ← timestamp_x;
timemax_x ← timestamp_x;
if left(x) ≠ nil then
    timemin_x ← min(timemin_x, timemin_left(x));
    timemax_x ← max(timemax_x, timemax_left(x));
if right(x) ≠ nil then
    timemin_x ← min(timemin_x, timemin_right(x));
    timemax_x ← max(timemax_x, timemax_right(x));
```

Let $L = \{y \mid y \in \texttt{subtreeT}(\texttt{left}(x))\}$ and $R = \{y \mid y \in \texttt{subtreeT}(\texttt{right}(x))\}$. Specially, if $\texttt{left}(x) = \texttt{nil}$, then $L = \varnothing$ (and similarly for R). Then $\texttt{subtreeT}(x) =$

$L \cup R \cup \{x\}$. We only analyze `timemin` here since `timemax` shares the same transition logic:

$$\begin{aligned}
&\min_{y \in \texttt{subtreeT}(x)} \texttt{timestamp}_y \\
&= \min(\min_{y \in L} \texttt{timestamp}_y, \min_{y \in R} \texttt{timestamp}_y, \texttt{timestamp}_x) \\
&= \min(\texttt{timemin}_{\texttt{left}(x)}, \texttt{timemin}_{\texttt{right}(x)}, \texttt{timestamp}_x)
\end{aligned}$$

Terms are ignored when $\texttt{left}(x)$ or $\texttt{right}(x)$ is `nil`. Algorithm 7 consists of the exact transition as the aggregation from $\texttt{left}(x)$ and $\texttt{right}(x)$'s result, which compute the correct value of $\texttt{timemin}_x$ and $\texttt{timemax}_x$ as result.

The key operation to change the structure of the splay tree is $\textsc{Rotate}(x)$, as we mentioned in Figure 2.6. For any non-root node x, the $\textsc{Rotate}(x)$ procedure will make the parent of the node x become the child of x, which means the distance of the node x to the root is decreased by exactly one.

Assume y is the parent of the node x before the rotation. After calling $\textsc{Rotate}(x)$, the set of the nodes corresponding to the vertex x and y are all changed, so we need to perform PushUp operation on them. Note that y is one of the children of x, thus we need to update the information of y before updating the node x, so we need to call $\textsc{PushUp}(y)$ and $\textsc{PushUp}(x)$ in order every time we finish a $\textsc{Rotate}(x)$.

We will show the following example of how the push-up performs on the Zig, Zig-Zag and Zig-Zig operation in the splay tree, which was mentioned in [194].

- The $\textsc{Zig}(x)$ function is called once x is not the root node of the splay but $\texttt{parentT}(x)$. It will perform a Rotate operation on vertex x, so that x becomes the root of the splay. The Figure 2.6 shows the procedure of $\textsc{Zig}(x)$. So after $\textsc{Zig}(x)$, we need to call $\textsc{PushUp}(y)$ and then $\textsc{PushUp}(x)$. The details of the operation are described in Figure 2.6.

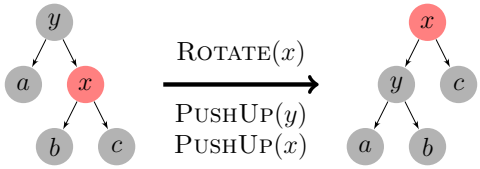


Figure 2.6: The figure corresponds to a call of $\textsc{Zig}(x)$, which is simply rotating the node x.

- The ZIG-ZIG(x) function is called once x and $y = \texttt{parentT}(x)$ is not the root node of the splay, and they are both left (or right) children. In the procedure, we will rotate the node y and then rotate the node X. After the rotation y becomes the parent of z and x becomes the parent of x. The details of the operation are described in Figure 2.7.

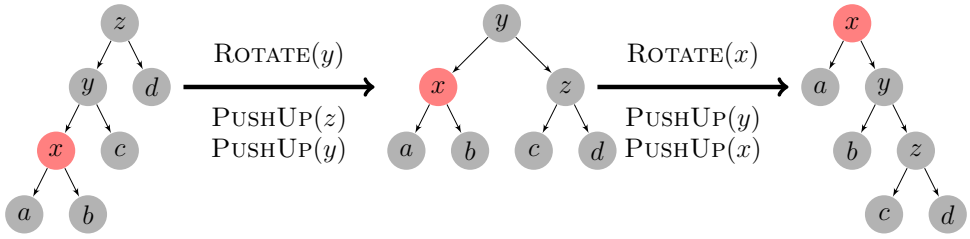


Figure 2.7: The figure corresponds to a call of ZIG-ZIG(x)

- The ZIG-ZAG(x) function is called once x and $\texttt{parentT}(x)$ is not the root node of the splay, and x is a left child and $\texttt{parentT}(x)$ is a right child, or vice-versa. In the procedure, we will rotate the node x twice. After the rotation x will be the parent of the node y and z. The details of the operation are described in Figure 2.8.

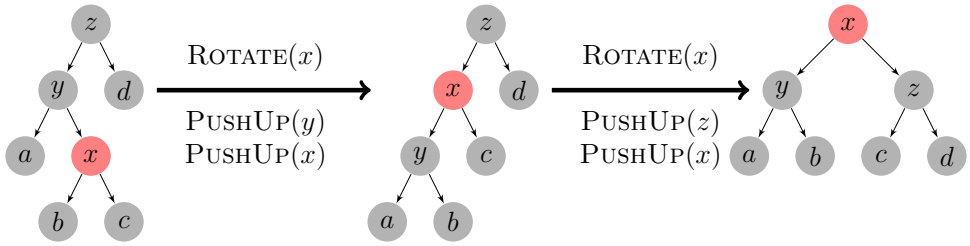


Figure 2.8: The figure corresponds to a call of ZIG-ZAG(x)

Finally, we need to consider when we should perform the PUSHDOWN operation. Once we traverse the splay tree from the root, we need to push down all the tags from the root to the current node. This implies every time we need to search a specific node, we need to push down all the nodes from the root to that node in order. This includes the INSERT, DELETE and SPLAY function in the splay tree. Furthermore, for the function FINDMIN (described in Algorithm 4), DELTAQUERY (described in Algorithm 12), FINDONEINTREE (described in Algorithm 10) and FINDONEBEFORETREE (described in Algorithm 11), it is equivalent to search a specific node in the splay tree. Since we will splay the node we searched in all these functions, the tags will be pushed down correctly in the SPLAY procedure.

2.6.4 Merging by Small-To-Large Technique

We will analyze the MERGE (Algorithm 8) operation by proving the following lemma.

Lemma 2.6.2. MERGE*(u, v) will merge all nodes from D_v into D_u. Note that there are no node in D_v after merging. During the execution of Algorithm 17, all* MERGE *operations take $O(n \log n)$ time in total.*

Algorithm 8: MERGE(u, v)

```
1 if size(D_u) < size(D_v) then
2 |   res_v ← 1;
3 |   Swap (D_u,D_v);
4 else
5 |   res_v ← 0;
6 for x ∈ D_v by increasing order do
7 |   insert(D_u, x);
```

When merging two splay trees D_u and D_v, we will need to use the classic small-to-large technique to merge them efficiently.

The small-to-large technique is easy to describe: when we want to merge two splay trees D_u and D_v , we always insert all nodes from the splay tree with a smaller size to the one with a larger size one by one in ascending order (Line 7).

In Algorithm 8, we need to guarantee that D_u contains all nodes from $D_v, v \in \texttt{children}(u)$ after merging. Therefore, we will simply swap D_u and D_v (Line 3) if $\texttt{size}(D_u) < \texttt{size}(D_v)$ (Line 1). Here we store a boolean value $\texttt{res}_v$ to keep track of such swapping (Line 2 and Line 5).

The analysis is also very trivial to reach a $O(n \log^2 n)$ upper bound: We can see that the time cost of all MERGE operations is equivalent to merging a series of splay trees into one. Note that every MERGE's time cost is proportional to the size of the smaller splay. In the next MERGE, the size value of the smaller splay is at most doubled than the previous one. Therefore, for each node, it will be inserted into other splay trees at most $O(\log n)$ times. Since the insertion on splay trees is $O(\log n)$ amortized, it costs $O(n \log^2 n)$ time in total. However, if we guarantee that all nodes are inserted in increasing order during the process (Line 6), we will be able to trigger Theorem 2.6.1 to reach a better bound.

Corollary 2.6.1. *Given a series of splay trees: $T_1, T_2, \cdots, T_k$ such that $\sum_{i=1}^{k} \texttt{size}(T_i) = n$, if we call* Merge *$k-1$ times in arbitrary order to merge them into one, the total time cost for all* Merge *operations is $O(n \log n)$. Note that between any two* Merge *operations, it is allowed to have other operations on splay trees separately.*

We will need a modified version of Theorem 2.6.1:

Theorem 2.6.2 ([42])**.** *The total time to perform n insertions on a splay tree of size m is $O(n \log \frac{m+n}{n})$ if the insertions are performed on items in increasing order of ranks.*

Now we are ready to prove Lemma 2.6.2.

Proof of Corollary 2.6.1. The goal is to proof the overall complexity is no more than $C \cdot n \log n$, where C is a deterministic constant. We can do the mathematical induction here. For $k = 1$, it is obviously true. Now assuming it is true for $k = 1 \cdots i-1$, now we look at the case where $k = i$. Focusing on the last $\textsc{Merge}(A, B)$ operation we performed where A and B are splay trees, we have $\texttt{size}(A) + \texttt{size}(B) = n$. By the inductive hypothesis, we know that the previous merging processes cost $C \cdot (\texttt{size}(A) \log \texttt{size}(A) + \texttt{size}(B) \log \texttt{size}(B))$ time. Without loss of generality, we assume $\texttt{size}(A) > \texttt{size}(B)$. Therefore, we will insert the nodes of B to A one by one in increasing order. By Theorem 2.6.2, this costs at most $C_1 \cdot \texttt{size}(B) \log \frac{n}{\texttt{size}(B)}$ time, where C_1 is a constant. Taking $C \geq C_1$, we get $C \cdot \texttt{size}(A) \log \texttt{size}(A) + C \cdot \texttt{size}(B) \log \texttt{size}(B) + C_1 \cdot \texttt{size}(B) \log(\frac{n}{\texttt{size}(B)}) \leq C \cdot \texttt{size}(A) \log n + C \cdot \texttt{size}(B) \log \texttt{size}(B) + C \cdot \texttt{size}(B) \log(\frac{n}{\texttt{size}(B)}) = C \cdot n \log n$, thus we complete the induction. □

By Corollary 2.6.1 and previous analysis, Lemma 2.6.2 is proved.

2.6.5 Splitting by Undoing Merges

We will analyze the Split (Algorithm 9) operation by proving the following lemma.

Lemma 2.6.3. *If the current D_u contains all key pairs from D_u and D_v before calling* Merge*(u, v) and no key pair from $D_{v'}$ exists if v' is after v in $\mathcal{I}$,* Split*(u, v) will extract nodes to D_v from D_u, reverting D_u, D_v from the corresponding call of* Merge*(u, v). After* Split*(u, v), no key pair from $D_{v'}$ exists if v' is no earlier than v in $\mathcal{I}$. During the execution of Algorithm 17, all* Split *operations take $O(n \log n)$ time in total.*

Algorithm 9: SPLIT(u,v)

```
1  while true do
2      x ← nil;
3      if res_v = 0 then
4          x ← FindOneInTree(u, v);
5      else
6          x ← FindOneBeforeTree(u, v);
7      if x = nil then
8          break;
9      Delete(D_u, x);
10     insert(D_v, x);
11 if res_v = 1 then
12     Swap(D_u, D_v);
```

The purpose of SPLIT(u, v) is to derive D_v back to the previous state. Based on the assumption that the current D_u contains nodes from the previous D_u and D_v only before MERGE(u, v), we can do this by finding nodes from D_v in increasing order on D_u. In this way, the process can be regarded as deleting and inserting nodes in increasing order in both D_u and D_v, which is a symmetric process to MERGE(u, v), thus sharing the same overall complexity.

Since MERGE(u, v) follows the small-to-large mechanism, it might execute a SWAP(D_u, D_v). Since we always plan on merging D_v into D_u, we need two similar functions to find the minimum rank node belonging to D_v or D_u. Here we refer to D_u and D_v before the possible swapping. These two functions are described in Algorithm 10 and Algorithm 11.

Lemma 2.6.4. FINDONEINTREE(u, v) *is able to find the minimum rank node belongs to the original* D_v *and* FINDONEBEFORETREE(u, v) *is able to find the minimum rank node belongs to the original* D_u*.*

Proof of Lemma 2.6.4. We first analyze FINDONEINTREE(u, v) here, and the analysis of FINDONEBEFORETREE(u, v) proceeds similarly.

Determining if a node x in D_u comes from the original D_v is equivalent to determining whether p is in `subtree`(v), where p is the tree vertex that node x is generated by NEWNODE during the execution of UPDATE(p).

Algorithm 10: FINDONEINTREE(u, v)

```
1  x ← root(D_u);
2  if x = nil or timemax_x < dfs_order_v then
3      return nil;
4  while true do
5      if left(x) ≠ nil and timemax_left(x) ≥ dfs_order_v then
6          x ← left(x);
7      else if timestamp_x ≥ dfs_order_v then
8          break;
9      else
10         x ← right(x);
11 SPLAY(x);
12 return x;
```

Such verification can be done by comparing the $\texttt{timestamp}_x$ and $\texttt{dfs_order}_v$. Since we assume that no key pair from $D_{v'}$ exists if v' is after v in $\mathcal{I}$, we can determine if x is in the original D_v by checking if $\texttt{timestamp}_x \geq \texttt{dfs_order}_v$ holds. If the inequality holds, it means that x is generated no earlier than visiting vertex v. Therefore, it must come from $\texttt{subtree}(v)$.

Now we further generalize this condition to a tree walk on the splay tree. We need to check if there is any node y in $\texttt{subtreeT}(x)$ satisfying this condition. Since we maintain `timemax` for every splay tree node, we can determine this by checking if $\texttt{timemax}_x \geq \texttt{dfs_order}_v$ holds.

Algorithm 10 is a tree walk supported by the above verification. We first initialize the x by $\texttt{root}(D_u)$ (Line 1). If x is empty or x does not satisfy the inequality (Line 2), there is no such node that exists in the entire D_u, thus, we return `nil`.

Since we are finding the one with the smallest rank, we first check if $\texttt{left}(x)$ satisfies this condition (Line 5). If so, we go to $\texttt{left}(x)$ (Line 6) and continue the walking. Otherwise, we check if the current node x satisfies (Line 7). If so, we find the one with the minimum rank, and thus we exit the loop (Line 8). If both verifications fail, since we already determine there is at least one node satisfying the condition, we go to $\texttt{right}(x)$ directly (Line 10) and continue walking.

After finding the desired node x, we will need to call SPLAY(x) to guarantee the amortized access cost (Line 11).

Algorithm 11: FindOneBeforeTree(u, v)

```
1  x ← root(D_u);
2  if x = nil or timemin_x ≥ dfs_order_v then
3      return nil;
4  while true do
5      if left(x) ≠ nil and timemin_left(x) < dfs_order_v then
6          x ← left(x);
7      else if timestamp_x < dfs_order_v then
8          break;
9      else
10         x ← right(x);
11 Splay(x);
12 return x;
```

In FindOneBeforeTree (Algorithm 11), the subtree verification condition becomes $\texttt{timemin}_x \geq \texttt{dfs_order}_v$ for any $\texttt{subtreeT}(x)$ on the splay tree. The rest analysis remains the same as above. □

Now we are ready to prove Lemma 2.6.3.

Proof. In Merge(u, v), we store $\texttt{res}_v$ to keep track of whether the swapping occurs. Therefore, by the value of $\texttt{res}_v$, we can tell if we use FindOneInTree (Line 4) or FindOneBeforeTree (Line 6) to find the minimum rank node in D_u that belongs to the original D_v. By Lemma 2.6.4 we know it will return the correct node if it exists. We use a variable x to store the search result (Line 2). If $x = \texttt{nil}$, then all nodes are found, and thus we exit the loop (Line 8). Every time we find a node x, we will delete it from D_u (Line 9) and insert it back to D_v (Line 10). In the end, D_v will be restored and no node from D_u belongs to D_v. Lastly, we also need to swap back to revert the previous swapping in Merge if necessary (Line 11 and Line 12).

One can notice that we are actually splitting nodes in the same *small-to-large* idea as Merge. Moreover, it is exactly the symmetric of nodes' insertions in Merge(u, v). Since on a splay tree, both Insert and Delete have the dynamic finger property. We can derive a similar theorem to Theorem 2.6.2 that the total time to perform n deletions on a splay tree of size m is $O(n \log \frac{m+n}{n})$. Same as Lemma 2.6.2, we can eventually derive the total time cost for all Split as $O(n \log n)$ as for all Merge operations. □

2.6.6 Difference Aggregation by Tree Walk

We will analyze DeltaQuery (Algorithm 12) by proving the following lemma.

Lemma 2.6.5. DeltaQuery(u, k) *will return the correct value of* $\delta(u, k)$*. During the execution of Algorithm 17, the* DeltaQuery *operation takes* $O(n \log n)$ *time in total.*

Algorithm 12: DeltaQuery(u, k)

```
1  now ← root(D_u);
2  las ← nil;
3  rank ← 0;
4  while now ≠ nil do
5      PushDown(now);
6      las ← now;
7      if moment_now ≤ k then
8          rank ← rank + size(left(now)) + 1;
9          now ← right(now);
10     else
11         now ← left(now);
12 Splay(las);
13 return k − rank;
```

Proof. By Lemma 2.6.1, $\delta(u, k) = k - |\{x \mid x \in D_u, \texttt{moment}_x \leq k\}|$.

Since we use a splay tree to maintain all these ordered key pairs, to count the number of key pairs for a certain prefix, we will conduct a top-down tree walk on the splay tree starting from the root. This process is described in Algorithm 12. We use `now` to represent the current visiting node, initialized as `root`(D_u) (Line 1) since we begin from the root. Additionally, we use *las* to store the previously visited node (updated in Line 6). We also initialize *rank* to be 0 (Line 3), denoting the number of key pairs we should count.

During the loop (Line 4), we are finding node `now` with maximum rank such that $\texttt{moment}_{\texttt{now}} \leq k$ while counting the number of key pairs. Specifically, if the current $\texttt{moment}_{\texttt{now}} \leq k$ (Line 7), by the property of the binary search tree, all key pairs in `subtreeT(left(now))` together with `now` satisfy the condition. Therefore, we directly increase *rank* by `size(left(now))`+1 (Line 8), then we go to `right(now)` for continue searching. Similarly, if $\texttt{moment}_{\texttt{now}} > k$, all key pairs in `subtreeT(right(now))` should be ignored. Thus we should go to

$\texttt{left}(\texttt{now})$. Note that during the loop, whenever we visit a new node, we need to call $\texttt{PushDown}(\texttt{now})$ to guarantee the correctness of $\texttt{moment}_{\texttt{now}}$.

After finding the pair with maximum rank, we also successfully count the number of key pairs (u, k') satisfying the condition $k' \leq k$. Before returning the value computed by the formula (Line 13), we also need to splay the last accessed node to the root by calling $\textsc{Splay}(las)$ (Line 12).

Since such a walk process is equivalent to the access on the splay tree, it has a $O(\log n)$ amortized cost and DeltaQuery will be called $O(n)$ times in Algorithm 3, the total time cost is $O(n \log n)$. □

2.6.7 Computing Partial Firing Numbers by Pop-Up Mechanism

We will analyze the ComputeC operation(Algorithm 13) by proving the following lemma.

Lemma 2.6.6. *When calling* $\textsc{ComputeC}(u)$*, if the current* D_u *is the collection of all key pairs from* $D_v, v \in \texttt{children}(u)$*,* $\textsc{ComputeC}(u)$ *will compute the correct* $\mathbf{c}^{\downarrow}(u)$*. During the execution of Algorithm 17, all* ComputeC *operations take* $O(n \log n)$ *time in total.*

Algorithm 13: $\textsc{ComputeC}(u)$

```
1  now ← 0;
2  count ← 0;
3  Q_u ← ∅ ;
4  while D_u ≠ nil do
5      x ← FindMin(D_u);
6      if moment_x = now or count + [u ≠ r] · (moment_x − 1 − now) ≤ σ'_u − degree(u) then
7          Delete(D_u, x);
8          Q_u.append(x);
9          count ← count + 1 + [u ≠ r] · (moment_x − now);
10         now ← moment_x;
11     else
12         break;
13 p ← [u ≠ r] · max(0, σ'_u − count − (degree(u) − 1));
14 return now + p;
```

Proof. By Lemma 2.4.7, we know that the value of $\mathbf{c}^{\downarrow}(u)$ is exactly the non-negative smallest integer k such that $\psi_u(k) < \texttt{degree}(u)$. Recall that

$$\psi_u(k) = \sigma_u - k \cdot \texttt{degree}(u) + \sum_{v \in \texttt{children}(u)} \delta(v, k).$$

Then, we have

$$\psi_u(k) = \sigma_u - \left(k \cdot |\texttt{children}(u)| - \left(\sum_{v \in \texttt{children}(u)} \delta(v, k) \right) \right) - k \cdot [u \neq r]. \tag{2.4}$$

Because we assume that ComputeC is called when D_u is updated to the collection of all key pairs in $\texttt{children}(u)$. This should eliminate the summation sign with regard to children in Equation (2.4) via introducing the formula below. That is, by Lemma 2.6.1, we have

$$\sum_{v \in \texttt{children}(u)} \delta(v, k) = k \cdot |\texttt{children}(u)| - |\{x \mid x \in D_u, \texttt{moment}_x \leq k\}|. \tag{2.5}$$

By substituting the term in Equation (2.4) we have

$$\psi_u(k) = \sigma_u - |\{x \in D_u \mid \texttt{moment}_x \leq k\}| - k \cdot [u \neq r]. \tag{2.6}$$

Now we will prove that Algorithm 13 is sufficient to find the minimum k. We use `now` to denote the current value of k. Since in D_u the nodes are ordered by the value of `moment` in increasing order, we can repeatedly extract the node with the smallest rank from D_u and see if it could increase the value of `now` while $\psi_u(\texttt{now}) \geq \texttt{degree}(u)$ holds.

In the beginning, we initialize `now` as 0 since k is non-negative (Line 1). As `now` increases, we use the variable `count` to keep track of $|\{x \mid x \in D_u, \texttt{moment}_x \leq \texttt{now}\}| + \texttt{now} \cdot [u \neq r]$. Since we extract the minimum rank node repeatedly (Line 4), we need to delete it from D_u (Line 7) to prevent redundant enumeration. We use an additional list $\boldsymbol{Q}_u$ for each u to store the deleted node temporarily (Line 8). The list $\boldsymbol{Q}_u$ is initialized as empty (Line 3) in the beginning.

In every turn, we first extract the minimum rank node x (Line 5). There are two different cases (Line 6):

- If $\texttt{moment}_x = now$, since the $\texttt{moment}_x \le now$ in the definition of $\texttt{count}$ is hold for the x now, we need to increase $\texttt{count}$ by 1.
- If $\texttt{moment}_x \neq now$, we test if $\texttt{now}$ can be increased to $\texttt{moment}_x - 1$ without triggering the terminate condition $\psi_u(\texttt{now}) < \texttt{degree}(u)$. Since there is no $y \in D_u$ such that $\texttt{now} < \texttt{moment}_y \le \texttt{moment}_x - 1$, by definition, $|\{x \mid x \in D_u, \texttt{moment}_x \le k\}| + k \cdot [u \neq r]$ equals to $\texttt{count} + [u \neq r] \cdot (\texttt{moment}_x - 1 - k)$. If such value $\le \sigma'_u - \texttt{degree}(u)$, then the desired k is at least larger than $\texttt{moment}_x - 1$, which allows us to increase $\texttt{now}$ to $\texttt{moment}_x$.

For both cases, $\texttt{now}$ will be increased (or kept) to $\texttt{moment}_x$ (Line 10). We update $\texttt{count}$ correspondingly (Line 9) and continue enumeration.

When the loop terminates, the term $|\{x \mid x \in D_u, \texttt{moment}_x \le k\}|$ has already been accumulated correctly. Now if we increase $\texttt{now}$ by p, the increment of $\texttt{count}$ will be $p \cdot [u \neq r]$. Assuming $u \neq r$, we can compute the maximum possible p to make $\psi_u(\texttt{now}) < \texttt{degree}(u)$ by simple calculation, which is the difference between $\sigma'_u - \texttt{count}$ and $\texttt{degree}(u) - 1$. We take the max value between this difference and 0 to avoid corner cases (Line 13). The desired value k will be $\texttt{now} + p$ (Line 14).

Since for each node x, it will only be deleted from any D_u and inserted into the corresponding $\boldsymbol{Q}_u$ once. By Lemma 2.6.12, there are $O(n)$ nodes in total. Since each deletion in D_u costs $O(\log n)$ in amortized, the total time cost will be $O(n \log n)$. □

2.6.7.1 DeltaSum Calculation

We will analyze the DeltaSum (Algorithm 14) operation by proving Lemma 2.6.7.

Algorithm 14: DeltaSum(u)

```
1 return c↓(u) · |children(u)| − |Q|;
```

Lemma 2.6.7. DeltaSum(u) *returns* $\sum_{v \in \texttt{children}(u)} \delta(v, \mathbf{c}^{\downarrow}(u))$ *correctly, which is equal to* $\mathbf{c}^{\downarrow}(u) \cdot |\texttt{children}(u)| - \texttt{size}(Q_u)$ *in the Algorithm 2. Each* DeltaSum *operation takes* $O(1)$ *time.*

Proof. By Equation (2.5), we have

$$\sum_{v \in \texttt{children}(u)} \delta(v, k) = k \cdot |\texttt{children}(u)| - |\{x \mid x \in D_u, \texttt{moment}_x \le k\}|$$

when D_u is exactly the union of $D_v, v \in \texttt{children}(u)$. After we merge all $D_v, v \in \texttt{children}(u)$ to D_u, we execute the ComputeC before calling DeltaSum. Note that the term $\left|\left\{x \mid x \in D_u, \texttt{moment}_x \leq \mathbf{c}^{\downarrow}(x)\right\}\right|$ should be the number of node x such that $moment_x \leq \mathbf{c}^{\downarrow}(x)$ in the original D_u before ComputeC is called. By the proof of Lemma 2.6.6, we know that ComputeC splits out all these nodes xs and stores in $\boldsymbol{Q}_u$ temporarily. Therefore, we only need to return $k \cdot |\texttt{children}(u)| - |\boldsymbol{Q}_u|$ by the definition, which costs $O(1)$ calculation. □

2.6.8 Moment Updating and Reverting

We analyze the Update(Algorithm 15) and Revert(Algorithm 16) operation by proving Lemma 2.6.8 and Lemma 2.6.13. We prove the following lemmas for any non-root vertex u and consider the root vertex in the last part of this section.

Lemma 2.6.8. *For any vertex $u \in V(G)$, if the current D_u is the union of all key pairs $(v,k), v \in \texttt{children}(u), k > \mathbf{c}^{\downarrow}(u)$, Update$(u)$ will update D_u correctly that it contains all key pairs (u,k). During the execution of Algorithm 17, all Update operations take $O(n \log n)$ time in total.*

Algorithm 15: Update(u)

```
1 IncTime(root(D_u), 0, −c↓(u));
2 num_u ← degree(u) − 1 − σ'_u;
3 for i in [1, num_u] do
4     insert(D_u, NewNode (0, dfs_order_u));
5 IncTime (root(D_u), 1, 0);
```

To analyze Update, first note that D_u contains all nodes x such that $(v, \texttt{moment}_x) \in D_v, v \in \texttt{children}(u), \texttt{moment}_x > \mathbf{c}^{\downarrow}(u)$ and they are all sorted by $\texttt{moment}_x$. Calling Update modifies the information represented by these nodes so that D_u will contain all key pairs of the current vertex u. We will show that this can be done without reordering the nodes.

By Definition 2.6.1, D_u should contain all key pairs (u,k) such that $\delta(u,k) = \delta(u,k-1)$. We give the following lemma to verify any positive integer k.

Lemma 2.6.9. *For any positive integer k, (u,k) is a key pair of u if and only if*

$$b - \left|\left\{x \in D_u \mid \texttt{moment}_x \leq \mathbf{c}^{\downarrow}(u) + (k-1-b)\right\}\right| < \texttt{degree}(u) - 1 - \sigma'_u \tag{2.7}$$

where

$$b = k - 1 - \delta(u, k-1). \tag{2.8}$$

.

Proof. By the definition of δ (Definition 2.4.3), $\delta(u,k) = \delta(u,k-1)$ if and only if vertex u will not be full after proceeding the following process:

- Add $k-1$ chips to the vertex u.
- Firing the full vertices in $\texttt{subtree}(u)$ until the configuration is local terminal in $\texttt{subtree}(u)$.
- Add one more chip to the vertex u.

Therefore, by definition of firing, we have:

$$\tilde{\sigma}_u + (k-1) - \texttt{degree}(u) \cdot \left(\mathbf{c}^{\downarrow}(u) + \delta(u,k-1)\right) + \sum_{v \in \texttt{children}(u)} \delta(v, \mathbf{c}^{\downarrow}(u) + \delta(u,k-1)) < \texttt{degree}(u) - 1 \tag{2.9}$$

On the left-hand side, $\tilde{\sigma}_u$ denotes the number of chips left on u after the configuration becomes local terminal in all $\texttt{subtree}(v), v \in \texttt{children}(u)$. Then if we add $k-1$ chips on u (Definition 2.4.3), all the $\tilde{\sigma}_u + (k-1)$ chips will cause $\mathbf{c}^{\downarrow}(u) + \delta(u,k-1)$ firings on u in total. In such scenario, every $v \in \texttt{children}(u)$ will receive $\mathbf{c}^{\downarrow}(u) + \delta(u,k-1)$ chips and then return $\delta(v, \mathbf{c}^{\downarrow}(u) + \delta(u,k-1))$ (Definition 2.4.3). Since adding one more chip will not cause a new firing, the left-hand side should be less than $\texttt{degree}(u) - 1$.

So the left-hand side of Equation (2.9) is equal to

$$\tilde{\sigma}_u + (k-1) - \left(\mathbf{c}^{\downarrow}(u) + \delta(u,k-1)\right) - \sum_{v \in \texttt{children}(u)} \Big(\left(\mathbf{c}^{\downarrow}(u) + \delta(u,k-1)\right) - \delta(v, \mathbf{c}^{\downarrow}(u) + \delta(u,k-1)) \Big) \tag{2.10}$$

Since $b = k-1-\delta(u,k-1)$, one can observe that b is the number of nodes x satisfying $\texttt{moment}_x \leq k-1$ in the final D_u. Now we can write Equation (2.10) as

$$\tilde{\sigma}_u + b - \mathbf{c}^{\downarrow}(u) - \sum_{v \in \texttt{children}(u)} \Big(\big(\mathbf{c}^{\downarrow}(u) + (k-1-b)\big) - \delta(v, \mathbf{c}^{\downarrow}(u) + (k-1-b)) \Big) \tag{2.11}$$

By Lemma 2.6.1, this is equal to

$$\tilde{\sigma}_u + b - \mathbf{c}^{\downarrow}(u) - \sum_{v \in \texttt{children}(u)} \sum_{i=1}^{\mathbf{c}^{\downarrow}(u)+(k-1-b)} [\delta(v,i) = \delta(v,i-1)] \tag{2.12}$$

Assuming we have a data structure T storing the union of all key pairs $(v,k), v \in \texttt{children}(u)$. This is equivalent to D_u before executing ComputeC(u). We can further transform Equation (2.12) to

$$\tilde{\sigma}_u + b - \mathbf{c}^{\downarrow}(u) - \big|\big\{x \in T \mid \texttt{moment}_x \leq \mathbf{c}^{\downarrow}(u) + (k-1-b)\big\}\big| \tag{2.13}$$

Noticed that we have $D_u \subset T$ and

$$\big|\big\{x \in T \backslash D_u \mid \texttt{moment}_x \leq \mathbf{c}^{\downarrow}(u) + (k-1-b)\big\}\big| = |\boldsymbol{Q}_u| \tag{2.14}$$

also

$$\tilde{\sigma}_u - \mathbf{c}^{\downarrow}(x) - |\boldsymbol{Q}_u| = \tilde{\sigma}_u + \textsc{DeltaSum}(u) - \texttt{degree}(u) \cdot \mathbf{c}^{\downarrow}(x) = \sigma'_u \tag{2.15}$$

Thus Equation (2.13) is equivalent to

$$\sigma'_u + b - \big|\big\{x \in D_u \mid \texttt{moment}_x \leq \mathbf{c}^{\downarrow}(u) + (k-1-b)\big\}\big| \tag{2.16}$$

In all, (u,k) is a key pair if, and only if, we have

$$b - \big|\big\{x \in D_u \mid \texttt{moment}_x \leq \mathbf{c}^{\downarrow}(u) + (k-1-b)\big\}\big| < \texttt{degree}(u) - 1 - \sigma'_u \tag{2.17}$$

□

Lemma 2.6.10. *For any $k \in [1, \texttt{degree}(u) - 1 - \sigma'_u]$, the inequality Equation (2.7) is satisfied. Therefore, (u, k) is a key pair of u by Lemma 2.6.9.*

Proof. We can prove it by doing the mathematical induction on k. Assuming the legal k forms a prefix $[1, p-1], p \leq \texttt{degree}(u) - 1 - \sigma'_u$, we have $\delta(u, p-1) = 0$ and thus we have $b = p - 1$. So, we have $p - 1 - b = 0$. Because in the current D_u, there is no node x such that $\texttt{moment}_x \leq \mathbf{c}^{\downarrow}(u)$. Therefore, the left-hand side of Equation (2.7) has only b left. Since $b = p - 1 < \texttt{degree}(u) - 1 - \sigma'_u$, the inequality holds for p. □

Lemma 2.6.11. *Let y be the node in D_u such that $\mathsf{rank}_{D_u}(y) = b - (\texttt{degree}(u) - 1 - \sigma'_u)$, and x be the node in D'_u such that $\mathsf{rank}_{D'_u}(x) = b$, where D'_u is the final D_u that stores all key pairs (u, k). Then the equation $\texttt{moment}_x = \texttt{moment}_y - \mathbf{c}^{\downarrow}(u) + b$ holds for all $b > \texttt{degree}(u) - 1 - \sigma'_u$.*

Proof. Firstly, we assume that the first $b \geq \texttt{degree}(u) - 1 - \sigma'_u$ key pairs are determined for D'_u. This is because the first $\texttt{degree}(u) - 1 - \sigma'_u$ key pairs are determined in Lemma 2.6.10.

Now we will find the next key pair (u, k) of rank $b+1$, which is the minimum k satisfying Lemma 2.6.9. Note that b here denotes the number of key pairs (u, k') where $k' \leq k - 1$, can be rewritten as $k - 1 - \delta(u, k-1)$ (Lemma 2.6.1) which happens to be the same b as in Equation (2.8). Therefore, we can use this b to test the correctness by Lemma 2.6.9. Note that, this works only when k is less or equal to the minimum possible one. That is equivalent to say, if there are multiple ks satisfying Lemma 2.6.9 concerning b, we should only take the minimum one.

Now we claim that k is equal to $\texttt{moment}_y - \mathbf{c}^{\downarrow}(u) + b + 1$ where y is the $b+1-(\texttt{degree}(u) - 1 - \sigma'_u)$-th key pair in current D_u.

To prove this, we first substitute this value into $\mathbf{c}^{\downarrow}(u) + (k - 1 - b)$, the term

$$\left|\left\{x \in D_u \mid \texttt{moment}_x \leq \mathbf{c}^{\downarrow}(u) + (k - 1 - b)\right\}\right| \tag{2.18}$$

becomes

$$\left|\{x \in D_u \mid \texttt{moment}_x \leq \texttt{moment}_y\}\right|,$$

which is no less than the rank of y, as $b + 1 - (\texttt{degree}(u) - 1 - \sigma'_u)$.

Therefore, the left-hand side of Lemma 2.6.9 is no greater than $(\texttt{degree}(u) - 1 - \sigma'_u) - 1$. Since this is less than the right-hand side of Equation (2.7), the inequality holds. Notice that if we decrease k by 1, the term Equation (2.18) will become strictly less than the rank of y, thus the left-hand side of Lemma 2.6.9 is larger than $(\texttt{degree}(u) - 1 - \sigma'_u) - 1$, which

is no less than the right-hand side of Lemma 2.6.9. Therefore, this claimed value k is the smallest possible k as desired.

Since $\left|\left\{x \in D_u \mid \texttt{moment}_x \leq \mathbf{c}^{\downarrow}(u) + (k-1-b)\right\}\right|$ is no greater than $|D_u|$, in Lemma 2.6.9, we have

$$b + 1 \leq \texttt{degree}(u) - 1 - \sigma'_u + |D_u|, \tag{2.19}$$

which is exactly the number of key pairs described in Lemma 2.6.10 and Lemma 2.6.11. Therefore, no key pair belonging to vertex u is missing. □

Lemma 2.6.12. NewNode *will be called $O(n)$ times only in the whole execution of Algorithm 17. It means that at any moment while executing Algorithm 17, there are $O(n)$ nodes storing in any D_u or $\boldsymbol{Q}_u$ in total.*

Proof. We can see that NewNode will be called num_u times (Line 3) in each Update(u). From Line 2 we can see that num is at most $\texttt{degree}(u)-1$. Therefore, in all Update$(u), u \in V(G)$, NewNode will be called at most $\sum_{u \in V(G)} \texttt{degree}(u) = O(n)$ times. Since no duplicating operation is involved throughout all operations, there are $O(n)$ nodes storing in any D_u or $\boldsymbol{Q}_u$ in total at any moment while executing Algorithm 17. □

Now we are ready to prove Lemma 2.6.8.

Proof of Lemma 2.6.8. With Lemma 2.6.10 and Lemma 2.6.11, we know how to modify current D_u into one with all key pairs belonging to u. Specifically, Lemma 2.6.10 implies the first $\texttt{degree}(u) - 1 - \sigma'_u$ key pairs and Lemma 2.6.11 implies all nodes in current D_u can be directly modified altogether without changing the relative order among them. This allows us to call IncTime to proceed with the update. This is vital because we are modifying $\texttt{moment}$ while D_u is sorted by $\texttt{moment}$.

Firstly, we call IncTime$(\texttt{root}(D_u), 0, -\mathbf{c}^{\downarrow}(u))$ to add a constant $-\mathbf{c}^{\downarrow}$ for every existing node in current D_u (Line 1). Then we create and insert $\texttt{degree}(u) - 1 - \sigma'_u$ nodes with $\texttt{moment} = 0$ (Line 2 to Line 4). Here we store $\texttt{num}_u = \texttt{degree}(u) - 1 - \sigma'_u$ for the future use. Lastly, we call IncTime$(\texttt{root}(D_u), 1, 0)$ to add a value of their rank to themselves. One can observe that these three operations will match the correct $\texttt{moment}$ value mentioned in Lemma 2.6.10 and Lemma 2.6.11. Therefore, Lemma 2.6.8 will update D_u correctly so that it contains all key pairs (u, k).

Each IncTime operation costs $O(1)$ runtime. By Lemma 2.6.12, there are $O(n)$ insertions for all Update during the execution of Algorithm 17. Since an insertion in D_u costs $O(\log n)$ in amortized time. Therefore, the total time cost is $O(n \log n)$. □

Now we will analyze REVERT, which is a process to revert the modification to D_u in both COMPUTEC(u) and UPDATE(u).

Algorithm 16: REVERT(u)

```
1  IncTime(root(D_u), −1, 0);
2  while D_u ≠ nil do
3      x ← FindMin(D_u);
4      if moment_x = 0 then
5          delete(D_u, x);
6      else
7          break;
8  IncTime(root(D_u), 0, c↓(x));
9  for x ∈ Q_u do
10     insert(rootD_u, x);
```

Lemma 2.6.13. *For any vertex $u \in V(G)$,* REVERT(u) *will revert D_u to the exact status before calling* COMPUTEC(u) *in Algorithm 2 that it contains all key pairs $(v, k), v \in$* `children`(u). *During the execution of Algorithm 17, all* REVERT *operations take $O(n \log n)$ time in total.*

Proof of Lemma 2.6.13. REVERT(u) can be divided into two parts:

- Line 1 to Line 8: Revert modification of UPDATE(u) on D_u from Line 1 to Line 5.
- Line 9 to Line 10: Revert modification of COMPUTEC(u) on D_u from Line 4 to Line 12.

It is easy to see that these revert operations are symmetric to the previous modification and thus produce the original D_u after calling REVERT(u). Since deletion and insertion on D_u both take in amortized $O(\log n)$, the time complexity for all REVERT costs the same as all UPDATE as $O(n \log n)$. □

2.6.9 Overall Analysis

We prove Theorem 2.6.3 from Lemma 2.6.5 (DELTAQUERY), Lemma 2.6.2 (MERGE), Lemma 2.6.6(COMPUTEC),Lemma 2.6.7(DELTASUM),Lemma 2.6.8 (UPDATE), Lemma 2.6.13 (REVERT), Lemma 2.6.3 (SPLIT) and Lemma 2.6.12 (Memory). After that, we prove the main result Theorem 2.1.2.

Theorem 2.6.3 (Data Structure Theorem)**.** *There exists a series of data structures* $\mathcal{D} = \{D_u, u \in V(G)\}$ *that satisfies the following:*

- *All operations of* Merge, Update, Revert, Split, ComputeC, DeltaSum *and* DeltaQuery *are correctly called and produce correct results among the entire execution of Algorithm 17.*
- *All operations of* Merge, Update, Revert, Split, ComputeC, DeltaSum *and* DeltaQuery *cost* $O(n \log n)$ *time in total among the entire execution of Algorithm 17.*
- $\mathcal{D}$ *takes* $O(n)$ *memory in total at any moment among the entire execution of Algorithm 17.*

Proof of Theorem 2.6.3. Firstly, we focus on Algorithm 2. For any leaf vertex u, D_u is set to empty, which is correctly set.

For the current vertex u, the D_v is correctly maintained as containing all key pairs (v, k) for all $v \in \texttt{children}(u)$ by inductive hypothesis.

We know that D_u contains the collection of all the key pairs from D_v for all $v \in D_u$. This is because we have called $\textsc{Merge}(u, v)$ for all $v \in \texttt{children}(u)$ in an arbitrary order $\mathcal{I}$. In each call of $\textsc{Merge}(u, v)$, by Lemma 2.6.2, D_u will contain all the nodes from D_v. This proves that D_u contains all the key pairs from D_v for all $v \in \texttt{children}(u)$. It implies that the assumption in Lemma 2.6.6 has been satisfied. Therefore, $\textsc{ComputeC}(u)$ will return the correct value of $\mathbf{c}^{\downarrow}(u)$ and transport $x \in D_u$ satisfying $moment_x \leq \mathbf{c}^{\downarrow}(u)$ to $\boldsymbol{Q}_u$. By Lemma 2.6.7, DeltaSum produces the correct result. Now D_u is the union of all key pairs $(v, k), v \in \texttt{children}(u), k > \mathbf{c}^{\downarrow}(u)$, which satisfies the assumption of Lemma 2.6.8. Therefore, $\textsc{Update}(u)$ will produce a correct D_u for u such that it contains all key pairs (u, k).

Now we analyze Algorithm 3. For a non-root vertex u, at the time we visit u, we can use the inductive hypothesis to assume the process on $\texttt{parent}(u)$ has been finished correctly. It means the value of $\mathbf{c}(\texttt{parent}(u))$ has been calculated correctly, and D_u is restored to the status before $\textsc{Merge}(u, v)$ happens. By Lemma 2.6.5, DeltaQuery produces the correct result $\delta(u, \mathbf{c}(\texttt{parent}(u)))$. By Lemma 2.6.13, $\textsc{Revert}(u)$ restore D_u to the union of all key pairs (v, k), where $v \in \texttt{children}(u)$. By enumerating in the reversed order of $\mathcal{I}$, $\textsc{Split}(u, v)$ are called repeatedly. We can see that such order of calling $\textsc{Split}(u, v), v \in \texttt{children}(u)$ will satisfy the assumption of Lemma 2.6.3. Therefore, Split will produce the correct D_v each time, which contains all key pairs of v.

The overall time complexity can be immediately derived from combining Lemma 2.6.5, Lemma 2.6.2, Lemma 2.6.6, Lemma 2.6.7, Lemma 2.6.8, Lemma 2.6.13, and Lemma 2.6.3 that all these subroutines cost $O(n \log n)$ time in total.

The space complexity is analyzed in Lemma 2.6.12. □

Algorithm 17: SOLVE(G, σ)

input : tree G, configuration σ
output: the terminal configuration σ^T of the instance $S(T, \sigma)$

```
1  r ← s;
2  σ' ← σ;
3  visit_time ← 1;
4  SolvePartial (r, G, σ');
5  SolveComplete (r, G);
6  for u ∈ V(G) do
7      σ_u ← σ_u − c(u) · degree(u);
8      for v ∈ neighbor(u) do
9          σ_v ← σ_v + c(u);
10 return σ;
```

By Theorem 2.6.3, we can prove Lemma 2.4.4 and Lemma 2.4.10. With them, we can now prove Theorem 2.1.2:

Theorem 2.1.2 (Sandpile Prediction on Trees)**.** *Given a sandpile instance $S(G, \sigma)$ such that G is a tree, there is an algorithm that determines whether S terminates and computes the terminal configuration of S in $O(n \log n)$ time, with $O(n)$ memory.*

Proof. Analysis the procedure of Algorithm 17:

- In Line 1 and Line 2 we will initialize the root r and the vector σ'. Since it is a memory copy operation, it uses $O(n)$ time and $O(n)$ extra memory.
- In Line 4 we call SOLVEPARTIAL(r, G, σ'). By Lemma 2.4.4 the procedure finishes in $O(n \log n)$ time.
- In Line 5 we call SOLVECOMPLETE(r, G, σ'). By Lemma 2.4.10 the procedure finishes in $O(n \log n)$ time.

- From Line 6 to Line 9, we will recover the terminal configuration based on the value of $\mathbf{c}(u)$ for all $\mathbf{c} \in V(G)$. The iteration of the pair (u, v) is equivalent to iterating all the edges in the graph G. Since $|E(G)| = |V(G)| - 1$ in a tree G, iterating over all edges (each edge will be iterated exactly twice) will use $O(n)$ time with $O(1)$ extra memory.

In addition to storing the tree structure with $O(n)$ memory, SolvePartial only needs a global variable σ' to pass during the recursion call, which is a vector of size n. Similarly, for SolveComplete, the algorithm only needs to store a variable u and k, which uses $O(1)$ memory on each vertex. We also need two vectors of size n to store the computed $\mathbf{c}^{\downarrow}$ and $\mathbf{c}$. By Section 2.6, there are also some additional arrays of $O(n)$ length to support operations. Therefore, excluding $\mathcal{D}$, we only need $O(n)$ memory. Since $\mathcal{D}$ takes $O(n)$ memory at any moment by Theorem 2.6.3 (proved in Section 2.6.9), the whole algorithm still takes $O(n)$ memory.

Therefore, the algorithm predicts the correct configuration in $O(n \log n)$ time with $O(n)$ memory. □

2.7 Sandpile Prediction on Paths

In this section, we mainly discuss how to modify the tree algorithm to reach $O(n)$ time complexity when the input graph is a path.

Theorem 2.7.1 (Sandpile Prediction on Paths)**.** *Given a sandpile instance $S(G, \sigma)$ such that G is $Path_n$, there is an algorithm that determines whether S terminates and computes the terminal configuration of S in $O(n)$ time, with $O(n)$ memory.*

Definition 2.7.1 (Path_n)**.** *$Path_n$ is defined as an undirected graph $G(V, E)$ such that $V = \{1, 2, \ldots, n\}$ and $E = \{(u, u+1) \mid 1 \leq u < n-1\}$.*

Since Path_n is also of the tree structure, if we call Algorithm 17 directly, we can solve any sandpile instance on Path_n with $O(n \log n)$ time and $O(n)$ memory by Theorem 2.1.2. We conjecture that the runtime is actually $O(n)$. The key idea to prove this result is through the *Deque Conjecture*, which is a corollary of the famous unproven *Dynamic Optimality Conjecure* [194]. The recent known result of the Deque Conjecture is by Seth Pettie [172].

Now we will show that by modifying SolveComplete (Algorithm 3) and Revert (Algorithm 16), Algorithm 17 will have an $O(n)$ runtime when the input graph G is Path_n. The modified algorithm does not rely on *Deque Conjecture*.

We fix the root at vertex 1, which we assume as the sink. In this way, every vertex $u \in [1, n-1]$ has exactly one child, $u+1$. Firstly, we will prove that SolvePartial(r, G, σ') (Algorithm 2) runs in $O(n)$ time.

Lemma 2.7.1. *Given an sandpile instance $S(G, \sigma)$ such that G is a $Path_n$,* SolvePartial(r, G, σ') *runs in $O(n)$ time with $O(n)$ extra memory.*

Proof. The proof is similar to the proof of Lemma 2.4.10, where the total time complexity relies on the time cost for all D_u operations. We assume currently we are at non-leaf vertex u. Since D_u is initialized to be $\varnothing$ and the current visit vertex u has exactly one child $v = u + 1$, Merge(u, v) will be called only once and merges D_v with an empty splay tree. By the small-to-large principle, D_u will inherit D_v directly taking $O(1)$ time. Therefore, throughout the whole execution of Algorithm 2, we are doing operations on one splay tree T.

By Lemma 2.6.6, the time cost of all ComputeC is dominated by $O(n)$ calls of FindMin(T) operation on T in total, which finds the minimum rank node each time. For Update(u), by Lemma 2.6.8, the time cost is dominated by the insertion of $O(n)$ nodes, each with $\texttt{moment} = 0$, which is always being inserted as the node with the minimum rank each time. Here we combine these two parts, and apply the dynamic finger theorem Theorem 2.6.1, since the rank difference between any two accesses is at most 1, the total time complexity is $O(n)$. Therefore, SolvePartial(r, G, σ') runs in $O(n)$ time.

The memory usage remains $O(n)$. □

In the original SolveComplete, although we can analyze Split(u, v) similar to Merge(u, v), for DeltaQuery and Revert, we cannot apply the dynamic finger theorem directly to achieve the linear runtime. Therefore, we propose the following alternate process Algorithm 18 (PathComplete) and Algorithm 19 (PathRevert) for the path case.

The following lemma shows an additional property for nodes in D_u, which is helpful to our path algorithm:

Lemma 2.7.2. *Consider the execution of* SolveComplete(u, G) *(Algorithm 3). For any node $x \in D_u$ such that $\texttt{moment}_x \le \mathbf{c}(\texttt{parent}(u))$, we have $\texttt{moment}_x \le \mathbf{c}(u)$ after calling* Revert(u) *(Algorithm 16).*

Proof. Let τ denote $|\{x \in D_u | \texttt{moment}_x \le \mathbf{c}(\texttt{parent}(u))\}|$. During the execution of Algorithm 3, we will change all $\texttt{moment}_x$ back to $\texttt{moment}_x - \mathsf{rank}_{D_u}(x) + \mathbf{c}^{\downarrow}(u)$ and compute

Algorithm 18: PATHCOMPLETE(u,G,*count*)

```
1  if u = r then
2      count ← 0;
3      c(u) ← 0;
4  else
5      count ← count + PATHQUERY(D_u, c(u − 1));
6      k ← c(parent(u)) − count;
7      c(u) ← c↓(u) + k;
8  count ← PATHREVERT(u,count);
9  if children(u) ≠ nil then
10     PATHCOMPLETE(children(u), G,count);
```

$\mathbf{c}(u) = \mathbf{c}^{\downarrow}(u) + k = \mathbf{c}(\texttt{parent}(u)) - \tau + \mathbf{c}^{\downarrow}(u)$ by Lemma 2.6.1. Since the relative order between nodes remains the same after calling REVERT(u, v), we only have to prove $\texttt{moment}_z \leq \mathbf{c}(u)$ where $\mathsf{rank}_{D_u}(z) = \tau$. Since we have $\texttt{moment}_z - \mathsf{rank}_{D_u}(z) + \mathbf{c}^{\downarrow}(u) = \texttt{moment}_z - \tau + \mathbf{c}^{\downarrow}(u) \leq \mathbf{c}(\texttt{parent}(u)) - \tau + \mathbf{c}^{\downarrow}(u) = \mathbf{c}(u)$, we have proved our lemma. □

Line 5 of Algorithm 17 (SOLVECOMPLETE(r, G)) will be replaced with PATHCOMPLETE$(r, G, 0)$ (Algorithm 18). An extra subroutine PATHQUERY (Algorithm 20) is also needed in our path algorithm:

Lemma 2.7.3. PATHQUERY(u, k) *returns the number of node* $x \in D_u$ *such that* $\texttt{moment}_x \leq k$ *and deletes them from* D_u.

Proof. During the process of PATHQUERY(u, k), we repeatedly find the node $x \in D_u$ where $\mathsf{rank}_{D_u}(x)$ is the minimum until D_u becomes empty (Line 2). For each x, we check if $\texttt{moment}_x \leq k$ (Line 4) holds. If so, we will delete it from D_u (Line 5) and increase the counter by 1 (Line 6). Otherwise, since nodes are ordered by `moment` from small to large, we have found all nodes satisfying the condition. Thus we exit the loop (Line 8). Since *count* keeps track of the number of nodes x with $\texttt{moment}_x \leq k$, we should return *count* as the result (Line 9). □

Now we are ready to analyze Algorithm 18.

Lemma 2.7.4 (Correctness of Algorithm 18)**.** *Algorithm 18 calculates the correct value of all* $\mathbf{c}(u)$ *for* $u \in V(G)$.

Algorithm 19: PATHREVERT(u,*count*)

```
1  IncTime(root(D_u), −1, −count);
2  if num_u ≤ count then
3      count ← count − num_u;
4  else
5      num_u ← num_u − count;
6      count ← 0;
7      while num_u > 0 do
8          delete(D_u, FindMin(D_u));
9          num_u ← num_u − 1;
10 IncTime(root(D_u), 0, c↓(x));
11 if count > 0 then
12     count ← count + |Q_u|;
13 else
14     for x ∈ Q_u do
15         insert(root(D_u), x);
16 return count ;
```

Proof. We will prove the correctness of Algorithm 18 by induction. We assume the computation on any vertex v visited before u is correct. For vertex u, we maintain a variable *count* in Algorithm 18 denoting the number of pairs $(u, \texttt{moment}_u)$ satisfying $v \leq \mathbf{c}(\texttt{parent}(u))$. When u is the root, we initialize *count* to 0. In the following, we assume u is not the root.

By Lemma 2.7.3, after executing Line 5, *count* will be increased by the number of key pairs (u, x) satisfying $x \leq \mathbf{c}(\texttt{parent}(u))$. By induction, *count* stores the number of key pairs $(\texttt{parent}(u), y)$ satisfying $y \leq \mathbf{c}(\texttt{parent}(\texttt{parent}(u)))$. By Lemma 2.7.2, we know these key pairs $(\texttt{parent}(u), k)$ satisfy $k \leq \mathbf{c}(\texttt{parent}(u))$ after being reverted. Therefore, after calling PATHCOMPLETE on u (Line 5), *count* will be correctly maintained. Moreover, all nodes satisfying $\texttt{moment}_x \leq \mathbf{c}(\texttt{parent}(u))$ are deleted from D_u. This process can be regarded as aggregating all previously counted key pairs to one counter since they remain to be legal in the following recursion. By Lemma 2.6.1, $k = \delta(u, \mathbf{c}(\texttt{parent}(u)))$ is computed as $\mathbf{c}(\texttt{parent}(u)) - \mathit{count}$ (Line 6).

PATHREVERT is a modified version REVERT, which also reverts UPDATE's modification on D_u. In the beginning, the last INCTIME operation in UPDATE (Line 1) still needs to be reverted. Previously in UPDATE(u), $\texttt{num}_u$ stores the number of insertions. Since now we aggregate nodes of $\texttt{rank} \in [1, \mathit{count}]$, the actual rank of any node $x \in D_u$ should be

Algorithm 20: PATHQUERY(u, k)

```
1 count ← 0;
2 while D_u ≠ ∅ do
3     x ← FINDMIN(root(D_u));
4     if moment_x ≤ k then
5         delete(D_u, x);
6         count ← count + 1;
7     else
8         break;
9 return count;
```

$\mathsf{rank}_{D_u}(x) + count$. Thus when we revert insertions in UPDATE(u), we have to check how many of them are already aggregated in *count*. We update $\texttt{num}_u$ and *count* correspondingly (Line 2 to Line 6). If there is any inserted node in D_u needs deleted, we can simply repeatedly acquire them by calling FINDMIN(D_u) and deleting them (Line 7 to Line 9). After undoing insertions, the first INCTIME operation should be reverted as well (Line 10). Lastly, to deal with nodes stored in bq_u, we first check if nodes in bq_u should be added back to D_u (Line 14 to Line 15). If not, we simply increase *count* (Line 12). This can be done by checking if *count* is positive (Line 11. We return the new *rank* back to PATHCOMPLETE in the end (Line 16). After PATHREVERT, $\mathbf{c}(u)$ is computed as $\mathbf{c}^{\downarrow}(u) + k$ by Lemma 2.4.9. Then we continue visiting u's only child, if it exists.

By applying mathematical induction on the arguments above, the algorithm proceeds the correct value of $\mathbf{c}(u)$ for all $u \in V(G)$, which proves the correctness of Algorithm 18. □

Lemma 2.7.5 (Time and Memory Used in Lemma 2.7.4)**.** *Algorithm 18 calculates the correct value of all* $\mathbf{c}(u)$ *for* $u \in V(G)$ *in a total of* $O(n)$ *time and* $O(n)$ *memory.*

Proof. Similar to the proof of Lemma 2.4.10, the total time complexity relies on the time cost for all D_u operations. PATHQUERY is implemented as deleting the node with the minimum rank after finding it. Thus in each PATHQUERY, every node will be deleted exactly once except for one node which will only be found but not deleted. In PATHREVERT($u, rank$), we can have a similar analysis for `insert`, `delete` and FINDMIN.

Overall, there are $O(n)$ times of operations of these three kinds in total. Notice that all these operations during the execution of Algorithm 18 access the node with the minimum rank. By applying the dynamic finger theorem Theorem 2.6.1, the total time complexity is $O(n)$. By Lemma 2.6.12, the memory usage is also $O(n)$ same as the previous SOLVECOMPLETE. □

Combining Lemma 2.7.1, Lemma 2.7.4, Lemma 2.7.5, we are able to prove Theorem 2.7.1.

2.8 Deferred Proofs

To prove Lemma 2.4.1, we will first give the following lemmas.

Lemma 2.8.1. *Let $S(G, \sigma)$ be a given sandpile instance. For two distinct vertices $u, v \in V(G)$ $(u \neq v)$, if $\sigma_u \geq \texttt{degree}(u)$ and $\sigma_v \geq \texttt{degree}(v)$, then:*

1. *It is possible to fire the vertex u and then fire the vertex v.*
2. *It is possible to fire the vertex v and then fire the vertex u.*
3. *Both order of firing vertex u, v obtains the same configuration. That is, $\texttt{fire}(\texttt{fire}(\sigma, u), v) = \texttt{fire}(\texttt{fire}(\sigma, v), u)$.*

Proof of Lemma 2.8.1. By $\sigma_u \geq \texttt{degree}(u)$ and $\sigma_v \geq \texttt{degree}(v)$, we know that $\sigma^{(u)} = \texttt{fire}(\sigma, u)$ and $\sigma^{(v)} = \texttt{fire}(\sigma, v)$ both exist.

Note that $\sigma_v^{(u)} \geq \sigma_v \geq \texttt{degree}(v)$, because firing vertex u won't decrease the number of the chips on all other vertices. Similarly we have $\sigma_u^{(v)} \geq \sigma_u \geq \texttt{degree}(u)$, so $\texttt{fire}(\sigma^{(u)}, v)$ and $\texttt{fire}(\sigma^{(v)}, u)$ both exist.

Since both of the configuration exist, we have $\texttt{fire}(\texttt{fire}(\sigma, u), v) = \sigma + F(u) + F(v)$ and $\texttt{fire}(\texttt{fire}(\sigma, v), u) = \sigma + F(v) + F(u)$. By the commutative property of the vector addition, $\sigma + F(v) + F(u) = \sigma + F(u) + F(v)$, which proves that both of the configurations are equal. □

Lemma 2.8.2. *Let $S(G, \sigma)$ be a given sandpile instance. Suppose it is possible to fire the vertices $u_1, u_2, \cdots, u_t$ in order and obtain another configuration σ'. Then for any $2 \leq j \leq t$ satisfying $u_k \neq u_j$ for all $1 \leq k < j$, the following conditions are hold*

- *It is possible to fire the vertices u_j, u_1, u_2, …, u_{j-1}, u_{j+1}, u_{j+2}, …, u_t in order.*
- *The configuration obtained by firing the vertices u_j, u_1, u_2, …, u_{j-1}, u_{j+1}, u_{j+2}, $\cdots$, u_t in order equals to σ'.*

Proof of Lemma 2.8.2. Consider the original firing sequence u_1, u_2, …, u_{j-1}, u_j, u_{j+1}, …, u_t. Since $u_{j-1} \neq u_j$, by Lemma 2.8.1 we can swap the order of the vertex u_{j-1} and u_j. After that, the vertex fired before the u_j is u_{j-2}, which $u_{j-2} \neq u_j$ also holds since $u_j \neq u_k$ holds for all $1 \leq k < j$. So we can swap u_{j-2} and u_j again. Repeatably swap u_j with the previous firing vertex until u_j becomes the first vertex to be fired. In the end we will fire the vertices $u_j, u_1, u_2, \cdots u_{j-1}, u_{j+1}, u_{j+2}, \cdots, u_t$ in order, so it's possible to fire the vertices in this order while not changing the obtained configuration. □

The configuration addition and fire operation give us the following corollary.

Corollary 2.8.1. *Let $S(G, \sigma)$ and $S(G, \sigma')$ be two sandpile instances. For any vertex $u \in V(G)$, if $\sigma_u \geq \texttt{degree}(u)$, then $\texttt{fire}(\sigma + \sigma', u) = \texttt{fire}(\sigma, u) + \sigma'$.*

Proof of Lemma 2.4.1. Assume there are arbitrary two sequence of vertices $u_1, u_2, \cdots, u_a$ and $v_1, v_2, \cdots, v_b$, such that we can get a terminal configuration $\sigma^{(u)}$ by firing vertex u_1, u_2, u_a in order, and a terminal configuration $\sigma^{(v)}$ by firing vertex $v_1, v_2, \cdots, v_b$ in order. We will prove $\sigma^(u)$ must equals to $\sigma^(v)$.

We can show that by mathematical induction. Let $k = \max(a, b)$. The lemma is obviously correct for $k = 0$, as they both equal to σ.

Otherwise, consider the vertex u_1 and v_1, there are two different cases:

- If $u_1 = v_1$, then $\texttt{fire}(\sigma, u_1) = \texttt{fire}(\sigma, v_1)$.
  - Let $\sigma' = \texttt{fire}(\sigma, u_1)$, then $\sigma^{(u)}$ is obtained by firing $a - 1$ vertices $u_2, u_3, \cdots, u_a$ in order, and $\sigma^{(v)}$ is obtained by firing $b - 1$ vertices $v_2, v_3, \cdots, v_b$.
  - By the induction hypothesis, the lemma is correct for $\max(a - 1, b - 1) \leq k - 1$, which means $\sigma^{(u)} = \sigma^{(v)}$, and each vertex will be fired the same number of times in $u_2, u_3, \cdots, u_a$ and $v_2, v_3, \cdots, v_b$.
  - Since $u_1 = v_1$, the vertex u_1 will be fired once more in both of the firing plans, so the number of the firings on u_1 remains equal.
- Otherwise, there must exists an $2 \leq i \leq a$ such that $u_i = v_1$.
  - This is because we have $\sigma_{v_1} \geq \texttt{degree}(v_1)$ in the beginning. Since the only way to decrease the number of the chips on a vertex is to perform a fire operation, we must perform at least one firing operation on v_1 to obtain a terminal configuration. It implies that there exists at least one $1 \leq i \leq a$ such that $u_i = v_1$.

– Let's take the smallest i such that $u_i = v_1$. The condition that $u_j \neq u_i$ for all $1 \leq j < i$ must be held. By Lemma 2.8.2, $\sigma^{(u)}$ equals to the configuration obtained by firing the vertices $u_i, u_1, u_2, \cdots, u_{i-1}, u_{i+1}, u_{i+2}, \cdots, u_a$ in order.
– Since $u_i = v_1$, by applying the proof of the case $u_1 = v_1$, we have $\sigma^{(u)} = \sigma^{(v)}$ and each vertex u were fired the same number of times in both plans.

In general, if the lemma is correct for $\max(a, b) \leq k - 1$, then it is also correct for $\max(a, b) = k$. By using mathematical induction, the lemma is true for all values of $k \in \mathbb{N}_{\geq 0}$. □

Proof of Lemma 2.4.2. Suppose σ' is obtained by firing vertex $v_1, v_2, \cdots, v_t \in \texttt{subtree}(u)$. Let $\sigma^{(0)} = \sigma$ and $\sigma^{(i)} = \texttt{fire}(\sigma^{(i-1)}, v_i)$ for all $1 \leq i \leq n$. Then $\sigma' = \sigma^{(t)}$.

By Definition 2.4.2, $\texttt{final}(\sigma^{(i)} + \sigma^*, u) = \texttt{final}(\texttt{fire}(\sigma^{(i-1)}, v_i) + \sigma^*, u)$. Since we can fire v_i in the configuration $\sigma^{(i-1)}$, we must have $\sigma_{v_i}^{(i-1)} \geq \texttt{degree}(v_i)$. By Corollary 2.8.1, $\texttt{fire}(\sigma^{(i-1)}, v_i) + \sigma^* = \texttt{fire}(\sigma^{(i-1)} + \sigma^*, v_i)$.

Since $v_i \in \texttt{subtree}(u)$, firing any vertex in $\texttt{subtree}(u)$ will not affect $\texttt{final}(\sigma, u)$. So

This gives us $\texttt{final}(\sigma', u) = \texttt{final}(\sigma_t, u) = \texttt{final}(\sigma_0, u) = \texttt{final}(\sigma, u)$. $\texttt{final}(\texttt{fire}(\sigma^{(i-1)} + \sigma^*, v_i), u) = \texttt{fire}(\sigma^{(i-1)} + \sigma^*, u)$. This gives us $\texttt{final}(\sigma^{(i)} + \sigma^*, u) = \texttt{final}(\sigma^{(i-1)} + \sigma^*, u)$ for all $1 \leq i \leq t$. It implies $\texttt{final}(\sigma' + \sigma^*, u) = \texttt{final}(\sigma^{(t)} + \sigma^*, u) = \texttt{final}(\sigma^{(0)} + \sigma^*, u) = \texttt{final}(\sigma + \sigma^*, u)$. □

Proof of Lemma 2.4.3. By Definition 2.4.2 $\texttt{final}(\sigma, u)$ is obtained by firing several vertex $v \in \texttt{subtree}(u)$. By applying Lemma 2.4.2 $\texttt{final}(\texttt{final}(\sigma, u) + \texttt{final}(\sigma', u), u) = \texttt{final}(\sigma + \texttt{final}(\sigma', u), u)$. Applying the lemma to $\texttt{final}(\sigma', u)$ again we will get $\texttt{final}(\sigma + \sigma(\sigma', u), u) = \texttt{final}(\sigma + \sigma', u)$. □

Proof of Lemma 2.4.5. We can prove the lemma by induction. The lemma is trivial for $k = 1$, as $\psi_u(0) = \sigma_u \geq \texttt{degree}(u)$, we can fire vertex u directly.

For all $k \geq 2$, and $\psi_u(k - 1) \geq \texttt{degree}(u)$. By Lemma 2.4.8, $\psi_u(k - 2) \geq \psi_u(k - 1) \geq \texttt{degree}(u)$, thus we can fire vertex u at least $k - 1$ times by inductive hypothesis.

Assume we have fired vertex u exactly $k - 1$ times, and we fired all full vertices in $\texttt{subtree}(v_i)$ for all $v_i \in \texttt{children}(u)$. By the inductive hypothesis, there are $\psi_u(k - 1)$ chips on vertex u. Since $\psi_u(k - 1) \geq \texttt{degree}(u)$, we can perform one firing operation on vertex u, and the number of chips on vertex u will become $\psi_u(k - 1) - \texttt{degree}(u)$.

Consider all $v_i \in \texttt{children}(u)$. Before the k-th firing on vertex u, it receives $k-1$ chips from u, and gives u back $\delta(v_i, k-1)$ chips. After the k-th operation on u, it will receive one more chip. For the initial configuration σ, adding k more chips on vertex v_i will let vertex u receive $\delta(v_i, k)$ chips. So there will be $(\delta(v_i, k) - \delta(v_i, k-1))$ more chips received from vertex v_i after making σ local terminal in $\texttt{subtree}(v_i)$. Thus the number of chips on vertex u will become $\psi_u(k-1) - \texttt{degree}(u) + \sum_{v \in \texttt{children}(u)} (\delta(v, k) - \delta(v, k-1))$, which is exactly $\psi_u(k)$. □

Proof of Lemma 2.4.6. We prove the lemma by induction. For all the leaf vertices u, $\delta(u, k) = k$ must be held. So the lemma is correct for all the leaf vertices.

Consider any vertex $u \in V(G)$, and for all vertices $v \in \texttt{children}(u)$ the inequality $\delta(v, k) \leq \delta(v, k+1) \leq \delta(v, k) + 1$ holds for all non-negative integers k by the inductive hypothesis. Consider $\sigma' = \texttt{final}(\sigma + k_u, u)$, there are two cases.

1. $\sigma'_u < \texttt{degree}(u) - 1$. Then putting one more chip on the vertex u does not make more firing operations available, since $\sigma'_u + 1 < \texttt{degree}(u)$. So $\delta(u, k+1) = \delta(u, k)$ in this case.

2. $\sigma'_u = \texttt{degree}(u) - 1$. Then we perform one firing operation on vertex u. All the children $v \in \texttt{children}(u)$ will receive one more chip after the operation, but since $\delta(v, k+1) \leq \delta(v, k) + 1$ holds for all k on vertex v, there will be at most one more chip received from vertex v after making σ being local terminal in $\texttt{subtree}(v)$ again. So there will be no more than $|\texttt{children}(u)|$ chips after doing all firing operations in $\texttt{subtree}(u)$. Since $|\texttt{children}(u)| < \texttt{degree}(u)$ for all $u \neq r$, no more fire operations on vertex u are possible. So there will be exactly one additional firing operation performed on vertex u, thus $\delta(u, k+1) = \delta(u, k) + 1$ in this case.

This shows $\delta(u, k+1) \in \{\delta(u, k), \delta(u, k)+1\}$. Thus $\delta(u, k) \leq \delta(u, k+1) \leq \delta(u, k)+1$. □

Proof of Lemma 2.4.8. By definition, $\psi_u(k) = \sigma_u - k \cdot \texttt{degree}(u) + \sum_{v \in \texttt{children}(u)} \delta(v, k)$, we have $\psi_u(k+1) - \psi_u(k) = -\texttt{degree}(u) + \sum_{v \in \texttt{children}(u)} (\delta(v, k+1) - \delta(v, k))$. By Lemma 2.4.6, $\delta(v, k+1) - \delta(v, k) \leq 1$, and this implies $\sum_{v \in \texttt{children}(u)} (\delta(v, k+1) - \delta(v, k)) \leq |\texttt{children}(u)| \leq \texttt{degree}(u)$. This proves $\psi_u(k+1) - \psi_u(k) \leq 0$, thus $\psi_u(k) \geq \psi_u(k+1)$. □

Proof of Lemma 2.4.7. By the definition of k we have either $k = 0$ or $\psi_u(k-1) \geq \texttt{degree}(u)$.

If $k = 0$, we have $\psi_u(0) = \sigma_u < \texttt{degree}(u)$, which means we can not perform any operation on vertex u. Otherwise, by Lemma 2.4.5, we can perform k firing operations on vertex u, and there are $\psi_u(k)$ chips located on vertex u after all these firings. Since $\psi_u(k) < \texttt{degree}(u)$ and σ became local terminal in all the subtree of v_i for $v_i \in \texttt{children}(u)$, the current configuration must be local terminal in the subtree of u. Thus $\mathbf{c}^{\downarrow}(u) = k$. □

Proof of Lemma 2.4.9. By Definition 2.4.2, a configuration that is local terminal in $\texttt{subtree}(r)$ is equivalent to being a terminal configuration. So $\mathbf{c}(r) = \mathbf{c}^{\downarrow}(r)$.

For every $u \in V(G)$ such that $u \neq r$, consider the following way to find the terminal configuration of σ.

1. For any vertex $v \in \texttt{subtree}(u)$ such that v is a full vertex, perform a firing operation on v. Repeat until there are no such $v \in \texttt{subtree}(u)$ exists.
2. For any vertex $w \in V(G)$ such that w is a full vertex, perform a firing operation on w. Then check if there is any vertex $v \in \texttt{subtree}(u)$ such that v is a full vertex. If so, find such $v \in \texttt{subtree}(u)$ repeatedly and fire the vertex v, until there is no such v exists. Repeat this process until there is no full vertex in σ.

The procedure will find a terminal configuration if it exists, since there will not be any full vertex after the procedure.

In the first stage, it's equivalent to performing a local finalize operation described in Definition 2.4.2. By definition, the vertex u will be fired exactly $\mathbf{c}^{\downarrow}(u)$ times in this stage.

Now we consider the second stage. Since the given graph is a tree, there is only one vertex, which is $\texttt{parent}(u)$ exactly, that serves as a neighbor of vertex u but does not belong to $\texttt{subtree}(u)$. So in the second stage, the only way that vertex u receives an additional chip is by firing vertex $\texttt{parent}(u)$.

Since vertex $\texttt{parent}(u)$ has never been fired in the first stage, all the firing operations on vertex $\texttt{parent}(u)$ will be happening in the second stage. So, the vertex u will receive exactly $\mathbf{c}(\texttt{parent}(u))$ chips.

Note that if there are $\mathbf{c}(\texttt{parent}(u))$ additional chips placed on vertex u, then $\texttt{parent}(u)$ will receive $\delta(u, \mathbf{c}(\texttt{parent}(u)))$ more chips after the configuration becomes a local terminal in $\texttt{subtree}(u)$. And it is equivalent to vertex u being fired $\delta(u, \mathbf{c}(\texttt{parent}(u)))$ times in this stage.

Adding the two stages together, vertex u is fired a total of $\mathbf{c}(u) = \mathbf{c}^{\downarrow}(u) + \delta(u, \mathbf{c}(parent(u)))$ times.

□

Chapter 3

Motif Counting on Bipartite Graphs

In this chapter, we first consider counting (p, q)-bicliques in bipartite graphs, which is crucial for a variety of applications, from recommendation systems to cohesive subgraph analysis. It remains computationally challenging due to the combinatorial explosion to exactly count the (p, q)-bicliques. In many scenarios, e.g., graph kernel methods, however, exact counts are not strictly required. Therefore, we aim to develop approximate counting algorithms to provide a scalable solution for large-scale (p, q)-biclique counting with acceptable runtime and accuracy.

Then, we involve the temporal dimension on bipartite graphs and focus on counting the butterfly motif, a complete 2×2 biclique, which represents the simplest yet significant subgraph structure and is crucial for analyzing complex network patterns. The temporal dimension of bipartite graphs, where edges activate within specific time frames, introduces the concept of historical butterfly counting, i.e., counting butterflies within a given time interval. This temporal analysis sheds light on the dynamics and evolution of network interactions, offering new insights into their mechanisms. Despite its importance, no existing algorithm can efficiently solve the historical butterfly counting task. To address this, we study on designing index algorithms for this newly introduced historical butterfly counting problem.

3.1 Approximate Biclique Counting

In this section, we tackle the (p, q)-biclique counting problem in large-scale bipartite graphs, which is crucial for applications like recommendation systems and cohesive subgraph analysis. To address scalability and accuracy issues in existing methods, we proposed a novel

sampling-based algorithm leveraging (p, q)-*brooms*, special spanning trees within (p, q)-bicliques. Utilizing graph coloring and dynamic programming, our method efficiently approximates (p, q)-biclique counts with unbiased estimates and provable error guarantees. Experimental results on nine real-world datasets show that our approach outperforms state-of-the-art methods, achieving up to $8\times$ error reduction and $50\times$ speed-up.

The results of this section come from the joint work [55] with Weinuo Li, Yingli Zhou, Hangrui Zhou, Qiuyang Mang, Can Wang, Yixiang Fang, Chenhao Ma. The primary authors are Jingbang, Weinuo and Yingli. The algorithm was initially proposed by the primary authors. Jingbang was in charge of the whole project and wrote the entire paper except for the experiment parts, which were mainly conducted by Weinuo and written by Yingli. Jingbang and Yingli designed the experiments.

3.1.1 Introduction

The bipartite graph stands as a cornerstone in graph mining, comprising two distinct sets of vertices where edges exclusively link vertices from different sets. This concept serves as a powerful tool for modeling relationships across various real-world domains, including recommendation networks [54], collaboration networks [31], and gene coexpression networks [121]. For example, in recommendation networks, users and items typically constitute two distinct vertex types. The interactions between these vertices form a bipartite graph, with edges representing users' historical purchase behaviors.

In the realm of bipartite graph analysis, counting (p, q)-bicliques has attracted much attention. A (p, q)-biclique is a complete subgraph between two distinct vertex sets of size p and q. The $(2, 2)$-bicliques, also known as butterflies, have played an important role in the analysis of bipartite networks [152, 205, 207]. Furthermore, there are more scenarios in which p, q is not fixed to 2. In general, counting (p, q)-bicliques serves as a fundamental operator in many applications, including cohesive subgraph analysis [38] and information aggregation in graph neural networks [223], and densest subgraph mining [157]. In higher-order bipartite graph analysis, the clustering coefficient is the ratio between the counts of (p, q)-bicliques and (p, q)-wedges, where counting (p, q)-wedges can be reduced to counting (p, q)-bicliques [226].

Despite its importance, counting (p, q)-bicliques is very challenging due to its exponential increase with respect to p and q [223]. For example, in one of the graph datasets `Twitter`, with fewer than 2×10^6 edges, there are more than 10^{13} $(5, 4)$-bicliques and more than 10^{18} $(6, 3)$-bicliques within. Therefore, enumeration-based counting methods including `BCList++` [223], `EPivoter` [226] that produce the exact solution are not scalable. On

the other hand, in many applications of biclique counting, an approximate count is often sufficient. In **graph kernel methods** [191], motifs serve as the basis for defining similarity measures between graphs in tasks like classification and anomaly detection. Since graph kernels depend on relative similarities rather than exact counts, approximate counts can preserve kernel performance while significantly reducing the computational overhead. This efficiency facilitates similarity computations in fields such as bioinformatics (e.g., comparing protein interaction networks) and cybersecurity (e.g., detecting similar attack patterns across networks). In **recommendation systems** [205], biclique counting helps uncover dense substructures among users or items, such as groups of users with shared interests or items commonly co-purchased. In large-scale environments such as e-commerce or streaming platforms, allowing a small error margin for counting can significantly reduce the runtime cost while still preserving the effectiveness of the downstream tasks.

From this perspective, we can see that developing algorithms that produce approximate answers to (p, q)-biclique counting is more practical and more applicable to large-scale data. Ye et al. propose a sampling-based method called `EP/Zz++` for approximate counting (p, q)-bicliques [226]. However, its precision is not satisfactory as p, q increases. For example, when querying $(5, 9)$-bicliques in the data set `Twitter`, `EP/Zz++` could produce an answer of the error ratio of more than 200%.[1] This shows the need to develop a better algorithm for approximate (p, q)-biclique counting, aiming to improve scalability and accuracy.

As results, we propose a new sampling-based method that produces a high-accuracy approximate counting of (p, q)-bicliques. We first adapt the coloring trick in this problem, inspired by several counting methods of k-cliques in general graphs [136, 225]. Then, we design a special type of subgraph that has a strong correlation with (p, q)-biclique, named (p, q)-brooms. The (p, q)-broom is a special type of spanning tree in a (p, q)-biclique, which has a fixed structure that can be taken advantage of when counting the amount. Using dynamic programming, we can count this motif efficiently and precisely. It also naturally adapts to the coloring. Finally, we develop a sampling method that takes advantage of the coloring and the intermediate result of dynamic programming. Compared to the h-zigzag pattern proposed by `Ep/Zz++`, our (p, q)-brooms have a structure that is closer to (p, q)-cliques, and it usually has a smaller amount in the graph. As a result, it provides a better error guarantee when sampling (p, q)-bicliques from them.

We empirically evaluate our algorithm against state-of-the-art exact and approximate counting algorithms on nine real-world datasets. Our contributions are summarized as follows:

[1] Let C and $\hat{C}$ be the exact and approximate counts of the (p, q)-biclique, respectively. The estimation error ratio is defined as $\frac{|\hat{C}-C|}{C}$.

- We design (p, q)-broom, a special type of spanning tree in a (p, q)-biclique, which can be efficiently counted via dynamic programming.
- Based on the dynamic programming result of counting (p, q)-brooms, we develop an efficient sampling-based algorithm that computes a high-accuracy counting of (p, q)-bicliques in bipartite graphs.
- Mathematical analysis shows that our algorithm is unbiased and obtains a provable error guarantee.
- Extensive experiments show that our algorithm consistently outperforms state-of-the-art approximate algorithms, with up to 8× reduction in approximation errors and up to 50× speed-up in running time.

3.1.1.1 Related Work

We first review the existing works on biclique and k-clique counting problems and then briefly review other motif counting methods.

Biclique Counting. The biclique counting problem focuses on enumerating (p, q)-bicliques within bipartite graphs, with particular emphasis on the $(2, 2)$-biclique, commonly known as a butterfly—a fundamental structural pattern with significant real-world applications. Given the importance of butterfly motifs, researchers have developed numerous algorithms for their efficient enumeration and counting [207, 185, 205]. The state-of-the-art method utilizes the vertex priority and cache optimization [207]. Recent advances have emerged in multiple directions: parallel computing techniques that optimize both memory and time efficiency [218], I/O-efficient methods [212] that minimize disk operations, and approximation strategies that provide accurate counts while reducing computational overhead. The field has further expanded to address more complex graph types, including uncertain graphs with probabilistic edges [231] and temporal graphs incorporating time-varying relationships [45]. The (p, q)-biclique counting is a general version of butterfly counting, which recently gained much attention [226, 223]. The state-of-the-art methods are extensively introduced and analyzed in the latter (See Section 3.1).

k-clique Counting. The evolution of k-clique counting algorithms began with Chiba and Nishizeki's backtracking enumeration method, which was later enhanced through more efficient ordering-based optimizations, including degeneracy ordering (Danisch et al. [81]) and color ordering (Li et al. [136]). While these algorithms perform efficiently for small k, their performance deteriorates with increasing k. `Pivoter` [115], introduced by Jain

and Seshadhri, marked a significant advancement by employing pivoting techniques from maximal clique enumeration, enabling combinatorial counting instead of explicit enumeration. However, `Pivoter`'s performance can degrade on large, dense graphs. To address scalability challenges, researchers have developed sampling-based approaches, including `TuranShadow` [116] and coloring-based sampling techniques [225].

Other Motif Counting. Beyond k-clique and (p, q)-biclique, numerous works are focusing on counting general motifs, such as four-cycles and other subgraph patterns [110, 230, 41, 150]. These methods can be broadly divided into two categories: traditional counting approaches [41, 150] and learning-based methods [110, 230]. The former is typically based on the sampling and enumeration techniques used to obtain the approximate and exact count of the motif, respectively. In contrast, learning-based approaches offer an alternative approximate solution by leveraging machine learning techniques. Beyond this primary categorization, subgraph counting can be further classified by scope: global counting determines subgraph frequencies across the entire graph, while local counting focuses on specific nodes or edges.

3.1.1.2 Preliminaries

Throughout this work, we study the bipartite graphs. A bipartite graph is an undirected graph $G = ((U, V), E)$, where U and V are disjoint sets of vertices and each edge $(u, v) \in E$ satisfies $u \in U, v \in V$. To denote the corresponding vertex and edge sets of a certain G, we may use $U(G)$, $V(G)$, and $E(G)$, respectively. For each vertex $u \in U(G)$, we denote its neighbor set as $N(u, G) = \{v \in V \mid (u, v) \in E\}$. Similarly, for a vertex $v \in V(G)$, its neighbor set is $N(v, G) = \{u \in U \mid (u, v) \in E\}$. For any vertex $x \in U \bigcup V$, its degree $d(x)$ is the size of its neighbor set. Now we define the (p, q)-biclique.

Definition 3.1.1 (Biclique)**.** *Given a bipartite graph $G = ((U, V), E)$, a (p, q)-biclique is a subgraph $G' = ((U', V'), E') \in G$ where $|U'| = p$, $|V'| = q$ and $E' = \{(u, v) \mid u \in U', v \in V'\}$ (a complete bipartite graph).*

Lastly, we define the problem that we mainly study:

Problem 2 (Biclique Counting)**.** *Given a bipartite graph G and two integers p and q, compute the number of (p, q)-bicliques in G.*

For example, in Figure 3.1, there are two $(3, 2)$-bicliques: **(1)** $U' = \{1, 3, 4\}, V' = \{1, 2\}$; **(2)** $U' = \{3, 4, 5\}, V' = \{2, 3\}$. Their edges are highlighted with pink lines and blue lines, respectively. Therefore, the number of $(3, 2)$-bicliques is 2. It is also known that

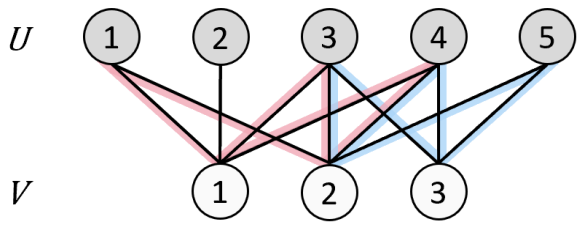


Figure 3.1: An illustrative example of two $(3, 2)$-bicliques.

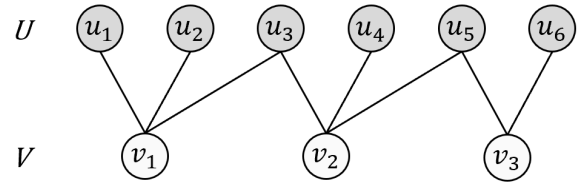


Figure 3.2: An illustrative example of a $(6, 3)$-broom.

Problem 2 is NP-Hard [223], which means there is no algorithm that runs in polynomial time. Throughout this work, we assume $p, q \geq 2$.

3.1.2 Algorithm

In this section, we propose a new sampling-based algorithm called *Colored Broom-based Sampling (CBS)* that solves Problem 2 approximately.

Inspired by methods for solving k-clique counting problems on general graphs [136, 225], we start by assigning a color to every vertex on the graph (Section 3.1.2.1). Our coloring method is simple, and produces assignments with a small number of colors empirically.

Instead of counting bicliques directly, our next step is to calculate the number of a special motif called "broom" that we design (Section 3.1.2.2). Specifically, the (p, q)-broom is defined as follows:

Definition 3.1.2 (Broom)**.** *Given a bipartite graph $G((U, V), E)$, two vertex sets $U' \in U$ and $V' \in V$ of size p and q, respectively. A vertex ordering is also given: $\{u_1, u_2, \ldots, u_p\}$ and $\{v_1, v_2, \ldots, v_q\}$. The corresponding (p, q)-broom is the subgraph $G'(U', V'), E') \in G$ that satisfies $E' = \{(u_i, v_{\lfloor \frac{(i-1)(q-1)}{p-1}+1 \rfloor}) \mid 1 \leq i \leq p\} \cup \{(u_{\lceil \frac{(i-1)(p-1)}{q-1} \rceil}, v_i) \mid 2 \leq i \leq q\}$.*

In Figure 3.2, we provide an illustration of a $(6, 3)$-broom. We can observe that such a subgraph is, in fact, a tree, which contains $p + q - 1$ edges and is connected. As shown in Figure 3.2, we can clearly see that each vertex v_i in V connects to the $2 - 3$ vertices u_j in U that i and j are close. Note that the defined ordering is only used to characterize the pattern, providing a better visualization, and it does not have to align with the original vertex index. Visually speaking, a (p, q)-broom is the "skeleton" of a (p, q)-biclique. The general idea here is to design a way to sparsify the dense biclique while trying best to preserve its identity. This broom pattern is very different from `EP/Zz++` [226]'s h-zigzag patterns, as they only use a simple path. We then empirically show that such pattern's count can lead to a much better approximation of biclique counting.

There are at least two observations for the broom:

1. A spanning tree captures a subgraph better than a simple path.
2. All $p + q - 1$ edges are allocated to each vertex nearly evenly.

This distribution design limits the total number of existing brooms, which makes the error analysis tight. Despite its complicated structures, we show in Section 3.1.2.2 that its amount can be computed precisely and efficiently using a dynamic programming process with respect to color assignment.

Lastly, by sampling, we extract the quantity relationship between the (p, q)-brooms and (p, q)-bicliques in the graph and eventually compute our approximate answer (Section 3.1.2.3). Shown in Section 3.1.3, our algorithm produces high-accuracy approximate counting with faster runtime in real-world datasets. We also prove unbiasedness and provide an error guarantee for our algorithm.

3.1.2.1 Coloring

The color assignment should guarantee that for any (p, q)-clique H, there are no two vertices in $U(H)$ that share the same color. Similarly, the same is true for any two vertices in $V(H)$.

Our coloring algorithm is shown in Algorithm 21. We run the coloring process `CalcColor` for the vertices in U and V separately (Line 1, 2). Note that here, we consider the neighbor symmetrically. After preprocessing the input graph G with Algorithm 21, each vertex $x \in U \bigcup V$ is colored, and we denote its color as $c(x)$.

S denotes the set of vertices that has not been colored. Whenever S is not empty (Line 7), we will try coloring some vertices with a new color, denoted as κ. For brevity,

we initialize κ to be 1 (Line 6), and whenever we need a new color, we increase κ by 1 (Line 16). Cnt is a temporary array that is used to guarantee the coloring is legal. Specifically, when we try coloring with color κ, $Cnt[u]$ stores the number of v that satisfies $v \in N(u, G)$ and there exists $w \in N(v, G)$ such that $c(w) = \kappa$. In each round of coloring, we start by initializing Cnt to be all zeros (Line 8). We use S_{next} to store vertices that have to be colored in the next round, initialized as $\emptyset$ (Line 9). We enumerate vertices in S in a randomized order (Line 10). For each enumerated vertex u, we first set its color to be κ (Line 11) and update Cnt correspondingly (Line 12). If there exists a vertex x such that $Cnt[x] \geq q$ (Line 13), then there could exist a (p, q)-biclique with two vertices with the same color. Therefore, such coloring is illegal, and we cannot color u with κ. In such case, we first revert the changes in Cnt (Line 14) and add u into S_{next} (Line 15). In the end, we have tried every vertex in S at least once and have deferred some vertices' coloring to upcoming rounds. We assign S_{next} to S and continue (Line 17).

It is not hard to see that the runtime of this coloring process is dominated by the total time of the total number of colors, i,e. κ. Note that querying, updating, and reverting cnt can be done in $O(|N(u, G)|)$ each operation. In all, the time complexity is $O(\kappa|E|)$. Shown in Table 3.1, on real-world datasets, κ is not large and our coloring algorithm runs very efficiently in turn. The only question left is how such an algorithm guarantees a correct color assignment. We prove it in the following lemma.

Lemma 3.1.1. *After executing* `Coloring`(G, p, q)*, for any (p, q)-clique H in G, there are no two vertices x, y either both in $U(H)$ or in $V(H)$ that share the same color, i.e. $c(x) = c(y)$.*

Proof. Let $U(H) = \{u_1, u_2, \ldots, u_p\}$, $V(H) = \{v_1, v_2, \ldots, v_q\}$. Because of symmetry, we only prove the case of U and p. We assume that there are two vertices $u_1, u_2 \in U(H)$ with the same color, i.e. $c(u_1) = c(u_2) = k$. WLOG, when we color vertices with the k-th color, we first set $c(u_1) \leftarrow k$. Then we will have $Cnt[u_2] \geq q$ since $\{v_1, v_2, \ldots, v_q\}$ are u_2's neighbors and they all have neighbor u_1 such that $c(u_1) = k$. So u_2 cannot be colored in this round. This contradicts the assumption.

□

3.1.2.2 Pattern Counting

After coloring, we enhance the motif we are counting with a condition of distinct color. However, it is still hard to count bicliques directly. Instead, we count the number of (p, q)-

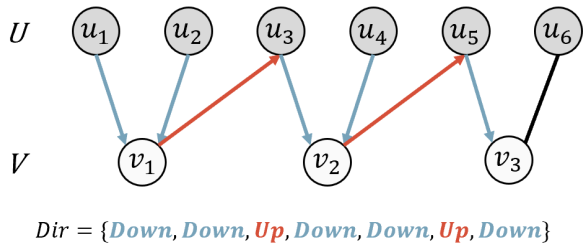


Figure 3.3: An example of edges' directions in a $(6, 3)$-broom, where edges with different directions are represented by `Up` and `Down` in Dir and colored in red and blue, respectively.

brooms. Since we assign a concrete integer for each color in Algorithm 21 for any (p, q)-biclique, to avoid overcounting, we assume its vertex ordering of U' and V' is following the color from small to large. With this exact ordering, by Definition 3.1.2, there is exactly one (p, q)-broom subgraph in each (p, q)-biclique. Note that the color of these (p, q)-brooms is also unique, which is convenient for counting. Algorithm 22 is able to compute the number of (p, q)-brooms of this type. We then show how to use this value to approximate the biclique count.

The whole process is dynamic programming where its state is represented by the total number of edges we have accumulated so far and the last edge we keep track of. To well define this dynamic programming state, we need to specify a few more properties in this type of (p, q)-broom:

A. Any (p, q)-broom H is connected and contains exactly $p + q - 1$ edges. That is to say, it is actually a tree.

B. If we sort all edges $(u, v) \in E(H)$ by the increasing order of the tuple $(c(u), c(v))$, there is only three types of edges:

1. The edge with the maximum tuple.
2. Edges sharing an endpoint in $U(H)$ with the next edge.
3. Edges sharing an endpoint in $V(H)$ with the next edge.

For **(2)** and **(3)**, we signal them with a **"direction"** as either `Up` or `Down`. For example, as shown in the $(6, 3)$-broom in Figure 3.3, (u_1, v_1) is with the smallest partial order. Its direction is `Down` since it shares the same endpoint v_1 with the next edge (u_2, v_1). Similarly, (u_2, v_1), (u_3, v_2), (u_4, v_2), (u_5, v_3) are also with direction `Down`, highlighted with blue arrows.

Algorithm 21: `CBS: Coloring`

Input: G: a bipartite graph; p, q: two parameters.
Output: G': a colored bipartite graph.

```
1  CalcColor(G, U(G), p, q);
2  CalcColor(G, V(G), q, p);
3  G' ← G;
4  return G';
5  Procedure CalcColor(G, S, p, q)
6      Initialize κ ← 1;
7      while |S| > 0 do
8          Initialize an array Cnt with zeros;
9          S_next ← ∅;
           /* enumerate vertices in a random order */
10         for u ∈ S do
11             c(u) ← κ;
12             Update Cnt;
13             if ∃x, Cnt[x] ≥ q then
14                 Revert Cnt's change;
15                 S_next ← S_next ∪ {u};
16         κ ← κ + 1;
17         S ← S_next;
```

(u_3, v_1), (u_5, v_2) are with direction `Up`, highlighted with reversed red arrows. (u_6, v_3) has no direction since it is the edge with the maximum tuple.

After fixing the structure and order of edges by **B**, we can simply use the number of edges and the last edge to represent the current structure of the growing broom. The rest is enumeration and direct transition. The details are as follows.

To begin with, we sort $U(G)$, $V(G)$ by the increasing order of colors $c(u), c(v)$ (Line 1), and sort $E(G)$ by the increasing order of the color tuple $(c(u), c(v))$ (Line 2). We use a 2D table $Dp[len][pre]$ to store the count of each state, indicating the current number of edges len and the last edge pre. We first initialize the whole table to 0 (Line 3). Then for the case where each (u, v) acts as the first edge in the broom, we set $Dp[1][(u, v)]$ to be 1 (Line 4, 4). Since there are $p+q-1$ edges in total besides the first one, the first dimension of the table should be 1 to $p + q - 1$. We iterate this dimension from small to large to accumulate the growing broom from 1..t-th edges to 1..$t + 1$-th edges (Line 5). We use $Dir[t]$ to denote the t-th edge's direction in the (p, q)-broom. When $Dir[t]$ is `Up` (Line 6),

Algorithm 22: `CBS: CountingIndex`

Input: G: a colored bipartite graph; p, q: two parameters.
Output: Dp: a $2D$ array; B: the total number of (p, q)-brooms in G.

```
1  Sort U(G), V(G) by the increasing order of c(u), c(v);
2  Sort E(G) by the increasing order of (c(u), c(v));
3  Initialize Dp with zeros;
4  for (u, v) ∈ E(G) do  Dp[1][(u, v)] ← 1  ;
5  for t ← 1 to p + q − 2 do
6      if Dir[t] = Up then
7          for (u, v) ∈ E(G) do
8              Dp[t + 1][(u, v)] ← Σ_{w∈N(u,G),c(w)<c(v)} Dp[t][(u, w)];
9      else
10         for (u, v) ∈ E(G) do
11             Dp[t + 1][(u, v)] ← Σ_{w∈N(v,G),c(w)<c(u)} Dp[t][(w, v)];
12 B ← Σ_{(u,v)∈E(G)} Dp[p + q − 1][(u, v)];
13 return Dp, B;
```

the $t+1$-th edge and the t-th edge share a common point in U. Therefore, for each edge (u, v) (Line 7), $Dp[t+1][(u, v)]$ should accumulate the number of ways coming from any $Dp[t][(u, w)]$ such that w is in $N(u, G)$ and $c(w) < c(v)$ (Line 8). Similarly, when $Dir[t]$ is `Down` (Line 9), the $t+1$-th edge and the t-th edge share a common point in V. In this case, $Dp[t+1][(u, v)]$ should accumulate the number of ways coming from any $Dp[t][(w, v)]$ such that w is in $N(v, G)$ and $c(w) < c(u)$ (Line 10, 11). After all computations, we return B as the sum of all $Dp[p + q - 1][\cdot]$ and the Dp table as the results (Line 12).

By using the prefix sum technique, for each iteration, we can finish all transition computations in $O(|E|)$. There are $O(p + q)$ rounds. Therefore, the total time complexity is $O((p + q)|E|)$.

3.1.2.3 Approximate Counting via Sampling

After `CountingIndex`(G, p, q), we have accumulated B (p, q)-brooms. To approximate the quantity relation between our computed (p, q)-brooms and (p, q)-bicliques, we can do the following sampling process:

1. Set two counters cnt_{broom} and $cnt_{biclique}$.

2. Uniformly sample a (p, q)-broom P from all B ones and increase cnt_{broom} by 1.

3. If the induced subgraph by $U(P)$ and $V(P)$ is a (p, q)-biclique, increase $cnt_{biclique}$ by 1.

After applying this sampling process sufficiently many times, we can roughly approximate the number of (p, q)-bicliques by $cnt_{biclique}/cnt_{broom} \times B$.

However, such a sampling process is too inefficient. We now propose Algorithm 23 to accelerate this process, utilizing the computed dynamic programming results from `CountingIndex`(G, p, q). Recall that $Dp[t][(u, v)]$ represents the number of ways to build a growing (p, q)-broom til the t-th edge, which is (u, v).

To begin with, we start by sampling the last edge of the (p, q)-broom, denoted as (u_{last}, v_{last}) (Line 1). Here, the sampling should follow the weight distribution of $\{Dp[p + q - 1][\cdot]\}$, indicating how many (p, q)-brooms end with each edge. The following process is similar to reverting the dynamic programming process. Instead of building the (p, q)-broom from the 1-st edge to the $(p+q-1)$-th edge, we do it reversely. We use (u_{cur}, v_{cur}), U', V' to denote the current growing (p, q)-broom. They are initialized as (u_{last}, v_{last}), $\{u_{last}\}$, $\{v_{last}\}$ respectively (Line 2, 3). Specifically, since (u_{last}, v_{last}) is the $(p+q-1)$-th edge, we are now adding the $(p+q-2)$-th edge til the 1-th edge gradually (Line 5). Assume we are processing the i-th edge now, and we know the $(i+1)$-th edge is (u_{cur}, v_{cur}). We initialize an edge set S to store the candidate edge for the i-th edge. Based on the direction signaling, we can know whether it should share the common endpoint in U or V. If $Dir[i]$ is `Up` (Line 7), then we should find all $w \in N(u_{cur}, G)$ such that $c(w) < c(v_{cur})$ and $U' \subseteq N(w, G)$. We assign S to be $\{(u_{cur}, w)\}$ (Line 8). The first condition is for obeying the color order, and the second condition indicates that adding w can still guarantee that the induced subgraph is biclique. Similarly, if $Dir[i]$ is `Down` (Line 9), we find all all $w \in N(v_{cur}, G)$ such that $c(w) < c(u_{cur})$ and $V' \subseteq N(w, G)$. Then we assign S to be $\{(w, v_{cur})\}$ (Line 10).

The idea to accelerate the sampling is, by building the (p, q)-broom, we strictly guarantee that it will correspond to a (p, q)-biclique while keeping track of the probability of sampling out this (p, q)-broom, which can be computed through the dynamic programming table. In the beginning, we initialize ans to be 1 (Line 4). Then, for the i-th edge, the probability contribution of sampling it out from S should be the number of the growing broom ends with $(u, v) \in S$ divided by the total number of possible brooms at this step, which is $\frac{\sum_{(u,v)\in S} Dp[i][(u,v)]}{Dp[i+1][(u_{cur},v_{cur})]}$. We multiply ans by this value (Line 11). If ans becomes 0, then there is no possible (p, q)-biclique from the current sampled (p, q)-broom. We return with 0 in this case (Line 12). Now we are ready to sample the i-th edge from S (Line 13). We do so following the normalized distribution of possible brooms from this step ($\{Dp[i][\cdot]\}$). To

Algorithm 23: `CBS: Sampling`

Input: G: a colored bipartite graph; p, q: two parameters; Dp: a $2D$ array; B: the total number of (p, q)-brooms in G.

Output: ans: estimate number of (p, q)-clique in G.

1 Sample the last edge (u_{last}, v_{last}) in $E(G)$ following the weight distribution of brooms;
2 $(u_{cur}, v_{cur}) \leftarrow (u_{last}, v_{last})$;
3 $U' \leftarrow \{u_{last}\}$, $V' \leftarrow \{v_{last}\}$;
4 $ans \leftarrow 1$;
5 **for** $i \leftarrow p + q - 2$ **to** 1 **do**
6 Initialize set $S \leftarrow \emptyset$;
7 **if** $Dir[i] =$ `Up` **then**
8 $S \leftarrow \{(u_{cur}, w) \mid w \in N(u_{cur}, G), c(w) < c(v_{cur}), U' \subseteq N(w, G)\}$;
9 **else**
10 $S \leftarrow \{(w, v_{cur}) \mid w \in N(v_{cur}, G), c(w) < c(u_{cur}), V' \subseteq N(w, G)\}$;
11 $ans \leftarrow ans \times \frac{\sum_{(u,v)\in S} Dp[i][(u,v)]}{Dp[i+1][(u_{cur}, v_{cur})]}$;
12 **if** $ans = 0$ **then break** ;
13 Sample the next edge (u_{next}, v_{next}) in S following the weight distribution of growing brooms;
14 $(u_{cur}, v_{cur}) \leftarrow (u_{next}, v_{next})$;
15 $U' \leftarrow U' \cup \{u_{next}\}, V' \leftarrow V' \cup \{v_{next}\}$;
16 **return** $ans \times B$;

continue the process, we assign (u_{cur}, v_{cur}), U', V' to (u_{next}, v_{next}), $U' \cup \{u_{next}\}$, $V' \cup \{v_{next}\}$, respectively (Line 14, 15).

In the end, we successfully sample a (p, q)-broom that corresponds to one (p, q)-biclique and compute the probability of sampling it out. We return this probability by multiplying B as the approximate count of (p, q)-bicliques (Line 16).

It is not hard to see that the time complexity is dominated by the cost of updating S, which requires enumerating all neighbors. The loop only lasts $O(p + q)$. Therefore, the time complexity is $O((p + q) \times \Delta)$, where Δ denotes the maximal degree in G.

Overall Algorithm. We provide Algorithm 24 as a black box to call all three subroutines properly and output the estimated answer of the biclique count. We start by coloring (Line 1) and pattern counting (Line 2). Then by the input sampling size parameter T, we repeatedly call `Sampling`(G, p, q, Dp, B) and aggregate the return to $\hat{C}$. The output approximate (p, q)-biclique count is the average of all T attempts (Line 7). Note that as a common trick, we will execute core-reduction for the original graph [223]: For any query

Algorithm 24: `CBS: Main`

Input: G: a bipartite graph; p, q: two parameters; T: sampling times.
Output: $\hat{C}$: estimate number of (p, q)-clique.

```
G ← Coloring(G, p, q);
Dp, B ← CountingIndex(G, p, q);
Ĉ ← 0;
for i ← 1 to T do
    Ĉ ← Ĉ + Sampling(G, p, q, Dp, B);
Ĉ ← Ĉ/T;
return Ĉ;
```

(p, q), we split the graph and reduce the query pair to $(p-1, q)$.

Unbiasedness. We now show that the `CBS` method is unbiased. Let (u_{cur}, v_{cur}), U', and V' represent the current growing (p, q)-broom. For clarity, we introduce the following definitions:

- $\mathcal{F}_{U',V'}$: The total value to be multiplied into *ans*, defined as the product of the last $i = (p+q) - (|U'| + |V'|)$ fractions.
- $\mathcal{B}_{U',V'}$: The number of (p, q)-brooms H such that U', V' are the maximal tuples of $U(H)$ and $V(H)$.
- $\mathcal{C}_{U',V'}$: The number of (p, q)-bicliques H' such that U', V' are the maximal tuples of $U(H')$ and $V(H')$.

After that, we prove the following lemma by mathematical induction.

Lemma 3.1.2. *For any growing (p, q)-broom U', V', $\mathbb{E}[\mathcal{F}_{U',V'}] = \mathcal{C}_{U',V'}/\mathcal{B}_{U',V'}$.*

Proof. **(a)** When $|U'| + |V'| = p + q$, we have $\mathcal{B}_{U',V'} = 1$. In this case, $\mathcal{F}_{U',V'} = \mathcal{C}_{U',V'} = 1$ if U', V' forms a (p, q)-biclique; otherwise $\mathcal{F}_{U',V'} = \mathcal{C}_{U',V'} = 0$. Hence, $\mathbb{E}[\mathcal{F}_{U',V'}] = \mathcal{C}_{U',V'}/\mathcal{B}_{U',V'}$ trivially holds in this case. **(b)** Suppose that the lemma holds for all growing (p, q)-brooms with $|U'| + |V'| = k + 1$. For any broom with $|U'| + |V'| = k$, let $P = \frac{\sum_{(u,v)\in S} Dp[i][(u,v)]}{Dp[i+1][(u_{cur},v_{cur})]}$. We then have

$$\mathbb{E}[\mathcal{F}_{U',V'}] = P \sum_{(u,v)\in S} \frac{Dp[i][(u,v)]}{\sum_{(u,v)\in S} Dp[i][(u,v)]} \mathbb{E}\left[\mathcal{F}_{U'\cup\{u\},V'\cup\{v\}}\right].$$

Since $u_{cur} = u$ or $v_{cur} = v$ always holds, we have that $|U' \cup \{u\}| + |V' \cup \{v\}| = |U| + |V| + 1 = k + 1$. According to the definition of Dp and $\mathcal{B}$, We can derive that $Dp[i][(u, v)] = \mathcal{B}_{U' \cup \{u\}, V' \cup \{v\}}$, and thus

$$\begin{aligned}\mathbb{E}\left[\mathcal{F}_{U',V'}\right] &= \frac{\sum\limits_{(u,v)\in S} Dp[i][(u,v)] \frac{\mathcal{C}_{U' \cup \{u\}, V' \cup \{v\}}}{\mathcal{B}_{U' \cup \{u\}, V' \cup \{v\}}}}{Dp[i+1][(u_{cur}, v_{cur})]} \\ &= \frac{\sum\limits_{(u,v)\in S} \mathcal{C}_{U' \cup \{u\}, V' \cup \{v\}}}{\mathcal{B}_{U',V'}} \\ &= \frac{\mathcal{C}_{U',V'}}{\mathcal{B}_{U',V'}},\end{aligned}$$

where the last equation is because $\mathcal{C}_{U' \cup \{u\}, V' \cup \{v\}} = 0$ for all pairs of $(u, v) \notin S$. □

Then we can derive the following theorem:

Theorem 3.1.1. *Let ans_i be the value of ans in the i-th sampling. Let $\hat{C} = \frac{1}{T} \sum_{i=1}^{T} ans_i \times B$. Then $\hat{C}$ is an unbiased estimator of the number of (p, q)-bicliques in G.*

Proof. We first have the following induction by Lemma 3.1.2:

$$\begin{aligned}&\mathbb{E}\left[ans_i \times B\right] \\ &= B \times \sum_{(u,v)\in E(G)} \frac{Dp[p+q-1][(u,v)]}{\sum Dp[p+q-1][(u,v)]} \mathbb{E}\left[\mathcal{F}_{\{u\},\{v\}}\right] \\ &= \sum_{(u,v)\in E(G)} Dp[p+q-1][(u,v)] \times \mathbb{E}\left[\mathcal{F}_{\{u\},\{v\}}\right] \\ &= \sum_{(u,v)\in E(G)} \mathcal{C}_{\{u\},\{v\}} = \mathcal{C}_{\emptyset,\emptyset}.\end{aligned}$$

Then we can derive that: $\mathbb{E}\left[\hat{C}\right] = \frac{1}{T} \sum_{i=1}^{T} \mathbb{E}\left[ans_i \times B\right] = \mathcal{C}_{\emptyset,\emptyset}$. Note that $\mathcal{C}_{\emptyset,\emptyset}$ is the number of (p, q)-bicliques in G according to the definition. Therefore, $\hat{C}$ is an unbiased estimator of the number of (p, q)-bicliques in G. □

Based on Theorem 3.1.1, we have obtained an unbiased estimator $\hat{C}$ of the number of (p, q)-bicliques in Algorithm 24.

Error Analysis. we now analyze the estimation error of our sampling algorithms. Our analysis relies on the classic Hoeffding's inequalities, which are shown below.

Lemma 3.1.3 (Hoeffding's inequality, [59, 107]). *For the random variables $X_i \in [0, M]$,$1 \leq i \leq n$, we let $X = \sum_{i=1}^{n} X_i$. Then for $\epsilon > 0$, we have*

$$\Pr(X \geq (1+\epsilon)\mathbb{E}[X]) \leq \exp(-\frac{2\epsilon^2 \mathbb{E}[X]^2}{nZ^2}),$$
$$\Pr(X \leq (1-\epsilon)\mathbb{E}[X]) \leq \exp(-\frac{2\epsilon^2 \mathbb{E}[X]^2}{nZ^2}).$$

Based on Lemma 3.1.3, we can derive the estimation error of our algorithms as shown in the following theorem.

Theorem 3.1.2. *Let C, B be the number of (p,q)-bicliques, (p,q)-brooms, respectively. Let $\hat{C} = \frac{1}{T}\sum_{i=1}^{T} ans_i \times B$ denote the estimated number of (p,q)-bicliques, where ans_i is the return result in the i-th sampling. Then, $\hat{C}$ is a $(1+\epsilon)$ approximation of C with probability $(1-\alpha)$ if $T \geq \frac{B^2}{2\epsilon^2 C^2}\ln(\frac{2}{\alpha})$.*

Proof. We can derive $ans_i \times B \in [0, B]$ since the values multiplied to ans are all probabilities in $[0, 1]$. And we have $\mathbb{E}\left[\hat{C}T\right] = CT$ based on Theorem 3.1.1. Given a positive value ϵ, applying Lemma 3.1.3 by plugging in $n = T$, $X_i = ans_i \times B$, $X = \sum_{i=1}^{n} X_i = \hat{C}T$, we then have

$$\Pr(X \geq (1+\epsilon)\mathbb{E}[X]) = \Pr(\hat{C}T \geq (1+\epsilon)CT) \leq \exp(-\frac{2\epsilon^2 (CT)^2}{TB^2}),$$
$$\Pr(X \leq (1-\epsilon)\mathbb{E}[X]) = \Pr(\hat{C}T \leq (1-\epsilon)CT) \leq \exp(-\frac{2\epsilon^2 (CT)^2}{TB^2}).$$

Further, we have

$$\Pr(\frac{|\hat{C} - C|}{C} \geq \epsilon) \leq 2\exp(-\frac{2\epsilon^2 C^2 T}{B^2})$$

Let $2\exp(-\frac{2\epsilon^2 C^2 T}{B^2}) \leq \alpha$, we can derive that $T \geq \frac{B^2}{2\epsilon^2 C^2}\ln(\frac{2}{\alpha})$. □

From Theorem 3.1.2, the sample size T is mainly determined by $(\frac{B}{C})^2$. That is to say, we need a larger sample size T to ensure high accuracy with a larger $(\frac{B}{C})^2$, where B is the number of (p,q)-brooms, C is the number of (p,q)-bicliques. As shown in Section 3.1.3.3, $(\frac{B}{C})^2$ is usually small. Therefore, our algorithm generally does not need a large T to achieve good accuracy in real-world datasets.

Table 3.1: Datasets used in experiments.

Graphs (Abbr.)	Category	$\|U\|$	$\|V\|$	$\|E\|$	$\max \kappa$
`github (GH)`	Authorship	56,519	120,867	440,237	508
`StackOF (SO)`	Rating	545,195	96,678	1,301,942	290
`Twitter (Wut)`	Interaction	175,214	530,418	1,890,661	2933
`IMDB (IMDB)`	Affiliation	685,568	186,414	2,715,604	158
`Actor2 (Actor2)`	Affiliation	303,617	896,302	3,782,463	189
`Amazon (AR)`	Rating	2,146,057	1,230,915	5,743,258	155
`DBLP (DBLP)`	Authorship	1,953,085	5,624,219	12,282,059	126
`Epinions (ER)`	Rating	120,492	755,760	13,668,320	13200
`Wikipedia-edits-de (DE)`	Authorship	1,025,084	5,910,432	129,885,939	118356

3.1.3 Experiments

3.1.3.1 Experimental Setting

Datasets. We use nine real datasets from different domains, which are available at SNAP [176], Laboratory of Web Algorithmics [166], and Konect [126]. Table 3.1 shows the statistics of these graphs.

Baselines. We compare our method with the baselines from highly related works. We summarize the core ideas of each baseline method as follows.

- `BCList++` [223]: the biclique listing-based algorithm, which is based on the Bron-Kerbosch algorithm [43]. The key idea of `BCList++` is to iteratively enumerate all (p, q)-bicliques containing each vertex through a node expansion process. Specifically, it employs an ordering-based search paradigm, where for each vertex, only its higher-order neighbors are considered during enumeration. Additionally, a graph reduction technique is applied to reduce the search space before biclique counting.

- `EPivoter` [226]: the state-of-the-art algorithm for exact biclique counting, which relies on the edge-based pivot technique. In `EPivoter`, an edge-based search framework is introduced. Unlike `BCList++`, it iteratively selects edges from the candidate set (i.e., the edge set used for biclique expansion) to expand the current biclique. Specifically, during the enumeration process, in each branch, it first select one edge as the pivot edge, and based on this edge, vertices can be grouped into four disjoint groups. By storing the entire enumeration tree along with the four vertex sets, biclique counting can then be efficiently performed using combination counting.

- `EP/Zz++` [226]: the state-of-the-art algorithm for approximate biclique counting. It first partitions the graph into two regions: a dense region (a subgraph containing only high-degree vertices) and a sparse region (a subgraph containing only low-degree vertices). For the sparse region, `EP/Zz++` utilizes `EPivoter` for exact counting, while for the dense region, it proposes a zigzag path-based sampling algorithm for approximate counting. Specifically, it leverages the fact that a (p, q)-biclique must contain a fixed number of $(min\{p, q\})$-zigzag paths. Based on this property, the algorithm first counts the number of h-zigzag paths and then samples T such paths, where $h = min\{p, q\}$. Finally, it estimates the number of h-bicliques based on their proportion with the sampled paths.
- `CBS`: our proposed approximation algorithm, which is introduced in Section 3.1.2.

Notice that we compare `EP/Zz++` instead of `EP/Zz` [226], since the former one is a better version of the latter in terms of efficiency and accuracy. We implement all the algorithms in C++ and run experiments on a machine having an Intel(R) Xeon(R) Platinum 8358 CPU @ 2.60GHz and 512GB of memory, with Ubuntu installed.

Parameter Settings. In our experiments, we following the existing work [226] setting T=10^5 as default values. We define the estimation error as $\frac{|\hat{C}-C|}{C}$, where C denotes the exact count of bicliques, and $\hat{C}$ represents its approximated value. To ensure the reliability of our results, we run each approximation algorithm 10 times, with the reported error representing the average across these executions. For some datasets, the exact count of bicliques cannot be computed (e.g., ER and DE), we evaluate the estimation error differently by using the average approximated biclique count from 10 runs as the exact count (i.e., C).

3.1.3.2 Overall Comparison Results

In this section, we compare `CBS` with three competitors (introduced in Section 3.1.3.1), w.r.t. overall efficiency, accuracy, and the effect of p and q, and the number of samples to demonstrate the superior of our algorithm.

1. Efficiency of All Algorithms. Figure 3.4 depicts the average running time of all the biclique counting algorithms on nine datasets for counting all $3 \leq p, q \leq 9$ bicliques. We make the following observations and analysis: (1) Our method `CBS` is up to two orders of magnitude faster than all competitors, this is mainly because our algorithm has a better theoretical guarantee. (2) on almost all datasets, `CBS` is at least 10× faster than all methods, except DBLP dataset. On this dataset, `BCList++` achieves the best performance, as graph

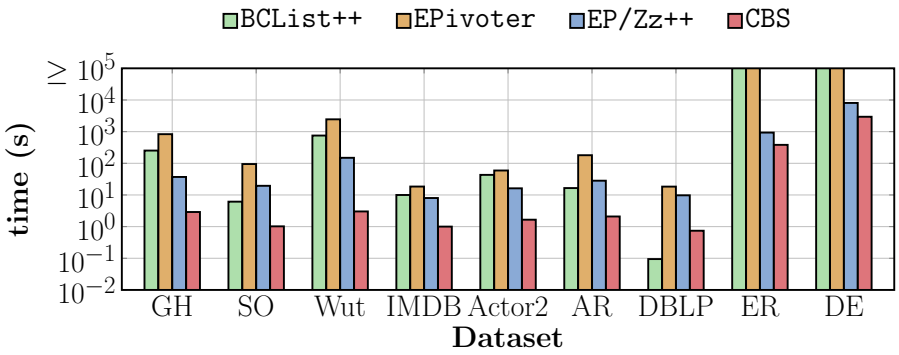


Figure 3.4: Average runtime of different biclique counting algorithms for all $3 \leq p, q \leq 9$.

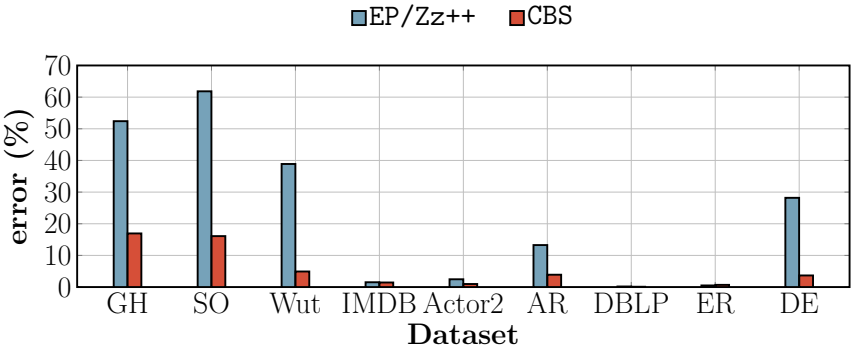
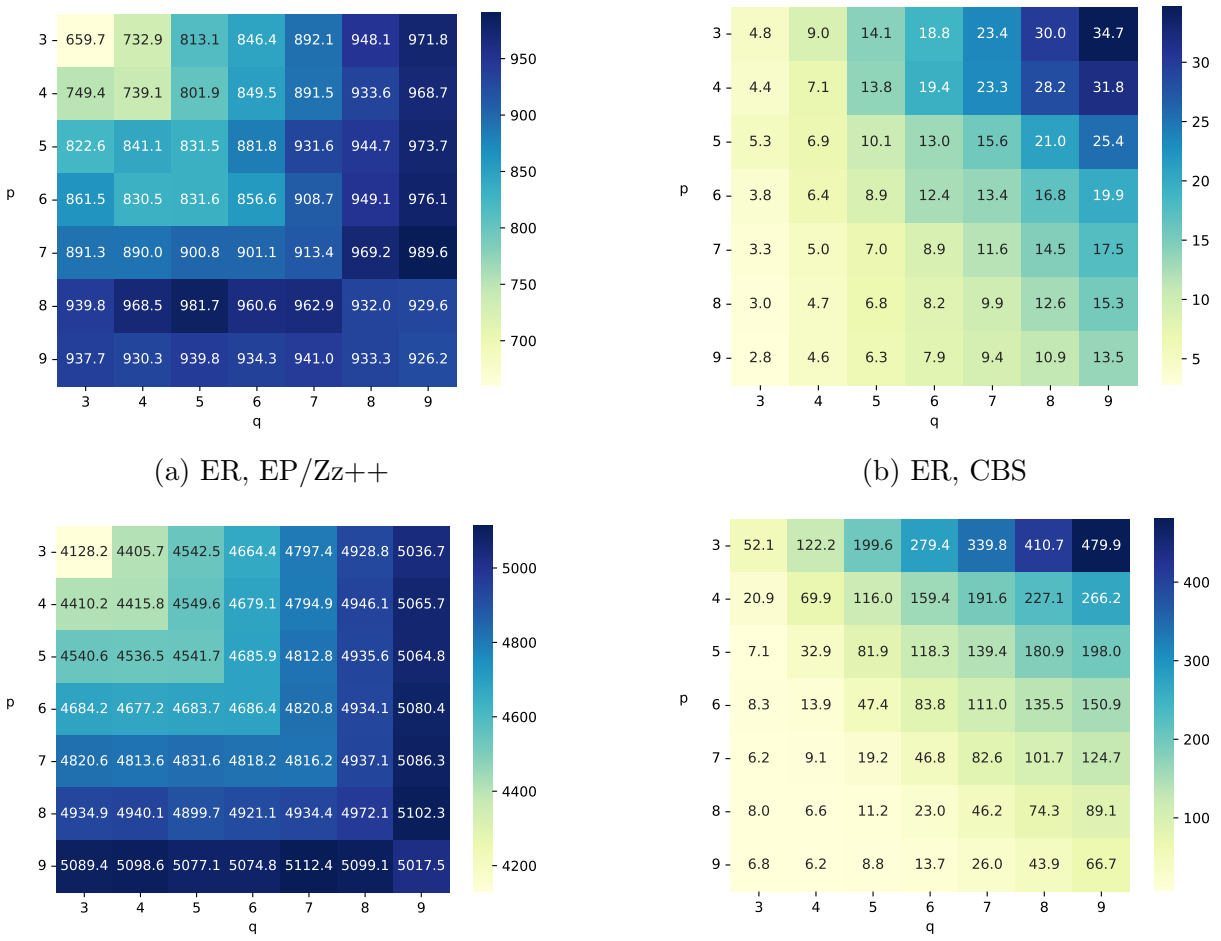


Figure 3.5: Average error of EP/Zz++ and CBS for all $3 \leq p, q \leq 9$.

reduction significantly reduces $|U|$ and $|V|$, allowing it to perform efficiently without extra initialization steps. Meanwhile, `CBS` still outperforms `EPivoter` and `EP/Zz++`. (3) On the two largest datasets, ER and DE, the exact algorithms fail to count the bicliques within 10^5 seconds, while both approximate algorithms successfully complete the task. Our method, `CBS`, demonstrates at least 3 times faster performance compared to `EP/Zz++`.

2. Accuracy of All Algorithms. Figure 3.5 illustrates the average error rates of `CBS` and `EP/Zz++` across all datasets for biclique sizes ranging from 3 to 9 (i.e., $3 \leq p, q \leq 9$). We can see that our algorithm demonstrates up to a 8× reduction in error compared to `EP/Zz++`, thanks to our carefully designed coloring scheme and (p, q)-broom-based sampling technique. Note that for the first three smaller datasets, `EP/Zz++` exhibits significant errors, with at least 38.9% inaccuracy, rendering its results unreliable. In contrast, our method, `CBS`, maintains a maximum error rate of 16.9% under the same number of sampling rounds. Combining these observations with the performance results shown in Figure 3.4, we can



(a) ER, EP/Zz++

(b) ER, CBS

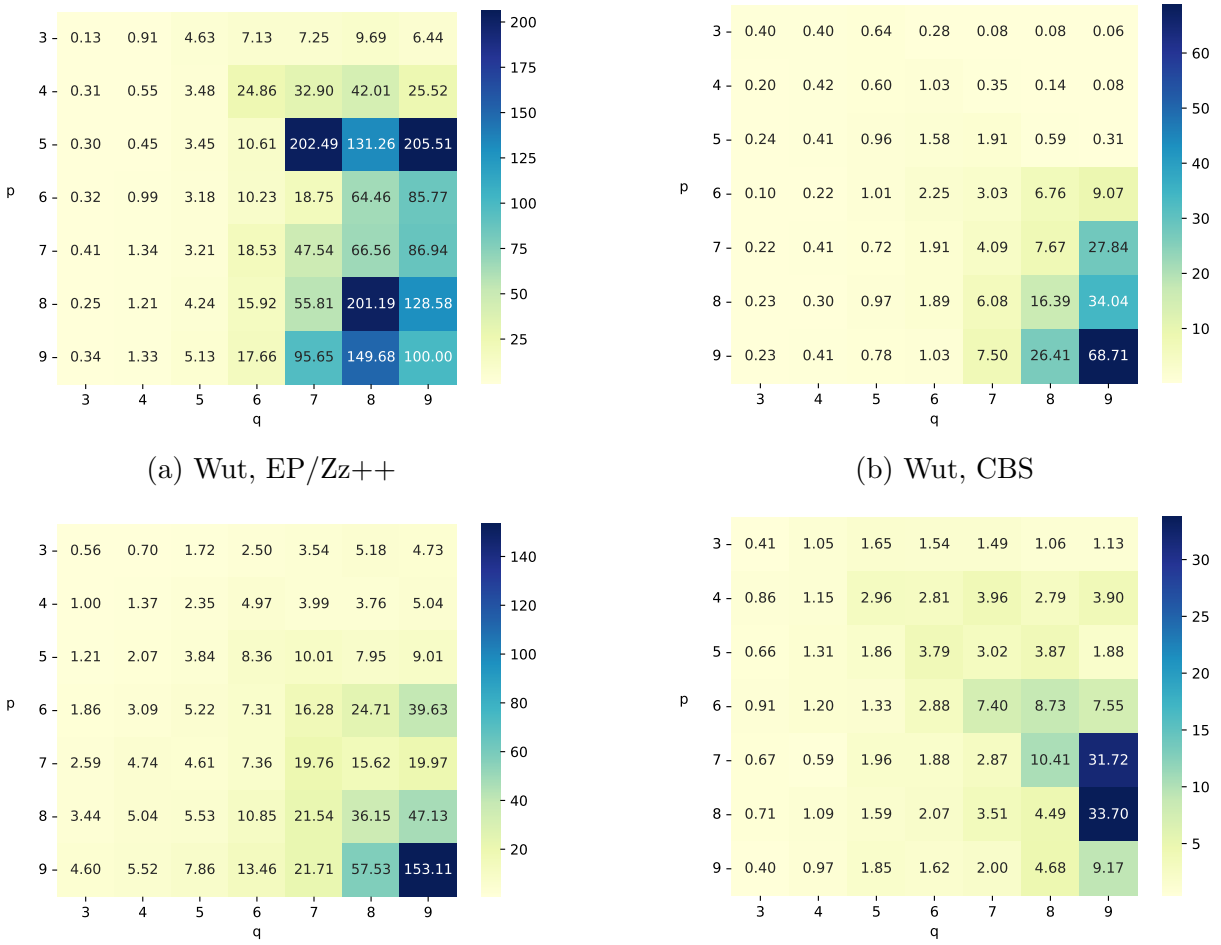
(c) DE, EP/Zz++

(d) DE, CBS

Figure 3.6: The heat-map of sampling time of EP/Zz++ and CBS with varying p and q (s).

conclude that our algorithm demonstrates an even greater advantage when considering the trade-off between accuracy and efficiency. This indicates that to achieve the same error rate, our algorithm would likely exhibit an even more substantial performance advantage over `EP/Zz++`.

3. Effect of p and q. Figures 3.6 and 3.7 illustrate the sampling time and error rate, respectively, of `CBS` and `EP/Zz++` across various p and q values. In each figure, rows represent p values and columns represent q values. Each cell displays the sampling time



(a) Wut, EP/Zz++

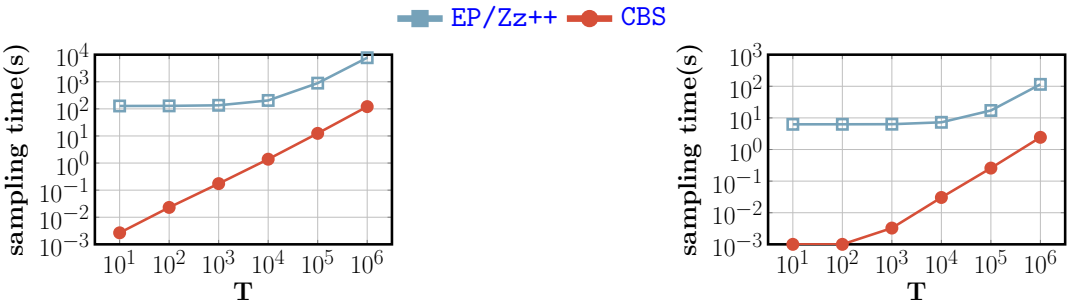
(b) Wut, CBS

(c) AR, EP/Zz++

(d) AR, CBS

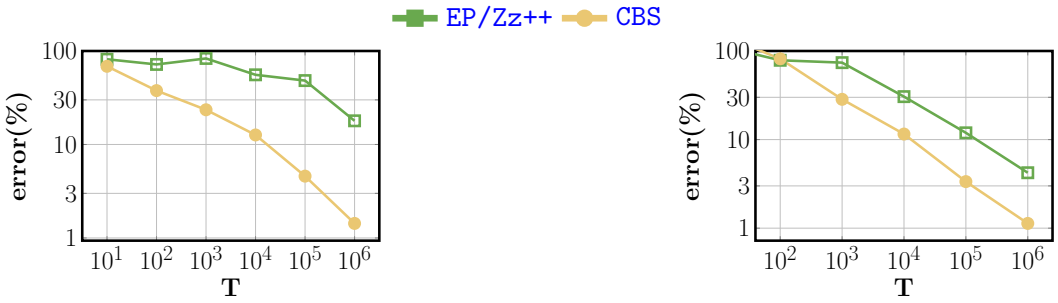
Figure 3.7: The heat-map of estimation errors of EP/Zz++ and CBS with varying p and q (%).

(in Figure 3.6) or the estimation error (in Figure 3.7) for counting the corresponding (p, q)-bicliques. Clear, our method, `CBS`, outperforms `EP/Zz++` by up to two orders of magnitude in both sampling time and accuracy. For instance, with $p = 5$ and $q = 3$ on the ER dataset, `EP/Zz++` requires 822.6 seconds for sampling, whereas our algorithm completes this stage in just 5.3 seconds. In terms of accuracy, on the Wut dataset with $p = 5$ and $q = 7$, `EP/Zz++` has an estimation error of 202.49, while our algorithm achieves a significantly lower error of 1.91. This observation aligns with our analysis in Section 3.1.2.



(a) ER (b) AR

Figure 3.8: Average sampling time of EP/Zz++ and CBS with varying T.



(a) Wut (b) AR

Figure 3.9: Average error of EP/Zz++ and CBS with varying T.

4. Effect of T. We evaluate the effect of sample numbers on sampling time and error rate. The results on three datasets are shown in Figures 3.8 and 3.9. Based on these results, we observe the following: (1) For sampling time, our algorithm consistently outperforms `EP/Zz++`, particularly when the sample numbers are relatively low (e.g., from 10^1 to 10^3). (2) Our algorithm consistently produces more accurate results than `EP/Zz++`, regardless of whether the sample size is high or low. (3) Even with very few samples, our algorithm achieves acceptable solutions. For instance, on the AR dataset, `CBS` requires only 10^3 samples to achieve a solution with a 30% error rate, whereas `EP/Zz++` requires over 10^4 samples.

3.1.3.3 Detailed Analysis

In addition, we present a more detailed analysis of `CBS`, including *Ablation Study*, *Time Cost of Different Stages*, and *Statistical of the Hyper-parameters*.

1.Ablation Study. To evaluate the effect of our coloring technique, we design a new variant of CBS by removing the vertex coloring step, denoted by BS. We then run them on two datasets and report the results in Figure 3.10. As we shall see, CBS generally achieves lower error with the same number of samples, aligning with our previous analysis. While the coloring technique slightly increases sampling time initially, the gap diminishes as sampling iterations increase. This demonstrates that in high-accuracy scenarios, our coloring technique can significantly improve accuracy without highly impacting sampling time.

T	AR		Wut	
	BS	CBS	BS	CBS
10^2	72.75	81.63	60.60	37.70
10^3	43.33	28.47	22.03	23.50
10^4	13.08	11.54	13.51	12.64
10^5	4.37	3.35	5.11	4.59
10^6	1.35	1.13	1.71	1.43

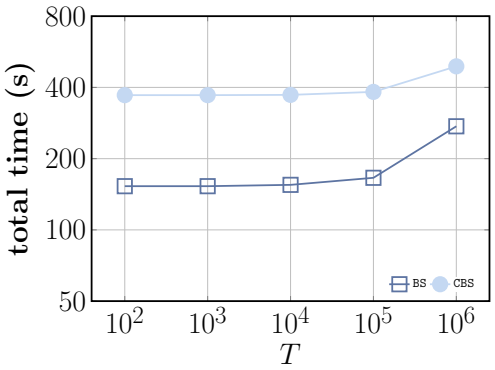


Figure 3.10: Results of BS and CBS with varying T.

2.Time Cost of Different Stages. For the two approximation algorithms, CBS and EP/Zz++, both require an initialization stage to precompute some auxiliary data for sampling. Specifically, in CBS, we need to assign a color number for each vertex (i.e., coloring) and count the number of (p, q)-brooms in bipartite graph, while in EP/Zz++, it needs to count the number of h-zigzag paths bipartite graph, where $h = \min\{p, q\}$. In Figure 3.11, we report the initializing and sampling times for these algorithms across all datasets. We observe that on more than half of the datasets, our algorithm requires less initializing and sampling time, which indicates that the number of (p, q)-brooms in the bipartite graph is typically less than h-zigzag paths (using in EP/Zz++). In terms of sampling time, our algorithm is up to two orders of magnitude faster than EP/Zz++. While on two datasets, GH and ER, EP/Zz++ is 10 times faster in initializing, CBS achieves 100 times lower sampling time, making the total time (i.e., initializing plus sampling) of our algorithm still highly efficient.

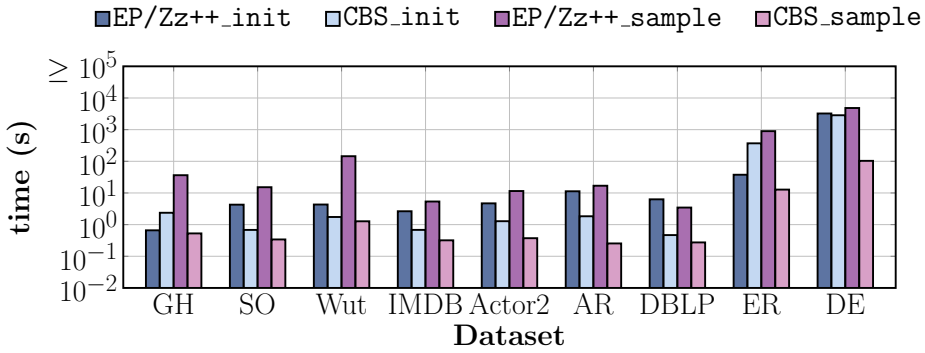


Figure 3.11: Average initializing and sampling time of EP/Zz++ and CBS for all $3 \leq p, q \leq 9$ ($T = 10^5$).

3.Statistical of the Hyper-parameters. Recall that in `CBS` and `EP/Zz++`, when ϵ is fixed, their required sample sizes are proportional to $\frac{B^2}{C^2}$ and $\frac{Z^2}{\rho^2}$, respectively. As shown in Table 3.2, we report the values of $\frac{B^2}{C^2}$ and $\frac{Z^2}{\rho^2}$ on the Amazon dataset for varying p and q. Similar trends are observed across other datasets. Based on Table 3.2, our algorithm `CBS` consistently requires fewer samples than `EP/Zz++`, explaining why `CBS` achieves the same or even lower estimation error with a smaller sample size. For instance, `CBS` requires up to 88× fewer samples than `EP/Zz++` on $p = 5$ and $q = 9$, demonstrating its efficiency in reducing sampling overhead while maintaining accuracy.

Table 3.2: The value of $\frac{B^2}{C^2}$ and $\frac{Z^2}{\rho^2}$ with varying p, q (Amazon).

(p, q)	$\frac{B^2}{C^2}$	$\frac{Z^2}{\rho^2}$
(3,4)	1.40E+03	1.79E+03
(3,5)	3.97E+03	1.69E+04
(3,9)	4.55E+04	2.85E+05
(4,5)	5.41E+04	5.98E+04
(4,8)	9.27E+04	2.48E+05
(5,3)	3.96E+02	6.17E+03
(5,6)	4.04E+05	1.20E+06
(5,9)	3.77E+04	3.34E+06
(6,4)	5.73E+03	5.80E+04
(6,7)	6.42E+06	5.32E+07
(7,4)	8.07E+03	1.54E+05
(7,7)	7.15E+06	2.44E+07
(8,4)	1.36E+04	3.16E+05
(8,8)	4.74E+07	1.04E+09
(9,4)	2.36E+04	5.32E+05
(9,9)	2.92E+08	8.01E+10

3.2 Historical Butterfly Counting

This section introduces the historical butterfly counting problem on temporal bipartite graphs. We propose a graph structure-aware indexing approach to facilitate efficient query processing with a tunable balance between query time and memory cost. Besides, we theoretically prove that our approach is especially advantageous on power-law graphs, by breaking the conventional complexity barrier associated with general graphs. To further mitigate the index overhead, we devised a high-quality approximation algorithm that leverages a compressed index structure, thereby enhancing overall efficiency. Extensive empirical tests show that our algorithm is up to five orders of magnitude faster than existing algorithms with controllable memory costs.

The results of this section come from the joint work [152] with Qiuyang Mang, Hangrui Zhou, Yu Gao, Yingli Zhou, Qingyu Shi, Richard Peng, Yixiang Fang, Chenhao Ma. The primary authors are Jingbang, Qiuyang, and Hangrui. The setting was proposed by Jingban, and the algorithm was proposed by the primary author together. Jingbang was in charge of the whole project and wrote the main paper except for: (1) the hardness result (by Qiuyang); (2) the experiment (conducted by Hangrui and written by Jingbang and Qiuyang).

3.2.1 Introduction

Due to its ability to model relationships between two distinct sets of entities, the bipartite graph, or network, holds significant importance across various fields, including disease control on people-location networks [86, 171], fraud detection on user-page networks [207, 142] and recommendation on customer-product networks [206, 106, 105, 208, 217]. To analyze the structure and dynamics of the network, counting motifs is one of the most popular methods [156, 208, 88, 118, 87] since motifs are considered the basic construction block of the network. The *butterfly* motif (2×2 biclique) represents fundamental interaction patterns within the graph. Counting it has wide applications ranging from biological ecosystems [207, 185, 205], where it helps in identifying mutual relationships between species, to social networks [234, 208], where it uncovers patterns of collaborations and affiliations.

Temporal bipartite graphs, in which edges typically carry timestamps, are often considered [45, 58, 87, 171] since real-world interactions (modeled as edges) usually occur at specific timestamps. Recently, Cai et al. [45] first considered counting butterflies on temporal bipartite graphs, which extends the analytical depth of traditional bipartite graph

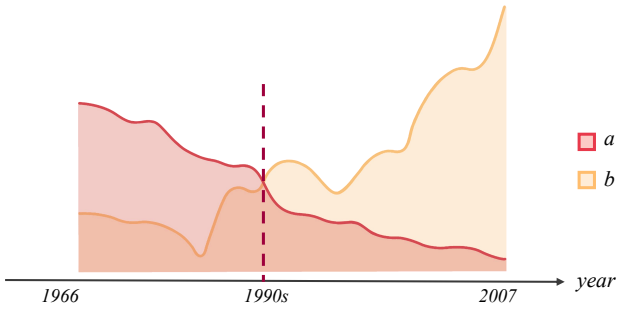


Figure 3.12: Jim Gray's activeness in database community (a) and astronomy (b) community.

analyses by incorporating the dimension of time, making it a powerful tool for uncovering dynamic patterns in complex systems.

However, merely counting butterflies across the entire timeline, as suggested by [45], may not accurately reflect the dynamic nature of relationships, failing to capture evolving trends. To address this, it is essential to consider the temporal dimension of interactions. By focusing on the occurrence of motifs within specified time frames, we introduce the concept of *historical butterfly counting*. This approach, which involves analyzing butterflies within discrete time windows on a temporal bipartite graph, offers enhanced insights into the timing and progression of interactions. It provides an in-depth understanding of network dynamics, uncovering the mechanisms behind network evolution and revealing opportunities for precise interventions across various domains.

Consider the example shown in Figure 3.12. By applying historical butterfly counting, we capture the emerging and diminishing trends in Jim Gray's research interests and his level of engagement within the database and astronomy communities. If we were to ignore the time dimension and simply count butterflies over the entire period from 1966 to 2007, we would obtain only two cumulative butterfly counts for each community. This aggregate approach would indicate Jim's activity across both fields but fail to reveal his gradual shift in focus over time. In contrast, historical butterfly counting provides a more detailed view, allowing us to pinpoint specific periods of change, such as the gradual shift from databases to astronomy around the 1990s. Hence, the historical butterfly counts offer a more granular view of evolving communities and reveal periods of gradual or sudden changes.

Regarding the approximate solution, we brief that in many applications of butterfly counting—such as graph kernel analysis [191] and network measurement tasks like detecting dense substructures in recommendation systems [185, 205]—an approximate count is often sufficient. Here, we expand some discussions:

In **recommendation systems**, butterfly counting can reveal dense substructures among users or items, such as groups of users with overlapping interests or items frequently co-purchased. For example, approximate butterfly counts can be used to identify clusters of products that are likely to be bought together, enhancing cross-selling recommendations without needing perfectly exact counts. Given the massive scale of data in e-commerce or streaming platforms, allowing a small error margin can lead to substantial memory savings, enabling real-time processing and improved system scalability.

Approximate butterfly counts are also valuable in **graph kernel methods**, where motifs are used to create similarity measures between graphs for tasks such as classification or anomaly detection. Since graph kernels rely on relative similarities rather than exact counts, approximate butterfly counts are sufficient to maintain kernel performance while reducing memory usage and computational cost. This approach enables efficient similarity computations in fields like bioinformatics (e.g., comparing protein interaction networks) or cybersecurity (e.g., identifying similar attack patterns across networks).

Applications We now discuss some other interesting applications of historical butterfly counting:

• **Bipartite Clustering Coefficient (BCC) Computation.** The *bipartite clustering coefficient* [140, 8, 167] is a traditional cohesiveness measure for bipartite graphs whose computing bottleneck is counting butterflies. Specifically, considering the scientific collaboration network (modeled as a temporal graph) in the given time window, a higher BCC suggests a stronger trend of cohesive collaboration within the research community. This coefficient positively correlates with the number of author pairs publishing multiple publications within a given time-window and inversely correlates with the number of author pairs collaborating only once. For example, in academia, a scientist may change their frequent collaborators or research interests over time. We also provide a case study on the global research collaboration trend and close collaboration time windows in Section 3.2.6 with BCC computation.

• **Identifying Close Communication Time-windows.** Directly counting butterflies can identify time windows during close communication within a specific community. For instance, we are interested in close collaboration. Among Kaiming He's 2-hop ego networks w.r.t. all possible time-windows, we find that the two-year time-window with the highest butterfly count is 2010 - 2011, with the corresponding research records detailed in Figure 3.13. During this period, Kaiming He, Jian Sun, Xiaoou Tang, Carsten Rother, and Christoph Rhemann established a series of cohesive research collaborations, resulting in the publication of three papers. In other words, Figure 3.13 depicts Kaiming's 2-hop ego network during 2010 - 2011, the two-year time interval with the highest butterfly count.

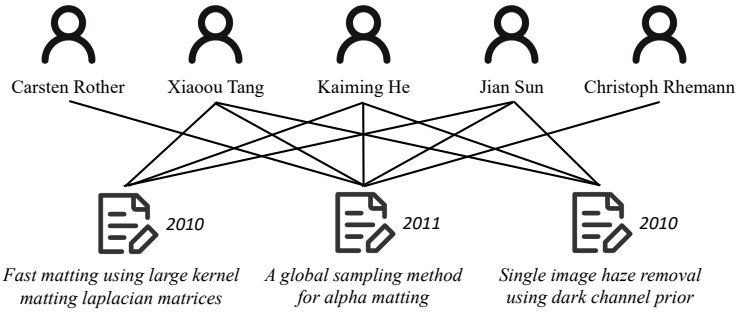


Figure 3.13: Finding the time-window of the closest collaboration.

Notably, 2010 and 2011, the last two years of Kaiming's Ph.D. under supervisor Xiaoou, marked his most cohesive collaborations.

Challenges and Contributions Counting butterflies in such a historical setting is challenging. One main reason is that the algorithm should be able to answer historical queries multiple times to analyze the changing trend. When the graph is large, it is inefficient to run existing butterfly counting algorithms from scratch for each query. Thus, we need to design algorithms that answer each query more efficiently after preprocessing. No algorithm can effectively solve this problem in existing works. This work fills this hole by proposing a new index algorithm with consistently high performance in large-scale graphs. The proposed algorithm `GSI` (graph structure-aware index) can take advantage of graph structures and balance between query time and memory usage, enabling it to outperform previous butterfly counting algorithms on both real-world data, and synthetic data with certain distributions. `GSI` can also be parallelized, providing faster query efficiency. It can also be compressed, providing smaller memory usage. To summarize, we have made the following contributions.

- We introduce the historical butterfly counting problem and prove its hardness. This enables in-depth trend and dynamics analysis over temporal graphs, and helps understand how temporal variations influence network structures over time.
- The graph structure-aware index (`GSI`) is designed to support efficient counting query with a controllable balance between query time and memory usage. Theoretically, we prove that the `GSI` approach transcends conventional computational complexity barriers associated with general graphs when applied to power-law graphs, a common characteristic of many real-world graphs.
- When memory becomes the bottleneck and exact counting is not required, we propose an index compression strategy to provide fast, high-quality, and unbiased approximations

of the counts based on the compressed index.

- Extensive experiments demonstrate that our query algorithm achieves up to five orders of magnitude speedup over the state-of-the-art solutions with manageable memory footprints.

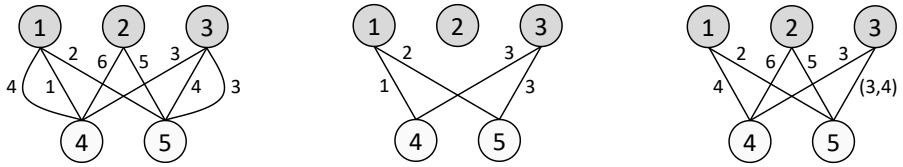
3.2.1.1 Related Work

In this section, we review the related works, including the butterfly counting on static bipartite graphs, motif counting on temporal graphs, and other historical queries on temporal graphs.

• **Butterfly Counting on Static Bipartite Graphs.** Butterfly is the most fundamental sub-structure in bipartite graphs. Significant research efforts have been dedicated to the study of counting and enumerating butterflies on static bipartite graphs [207, 185, 205]. Wang et al. [205] first proposed the butterfly counting problem and designed an algorithm by enumerating wedges from a randomly selected layer. Sanei-Mehri et al. [185] further developed a strategy for choosing the layer to obtain better performance. Recently, Wang et al. [207] utilized the vertex priority and cache optimization to achieve state-of-the-art efficiency. Additionally, the parallel algorithms [185, 192], I/O efficient algorithm [212], sampling-based algorithms [185, 137, 191], GPU-based algorithm [191, 218], and batch update algorithm [209] have also been developed for the butterfly counting problem. In addition, the butterfly counting problems in steam, uncertain, and temporal bipartite graphs have also been studied [45, 191, 186, 231].

• **Motif Counting on Temporal Graphs.** The problem of temporal motif counting has been extensively studied recently [35, 99, 139, 145, 128]. Kovanen et al. [128] introduced the concept of Δ-adjacency, which pertains to two temporal edges sharing a vertex and having a timestamp difference of at most Δ, and consider the temporal ordering aspect. The Δ-temporal motif counting with [170, 180] and without temporal ordering [169] are also been studied. Furthermore, there are numerous approximation algorithms available for solving counting problems [144, 187]. When it comes to enumeration problems, isomorphism-based algorithms are the most commonly used [138, 146].

• **Other Historical Queries on Temporal Graphs.** The historical queries on the temporal graphs aim to compute the specific structure in the snapshot of an arbitrary time window. The historical reachability [214], k-core [227], structural diversity [57], and connected components [219] of temporal graphs have been defined, and index-based solutions have also been proposed.



(a) a temporal bipartite graph G with duplicated edges. **(b)** a projected graph $G_{[1,3]}$, whose butterfly count is 1. **(c)** a projected graph $G_{[2,6]}$, whose butterfly count is 3.

Figure 3.14: A temporal bipartite graph and its projected graphs in two time-windows, associated with their butterfly counts.

Note that our work, *the historical butterfly counting*, is significantly different with *the temporal butterfly counting* in [45], as our work focuses on more generalized scenarios of temporal graph mining, i.e., analyzing the relationship between vertices in a time interval and without any temporal ordering limitations. Specifically, they only focus on temporal butterfly counting/listing such that the duration time is no larger than a given threshold, while our historical butterfly counting considers querying on arbitrary time windows.

3.2.2 Preliminaries

Throughout the work, we use the big-O notation to express the upper bounds of the *absolute value*s of functions. For example, $O(\frac{1}{n})$ includes functions between $-\frac{c}{n}$ and $\frac{c}{n}$ for some constant c and sufficiently large n. We say $f(n) \geq g(n) + O(h(n))$ if $f(n) \geq g(n) - c|h(n)|$ for some constant c and sufficiently large n. We use $\widetilde{O}(f(n))$ to denote all functions bounded by $O(f(n)\log^c n)$ for some constant c. We use $[n]$ to denote the set $\{1, \ldots, n\}$. For an undirected graph, we denote every vertex u's degree as deg_u.

3.2.2.1 Problem Definitions

Two common motifs are being widely studied on bipartite graphs: wedges and butterflies. We give their formal definition as follows:

Definition 3.2.1 (Wedge [207])**.** *Given a bipartite graph $G = (V = (U, L), E)$, a wedge $\langle x \rightsquigarrow y \rightsquigarrow z \rangle$ is a 2-hop path consisting of edges (x, y) and (y, z).*

Definition 3.2.2 (Butterfly [207])**.** *Given a bipartite graph $G = (V = (U, L), E)$, and the four vertices $x, y, z, w \in V$ where $x, z \in U$ and $y, w \in L$, $b : \langle x, y, z, w \rangle$ is a butterfly iff the subgraph induced by x, y, z, w is a $(2, 2)$-biclique of G; that is, x and z are all connected to y and w, respectively.*

Throughout the work, we study the temporal bipartite graphs. A temporal bipartite graph is an undirected graph $G = (V = (U, L), E)$, where each edge $e \in E$ is a triple (u, v, t) with two vertices $u \in U, v \in L$ and a timestamp t. Our major focus is the historical type query on temporal bipartite graphs; that is, we query on an extracted graph from G with respect to a certain time window. We denote it as the *projected graph* and its formal definition is as follows:

Definition 3.2.3 (Projected graph [88])**.** *Given a temporal bipartite graph $G = (V, E)$ and a time-window $[t_s, t_e](t_s \leq t_e)$, the projected graph $G_{[t_s,t_e]}$ of G is an undirected bipartite graph without timestamps, where its vertex set $V_{[t_s,t_e]}$ is V and the edge set $E_{[t_s,t_e]}$ is $\{(u, v) \mid \exists (u, v, t) \in E \ \wedge \ t \in [t_s, t_e]\}$.*

Now, we are ready to state the major problem formally:

Problem 3 (Historical butterfly counting)**.** *Given a temporal bipartite graph G and a time-window $[t_s, t_e]$, find the number of butterflies in the projected graph $G_{[t_s,t_e]}$.*

In Figure 3.14, we are given a temporal bipartite graph G in (a), in which the number represents the timestamp of each edge. We consider two time-windows $[1, 3]$ and $[2, 6]$. The corresponding projected graphs $G_{[1,3]}$ and $G_{[2,6]}$ are shown in (b) and (c), respectively. The historical butterfly counting query associated with these two-time windows is, in fact, counting on these two projected graphs. Therefore, the answer is 1 ($\langle 1, 3, 4, 5\rangle$) for $[1, 3]$ and 3 ($\langle 1, 2, 4, 5\rangle$, $\langle 1, 3, 4, 5\rangle$, $\langle 2, 3, 4, 5\rangle$) for $[2, 6]$.

3.2.2.2 Some Key Techniques in Motif Counting

In the previous works on motif counting [207, 212], the value of $\frac{1}{|E|}\sum_{(u,v)\in E} \min(\deg_u, \deg_v)$ is widely used for complexity analysis for the given input graph $G = (V, E)$. By [61], this can be simplified as $O(\delta)$, where δ is defined as the arboricity of the given graph. We will adapt the δ notion for brevity.

Our proposed algorithms widely use two important techniques: the *Vertex Priority* method and the *Chazelle's structure*.

Vertex Priority The vertex priority method reduces the number of wedges we need to consider. To begin with, we define the vertex priority as follows:

Definition 3.2.4 (Vertex priority [207]). *For any pair vertices x, y in a temporal bipartite graph G, we define x is prior to y ($pr(x) \prec pr(y)$) if and only if: $\overline{deg}_x$ ¿ $\overline{deg}_y$, or $\overline{deg}_x = \overline{deg}_y$ and $id(x) < id(y)$, where $\overline{deg}_u$ denotes the degree of u when only considering unique edges in G, and $id(u)$ denotes the unique ID of u.*

By [207], we only need to consider the wedges $\langle x \rightsquigarrow y \rightsquigarrow z \rangle$ that satisfy $pr(x) \prec pr(y) \wedge pr(x) \prec pr(z)$ in order to count the butterflies without repetition or missing. Each butterfly $\langle x, y, z, w \rangle$ is constructed from two such wedges $\langle x \rightsquigarrow y \rightsquigarrow z \rangle$ and $\langle x \rightsquigarrow w \rightsquigarrow z \rangle$.

2D-range Counting Our algorithm will transform counting butterflies into counting points on a 2-dimensional plane, known as the *2D-range counting* problem. Specifically, let P be a set of n points in 2-d space $\mathbb{R}^2$. The *2D-range counting* problem is: Given an orthogonal rectangle Q of the form $[x_1, x_2] \times [y_1, y_2]$, find the size of $|Q \cap P|$. For an instance of the 2D-range counting problem, we use a classic data structure known as Chazelle's structure [52] to handle all 2D-range counting queries after preprocessing:

Theorem 3.2.1 (Chazelle's structure [52]). *A Chazelle's structure $\mathcal{CS}$ is a data structure that can answer each 2D-range counting in $O(\log n)$ time and $O(\frac{n \log n}{\omega})$ memory usage, where ω is the word size. The preprocessing time is $O(n \log n)$.*

In practice, 2^ω significantly exceeds the number of points involved in our method's 2D-range counting task. Therefore, we assume that $\log n$ is $O(\omega)$ for Chazelle's structures used and simplify the memory usage into $O(n)$ for brevity. In later sections, we use $\mathcal{CS}$ to denote such a data structure.

3.2.2.3 Baselines

Our baseline solutions are built upon the *states-of-the-arts* methods for exact butterfly counting (i.e., `BFC-VP++` [207]) and approximate butterfly counting (i.e., Weighted Pair Sampling (`WPS`) [228]) on the static graphs. We need to extract the static graph from the temporal bipartite graph for a given time-windows first, and then run the following solutions:

- `BFC-VP++`: The `BFC-VP++` algorithm sorts vertices by their proposed vertex priority (Definition 3.2.4) and efficiently identifies almost all minimally redundant wedges that can form a butterfly, offering a practical, efficient, and cache-friendly solution. In addition, `BFC-VP++` can be highly parallelized. We will also compare our methods with its parallel version in the following.

- `WPS`: The basic idea behind `WPS` is to estimate the total butterfly count with the number of butterflies containing two randomly sampled vertices from the same side of the two vertex sets. The method has been proven as unbiased and theoretically efficient in power-law graph models.

3.2.3 Algorithm

In this section, we consider solving Problem 3 exactly by index-based solutions. To begin with, in Section 3.2.3.1, we give a hardness result showing that $\widetilde{O}(m^2/\lambda^2)$ space is needed to answer queries exactly in $\widetilde{O}(\lambda)$ time, where m denotes the number of edges in the given temporal bipartite graph G. Correspondingly, we provide an algorithm that meets such bound in Section 3.2.3.2, named as *Enumeration-based Index (`EBI`)*. `EBI` achieves $\widetilde{O}(1)$ query time but it needs expensive memory usage for large graph data. In Section 3.2.3.3, we introduce a different algorithm named *Combination-based Index (`CBI`)* that can be constructed under practical memory constraints.

To bridge the gap between theory and practice, we propose a new algorithm in Section 3.2.3.4 that effectively combines `EBI` and CBI, demonstrating strong performance on real-world graph data. This algorithm, named *Graph Structure-aware Index (`GSI`)*, intelligently allocates graph data to the two indices which allows it to take advantage of the underlying graph structure. In addition, without compromising the performance, `GSI` can also handle duplicate edges with proper modification, which is discussed in Section 3.2.7.

3.2.3.1 Problem Hardness

Our result is motivated by the hardness result in [70] where they reduce the Set Disjointness problem (Definition 3.2.5) into Problem 3. While [70] considers timestamps on vertices, we prove for the setting where timestamps are on edges separately.

Definition 3.2.5 (Set Disjointness)**.** *Given a collection of $s \geq 2$ sets S_1, S_2, ...,S_s, a query $(a, b) \in [s]^2$ asks for whether $S_a \cap S_b$ is empty.*

The strong set disjointness conjecture [94, 95] is as follows:

Theorem 3.2.2. *For the set disjointness problem, any data structure with query time λ must use $\widetilde{\Omega}(n^2/\lambda^2)$ space where n is the sum of the sizes of $S_1, \ldots, S_s$.*

Specifically, we prove the following result for Problem 3:

Theorem 3.2.3. *Consider Problem 3. Let m denote the number of edges in G. Fix any $\lambda \in [1, m]$ and $\delta > 0$. Suppose that we have a data structure for Problem 3 using $\widetilde{O}(m^{2-\delta}/\lambda^2)$ space and exactly answers each query in $\widetilde{O}(\lambda)$ time. Then, for any set disjointness problem with $\sum_{i=1}^{s} |S_s| = N$, we have a data structure that uses $\widetilde{O}(N^{2-\delta}/\lambda^2)$ time and that answers each query in $\widetilde{O}(\lambda)$ time.*

Proof. Suppose that we have a data structure for Problem 3 using $\widetilde{O}(m^{2-\delta}/\lambda^2)$ space and exactly answers each query in $\widetilde{O}(\lambda)$ time. We can utilize the data structure for a set disjointness instance as follows:

Let there be s sets $S_1, \ldots, S_s$. Without loss of generality, we let $1, \ldots, t$ denote all distinct elements in $S_1, \ldots, S_s$, *i.e.*, $\bigcup_{i=1}^{s} S_i = [t]$. We construct a graph $G = (V = (U, L), E)$ such that $U = [s]$ and $L = [t+1]$. An edge (i, j) exists if $j \in S_i$ or $j = t+1$. We assign timestamp i to the edge (i, j).

We build the data structure for the graph G defined above. For two integers $a \leq b \in [s]$, we can query the data structure for the number of butterflies in the time window $[a, b]$. Since the timestamp on edge (i, j) is equal to i, such butterflies are exactly those with the form $\langle u, v, w, x \rangle$ such that $u, w \in [a, b]$. The number of such butterflies is

$$Q(a, b) \stackrel{\text{def}}{=} \sum_{i=a}^{b} \sum_{j=i+1}^{b} \binom{|S_i \cap S_j| + 1}{2}.$$

Notice that $Q(a, b)$ is the two-dimensional prefix sum of $\binom{|S_a \cap S_b|+1}{2}$. In other words, we can calculate $\binom{|S_a \cap S_b|+1}{2}$ by $Q(a, b) - Q(a+1, b) - Q(a, b-1) + Q(a+1, b-1)$, where we define $Q(x, y) = 0$ for $x \geq y$. We can answer whether $|S_a \cap S_b| = 0$ by checking whether this value is equal to zero.

□

For example, in the Figure 3.15, we reduce the set disjointness problem for three sets $\{1, 2\}$, $\{2, 3\}$ and $\{3, 4\}$ to a historical butterfly counting problem on the graph we constructed. For determining whether $S_1 \cap S_2$ is empty, we count the number of butterflies of the form $\langle 1, v, 2, x \rangle$, $Q(1, 2)$, which is equal to $\binom{|S_1 \cap S_2|+1}{2}$. For determining whether $S_1 \cap S_3$ is empty, we calculate $\binom{|S_1 \cap S_3|+1}{2} = Q(1, 3) - Q(2.3) - Q(1, 2) + Q(2, 2) = 2 - 1 - 1 + 0 = 0$.

This hardness result shows that to achieve efficient query runtime (e.g., $\widetilde{O}(\lambda)$), large memory space is necessary (e.g., $\widetilde{O}(m^2/\lambda^2)$). Even though we can design theoretically optimal algorithms that reach the lower bound, they might not be practical when the

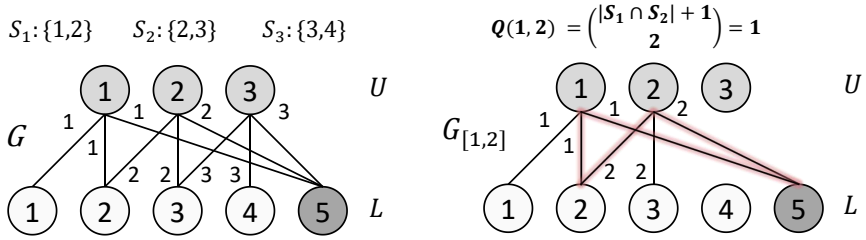


Figure 3.15: An example for reducing the set disjointness problem into exact historical butterfly counting in temporal bipartite graphs.

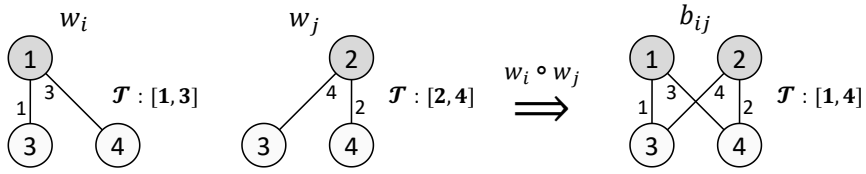


Figure 3.16: An illustrative example for active timestamps of wedges and the butterfly constructed from them.

input graph is large. One possible way to overcome such a challenge from a practical perspective is to develop algorithms that take advantage of real-world graphs' properties.

3.2.3.2 Enumeration-based Index

Since the structure of a temporal graph changes over time, each subgraph has its own life cycle, i.e., the only interval of time in which it exists. We denote such an interval as the active timestamp of this subgraph. We give a formal definition as follows:

Definition 3.2.6 (Active Timestamp)**.** *For any subgraph P of a bipartite temporal graph G, we define the active timestamp $\mathcal{T}(P)$ as a pair of ordered timestamp $[l, r](l \leq r)$, such that P exist in the projected graph $G_{[t_s,t_e]}$ if and only if $t_s \leq l$ and $r \leq t_e$.*

In Figure 3.16, there are two wedges w_i and w_j. For w_i, since it contains two edges $(1, 3, 1)$ and $(1, 4, 3)$, it only exists in a projected graph $G_{[t_s,t_e]}$ satisfying $t_s \leq 1$ and $t_e \geq 3$. The active timestamp $\mathcal{T}(w_i)$ is $[1, 3]$. Similarly, for w_j, its active timestamp is $[2, 4]$.

The proposed *Enumeration-based Index (`EBI`)* algorithm is shown in Algorithm 25 and Algorithm 26. For brevity, we assume that no duplicate edge exists. We discuss how to handle duplicate edges without compromising the performance in Section 3.2.7.

• **Construction** The goal is to enumerate every butterfly and compute its active timestamp. After initializing a 2D-range counting data structure T_E by all butterflies' timestamps, we can answer every historical butterfly counting query through a single query on T_E. We initialize it to be empty in the beginning (Line 1).

To find all the butterflies in G, we enumerate all the wedges. We consider two wedges w_i, w_j of G with shared endpoints but different middle vertices, denoted as $w_i : \langle x \rightsquigarrow y_i \rightsquigarrow z \rangle$ and $w_j : \langle x \rightsquigarrow y_j \rightsquigarrow z \rangle$. Let $\mathcal{T}(w_i) = [l_i, r_i]$ and $\mathcal{T}(w_j) = [l_j, r_j]$. The temporal butterfly constructed from them is denoted as $b_{ij} = w_i \circ w_j$, where $\mathcal{T}(b_{ij}) = [\min(l_i, l_j), \max(r_i, r_j)]$.

In Figure 3.16, we demonstrate how to construct a butterfly b_{ij} with two wedges w_i and w_j. Since b_{ij} contains edges $(1, 3, 1)$, $(1, 4, 3)$, $(2, 3, 4)$, and $(2, 4, 2)$, its active timestamp can be computed by $l = \min\{t | \ \exists e = (u, v, t) \in edges(b_{ij})\} = 1$, and $r = \max\{t | \ \exists e = (u, v, t) \in b_{ij}\} = 4$. By the definition of the $\circ$ operation, it can also be computed from $\mathcal{T}(w_i)$ and $\mathcal{T}(w_j)$. This also indicates that a butterfly $b_{ij} = w_j \circ w_j$ exists in a given time-window, if and only if the active timestamp of w_i and w_j are both included in the window. Such observation helps us develop an alternative index approach in the later section.

We follow the method in [207] to enumerate every wedge $w_i : \langle x \rightsquigarrow y \rightsquigarrow z \rangle \, [l_i, r_i]$ in G without repetition or missing (Line 3). The complexity is bounded by the total number of wedges in G. Let $W[(x, z)]$ be a set of wedges whose endpoints are x, z. W is the family of sets $W[(x, z)]$. W is initialized to be empty (Line 2). For the current wedge w_i, we first check if its endpoints (x, z) exists in W (Line 4) (by a hash table, for example). If not, we initialize the corresponding set $W[(x, z)]$ to be empty (Line 4).

Since any two different wedges with the same pair of endpoints (x, z) construct a unique butterfly, we enumerate every w_j in
$W[(x, z)]$ (Line 5) and construct a butterfly $b_{ij} = w_i \circ w_j$ with $\mathcal{T}(b_{ij}) = (t_l, t_r)$ (Line 6). Correspondingly, we insert a single 2D point (t_l, t_r) into T_E. After enumeration, we need to update $W[(x, z)]$ by inserting w_i in it (Line 8). After finding all the butterflies, we run the preprocessing for T_E and return it (Line 10).

In Figure 3.17, we provide an example of our construction process. There are six wedges $w_1 : \langle 1 \rightsquigarrow 4 \rightsquigarrow 3 \rangle$, $w_2 : \langle 1 \rightsquigarrow 5 \rightsquigarrow 3 \rangle$, $w_3 : \langle 1 \rightsquigarrow 4 \rightsquigarrow 2 \rangle$, $w_4 : \langle 1 \rightsquigarrow 5 \rightsquigarrow 2 \rangle$, $w_5 : \langle 2 \rightsquigarrow 4 \rightsquigarrow 3 \rangle$, $w_6 : \langle 2 \rightsquigarrow 5 \rightsquigarrow 3 \rangle$ where $\mathcal{T}(w_1) = [4, 4]$, $\mathcal{T}(w_2) = [2, 4]$, $\mathcal{T}(w_3) = [4, 6]$, $\mathcal{T}(w_4) = [2, 5]$, $\mathcal{T}(w_5) = [4, 6]$, $\mathcal{T}(w_6) = [4, 5]$. We group them by (x, z). There will be three groups (sets) of wedges: $W[(1, 3)] = \{w_1, w_2\}$, $W[(1, 2)] = \{w_3, w_4\}$, $W[(2, 3)] = \{w_5, w_6\}$. For each group, we construct a butterfly b_{ij} from each pair of different wedges w_i, w_j from the group. We have $b_1 = w_1 \circ w_2$, $b_2 = w_3 \circ w_4$, $b_3 = w_5 \circ w_6$. We compute their active timestamp through simple calculation: $\mathcal{T}(b_1) = [2, 4]$, $\mathcal{T}(b_2) = [2, 6]$, $\mathcal{T}(b_3) = [4, 6]$. In the end, we insert $(2, 4)$, $(2, 6)$, $(4, 6)$ into T_E. Note that in this showcase, for demonstrating

the group dividing, we do not follow the vertex priority in Definition 3.2.4. Otherwise, we only consider three wedges and they are all in the group $W[(4,5)]$.

• **Answering Query** Since each butterfly with active timestamp $[l,r]$ is represented by a point (l,r) in T_E, a historical butterfly counting of time-window $[t_s,t_e]$ can be interpreted as a 2D-range query counting the number of points in the rectangle $[t_s,\infty]\times[-\infty,t_e]$. We query T_E for it and return the answer directly (Line 1).

In Figure 3.17, after the preprocessing of T_E, we can answer any 2D-range counting query concerning those three points, which represent the three butterflies. When we want to query the number of butterflies in time-window $[1,5]$, we are asking how many butterflies' active timestamps are in the range of $[1,5]$. We interpret this as querying the number of points in the rectangle $[1,\infty]\times[-\infty,5]$. There is only one point $(2,4)$ in the rectangle. Therefore, there is only one butterfly in the projected graph of the queried time-window. The calculation is done by a single query on T_E. It is a typical 2D-range counting query.

• **Time and Space Complexity** The time and space complexity of `EBI` is summarized as follows:

Theorem 3.2.4. *Denote B as the number of butterflies of the input graph G and δ as the arboricity, `EBI`'s construction (Algorithm 25) runs in $O(m\delta + B\log m)$ and takes $O(B)$ space. After the construction, each query (Algorithm 26) takes $O(\log m)$ to answer.*

Proof. The construction process requires enumerating all wedges. By [207], this takes $O(m\delta)$. For each wedge $w_i : \langle x \rightsquigarrow y \rightsquigarrow z\rangle$, it is inserted in $W[(x,z)]$. By enumerating every existing w_j in $W[(x,z)]$, every butterfly $b_{ij} = w_i \circ w_j$ is enumerated exactly once. Since the $\circ$ operation takes $O(1)$ and the insertion to T_E takes $O(\log m)$ by Theorem 3.2.1, the total preprocess time for all butterflies takes $O(B\log m)$. Therefore, the total runtime for Algorithm 25 is $O(m\delta + B\log m)$ as desired. Every wedge will be stored in $W[(x,z)]$ where x,z are its endpoints. All the wedges take $O(m\delta)$ space in total. T_E contains $O(B)$ points in total. By Theorem 3.2.1, it takes $O(B)$ space. The total memory usage sum up to $O(B+m\delta)$. Each query on `EBI` is a range query on T_E. By Theorem 3.2.1, this takes $O(\log m)$ time and no additional space. □

The `EBI` algorithm reaches the bound in Section 3.2.3.1: Since δ is bounded by $O(\sqrt{m})$ and B is bounded by $O(m^2)$, the memory usage is bounded by $\widetilde{O}(m^2)$. Therefore, `EBI` answers each query in $\widetilde{O}(1)$ time and take $\widetilde{O}(m^2)$ space. This indicates that `EBI`'s asymptotic memory usage cannot be further improved without affecting the $\widetilde{O}(1)$ query time. On the other hand, though reaching theoretical optimality, `EBI`'s performance on large-scale data is not ideal because enumerating and storing all butterflies is challenging for these graphs.

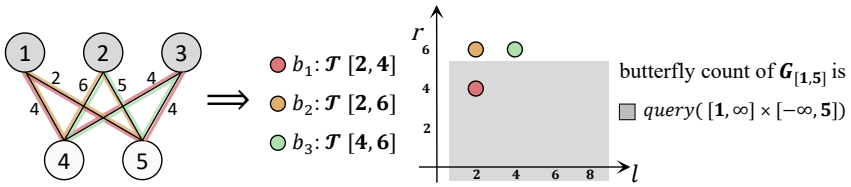


Figure 3.17: An example for transforming historical butterfly counting into 2D-range counting.

Algorithm 25: EBI-Construction

Input: a bipartite temporal graph: $G = (V, E)$;
Output: a $\mathcal{CS}$ (Theorem 3.2.1) for indexing (EB-index): T_E;

Initialize T_E with an empty $\mathcal{CS}$;
$W \leftarrow$ an empty hashmap mapping pairs of vertices to sets;
for *each wedge* $w_i : \langle x \rightsquigarrow y \rightsquigarrow z \rangle\ [l_i, r_i] \in G$ **do**
    if $(x, z) \notin W.keys()$ **then** $W[(x, z)] \leftarrow \emptyset$;
    for $w_j \in W[(x, z)]$ **do**
        $b_{ij} \leftarrow w_i \circ w_j$;
        $T_E.insert(t_l, t_r)$; // $(t_l, t_r) = \mathcal{T}(b_{ij})$
    $W[(x, z)] \leftarrow W[(x, z)] \cup w_i$;
Preprocess T_E (Theorem 3.2.1);
return T_E;

For example, in the `Wiktionary` dataset[2], the butterfly count exceeds 10^{16} where the graph contains about 4×10^7 edges[3].

3.2.3.3 Combination-based Index

If the graph is large, it is not realistic to pre-compute and store all butterflies to handle historical queries, as this requires too much space. On the other hand, we are only interested in the number of butterflies, so it is not necessary to construct them explicitly.

Recall that a butterfly exists in a given time window if and only if the active timestamp of the two wedges are both included in the window. Consider a *group of wedges* $S = \{\langle x \rightsquigarrow y \rightsquigarrow z \rangle\}$ where x, z are fixed. Any two wedges from S form a butterfly. Therefore,

[2] http://konect.cc/
[3] Here, #butterflies is larger than the square of #edges due to duplicate edges, which will be handled in Section 3.2.7

Algorithm 26: EBI-Querying

Input: a time-window: $[t_s, t_e]$; a EB-index: T_E;
Output: temporal butterfly counting: $num_{[t_s,t_e]}$;
1 **return** $num_{[t_s,t_e]} \leftarrow T_E.query([t_s, \infty] \times [-\infty, t_e])$;

the total number of butterflies that comes from S is $\binom{|S|}{2}$. This idea leads us to a different index algorithm, named *Combination-based Index (`CBI`)*. The implementation details are provided in Algorithm 27 and Algorithm 28. Similarly, we assume no duplicate edge for brevity. Resolving them is deferred to Section 3.2.7.

• **Construction** In `EBI`, we use one 2D-counting data structure to store butterflies. Instead, in `CBI`, we map every type of wedge to a separate data structure. We initialize a hashmap T_C in the beginning (Line 1). We still need to enumerate every wedge $w_i : \langle x \rightsquigarrow y \rightsquigarrow z \rangle [l_i, r_i]$ in G (Line 2). A wedge with endpoints x, z is maintained by the 2D-counting data structure $T_C[(x, z)]$. If $T_C[(x, z)]$ does not exist(Line 3), we initialize an empty $\mathcal{CS}$ (Theorem 3.2.1) for $T_C[(x, z)]$ (Line 4). Unlike in `EBI`, where we compute every butterfly that is constructed from the new wedges w_i and a previous wedge $w_j \in W[(x, z)]$, we directly insert a point $[l_i, r_i]$ (w_i's active timestamp) into the corresponding $T_C[(x, z)]$ (Line 5). After the enumeration, we preprocess all data structures $T_C[(x, z)]$ and return T_C (Line 7).

• **Answering Query** To answer a historical butterfly counting of time-window $[t_s, t_e]$, for every type of wedges, we first need to compute the number of wedges that exist in the time-window, then compute the number of butterflies by a simple binomial coefficient. We set a counter $num_{[t_s,t_e]}$ to be 0 initially (Line 1). We enumerate every existing data structure in T_C (Line 2). Similar to what we do in `EBI`, we query on the corresponding 2D-counting data structure $T_C[(x, z)]$ with a query $[t_s, \infty] \times [-\infty, t_e]$ to get the number of wedges that are active in the time-window (Line 3). Denoting the query answer as *count*, we add $\binom{count}{2}$ to $num_{[t_s,t_e]}$, which is the number of butterflies that are constructed from these wedges. After enumerating all the data structures, we return $num_{[t_s,t_e]}$ as the total number of butterflies (Line 7).

• **Time and Space Complexity** The time and space complexity of `CBI` is summarized as follows:

Theorem 3.2.5. *Denote $\widetilde{w}$ as the number of wedge groups and δ as the arboricity. `CBI`'s construction (Algorithm 27) runs in $O(m\delta \log n)$ and takes $O(m\delta)$ space. After the construction, `CBI` answers each query (Algorithm 28) in $O(\widetilde{w} \log n)$ time.*

Proof. The construction process (Algorithm 27) also requires enumerating all wedges,

Algorithm 27: CBI-Construction

Input: a bipartite temporal graph: $G = (V, E)$;
Output: a hashmap mapping pairs of vertices to multiple $\mathcal{CS}$s: T_C;

```
T_C ← an empty hashmap mapping pairs of vertices to CSs;
for each wedge w_i : ⟨x ⇝ y ⇝ z⟩ [l_i, r_i] ∈ G do
    if (x, z) ∉ T_C.keys() then
        Initialize T_C[(x, z)] with an empty CS;
    T_C[(x, z)].insert(l_i, r_i);
Preprocess each data structure in T_C (Theorem 3.2.1);
return T_C;
```

which takes $O(m\delta)$ by [207]. For each group of wedges with the same endpoints, a $\mathcal{CS}$ needs to be initialized with all wedges from this group. Since the number of wedges is bounded by $O(m\delta)$ and each group contains at most $O(n)$ wedges, the total time for preprocessing all the data structures is $O(m\delta \log n)$ by Theorem 3.2.1. Similar to Theorem 3.2.4, it takes $O(m\delta)$ memory in total. When answering a query (Algorithm 28), all groups of wedges need to be enumerated, and their corresponding $\mathcal{CS}$ will be queried exactly once. The total query time will be $O(\widetilde{w} \log n) = \widetilde{w} \times O(\log n)$ by Theorem 3.2.1. Memory usage is dominated by 2D-range counting data structures for each group of wedges. □

As a result, `CBI` manages to resolve the memory issue of `EBI` when there are too many butterflies. Although $\widetilde{w}$ is bounded by the number of wedges (*i.e.,* $m\delta$), in practice, it is significantly smaller (*i.e.,* $\widetilde{w} \ll m\delta$). For example, in the `Wiktionary` dataset, there is about $m\delta \approx 1.3 \times 10^9$ wedges but only $\widetilde{w} \approx 5 \times 10^7$ wedge groups with different (x, z).

By further observing the distribution of wedges on real-world data, we see a concentration phenomenon that our algorithm has not addressed: Most wedges are grouped in a few groups, while the rest contain only a very small number of wedges. Therefore, building a separate data structure for every wedge group might be too expensive and unnecessary, especially for groups that only contribute a small amount of butterflies. For example, in the `Wiktionary (WT)` dataset, more than 81.3% wedge groups contain less than 10 wedges.

3.2.3.4 Graph Structure-aware Index

Recall that the number of butterflies dominates the memory usage of `EBI`, and the number of wedge groups greatly affects the query time of `CBI`. They fail to take into account the actual structure of the input graph. If we assume that most wedge groups only contain

Algorithm 28: CBI-Querying

Input: a time-window: $[t_s, t_e]$; a CB-index: T_C;
Output: temporal butterfly counting: $num_{[t_s,t_e]}$;

1 $num_{[t_s,t_e]} \leftarrow 0$;
2 **for** $(x, z) \in T_C.keys()$ **do**
3 $\quad count \leftarrow T_C[(x, z)].query([t_s, \infty] \times [-\infty, t_e])$;
4 $\quad num_{[t_s,t_e]} \leftarrow num_{[t_s,t_e]} + \binom{count}{2}$
5 **return** $num_{[t_s,t_e]}$;

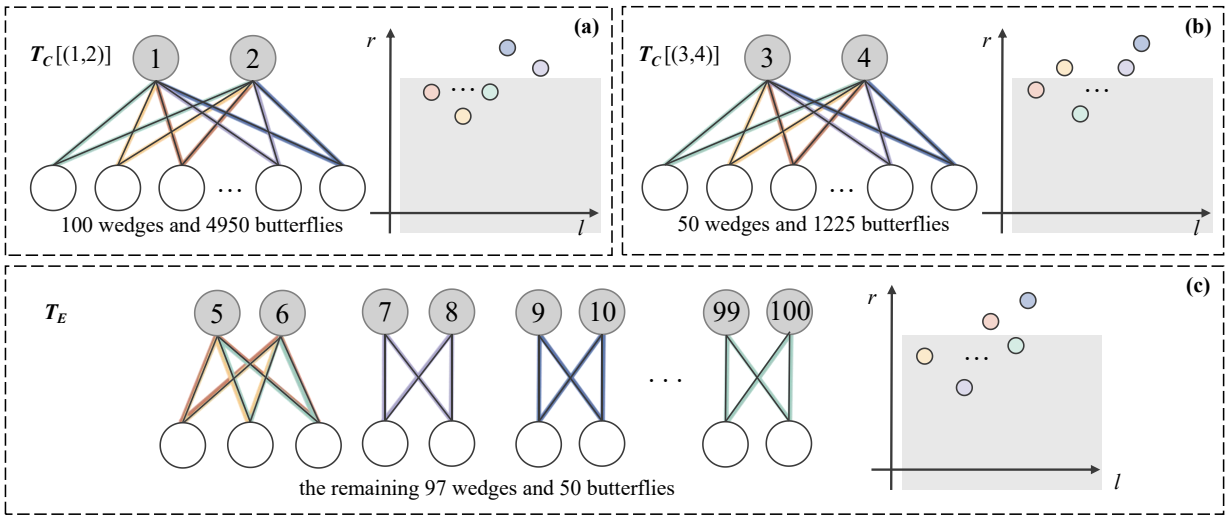


Figure 3.18: An illustrative example for demonstrating GSI.

a very small amount of wedges, we might be able to take advantage of both EBI and CBI by distributing each wedge group into one of EBI and CBI by its size. As the main result, we propose a new index algorithm named *Graph Structure-aware Index (GSI)* with such an idea. Besides its promising performance on real-world data, it can also be parallelized to reach an even better performance. The implementation details are provided in Algorithm 29 and Algorithm 30.

• **Construction** The important assumption we make here is that a small number of groups contain most of the wedges. Our key idea is to store them in the same way as CBI, building a 2D-range counting data structure for each index, maintained by a hashmap T_C. For the rest of the wedges, we precompute all butterflies constructed from them and store them in one data structure T_E like EBI. In the beginning, in addition to T_E (Line 1) and T_C (Line 2), we also initialize a hashmap W to group wedges directly (Line 3). We enumerate

all $w_i : \langle x \rightsquigarrow y \rightsquigarrow z \rangle [l_i, r_i]$ in G (Line 4) and store them into the corresponding set mapped by $W[(x, z)]$ (Line 6) . If the key does not exist in W, we need to initialize the set to be empty (Line 5). After grouping wedges, we sort the sets in W with decreasing order of sizes. We also need to compute the total number of butterflies and store it in num and $numB$ (Line 8 to Line 10).

We introduce a parameter α to control the cutoff between the two methods. More specifically, we build a data structure for some sets in W such that the total number of butterflies constructed from them does not exceed α fraction of all the butterflies in G. The butterflies constructed from the other sets in W are maintained by one data structure per set.

We enumerate the sets in W by decreasing the order of sizes (Line 11). num represents the number of butterflies we processed. If we have not processed the majority of butterflies, i.e., when num is no less than $\alpha \cdot numB$ (Line 12), we build a $\mathcal{CS}$ for every wedge in $W[(x, z)]$, inserting points representing their active timestamps (Line 13 to Line 15). num is subtracted by $\binom{|W[(x,z)]|}{2}$, indicating that these of butterflies are recorded by $T_C[(x, z)]$ (Line 21).

When num is smaller than $\alpha \cdot numB$, we have already handled the majority of butterflies. Now, we need to process butterflies built from the rest of the wedges groups. Though each of these groups only contains a relatively small amount of wedges, the number of groups might be large. Here we apply the idea of `EBI`. We enumerate every possible wedge pair in the current enumerated $W[(x, z)]$ (Line 17 to Line 18), compute the butterfly $b_{ij}[t_l, t_r] \leftarrow w_i \circ w_j$ (Line 19) and insert a point (t_l, t_r) into the global data structure T_E (Line 20). In the end, after preprocessing every $\mathcal{CS}$ in T_C and T_E, we return them and finish the construction (Line 23).

An example of construction is shown in Figure 3.18. In the input graph, there are 100 wedges with endpoints $(1, 2)$ and 50 wedges with endpoints $(3, 4)$, while there are only 97 wedges in other groups. We treat these two groups with the `CBI` method and build $T_C[(1, 2)]$ (a), $T_C[(3, 4)]$ (b) correspondingly. We compute all 50 butterflies for the remaining wedges and store them in T_E (c).

• **Answering Query** For a historical butterfly counting of time-window $[t_s, t_e]$, we need to accumulate answers from both T_C and T_E. As in `EBI`, we directly query the 2D range $[t_s, \infty] \times [-\infty, t_e]$ on T_E (Line 1). Then, as in `CBI`, we enumerate all groups of wedges from T_C (Line 2). We query $T_C[(x, z)]$ with the same 2D range (Line 3) and add up the number of butterflies constructed from each group of wedges (Line 4). We return the answer as the sum of these two parts (Line 5). In the Figure 3.18 showcase, we query from all three $\mathcal{CS}$s: $T_C[(1, 2)]$, $T_C[(3, 4)]$, T_E and sum up the answer.

Algorithm 29: GSI-Construction

Input: a bipartite temporal graph: $G = (V, E)$; a parameter for space using: α;
Output: a $\mathcal{CS}$ for indexing (EB-index): T_E; a hashmap of $\mathcal{CS}$ for indexing: T_C;

Initialize T_E with an empty $\mathcal{CS}$;
$T_C \leftarrow$ an empty hashmap mapping pairs of vertices to $\mathcal{CS}$s;
$W \leftarrow$ an empty hashmap mapping pairs of vertices to sets;
for *each wedge* $w_i : \langle x \rightsquigarrow y \rightsquigarrow z \rangle\, [l_i, r_i] \in G$ **do**
    if $(x, z) \notin W.keys()$ **then** $W[(x, z)] \leftarrow \emptyset$;
    $W[(x, z)] \leftarrow W[(x, z)] \cup w_i$;
Sort sets in W with decreasing order of sizes;
$num \leftarrow 0$;
for $(x, z) \in W.keys()$ **do** $num \leftarrow num + \binom{|W[(x,z)]|}{2}$;
$numB \leftarrow num$;
for $(x, z) \in W.keys()$ **do**
    if $num \geq \alpha \cdot numB$ **then**
        Initialize $T_C[(x, z)]$ with an empty $\mathcal{CS}$;
        for $w_i : [l_i, r_i] \in W[(x, z)]$ **do**
            $T_C[(x, z)].insert(l_i, r_i)$;
    else
        for $w_i : [l_i, r_i] \in W[(x, z)]$ **do**
            for $w_j : [l_j, r_j] \in W[(x, z)] \wedge (j > i)$ **do**
                $b_{ij} \leftarrow w_i \circ w_j$;
                $T_E.insert(t_l, t_r)$; // $(t_l, t_r) = \mathcal{T}(b_{ij})$
    $num \leftarrow num - \binom{|W[(x,z)]|}{2}$;
Preprocess T_E and each data structure in T_C (Theorem 3.2.1);
return T_E, T_C;

• **Time and Space Complexity** The time and space complexity of `GSI` is summarized as follows:

Theorem 3.2.6. *Denote β as the number of $\mathcal{CS}$ building for wedge groups of the input graph G and δ as the arboricity. `GSI`'s construction (Algorithm 29) runs in $O(m\delta \log n + \alpha B \log m)$ and takes $O(m\delta + \alpha B)$ space. After the construction, each query (Algorithm 30) takes $O(\beta \log n + \log m)$ to answer.*

Proof. The construction process (Algorithm 29) requires enumerating all wedges, which takes $O(m\delta)$ by [207]. The rest of the algorithm can be divided into two parts: a `CBI`

Algorithm 30: GSI-Querying

Input: a time-window: $[t_s, t_e]$; GSI-indexes: T_E, T_C;
Output: temporal butterfly counting: $num_{[t_s,t_e]}$;

```
1 num_[ts,te] ← T_E.query([ts, ∞] × [−∞, te]);
2 for (x, z) ∈ T_C.keys() do
3 |   count ← T_C[(x, z)].query([ts, ∞] × [−∞, te]);
4 |   num_[ts,te] ← num_[ts,te] + (count 2)
5 return num_[ts,te];
```

instance of β wedges group containing $(1-\alpha)B$ butterflies and an `EBI` instance containing αB butterflies. By Theorem 3.2.4 and Theorem 3.2.5, the total time complexity is $O(m\delta \log n + \alpha B \log m)$ and the corresponding query complexity (Algorithm 30) is $O(\beta \log n + \log m)$. The memory usage is $O(m\delta + \alpha B)$. □

Note that β is determined by α and bounded by the number of wedge groups $\widetilde{w}$. In real-world datasets with practical memory constraints, β is significantly smaller than $\widetilde{w}$ (*i.e.*, $\beta \ll \widetilde{w}$). For example, in the `WT` dataset and under a 500 GB memory constraint, the β of `GSI` is 6986 when setting the optimal α that maximize the memory utilization, while the $\widetilde{w}$ is $3,925,064$.

As a result, `GSI` manages to address the concentration phenomenon of the wedges distribution on real-world graphs, taking advantage of both `EBI` (for scattered wedges) and `CBI` (for concentrated wedges). It also has flexibility regarding the different limitations of memory. In addition, under the well-known power-law model [4], `GSI` has a non-trivial complexity, which is analyzed in Section 3.2.4.

Automatically Determining α The parameter α reflects a trade-off between query time and memory usage. Therefore, selecting a proper α based on the input graph and the memory limit is important. We introduce an easy method to automatically select an efficient α before executing `GSI`. In the beginning, we set $\alpha = 0$, which means that all wedge groups will be maintained separately with a $\mathcal{CS}$ data structure. We sort wedge groups by size from small to large and enumerate them one by one. For each enumerated wedge group i with w_i wedges, we increase α by $\binom{w_i}{2}/B$, where B is the number of butterflies, precomputed in Line 8 to Line 9. We can estimate the memory usage by simple calculation as Chazelle's structure's memory usage can be precomputed based on the number of inserted points in $O(1)$. We stop and determine α if the estimated memory exceeds the limit

with the next wedge group. This process takes $O(m\delta)$ time and avoids actually building indexes.

Parallelized Querying After the construction process, when answering a query (Algorithm 30), we are, in fact, summing up the answer from several independent $\mathcal{CS}$ data structures, which motivates us to utilize parallelization to speed up. Assume we have multiple threads, and these threads can handle different $\mathcal{CS}s$ simultaneously. Since no conflict occurs when these threads read the indexes simultaneously, we consider Algorithm 30 is highly parallelizable when the number of $\mathcal{CS}$s greater than the number of threads. To demonstrate, we implement and evaluate a parallel version of `GSI` in Section 3.2.5.1.

3.2.3.5 Index Compression

`EBI` has already reached the memory lower-bound in Theorem 3.2.3. Meanwhile, `GSI` provides a lower and adjustable memory usage in practice, with a better theoretical guarantee in the power-law graph. Both our algorithms are designed for exact solutions. However, in some applications of butterfly counting, such as graph kernel analysis [191] and network measurement [185, 205], using approximation counts is sufficient. In this section, we show that if we allow a small error ratio (e.g., $\leq 10^{-6}\%$) of the butterfly counting, we can reach a much lower memory usage on real-world graphs. Two index compression methods for approximate historical butterfly counting with strong theoretical guarantees are introduced.

Single-sided Compression (SGSI) Intuitively, most butterflies in Algorithm 30 come from T_C (Line 2 to Line 3) while the actual memory usage of T_E is much larger in general. This inspires us only to maintain a small portion (e.g., $\frac{1}{1000}$) of all T_E's butterflies. We denote λ_1 as the single-sided compression ratio, which indicates we only preserve $\lambda_1 \cdot \alpha B$ butterflies, where αB is the total number of butterflies in T_E. Each butterfly is preserved with probability λ_1 independently. In other words, we modify the `GSI` algorithm so that when `GSI` inserts a 2D point into T_E, the insertion is actually performed with probability λ_1. We denote the compressed T_E by $\widetilde{T_E}$. To answer queries, we return $\widetilde{T_E}.query([t_s, \infty] \times [-\infty, t_e])/\lambda_1$ (Line 1 in Algorithm 30). This results in a compressed memory usage of $O(m\delta \log n + \lambda_1 \cdot \alpha B \log m)$.

Double-sided Compression (DGSI) `SGSI` only compresses on T_E, which does not affect the query efficiency. If we further compress T_C, we can reach a faster query time and

even lower memory usage. Specifically, in DGSI, besides compressing T_E as in SGSI, we only maintain λ_2 portion of wedges for each group in T_C's $\mathcal{CS}$s. Each wedge is preserved with probability λ_2 independently. We denote the compressed T_C by $\widetilde{T_C}$. To answer queries, each $\widetilde{\mathcal{CS}}$ in $\widetilde{T_C}$ contributes $\binom{count}{2}/\lambda_2^2$ (Line 4 in Algorithm 30) to the total answer where *count* is the number of wedges in $\widetilde{\mathcal{CS}}$ in the projected graph. This results in an $O(\beta \log(\lambda_2 n) + \log(\lambda_1 m))$ query time, which is more efficient than our exact algorithm, GSI.

We now prove that both compression methods are unbiased and bound their errors by the following theorems.

Theorem 3.2.7 (Compressing T_E)**.** *For any historical butterfly counting query $[t_s, t_e]$ on a bipartite graph G, let B denote the correct number of butterflies in $G_{[t_s,t_e]}$ that is maintained by T_E, i.e., $B = T_E.query([t_s, \infty] \times [-\infty, t_e])$. Let B' denote the number of butterflies computed by the compressed T_E in SGSI with compression ratio λ_1, i.e., $B' = \widetilde{T_E}.query([t_s, \infty] \times [-\infty, t_e])/\lambda_1$. B' is unbiased, i.e., the expectation of B' is equal to B. The absolute error $|B - B'|$ is $O(B^{0.5} \log^{0.5}(n) \lambda_1^{-0.5})$ with high probability, where n is the number of vertices.*

Proof of Theorem 3.2.7. The approximation B' is unbiased because each butterfly from T_E that contributes to B is preserved with probability λ_1, and each of these preserved butterfly contributes $1/\lambda_1$ to B'.

B' is the sum of independent random variables taking values in $\{0, 1/\lambda_1\}$. By Chernoff bound, we have

$$\mathsf{Pr}\left[|B' - \mathbb{E}\left[B'\right]| \geq \delta \mathbb{E}\left[B'\right]\right] = O(\exp(-\delta^2 \lambda_1 \mathbb{E}\left[B'\right]/3)).$$

Since B' is unbiased, $\mathbb{E}\left[B'\right] = B$. Substituting $\delta = c \log^{0.5}(n) \lambda_1^{-0.5} B^{-0.5}$ for some large enough c into the equation above gives

$$\begin{aligned}
&\mathsf{Pr}\left[|B' - B| \geq \delta B\right] \\
=&O(\exp(-\delta^2 \lambda_1 B/3)) \\
=&O(\exp(-c^2 \log(n))) = O(n^{-c^2}).
\end{aligned}$$

□

Theorem 3.2.8 (Compressing T_C)**.** *For any historical butterfly counting query $[t_s, t_e]$ on a bipartite graph G, let S denote the correct number of butterflies in $G_{[t_s,t_e]}$ that is maintained by T_C, i.e., the total increment of $num_{[t_s,t_e]}$ in Line 4 of Algorithm 30. Let S' denote the number of butterflies computed by the compressed $\widetilde{T_C}$ in DGSI with compression ratio λ_2. S' is unbiased, i.e., the expectation of S' is equal to S. The absolute error $|S - S'|$ is $O(S^{0.75} \ln(n)^{0.75} \lambda_2^{-1.75})$ with high probability, where n is the number of vertices.*

Proof of Theorem 3.2.8. We consider any two different wedges w_1, w_2 from the same $\mathcal{CS} \in T_C$ such that both appears in the projected graph $G_{[t_s,t_e]}$ for the time-window $[t_s, t_e]$. Let $\widetilde{\mathcal{CS}}$ be the corresponding data structure of $\mathcal{CS}$ in $\widetilde{T_C}$. Let C' be the number of wedges in $G_{[t_s,t_e]}$ that are inserted into $\widetilde{\mathcal{CS}}$. Let C be the total number of wedges in $G_{[t_s,t_e]}$ that are inserted into $\mathcal{CS}$. The pair of wedges, (w_1, w_2), contributes $1/\lambda_2^2$ to the answer $\binom{C'}{2}/\lambda_2^2$ if both of them are preserved $\widetilde{\mathcal{CS}}$. The expected contribution is exactly 1 because w_1 and w_2 are preserved independently with probability λ_2 each. Thus, by the linearity of expectation, $\mathbb{E}\left[\binom{C'}{2}/\lambda_2^2\right]$ is exactly $\binom{C}{2}$, i.e., the number of pairs (w_1, w_2). Summing over all $\mathcal{CS} \in T_C$ gives the unbiasedness.

Let δ be an undetermined coefficient. By Chernoff bound,

$$\begin{aligned}&\mathsf{Pr}\left[|C' - \lambda_2 C| \geq \delta C\right]\\ \leq& O(\exp(-\Omega(-\delta^2 \lambda_2 C))).\end{aligned}$$

Let $\delta = \log^{0.5}(n)\lambda_2^{-0.5}C^{-0.5}$. We have that with high probability,

$$|C' - \lambda_2 C| = O(C^{0.5}\log^{0.5}(n)\lambda_2^{-0.5}).$$

This means that $\binom{C'}{2}/\lambda_2^2$ is within multiplicative error $O(C^{-0.5}\lambda_2^{-1.5}\log^{0.5}(n))$ of the correct answer $\binom{C}{2}$.

Conditioning on the error bound above hold for every $\mathcal{CS}$ in T_C, we may use Hoeffding's inequality. For each $\mathcal{CS}_i \in T_C$ such that $|\mathcal{CS}_i| \geq 2$, $\binom{C'_i}{2}/\lambda_2^2$ is an independent random variable taking values from

$$\binom{C_i}{2} \pm O(C^{1.5}\lambda_2^{-3.5}\log^{0.5}(n)),$$

where C_i and C'_i are the number of wedges in $\mathcal{CS}_i \cap G_{[t_s,t_e]}$ and the number of preserved wedges in $\mathcal{CS}_i \cap G_{[t_s,t_e]}$. Let $S' = \sum_i \binom{C'_i}{2}/\lambda_2^2$, and let $S = \sum_i \binom{C_i}{2}$. We have, for any parameter t,

$$\begin{aligned}&\mathsf{Pr}\left[|S' - S| \geq t\right]\\ \leq& O\left(\exp\left(-\Omega\left(t^2/\left(\lambda_2^{-3.5}\log^{0.5}(n)\sum_i C_i^3\right)\right)\right)\right)\end{aligned}$$

by Hoeffding's inequality. We may set $t = c\lambda_2^{-1.75}\log^{0.75}(n)S^{0.75}$ for large enough constant c to ensure that $|S' - S| < t$ with high probability. □

Theorem 3.2.7 directly bounds the error of `SGSI`. To bound the error of `DGSI`, we may add the error bounds for compressing the T_E part (Theorem 3.2.7) and compressing the T_C part (Theorem 3.2.8).

3.2.4 Analysis on Power-law Graphs

• **Power-law Bipartite Graphs** Previous research [98, 203] shows that many real-world bipartite graphs follow the *power-law* distribution with γ generally between 2 to 3, similar to the *scale-free* graphs model for unipartite graphs. By leveraging the properties of the power-law degree distribution, there are many studies on analyzing algorithms based on such settings [228, 224, 131, 40]. In this section, we prove that `GSI` runs in $\widetilde{O}(\lambda)$ time and $O(m^{2-\delta}/\lambda^2)$ memory for some $\delta > 0$ with high probability on power-law bipartite graphs [4] with $\gamma \in (2, 3)$, in contrast to the hardness result in Section 3.2.3.1 for the general case. We use the following model for G.

Definition 3.2.7. *Let n_1 be the number of vertices in U. Let n_2 be the number of vertices in L. Let γ_i, Δ_i $(i = 1, 2)$ be the exponents and max degrees of two power-law distributions. A power-law bipartite graph is obtained by the following process:* ***(1)*** *For each vertex x in U, sample $d_x \in [1, \Delta_1]$ such that $\mathsf{Pr}[d_x = k] \propto k^{-\gamma_1}$.* ***(2)*** *For each vertex y in L, sample $d_y \in [1, \Delta_2]$ such that $\mathsf{Pr}[d_y = k] \propto k^{-\gamma_2}$.* ***(3)*** *Let $m = \mathbb{E}\left[\sum_{x\in U} d_x\right] = \mathbb{E}\left[\sum_{y\in L} d_y\right]$ be the expected number of edges in G.* ***(4)*** *For each pair of vertices $x \in U$, $y \in L$, sample the existence of the edge (x, y) such that $\mathsf{Pr}[(x, y) exists] = \frac{d_x d_y}{m}$.*

For Definition 3.2.7 to be well-defined, we need to choose parameters such that

$$\mathbb{E}\left[\sum_{x\in U} d_x\right] = \mathbb{E}\left[\sum_{y\in L} d_y\right]$$

, i.e., the expected sums of degrees for vertices in U and L are equal. We note that the sampled d_x and d_y values are intermediate variables of the sampling process. They are not necessarily the same as the degrees $\deg_x$ and $\deg_y$ of vertex x and y. We can interpret d_x as the expected degree of x. Based on Definition 3.2.7, there are two following two types of power-law bipartite graphs to model the real-world networks comprehensively.

Definition 3.2.8 (Double-sided Power-law Bipartite Graph)**.** *A double-sided power-law bipartite graph is a power-law bipartite graph (Definition 3.2.7) satisfying $\gamma_1 \in (2, 3)$, and $\gamma_2 \in (2, 3)$.*

Definition 3.2.9 (Single-sided Power-law Bipartite Graph)**.** *A single-sided power-law bipartite graph is a power-law bipartite graph (Definition 3.2.7) satisfying* $\gamma_1 \in (2,3)$, $\gamma_2 = 0$, *and* $\Delta_1 > \Delta_2$.

Twitter exemplifies a single-sided power-law bipartite graph, where the distribution of followers per user varies significantly, often displaying a vast disparity. In contrast, the number of accounts each user follows tends to be more uniform and comparatively narrow in range. In contrast, movie-actor networks exhibit a double-sided power-law distribution. Most actors appear in just a few films, and the majority of films feature only a small number of actors. However, a select group of highly connected actors mirrors the highly-followed users on social media platforms, while films with large casts resemble "super-connected" nodes.

• **Time and Space Complexity** We calculate the expected time and space of `GSI` for each type of power-law bipartite graph.

Theorem 3.2.9 (`GSI` on double-sided power-law bipartite graphs)**.** *Let* G *be a single-sided power-law bipartite graph. We can set* α *in Algorithm 29, such that the expected space usage of* ***GSI*** *is* $O(\Delta^{6-2\gamma})$*, and that the query time is* $\widetilde{O}(1)$*. We can also set* α *such that the expected space usage is* $O\left(n\Delta^{3-\gamma}\right)$*, and that the expected query time is* $\widetilde{O}(n^2 \min(\Delta_1, \Delta_2)^{2-2\gamma})$.

We note that for both settings of Theorem 3.2.9, the time and space complexities surpass the lower bound in Theorem 3.2.3. This shows that the historical butterfly counting problem on power-law graphs are not as hard as the general case, and that `GSI` successfully exploits the features of the power-law graphs to reduce the computation cost. The first setting has $\widetilde{O}(1)$ time complexity and $o(n^2)$ space complexity since $\Delta \leq n$ and $\gamma > 2$. The second setting has $\widetilde{O}(\lambda) = \widetilde{O}(n^2 \min(\Delta_1, \Delta_2)^{2-2\gamma})$ time complexity and $o(n^2/\lambda^2)$ space complexity when $k > n^{\frac{3}{4\gamma-4}} \Delta^{\frac{3-\gamma}{4\gamma-4}}$.

Theorem 3.2.10 (`GSI` on single-sided power-law bipartite graphs)**.** *Let* G *be a single-sided power-law bipartite graph.*

We can set α *in Algorithm 29, such that the expected space usage of* ***GSI*** *is* $O\left(\Delta_2 + \left(\frac{\Delta_1^{6-\gamma_1}}{\Delta_2^5}\right)\right)$ *and that the expected query time is* $\widetilde{O}\left(1\right)$.

Similar to the double-sided case, Theorem 3.2.10 solves the historical butterfly counting problem with a better trade-off than Theorem 3.2.3.

The complete proofs of Theorem 3.2.9 and Theorem 3.2.10 are shown below.

Proof for Theorem 3.2.9 and Theorem 3.2.10. By Theorem 3.2.6, we know that the query time for `GSI` is nearly linear in the number of keys (x, z) in T_C. The space usage for `GSI` is bounded by the number of butterflies not maintained by $T_C[(x, z)]$, plus the total number of wedges in each $W[(x, z)]$ for $(x, z) \in T_C.keys()$.

To bound the query time and space usage, let k be a parameter between 1 and $\Delta = \max(\Delta_1, \Delta_2)$. Let $P_{\geq k}$ be the set of unordered pairs (x, z) such that x, z are on the same side of the bipartite graph G, and that $d_x, d_z \geq k$. Let $\# \bowtie_{\geq k}$ be the number of butterflies B such that $\exists\{x, z\} \in P_{\geq k}$, $\{x, z\} \subseteq B$. Here we abuse notation and use B to mean the vertices of a butterfly B.

In Line 11 of Algorithm 29, we construct $T_C[(x, z)]$s for pairs (x, z) with the largest $W[(x, z)]$s until the total number of butterflies in these largest $W[(x, z)]$s exceeds $(1 - \alpha)numB$. The rest of the butterflies will be maintained by T_E.

Intuitively, for proving the efficiency of `GSI`, we would like to show that a small number of $\{x, z\}$ (those in $P_{\geq k}$) covers a large fraction ($\# \bowtie_{\geq k} /numB$) of the total number of butterflies. Formally, we can prove that for some k, `GSI` has query complexity $\widetilde{O}(|P_{\geq k}|)$ and expected space complexity.

$O\left(\# \bowtie_{\geq 1} - \# \bowtie_{\geq k} + \sum_{(x,z)\in P_{\geq k}} |T_C[(x, z)]|\right)$. To show this, let's consider an algorithm similar to Algorithm 29. The modified algorithm changes the condition on Line 12 from "$num \geq \alpha \cdot numB$" to "$\{x, z\} \in P_{\geq k}$". Then T_C contains $|P_{\geq k}|$ elements (one for each $(x, z) \in P_{\geq k}$), and T_E contains $\# \bowtie_{\geq 1} - \# \bowtie_{\geq k}$ butterflies. The time complexity of the modified algorithm is $\widetilde{O}(|P_{\geq k}|)$ and that the expected space usage of it is

$$O\left(\# \bowtie_{\geq 1} - \# \bowtie_{\geq k} + \sum_{(x,z)\in P_{\geq k}} |T_C[(x, z)]|\right).$$

In the original `GSI` (Algorithm 29), for a fixed choice of k, we can choose α properly such that the condition on Line 12 evaluates to true for the first $|P_{\geq k}|$ iterations, i.e., after the first $|P_{\geq k}|$ iterations of the loop, num is no more than $\alpha \cdot numB$. The query time of `GSI` is bounded by $\widetilde{O}(|P_{\geq k}|)$. The space usage of `GSI` is bounded above by that of the modified algorithm, because the sets in W are sorted with decreasing order of sizes. Each set $W[(x, z)]$ costs $|W[(x, z)]|$ space if it is maintained in T_C and $\binom{|W[(x,z)]|}{2}$ space if it is maintained in T_E. `GSI` costs less space because it maintains larger sets in T_C, compared to the modified algorithm. These will automatically translates to the same bounds for `GSI`.

For simplicity, we use V_1, V_2 to denote U, L. We first calculate the expected number of edges, m, of G. m can be calculated by Δ_i, γ_i, n_i for either $i = 1$ or $i = 2$. For the model to be consistent, we require that the m's calculated by $i = 1$ and $i = 2$ are equal.

Lemma 3.2.1. *For any $i \in \{1,2\}$, if $\gamma_i \in (2,3)$,*

$$m = \left(1 + O\left(\Delta_i^{2-\gamma_i}\right)\right) \frac{n_i}{s_i(\gamma_i - 2)}.$$

Proof. Let x be a fixed vertex in V_i.

$$\begin{aligned} m =& n_i \mathbb{E}\left[\deg_x\right] && \text{(by linearity of expectation)} \\ =& n_i \mathbb{E}\left[d_x\right] \\ =& n_i \frac{1}{s_i} \sum_{i=1}^{\Delta_i} i^{1-\gamma_i} \\ =& \left(1 + O\left(\Delta_i^{2-\gamma_i}\right)\right) \frac{n_i}{s_i(\gamma_i - 2)}. \end{aligned}$$

□

Next, we estimate the expectation of $|P_{\geq k}|$ for bounding the query time.

Lemma 3.2.2. $\mathbb{E}\left[|P_{\geq k}|\right] = \left(1 + O\left(\frac{1}{n_1} + \frac{1}{n_2}\right)\right)\left(\frac{n_1^2 s_{1,k}^2}{2s_1^2} + \frac{n_2^2 s_{2,k}^2}{2s_2^2}\right).$

Proof. Recall that for any pair of vertices x, z, $(x,z) \in P_{\geq k}$ if $x, z \in V_i$ and $d_x, d_z \geq k$. The expected number of such pairs for a fixed $i \in \{1,2\}$ is

$$\begin{aligned} & \binom{n_i}{2} \mathsf{Pr}\left[d_x \geq k\right] \mathsf{Pr}\left[d_z \geq k\right] \\ =& \binom{n_i}{2} \left(\frac{\sum_{d=k}^{\Delta_d} d^{-\gamma_i}}{s_i}\right)^2 \\ =& \left(1 + O\left(\frac{1}{n_i}\right)\right) \frac{n_i^2 s_{i,k}^2}{2s_i^2}. \end{aligned}$$

Summing over $i = \{1,2\}$ gives

$$\mathbb{E}\left[|P_{\geq k}|\right] = \left(1 + O\left(\frac{1}{n_1} + \frac{1}{n_2}\right)\right)\left(\frac{n_1^2 s_{1,k}^2}{2s_1^2} + \frac{n_2^2 s_{2,k}^2}{2s_2^2}\right).$$

□

Lastly, we calculate $\mathbb{E}\left[\# \bowtie_{\geq k}\right]$, and $\mathbb{E}\left[\# \bowtie_{\geq 1}\right]$. The difference of these two numbers will bound the space usage.

Lemma 3.2.3.

$$\begin{aligned}
&\mathbb{E}\left[\# \bowtie_{\geq k}\right] \\
&\geq \left(1 + O\left(\Delta_1^{\gamma_1 - 3} + \Delta_2^{\gamma_2 - 3}\right)\right) \frac{n_1^2 n_2^2}{4m^4} \\
&\quad \frac{\left(\Delta_1^{3-\gamma_1} - k^{3-\gamma_1}\right)\left(\Delta_2^{3-\gamma_2} - k^{3-\gamma_2}\right)}{s_1 s_2 (3-\gamma_1)(3-\gamma_2)}.
\end{aligned}$$

Proof. To get a lower bound for $\mathbb{E}\left[\# \bowtie_{\geq k}\right]$, we use the quantity $\# \bowtie_{both \geq k}$ be the number of butterflies $B = (x, y, z, w)$ such that both (x, z) and (y, w) are from $P_{\geq k}$.

$\# \bowtie_{both \geq k}$ is no more than $\# \bowtie_{\geq k}$ which is the number of butterflies such that at least one of (x, z) and (y, w) are from $P_{\geq k}$.

We sum up the probability that x, y, z, w form a butterfly for $x, z \in V_1, y, w \in V_2$.

$$
\begin{aligned}
&\mathbb{E}\left[\# \bowtie_{both \geq k}\right] \\
=&\binom{n_1}{2}\binom{n_2}{2} \sum_{d_1,d_3\in[k,\Delta_1],d_2,d_4\in[k,\Delta_2]} \mathsf{Pr}\left[d_x = d_1\right]\mathsf{Pr}\left[d_y = d_2\right] \\
&\mathsf{Pr}\left[d_z = d_3\right]\mathsf{Pr}\left[d_w = d_4\right]\frac{d_x d_y}{m}\frac{d_x d_w}{m}\frac{d_z d_y}{m}\frac{d_z d_w}{m} \\
=&\binom{n_1}{2}\binom{n_2}{2} \sum_{d_1,d_3\in[k,\Delta_1],d_2,d_4\in[k,\Delta_2]} \mathsf{Pr}\left[d_x = d_1\right]\mathsf{Pr}\left[d_y = d_2\right] \\
&\mathsf{Pr}\left[d_z = d_3\right]\mathsf{Pr}\left[d_w = d_4\right]\frac{d_x^2 d_y^2 d_z^2 d_w^2}{m^4} \\
=&\binom{n_1}{2}\binom{n_2}{2}\frac{1}{m^4}\left(\sum_{d_1,d_3\in[k,\Delta_1]} \mathsf{Pr}\left[d_x = d_1\right]\mathsf{Pr}\left[d_z = d_3\right] d_x^2 d_z^2\right) \\
&\left(\sum_{d_2,d_4\in[k,\Delta_2]} \mathsf{Pr}\left[d_y = d_2\right]\mathsf{Pr}\left[d_w = d_4\right] d_y^2 d_w^2\right) \\
=&\binom{n_1}{2}\binom{n_2}{2}\frac{1}{m^4}\left(\sum_{d_1\in[k,\Delta_1]} \mathsf{Pr}\left[d_x = d_1\right] d_x^2\right)^2 \\
&\left(\sum_{d_2\in[k,\Delta_2]} \mathsf{Pr}\left[d_y = d_2\right] d_y^2\right)^2 \\
=&\binom{n_1}{2}\binom{n_2}{2}\frac{1}{m^4}\mathbb{E}^2\left[\mathbf{1}_{d_{v_1}\geq k} d_{v_1}^2\right]\mathbb{E}^2\left[\mathbf{1}_{d_{v_2}\geq k} d_{v_2}^2\right]
\end{aligned}
$$

where v_1 (v_2) is an arbitrary vertex from V_1 (V_2). We next calculate the expectations in

the equation above. For any $i \in \{1, 2\}$,

$$\begin{aligned}
&\mathbb{E}\left[\mathbf{1}_{d_{v_i} \geq k} d_{v_i}^2\right] \\
=&\sum_{d=k}^{\Delta_i} \mathsf{Pr}\left[d_{v_i} = d\right] d^2 \\
=&\frac{1}{s_i} \sum_{d=k}^{\Delta} d^{2-\gamma_i} \\
=&\left(1 + O\left(\Delta_i^{\gamma_i - 3}\right)\right) \frac{\Delta_i^{3-\gamma_i} - k^{3-\gamma_i}}{s_i(3-\gamma_i)}.
\end{aligned}$$

Thus, we have

$$\begin{aligned}
&\mathbb{E}\left[\# \bowtie_{both \geq k}\right] \\
=&\binom{n_1}{2}\binom{n_2}{2}\frac{1}{m^4}\mathbb{E}^2\left[\mathbf{1}_{d_{v_1} \geq k} d_{v_1}^2\right]\mathbb{E}^2\left[\mathbf{1}_{d_{v_2} \geq k} d_{v_2}^2\right] \\
=&\binom{n_1}{2}\binom{n_2}{2}\frac{1}{m^4}\Pi_{i=1}^2\left(1 + O\left(\Delta_i^{\gamma_i - 3}\right)\right)\frac{\Delta_i^{3-\gamma_i} - k^{3-\gamma_i}}{s_i(3-\gamma_i)} \\
=&\left(1 + O\left(\Delta_1^{\gamma_1 - 3} + \Delta_2^{\gamma_2 - 3}\right)\right)\frac{n_1^2 n_2^2}{4m^4} \\
&\frac{\left(\Delta_1^{3-\gamma_1} - k^{3-\gamma_1}\right)\left(\Delta_2^{3-\gamma_2} - k^{3-\gamma_2}\right)}{s_1 s_2 (3-\gamma_1)(3-\gamma_2)}.
\end{aligned}$$

□

Similar to Lemma 3.2.3, we can estimate the expectation of $numB = \# \bowtie_{\geq k}$.

Lemma 3.2.4.

$$\begin{aligned}
&\mathbb{E}\left[\# \bowtie_{\geq 1}\right] \\
=&\left(1 + O\left(\Delta_1^{\gamma_1 - 3} + \Delta_2^{\gamma_2 - 3}\right)\right)\frac{n_1^2 n_2^2}{4m^4} \\
&\frac{\Delta_1^{3-\gamma_1}\Delta_2^{3-\gamma_2}}{s_1 s_2 (3-\gamma_1)(3-\gamma_2)}.
\end{aligned}$$

Proof. The proof is identical to that of Lemma 3.2.3 with $k = 1$. We may replace the $\geq$ to $=$ because when $k = 1$, $\# \bowtie_{both \geq k}$ is equal to $\# \bowtie_{\geq k}$. □

Double-sided power-law bipartite graphs In the double-sided power-law model, both γ_1 and γ_2 are in the range $(2, 3)$. In this case, we have that s_i is a constant for $i = 1, 2$. We also have that $m = O(n)$.

We define $n = max(n_1, n_2)$, $\Delta = max(\Delta_1, \Delta_2)$, and $\gamma = min(\gamma_1, \gamma_2)$.

We may calculate

$\mathbb{E}\left[\# \bowtie_{\geq 1} - \# \bowtie_{\geq k}\right]$ by Lemma 3.2.4 and Lemma 3.2.3.

$$\begin{aligned}
&\mathbb{E}\left[\# \bowtie_{\geq 1} - \# \bowtie_{\geq k}\right] \\
=&O\left(\left(\Delta_1^{\gamma_1-3} + \Delta_2^{\gamma_2-3}\right) \frac{n_1^2 n_2^2}{m^4} \left(k^{3-\gamma_1}\Delta_2^{3-\gamma_2} + k^{3-\gamma_2}\Delta_1^{3-\gamma_1}\right)\right) \\
=&O\left(\frac{n_1^2 n_2^2}{m^4}\left(\left(\frac{k}{\Delta_1}\right)^{3-\gamma_1}\Delta_2^{3-\gamma_2} + \left(\frac{k}{\Delta_2}\right)^{3-\gamma_2}\Delta_1^{3-\gamma_1} + k^{3-\gamma_1} + k^{3-\gamma_2}\right)\right) \\
=&O\left(\Delta^{6-2\gamma}\right)
\end{aligned}$$

for $k \leq \Delta$. When k is no more than the smaller of Δ_1 and Δ_2, we have a sharper bound.

$$\begin{aligned}
&\mathbb{E}\left[\# \bowtie_{\geq 1} - \# \bowtie_{\geq k}\right] \\
=&O\left(\frac{n_1^2 n_2^2}{m^4}\left(\Delta_2^{3-\gamma_2} + \Delta_1^{3-\gamma_1} + k^{3-\gamma_1} + k^{3-\gamma_2}\right)\right) && (k \leq \Delta_1, k \leq \Delta_2) \\
=&O\left(\frac{n^4}{m^4}\Delta^{3-\gamma}\right) \\
=&O\left(\Delta^{3-\gamma}\right)
\end{aligned}$$

for $k \leq \min(\Delta_1, \Delta_2)$. By Lemma 3.2.2,

$$\begin{aligned}
&\mathbb{E}\left[|P_{\geq k}|\right] \\
=&O\left(n^2 k^{2-2\gamma}\right).
\end{aligned}$$

Note that the two bounds above on space usage do not depend on k. Thus, we may choose the largest possible k to reduce the query time in each case.

- We may choose $k = \Delta$ so that all butterflies are maintained by T_E. The query time is $\widetilde{O}(1)$ and the expected space usage is $\mathbb{E}\left[\# \bowtie_{\geq 1}\right] = O(\Delta^{6-2\gamma})$.

- We may also choose $k = \min(\Delta_1, \Delta_2)$ so that the expected query time is $\widetilde{O}(\mathbb{E}\left[|P_{\geq k}|\right]) = \widetilde{O}(n^2 \min(\Delta_1, \Delta_2)^{2-2\gamma})$ and the expected space usage for T_E is $O(\Delta^{3-\gamma})$. Note that we need to consider the space usage of T_C. This can be bounded by

$$\begin{aligned}
&\sum_{i=1}^{2} n_i \mathbb{E}\left[\deg_{x_i}^2\right] \\
=&\sum_{i=1}^{2} n_i \frac{\sum_{d=1}^{\Delta_i} d^{2-\gamma_i}}{s_i} \\
=&O\left(\sum_{i=1}^{2} n_i \Delta_i^{3-\gamma_i}\right) \\
=&O\left(n\Delta^{3-\gamma}\right)
\end{aligned}$$

  where $x_i \in V_i$ for $i = 1, 2$. The total expected space usage is $O\left(n\Delta^{3-\gamma}\right)$.

Single-sided power-law bipartite graphs In the single-sided power-law model, we have $\gamma_1 \in (2, 3)$, $\gamma_2 = 0$, and $\Delta_1 > \Delta_2$. In this case, s_1 is a constant and $s_2 = \Delta_2$. We also have that $m = \Theta(n_1) = \Theta(n_2\Delta_2)$.

We define $n = max(n_1, n_2)$, $\Delta = max(\Delta_1, \Delta_2)$.We choose $\Delta_2 < k \leq \Delta_1$. We may calculate $\mathbb{E}\left[\# \bowtie_{\geq 1} - \# \bowtie_{\geq k}\right]$ by Lemma 3.2.4 and Lemma 3.2.3.

$$\begin{aligned}
&\mathbb{E}\left[\# \bowtie_{\geq 1} - \# \bowtie_{\geq k}\right] && (3.1)\\
=&O\left(\left(\Delta_1^{\gamma_1-3} + \Delta_2^{\gamma_2-3}\right) \frac{n_1^2 n_2^2}{m^4} \left(k^{3-\gamma_1}\Delta_2^{3-\gamma_2} + k^{3-\gamma_2}\Delta_1^{3-\gamma_1}\right)\right) && (3.2)\\
=&O\left(\frac{n_1^2 n_2^2}{m^4}\left(\left(\frac{k}{\Delta_1}\right)^{3-\gamma_1}\Delta_2^{3-\gamma_2} + \left(\frac{k}{\Delta_2}\right)^{3-\gamma_2}\Delta_1^{3-\gamma_1} + k^{3-\gamma_1} + k^{3-\gamma_2}\right)\right) && (3.3)\\
=&O\left(\frac{n_2^2}{m^2}\left(\frac{\Delta_1^{6-\gamma_1}}{\Delta_2^3} + \Delta_2^3\right)\right) && (k \leq \Delta_1, \Delta_1 > \Delta_2)\\
=&O\left(\Delta_2 + \left(\frac{\Delta_1^{6-\gamma_1}}{\Delta_2^5}\right)\right). && \left(\tfrac{n_2}{m} = \Theta\left(\tfrac{1}{\Delta_2}\right)\right)
\end{aligned}$$

Note that the bound above does not depend on k. Thus, it is an upper bound of $\mathbb{E}\left[\# \bowtie_{\geq 1}\right]$. We may set $\alpha \geq 1$ so that `GSI` maintain all butterflies in T_E. This results in an expected query time of $\widetilde{O}(1)$. □

Table 3.3: The summary of datasets, including the size of vertices (*i.e.,* $|U|$ and $|L|$) and edges (*i.e.,* $|E|$), butterfly counts (*i.e.,* B), and the γ value of power-law exponent assuming the graph's degree follows a power-law distribution (*i.e.,* $\hat{\gamma}$). For synthetic datasets, we provide the γ value for each side (*i.e.,* γ_1 and γ_2).

Real-world dataset	$\|U\|$	$\|L\|$	$\|E\|$	B	$\hat{\gamma}$
Wikiquote (WQ)	961	640,482	776,458	5e9	4.281
Wikinews (WN)	2,200	35,979	907,499	3e11	2.654
StackOverflow (SO)	545,196	96,680	1,301,942	1e7	2.797
CiteULike (CU)	153,277	731,769	2,411,819	6e8	2.325
Bibsonomy (BS)	204,673	767,447	2,555,080	8e8	2.209
Twitter (TW)	175,214	530,418	4,664,605	1e12	2.460
Amazon (AM)	2,146,057	1,230,915	5,838,041	3e7	2.957
Edit-ru (ER)	7,816	1,266,349	8,349,235	1e13	5.301
Edit-vi (EV)	72,931	3,512,721	25,286,492	3e14	2.017
Wiktionary (WT)	66,140	5,826,113	44,788,448	1e16	1.826
Synthetic dataset	$\|U\|$	$\|L\|$	$\|E\|$	γ_1	γ_2
DoublePower-law (DPW)	25,599,878	25,599,878	100,000,000	2.1	2.1
SinglePower-law (SPW)	21,020,985	199,802	100,000,000	2.1	0

3.2.5 Experiments

• **Datasets** We use 10 large-scale real-world datasets to evaluate our algorithm. All real-world dataset sources and more detailed statistics are available at KONECT[4]. In addition, we randomly generate 2 power-law bipartite graphs with billion edges according to the models in Section 3.2.4, where their timestamps are also uniformly sampled from $[1, |E|]$. The dataset statistics are presented in Table 3.3.

• **Algorithms** Our algorithms are implemented in C++ with STL used. The implementation of `BFC-VP++` is obtained from their authors. As `WPS` is an approximation method, the reported running time for `WPS` represents the time required first to reach a relative error of 10%.

• **Hyperparameter Settings** The only hyperparameter for our methods is α. In our study of space-query trade-offs (Section 3.2.5.3), we determine the appropriate α to constrain memory usage. We employ the same algorithm for all other experiments to optimize α for maximal memory utilization.

All experiments are conducted on a Ubuntu 22.04 LTS workstation, equipped with an Intel(R) Xeon(R) Gold 6338R 2.0GHz CPU and 512GB of memory.

[4]from http://konect.cc/, where the $\hat{\gamma}$ of `ER` only considers the tail degrees (i.e., γ_t) because the global value missed in the website.

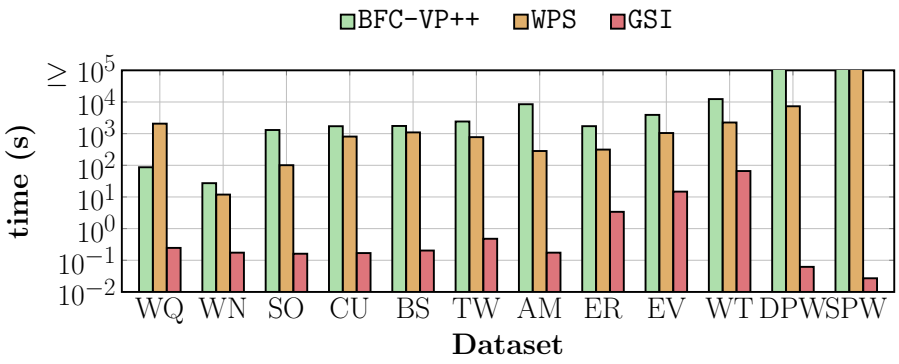


Figure 3.19: Time cost of 5,000 random queries for historical butterfly counting on datasets, where all algorithms use sequential versions.

3.2.5.1 Efficiency

Query Processing Shown in Figure 3.19, the efficiency of `GSI`, as well as the two baseline algorithms `BFC-VP++`, `WPS`, is compared on various datasets, with 5000 random queries.

On real-world datasets, `GSI` demonstrates a significant speedup ranging from 100× to 10000× comparing to `BFC-VP++`, 100× to 1000× comparing to `WPS`. In most datasets, `GSI` needs less than 1 seconds to answer all 5000 queries online, even when the dataset has around 4×10^6 vertices (`AM`). In general, the query time is proportional to the number of edges. For the largest dataset `WT`, which has around 6×10^6 vertices and 4.5×10^8 edges, all queries can be answered within 10^2 seconds, which means less than 0.1 seconds for each query in average. This shows the consistency and efficiency of our algorithms under very large real-world datasets.

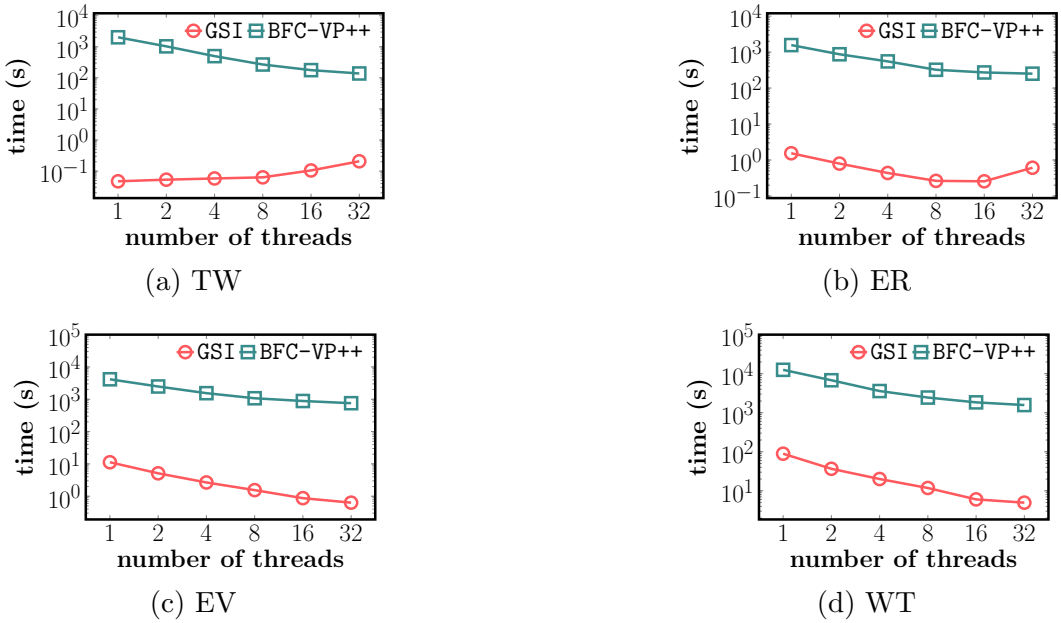
On the synthetic dataset, we generated, our algorithm shows an extremely large speedup greater than 100000×. Especially on `SPW`, while both `BFC-VP++` and `WPS` cannot process all queries within 10^5 seconds, our algorithms produce answers within 10^{-1} seconds.

Index Construction The reported time for index construction is reported in Table 3.4. For most datasets, even considering the time of index construction, `GSI`'s performance outperforms the two baselines, indicating that the index construction performance of `GSI` is both reasonable and practical.

Empirical Study on Power-law Graphs In this section, we further analyze the performance of `GSI` and baselines on the power-law graphs. In addition to two synthetic

Table 3.4: The index construction time for GSI in Figure 3.19.

Dataset	WQ	WN	SO	CU	BS	TW
time (s)	74	6	9	129	207	217
Dataset	AM	ER	EV	WT	DPW	SPW
time (s)	39	3,654	3,842	3,964	2,566	341



(a) TW (b) ER

(c) EV (d) WT

Figure 3.20: Evaluating the parallelization: time cost for 5,000 random queries against the number of threads.

graphs in Table 3.3, we generate power-law graphs with various γ_1 and γ_2 between $(2, 3)$ and uniformly sampled timestamps. Specifically, to ensure that baselines can finish in practical time (*i.e.,* less than $\leq 10^5$ seconds), we reduce the number of edges in these graphs to 10^7 instead of 10^8. in DPW and SPW. Shown in Figure 3.21, both BFC-VP++ and WPS require at least 10^3 seconds in all synthetic data we generate while our algorithm can process all 5,000 queries within 10^{-1} seconds. This gap is even more significant compared to results on real-world data. Combined with the theoretical analysis in Section 3.2.4, GSI has outstanding performance both theoretically and practically on Power-law graphs. This further reveals the advantage that GSI admits and makes full use of additional properties on graphs.

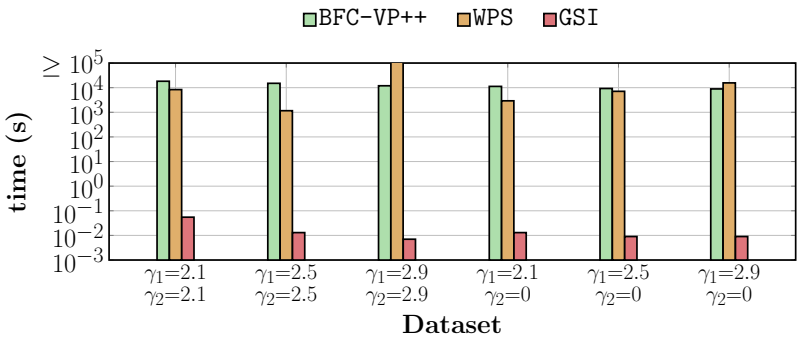


Figure 3.21: Time cost of 5,000 random queries for historical butterfly counting on different power-law bipartite graphs, where all algorithms use sequential versions.

Parallelization In addition, we compare GSI's parallelized querying with BFC-VP++'s parallelized version. In Figure 3.20, we test the running time for 5,000 queries with various numbers of threads from 1, 2, 4, 8, 16, and 32. In two large-scale datasets EV and WT, our algorithm obtains higher parallelism compared to the baseline, which has spent efforts in optimizing for higher parallelism. On the other hand, in two other two large-scale datasets TW and ER, despite its fast runtime, our algorithm's performance does not always improve as the number of threads increases, which is not surprising and is under our expectations. This is because every historical query is essentially querying on multiple $\mathcal{CS}$s and summing up the answer. In the parallel version, we allocate threads to these data structures to increase efficiency. However, given a fixed memory limitation, GSI may create less $\mathcal{CS}$s if the number of motifs is relatively small. In such a circumstance, if the number of threads approaches or exceeds the number of data structures we built in construction, the runtime may increase. In general, our parallelized GSI still obtains a faster runtime and, most likely, better parallelism on very large-scale datasets.

3.2.5.2 Comparison with TBC++

We do not adapt TBC++ and compare it in our problem because **(1):** running TBC++ each time for the project graph of each query will lead to the algorithm degrading to BFC-VP++, which we have already included as a baseline strategy. *e.g.,* TBC++ needs 149 seconds for 5,000 random queries on WQ, where BFC-VP++ only needs 86 seconds; **(2):** building index based on TBE+ requires unaffordable memory storage since the number of listed temporal butterflies will be too huge when the threshold gets larger, which can be the maximum length of the queried time window in our problem. *e.g.,* TBE+ can not run on WT due to the number of $\geq 10^{10}$ butterflies in the graph [45]. Therefore, to comprehensively compare

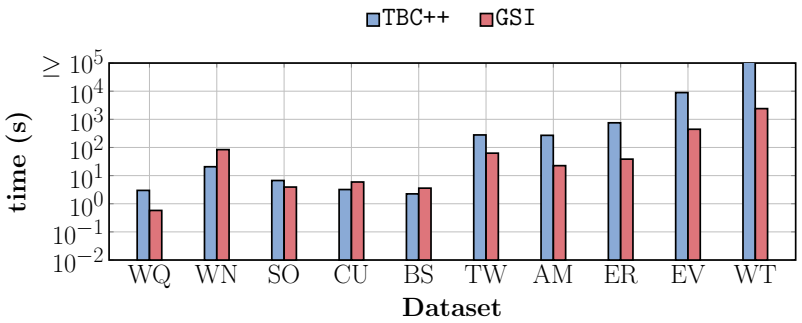


Figure 3.22: Comparison with [45] on the TBC task with random δ, where the time of GSI includes both index time and query time.

with [45], we instead adapt our algorithms to the temporal butterfly counting problem, which is the key problem [45] aiming to solve and can also be naturally generalized by GSI.

The definition of the temporal butterfly is defined as follows:

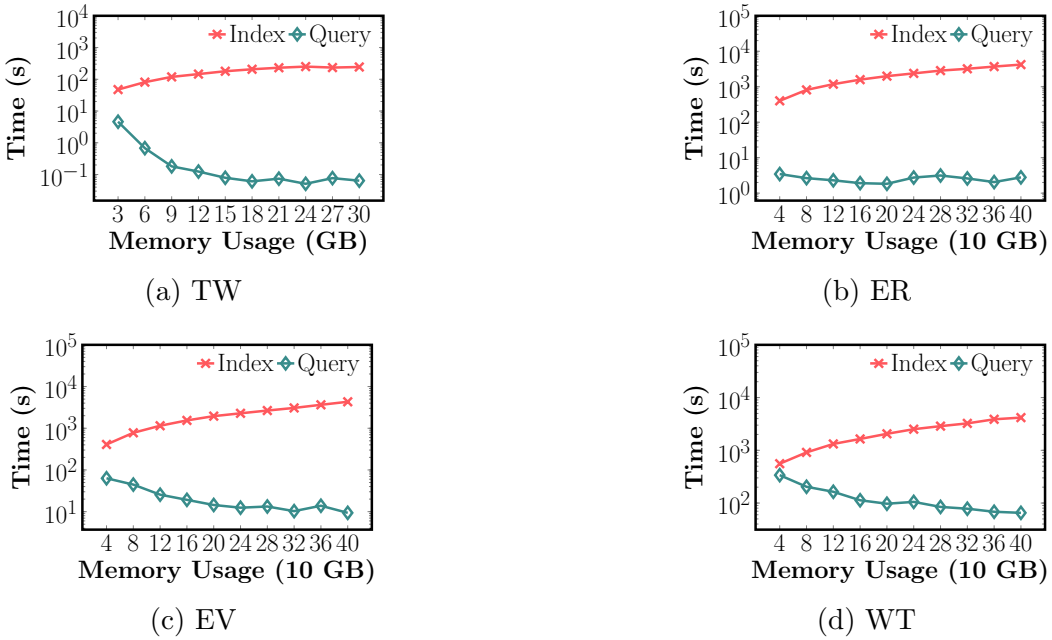
Definition 3.2.10 (Temporal Butterfly [45])**.** *Given a duration threshold δ, a temporal butterfly is a sequence of 4 temporal edges in chronological order $\langle e_1, e_2, e_3, e_4 \rangle$, s.t., (1) $e_1.t < e_2.t < e_3.t < e_4.t$, (2) $e_4.t - e_1.t \leq \delta$, and (3) the induced graph is a butterfly.*

For each dataset, we randomly select a threshold value (δ) and compare the total time taken to count the temporal butterflies for these thresholds between GSI and TBC++. Specifically, the adaptation of GSI employs the same mechanism for combining the #wedges and #butterfly indices but utilizes a new underlying data structure, rather than Chazelle's [52], tailored specifically to this problem.

As shown in Figure 3.22, GSI outperform TBC++ up to $\frac{1}{100}\times$ in the five largest datasets (TW, AM, ER, EV, WT). In the other relatively small datasets, our runtime is close to TBC++. This is because our runtime includes the time to build the index, and we only query on one fixed δ. If we consider calculating TBC for varying δs, our index-based algorithms will significantly outperform TBC++ even in those datasets. For example, for 50 randomly selected δs on WN, GSI only needs 132 seconds in total, where TBC++ needs more than 1,000 seconds.

3.2.5.3 Space-Time Trade-Offs

Since GSI is a flexible algorithm that strikes to balance the trade-off between space and time, we further study its performance under different memory limitations. Shown in



(a) TW (b) ER

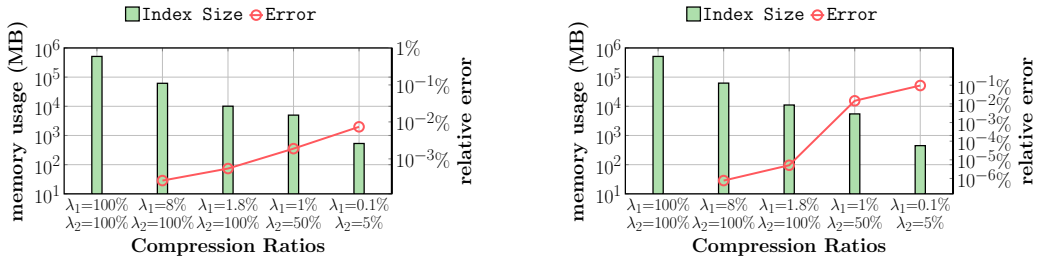
(c) EV (d) WT

Figure 3.23: Space-query trade-offs study

Figure 3.23, the query time can be efficiently reduced if provided more memory space: When the applicable memory is enlarged to 10×, the query time is decreased to $\frac{1}{10}\times$ to $\frac{1}{100}\times$. Furthermore, it is shown that even when the memory space is very limited, `GSI` can still obtain a good query efficiency. For example, on `WT`, only given 40 Gigabytes memory, both the index-construction time and query time are less than 600 seconds, which still holds a significant advantage compared to our baseline in Figure 3.19. In all, this study further reveals `GSI`'s flexibility on the trade-off between space and time. In practice, it can fit different circumstances while providing high-efficiency performance.

3.2.5.4 Study on Compressed Indexes

In addition, shown in Figure 3.24, we evaluate the index compression on two large-scale datasets `EV` and `WT`. With the single-sided compression algorithm `SGSI`, we are able to guarantee a relative error $\leq 10^{-3}$ while compressing memory usage to $\frac{1}{50}$ (i.e., 500G $\rightarrow$ 1G). With the double-sided compression algorithm `DGSI`, we achieve $\leq 10^{-1}$ relative error while compressing memory for around 1000 times (500G $\rightarrow$ 500MB). This experiment validates the effectiveness of our `GSI` algorithm with index compression based on sampling. Moreover, it empirically provides a trade-off between memory and accuracy, i.e., if we need



(a) EV (b) WT

Figure 3.24: Evaluating memory usage and relative error in various compression ratio settings. Note that both our compression algorithms are not affect the query effciency.

a higher accuracy guarantee, we use `SGSI`. To compress the memory further, we may use `DGSI` instead.

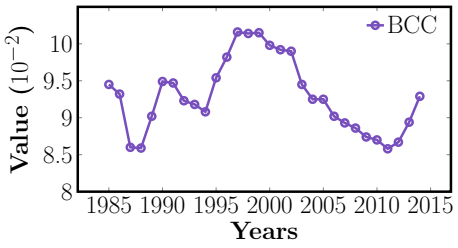
3.2.6 Case Study

To begin with, we acquired the author-publication datasets [5], which contains 10,419,221 author-publication records before 2016. Based on that, we build an author-publication bipartite temporal graph containing 14,223,972 vertices and 10,419,216 edges. Consider the projected graph $G_{[t_s,t_e]}$, its corresponding BCC should be $\frac{4\times\bowtie_{[t_s,t_e]}}{\ltimes_{[t_s,t_e]}}$, where $\bowtie_{[t_s,t_e]}$ is the butterfly count in the projected graph $G_{[t_s,t_e]}$ and $\ltimes_{[t_s,t_e]}$ is the number of 3-paths in $G_{[t_s,t_e]}$.

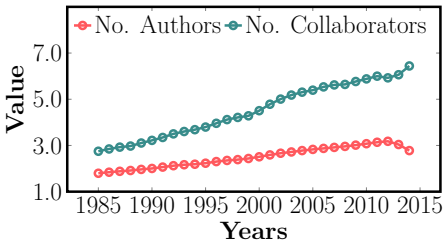
In this study, we investigate the trend of the BCC over various two-year time windows. To achieve that, we need to efficiently query several historical butterfly counts by `GSI`. The result is shown in Figure 3.25 (a), where we set the length of the time windows as two years (e.g., 1995 - 1996). Surprisingly, the value of BCCs doesn't consistently rise alongside the research community's overall trend toward collaboration. This implies that the research community might not always achieve greater cohesiveness through collaboration. In contrast, in Figure 3.25 (a), BCCs exhibit an initial general increase (1985 - 2000) followed by a decrease (2000 - 2015). We delve deeper into the underlying reasons for these trends and analyze the changing pattern of **(1)** *the average number of publications per author within each time-window* and **(2)** *the average number of unique collaborators per author within each time-window*, shown in Figure 3.25 (b). As the average number of publications generally rises from 1985 to 2015, it suggests a higher probability of collaboration,

[5] https://github.com/THUDM/citation-prediction

consequently increasing the butterfly counts in the author-publication bipartite temporal graph. Nevertheless, a greater likelihood of collaboration does not necessarily equate to more cohesive collaboration. As depicted in Figure 3.25 (b), there is a notable increase in the average number of unique collaborators after 2000, indicating a rise in 3-paths counts within the graph, which may be an important factor causing BCCs decreases again from 2000 to 2010. Entering 2010, BCCs come across another rising trend. There are many possible reasons, one of which could be the significant increase in total publications.



(a) BCC



(b) Authors vs. Collaborators

Figure 3.25: Case Study (1): The trend of closeness in global research collaboration.

3.2.7 Handling Duplicate Edges

The key challenge when the graph has duplicate edges is that the life cycle of a wedge or a butterfly is no longer an active timestamp as defined in Definition 3.2.6. Now we show that the life cycle can be decomposed into several redefined active timestamps (Definition 3.2.11) for graphs with duplicate edges. We will prove that the decomposition does not increase the time complexity or memory usage of `GSI` (Lemma 3.2.5 and Theorem 3.2.11).

Definition 3.2.11 (Active Intervals for Graphs with Duplicate Edges)**.** *Given a bipartite temporal graph G with duplicate edges, a subgraph P, we define the active intervals $\tilde{\mathcal{T}}(P)$ as a tuple of timestamps of the form $[l, r_1, r_2](l \leq r_1 \leq r_2)$ such that P is active in the query time-window $[t_s, t_e]$ if and only if for exactly one of the timestamps $[l, r_1, r_2]$, $t_s \leq l$ and $r_1 \leq t_e < r_2$.*

For applying `GSI` to a graph with duplicate edges, we first modify $\mathcal{CS}$ such that it can answer 2D-range queries on timestamps of the form $[l, r_1, r_2]$, i.e., counting the number of $[l, r_1, r_2]$ such that $t_s \leq l$ and $r_1 \leq t_e \leq r_2$ given the query time-window $[t_s, t_e]$. This can be done by applying the inclusive-exclusive principle on 2D ranges. Then we generate the active intervals (tuple of timestamps $[l, r_1, r_2]$) for each wedge. We feed each $[l, r_1, r_2]$ to `GSI` (with the modified $\mathcal{CS}$).

Now, we provide the intuition and detailed analysis of Algorithm 31.

Given a butterfly or wedge P, we consider a subgraph S constructed by choosing exactly one edge (u, v, t) for each $(u, v) \in E(P)$. We denote the set of all possible Ss by $\mathcal{S}$. Due to duplicate edges, $|\mathcal{S}| > 1$. For each $S \in \mathcal{S}$, its active timestamp is $[l_S, r_S]$ where $l_S = \min_{(u,v,t)\in S} t$ and $r_S = \max_{(u,v,t)\in S} t$. For a time-window $[t_s, t_e]$, $P \in G_{[t_s,t_e]}$ if and only if there exists an $S \in \mathcal{S}$ satisfying $t_s \leq l_S \leq r_S \leq t_e$. Therefore, for $S_1, S_2 \in \mathcal{S}$, if $l_{S_1} \leq l_{S_2} \leq r_{S_2} \leq r_{S_1}$, then S_1 is not necessary to be considered for any query time-window. In this way, we manage to reduce the number of Ss to be stored to answer historical queries concerning P.

We still need to resolve the issue of overcounting. For a time-window $[t_s, t_e]$, if there are multiple Ss such that their timestamps are all included in $[t_s, t_e]$, we should not count P as multiple wedges or butterflies (Definition 3.2.3). This is the reason we consider the redefined active timestamp in Definition 3.2.11. Let the reduced set of Ss be $\{[l_1, r_1], [l_2, r_2], \cdots, [l_k, r_k]\}$ satisfying $l_i < l_{i+1}, r_i \leq r_{i+1}$ for each $1 \leq i < k$. We create a redefined timestamp $[l_i, r_i, r_{i+1}]$ for each $1 \leq i < k$ and a redefined timestamp $[l_k, r_k, \infty]$ for the last timestamp $[l_k, r_k]$. We can see that exactly one of the redefined timestamps becomes active when $P \in G_{[t_s,t_e]}$: If there exists $1 \leq i < k$ such that $r_i \leq t_e < r_{i+1}$, only $[l_i, r_i, r_{i+1}]$ is active. Otherwise, we have $t_e \geq r_k$. In such case, $[l_k, r_k, \infty]$ is active. In addition, if $P \notin G_{[t_s,t_e]}$, no redefined timestamps becomes active. For any timestamp $[l_i, r_i]$, $[t_s, t_e]$ does not include $[l_i, r_i]$. This implies that the redefined timestamp $[l_i, r_i, r_{i+1}]$ (or $[l_k, r_k, \infty]$ when $i = k$) is not active.

Algorithm 31 computes the active intervals for a wedge $\langle x \rightsquigarrow y \rightsquigarrow z \rangle$. Given the two timestamp sets $L_{x,y}$ and $L_{y,z}$, we will produce all necessary $S \in \mathcal{S}$ and store the redefined timestamps in A. We can safely assume that there is no duplicate timestamp in either $L_{x,y}$ or $L_{y,z}$. To begin with, we initialize S to be $\emptyset$ (Line 3). We sort $L_{x,y}$ and $L_{y,z}$ from small to large (Line 2). We also insert an ∞ timestamp to both timestamp sets (Line 1) to avoid boundary cases. We will enumerate every element in these sets, and we denote them as $L_{x,y}[i](1 \leq i \leq |L_{x,y}|$ and $L_{y,z}[j](1 \leq j \leq |L_{y,z}|$. Consider a pair (i, j). When $L_{x,y}[i] \leq L_{y,z}[j]$, we will adjust i to i' such that $L_{x,y}[i + 1] > L_{y,z}[j]$. That is to say, we will adjust the smaller side as much as possible without breaking the inequality $L_{x,y}[i] \leq L_{y,z}[j]$ (Line 5, Line 14, Line 7, and Line 17). We do the same for the case which $L_{x,y}[i] > L_{y,z}[j]$ (adjusting j instead). For the current enumerated pair (i, j), the corresponding redefined timestamp of the wedge formed by $(x, y, L_{x,y}[i])$ and $(y, z, L_{y,z}[j])$ is $[l, r, \max(L_{x,y}[i], L_{y,z}[j])]$, where l and r are the minima and maxima of the two timestamps $L_{x,y}[i]$s and $L_{y,z}[j]$s in the previous iteration (Line 10 and Line 11). We insert it into A (Line 11). After inserting, we move to the next pair by increasing one of the indices i or j. When $L_{x,y}[i] \leq L_{y,z}[j]$, by setting i to $i + 1$, we will have $L_{x,y}[i] > L_{y,z}[j]$ (Line 12 to

Line 14). Then we adjust j again to the largest possible value satisfying $L_{y,z}[j] \leq L_{x,y}[i]$ (Line 14). If $L_{x,y}[i] > L_{y,z}[j]$, we increase j and adjust i instead (Line 13 and Line 17). The whole process will terminate when no more feasible pair needs to be enumerated (Line 9). Then, all the redefined timestamps are stored in A. We return A as the output (Line 21).

Lastly, we prove that counting the number of activated redefined timestamps can be converted to computing the difference between two 2D-range counting queries.

Previously, we regarded the active timestamp as a single point in the 2-D plane and inserted it into the 2D-counting data structure. For any time window, the answer is computed by a single query on the data structure. Under the current definition of the active timestamp, we can still efficiently answer the query using two 2D-range counting data structures T and $\overline{T}$ instead. That is to say, after construction, we will be able to answer the number of redefined timestamps $[l, r_1, r_2]$ satisfying $t_s \leq l$ and $r_1 \leq t_e < r_2$. Specifically, for $[l, r_1, r_2]$, we insert (l, r_1) and (l, r_2) into T and $\overline{T}$ respectively. To count the number of active timestamps of a time-window $[t_s, t_e]$, we first query $[t_s, \infty] \times [-\infty, t_e]$ on both T and $\overline{T}$, denoted as num and $\overline{num}$ respectively. Then, we return $num - \overline{num}$ as the answer. The formal proof is as follows.

$$\sum_{P \in G} \sum_{[l_i, r_{1,i}, r_{2,i}] \in \tilde{\mathcal{T}}(P)} \mathbb{1}\{l_i \geq t_s \wedge t_e \in [r_{1,i}, r_{2,i})\} =$$

$$\sum_{P \in G} \sum_{[l_i, r_{1,i}, r_{2,i}] \in \tilde{\mathcal{T}}(P)} \mathbb{1}\{l_i \geq t_s \wedge t_e \geq r_{1,i}\} - \mathbb{1}\{l_i \geq t_s \wedge t_e \geq r_{2,i}\} =$$

$$\left(\sum_{P \in G} \sum_{[l_i, r_{1,i}] \in \tilde{\mathcal{T}}(P)} \mathbb{1}\{l_i \geq t_s \wedge t_e \geq r_{1,i}\} \right) -$$

$$\left(\sum_{P \in G} \sum_{[l_i, r_{2,i}] \in \tilde{\mathcal{T}}(P)} \mathbb{1}\{l_i \geq t_s \wedge t_e \geq r_{2,i}\} \right) =$$

$$T.query([t_s, \infty] \times [-\infty, t_e]) - \overline{T}.query([t_s, \infty] \times [-\infty, t_e])$$

Lastly, we prove that if the size (number of $[l, r_1, r_2]$ in the tuple) of the active intervals of each wedge is bounded (Lemma 3.2.5), both `EBI` and `CBI`'s time complexity will not be compromised.

Lemma 3.2.5. *For any two vertices $u, v \in V(G)$, we denote $cnt_{u,v}$ as the number of edges $(u, v, t) \in E(G)$. There exists an algorithm (Algorithm 31 that returns its active intervals (Definition 3.2.11) of size $O(\min(cnt_{x,y}, cnt_{y,z}))$ for any wedge $\langle x \rightsquigarrow y \rightsquigarrow z \rangle$.*

Proof of Lemma 3.2.5. WLOG, we assume $cnt_{x,y} \leq cnt_{y,z}$. It is easy to see the time complexity of Algorithm 31 is $O(|S| \log m)$, and we will then show $|S| \leq 3cnt_{x,y}$.

For each wedge $\langle x \rightsquigarrow y \rightsquigarrow z \rangle\ [l_i, r_{1,i}, r_{2,i}] \in S$, we have at least one of l_i or $r_{1,i}$ comes from a timestamp of $L_{x,y}$. In other words, for each pair (l, r) in lines 9 to 18 of Algorithm 31, there exists an $i \in [1, cnt_{x,y}]$ such that at least one of l or r equals to $L_{x,y}[i]$. We consider these cases as follows:

- $l = L_{x,y}[i], r \neq L_{x,y}[i]$: We move $i \leftarrow i + 1$ in the next iteration, which implies this case will only occur for at most $cnt_{x,y}$ times.
- $l \neq L_{x,y}[i], r = L_{x,y}[i]$: let $\hat{l}$ and $\hat{r}$ be the values of l and r in the next iteration, respectively. We have $\hat{r} \geq \hat{l} \geq r$, which implies this case will only occur for at most $cnt_{x,y}$ times.
- $l = L_{x,y}[i], r = L_{x,y}[i]$: let $\hat{l}$ and $\hat{r}$ be the value of l and r in the next iteration, respectively. We have either $r' \geq \hat{l} > r$ or $\hat{r} > \hat{l} \geq r$, which implies this case will only occur for at most $cnt_{x,y}$ times.

Since each wedge belongs to one of the three cases above, we prove that $|S| \leq 3cnt_{x,y}$.

□

With this lemma, we are ready to prove that the time complexity for our algorithms will not be compromised by duplicate edges:

Theorem 3.2.11 (Time complexity with Duplicate Edges)**.** ***(i)*** *There exists a modification for* `EBI` *that can run in $O(\log m)$ time and $O(m^2)$ memory usage for bipartite temporal graphs with duplicate edges;* ***(ii)*** *There exists a modification for* `CBI` *that can run in $O(\tilde{w} \log m)$ time and $O(m\delta)$ memory usage for bipartite temporal graphs with duplicate edges.*

Proof of Theorem 3.2.11. **(i)**: Considering a butterfly consisting of two wedges $\langle x \rightsquigarrow y_1 \rightsquigarrow z \rangle$ and $\langle x \rightsquigarrow y_2 \rightsquigarrow z \rangle$, the size of its active intervals is $O(\min(cnt_{x,y_1}, cnt_{y_1,z}) \times \min(cnt_{x,y_2}, cnt_{y_2,z}))$ by Lemma 3.2.5. Since

$$\sum_{\text{each butterfly } \langle x,y_1,z,y_2 \rangle \in G} \min(cnt_{x,y_1}, cnt_{y_1,z}) \times \min(cnt_{x,y_2}, cnt_{y_2,z})$$
$$\leq \sum_{\text{each butterfly } \langle x,y_1,z,y_2 \rangle \in G} cnt_{x,y_1} cnt_{y_2,z} \leq m^2.$$

, the bound of the number of points in $\mathcal{CS}s$ $O(m^2)$ for `EBI` still holds.

(ii): Considering a wedge $\langle x \rightsquigarrow y \rightsquigarrow z \rangle$, the the size of its active intervals in its active timestamp is $O(\min(cnt_{x,y}, cnt_{y,z}))$ by Lemma 3.2.5. The total number of tuples for all wedges' active timestamps is:

$$\begin{aligned}
&\sum_{y \in V(G)} \sum_{\substack{x \in N(y), \\ pr(x) \prec pr(y)}} \sum_{\substack{z \in N(y), \\ pr(x) \prec pr(z)}} \min(cnt_{x,y}, cnt_{y,z}) \\
&\leq \sum_{y \in V(G)} \sum_{\substack{x \in N(y), \\ pr(x) \prec pr(y)}} cnt_{x,y} deg_x \\
&\leq \sum_{(x,y) \in E(G)} \min(deg_x, deg_y) = m\delta.
\end{aligned}$$

Therefore, the bound of the number of points in $\mathcal{CS}s$ for `CBI`, $O(m\delta)$, also holds. □

Algorithm 31: Computing $\langle x \rightsquigarrow y \rightsquigarrow z \rangle$'s Active Timestamps

Input: A list of unique timestamps on the duplicate edges between x and y: $L_{x,y}$;
A list of unique timestamps on the duplicate edges between y and z : $L_{y,z}$;

Output: The active intervals $A = \{[l_i, r_{1,i}, r_{2,i}]\}$ for $\langle x \rightsquigarrow y \rightsquigarrow z \rangle$

$L_{x,y} \leftarrow L_{x,y} \cup \{\infty\}, L_{y,z} \leftarrow L_{y,z} \cup \{\infty\}$;
Sort $L_{x,y}$ and $L_{y,z}$ in ascending order;
$S \leftarrow \emptyset, i \leftarrow 1, j \leftarrow 1$;
if $L_{x,y}[1] \leq L_{y,z}[1]$ **then**
    $i \leftarrow$ the maximum k such that $L_{x,y}[k] \leq L_{y,z}[1]$;
else
    $j \leftarrow$ the maximum k such that $L_{y,z}[k] \leq L_{x,y}[1]$;
$l \leftarrow \infty, r \leftarrow \infty$;
while $i < |L_{x,y}|$ *and* $j < |L_{y,z}|$ **do**
    $l' \leftarrow \min(L_{x,y}[i], L_{y,z}[j]), r' \leftarrow \max(L_{x,y}[i], L_{y,z}[j])$;
    if $r \neq \infty$ **then** $A = A \cup \{[l, r, r']\}$;
    if $L_{x,y}[i] \leq L_{y,z}[j]$ **then**
        $i \leftarrow i + 1$;
        $j \leftarrow$ the maximum k such that $L_{y,z}[k] \leq L_{x,y}[i]$;
    else
        $j \leftarrow j + 1$;
        $i \leftarrow$ the maximum k such that $L_{x,y}[k] \leq L_{y,z}[j]$;
    $l \leftarrow l', r \leftarrow r'$;
if $r \neq \infty$ **then**
    $A = A \cup \{[l, r, \infty]\}$
return S;

Chapter 4

Balanced Subgraph Finding on Signed Graphs

In this chapter, we present a new and versatile model to identify polarized communities in signed graphs. The model accommodates inherent imbalances in polarized communities through a tolerance feature. In addition, we propose a region-based heuristic algorithm. Through a wide variety of experiments on graphs of up to 40 M edges, it demonstrates effectiveness and efficiency beyond the state-of-the-art methods in addressing both traditional and generalized MBS problems. We also adapt our model and algorithm to 2PC, a related but essentially different task, further verifying their generalizability.

The results of this chapter come from the joint work [56] with Qiuyang Mang, Hangrui Zhou, Richard Peng, Yu Gao and Chenhao Ma. The primary authors are Jingbang, Qiuyang and Hangrui. Hangrui conducted the experiment, and together, all primary authors designed the experiment. Jingbang wrote the whole paper. The algorithm was proposed by the three primary authors in early discussions, with Qingyu Shi also participating in the discussions.

4.1 Introduction

Social media platforms, integral to our digital connectivity, transform interactions into analyzable social networks. By deploying graph algorithms, we discern network properties like community detection [221, 90] and partitioning [44], informing user experience

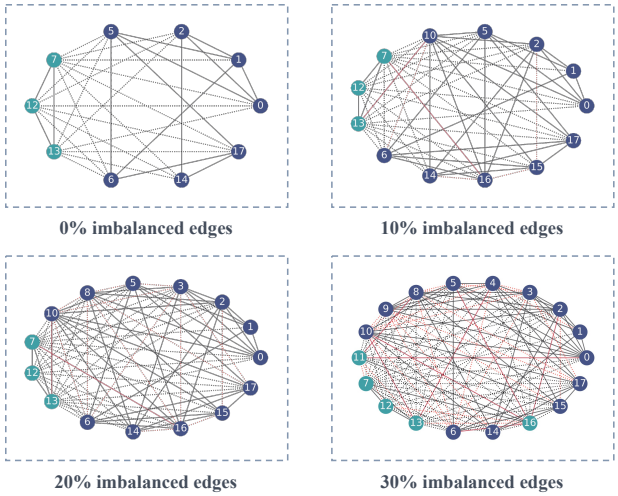


Figure 4.1: Balanced subgraphs are found in CLOISTER with different tolerance, where solid edges are positive, dashed edges are negative, black edges are balanced, and red edges are imbalanced.

improvements and recommendation systems [217]. Yet, these platforms can also engender echo chambers that reinforce divisive ideologies, challenging democratic health. Consequently, detecting and countering polarization in social networks is a critical area of research [27, 164], pivotal for developing defenses against misinformation [62, 25].

A classical model that applies to social networks to deal with polarization is the signed graphs. The signed graph model overcomes the limitation that normal graphs cannot capture users' dispositions. Generally speaking, while the vertex set represents users, there are two kinds of edges between vertices indicating agreement or disagreement. We often refer to them as the positive and the negative edges. The signed graph model was first introduced by Harary in 1953 [101] to study the concept of *balance*. The concept of *balance* is important in signed graphs. Generally speaking, a signed graph is balanced if it can be decomposed into two disjoint sets such that positive edges are between vertices in the same set while negative edges are between vertices from different sets. Such a concept has many practical applications, especially in the polarization study. In a social network, a balanced graph suggests two communities exist with contrasting relationships while maintaining inner cohesion.

There are two lines of work when studying social network polarization with signed graphs. Since most graphs coming from real scenarios are not balanced, if we can find the maximum balanced subgraph (MBS) instead, it usually reveals the largest polarized communities along with several important properties. On the other line of work, instead of

extracting the maximal subgraph, it focuses on removing edges to guarantee the balance of the original graph. The minimum number of edges whose deletion makes all connected components balanced is called the *Frustration Index* of the given signed graph.

However, such a notion of balance is strict in that no edge can disobey the condition. In reality, such strictness does not usually appear. For example, though being dominated, there usually exists some voices of disagreement on the majority idea, even in the most extreme community. Besides, two individuals in different political parties might reach a consensus on certain issues. In other words, there usually exist some *imbalanced* edges in signed graphs extracted from the community, which are against the strict notion of balance. Failing to handle these imbalanced edges might prevent us from identifying, extracting, and characterizing the community properly in real scenarios. We provide an example in Figure 4.1[1]: As we increase the limitation for imbalanced edges, the community is getting significantly larger and denser. Therefore, we may be unable to capture the actual community if we prohibit imbalanced edges when computing.

Many questions arise here due to the many limitations:

- Can these two lines of work be unified?
- Can we develop algorithms finding communities that are non-strictly balanced?

As a result, we answer these questions affirmatively. We design a tailored function to measure the tolerance of the imbalance. To properly describe such tolerance, we adapt the frustration index as part of the tolerance function. In this way, we can either allow no tolerance to extract the MBS in the network or a loose tolerance to extract communities that might contain voices of disagreement. We further raise new problems based on this tolerance model and present a new region-based heuristic algorithm that computes maximal balanced subgraphs under the tolerance setting. Our new algorithm is versatile, delivering high-quality solutions across a range of problems based on the tolerance model, and it's also efficient, with an expected output time proportional to the size of the result. By setting different tolerances, we utilize our algorithm to handle different tasks and compare it with the performance of state-of-the-art algorithms[2] on 8 real-world datasets.

Our contributions are summarized as follows:

- We introduce a novel, generalized, and practical model for identifying balanced subgraphs with tolerance in signed graphs.

[1] konect.cc/networks/moreno_sampson

[2] Here, the SOTA algorithms are adapted to fit our new tasks under various tolerances.

- We have developed an efficient region-based heuristic randomized algorithm, characterized by an expected time complexity proportional to the size of the output, and coupled with a guarantee of result quality.
- Extensive experiments show that our algorithm consistently outperforms the baselines in terms of the quality of the returned subgraphs and achieves up to 100× speedup in terms of running time.
- The effectiveness of our algorithm is also evident in its application to polarized community detection.

4.1.1 Related Work

Signed Graphs Signed graphs were first introduced by Harary in 1956 to study the notion of *balance* [101]. Cartwright and Harary generalized Heider's theory of balance onto signed graphs [47]. Harary also developed an algorithm to detect balance in signed graphs [103]. There are other works on studying the minimum number of sign changes to make the graph balance [7]. Spectral properties have also been studied recently. Hou et al. have studied the smallest eigenvalue in signed graphs' Laplacian [112, 111].

Many works focus on community detection or partition in signed graphs. Anchuri et al. give a spectral method that partitions the signed graph into non-overlapping balanced communities [12]. Doreian and Mrvar propose an algorithm for partitioning a directed signed graph and minimizing a measure of *imbalance* [75]. Yang et al. give a random-walk-based method to partition into cohesively polarized communities [222]. The more recent work is by Niu et al [165]. They leverage the balanced triangles, which model cohesion and polarization simultaneously, to design a good heuristic.

Maximum Balanced Subgraphs The Maximum Balanced Subgraphs (MBS) problem has two variants: maximizing the number of vertices (MBS-V) and edges (MBS-E). Poljak et al. give a tight lower bound in 1986 [175] on the number of edges and vertices of the balanced subgraph. the MBS-E problem is in fact **NP**-hard since it can be formulated as a generalization of the standard MaxCut problem. To find the exact solution, there are some algorithms in the context of *fixed-parameter tractability* (FPT) are developed [113, 67]. More studies are on extracting large balanced subgraphs. DasGupta et al. have developed an algorithm based on semidefinite programming relaxation (SDP) [68]. For the MBS-V problem, Figueiredo and Frota propose several heuristic methods in 2014 [89].

In 2020, Ordozgoiti et al. proposed a new algorithm named Timbal [168] that extracts large balanced subgraphs regarding both vertices and edges. Timbal relies on signed spectral theory and the bound for perturbations of the graph Laplacian. However, Timbal is not stable and the balanced subgraph it found is sometimes small and unsatisfying as shown in our experiments.

Frustration Index The frustration index is first introduced in 1950s [1, 102]. Computing the frustration index is related to the EdgeBipartization problem, which requires minimization of the number of edges whose deletion makes the graph bipartite. Since EdgeBipartization is **NP**-hard, computing the frustration index is also **NP**-hard. The MaxCut is also a special case of the frustration index problem. Assuming Khot's unique games conjecture [122], it is still **NP**-hard to approximate within any constant factor. For the non-constant factor case, there are works that produce a solution approximated to a factor of $O(\sqrt{\log n})$ [3] or $O(\log k)$ [18] where n is the number of vertices and k is the frustration index. Coleman et al. have given a review on different approximation algorithms [66]. Hüffner et al. show that the frustration index is *fixed parameter tractable* and can be computed in $O(2^k m^2)$, where m is the number of edges and k is the fixed parameter (the frustration index) [114]. There are also algorithms using binary programming models to compute the exact frustration index [14, 15].

4.2 Problem Specification

A signed graph is an undirected simple graph $G = (V, E^+, E^-)$ where V is the vertex set and E^+, E^- represent the positive and negative signed edge sets. We first give the formal definition of balanced graphs as follows, which is the same as the previous works [168, 165]. Note that we and these works require the graph to be connected for the community detection proposal.

Definition 4.2.1 (Balanced Graph)**.** *Given a signed graph $G = (V, E^+, E^-)$, G is balanced if G is connected and there exists a partition $V = V_1 \cup V_2, V_1 \cap V_2 = \emptyset$ such that for each edge $(i, j) \in E^+$, vertices i and j belong to the same set within V_1 and V_2, while for each edge $(i, j) \in E^-$, they belong to the different sets.*

Graphs are usually not balanced, especially when they are from practical scenarios. Therefore, people turn to study to find the maximum balanced subgraph (MBS) from the given graph. There are usually two variants of problems: maximizing the number of vertices [168] and edges [68].

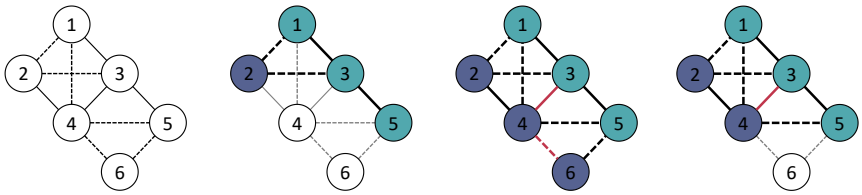


Figure 4.2: A signed graph (a), its MBS-V and MBS-E (b), its TMBS-V and TMBS-E (c), and its β-TMBS (d) with $\beta = \frac{1}{3}$.

Problem 4 (MBS-V)**.** *Given a signed graph $G = (V, E^+, E^-)$, find the graph $G' = (V', E'^+, E'^-)$ induced by $V' \subseteq V$ such that G' is balanced and $|V'|$ is maximized.*

Problem 5 (MBS-E)**.** *Given a signed graph $G = (V, E^+, E^-)$, find the graph $G' = (V', E'^+, E'^-)$ induced by $V' \subseteq V$ such that G' is balanced and $|E'^+ \cup E'^-|$ is maximized.*

We illustrate the concepts of MBS-V and MBS-E with an example in Figure 4.2, along with other problems to be discussed below. Here, we give a graph with 6 vertices numbered from 1 to 6 where solid edges are positive and dashed edges are negative, presented in (a). When solving Problem 4 or 5 on this graph, the optimal subgraph is constructed by vertex $1, 2, 3, 5$, where $V_1 = \{1, 3, 5\}$ and $V_2 = \{2\}$. They are painted in two different colors in (b).

Another well-known problem in this topic is computing the minimum number of edges whose deletion makes all connected components balanced. This minimum amount of the edge removal is called the *Frustration Index* of the given signed graph G, denoted as $L(G)$.

Now, we are ready to introduce the concept of *Balance with β-tolerance* in signed networks, which is central to our work.

Definition 4.2.2 (Balanced Graph under β-Tolerance)**.** *Given a signed graph $G = (V, E^+, E^-)$ and a tolerance parameter $\beta \in (0, 1]$, G is balanced under β-Tolerance if G is connected and $\frac{L(G)}{|E'^+ \cup E'^-|} \leq \beta$.*

In other words, when a graph is balanced under β-tolerance, it implies that by removing at most $\beta|E^+ \cup E^-|$ edges from it, we can ensure that all connected components of the remaining graph become strictly balanced. Expanding upon the original definition of balance, this concept allows for the tolerance of a maximum of $\beta|E^+ \cup E^-|$ edges that disobey the partitioning of polarized communities.

Under such β-tolerance restriction, finding the maximum balanced subgraph is generalized into the following two problems.

Problem 6 (TMBS-V). *Given a signed graph $G = (V, E^+, E^-)$ and a tolerance parameter $\beta \in (0, 1]$, find the graph $G' = (V', E'^+, E'^-)$ induced by $V' \subseteq V$ such that G' is balanced under β-Tolerance and $|V'|$ is maximized.*

Problem 7 (TMBS-E). *Given a signed graph $G = (V, E^+, E^-)$ and a tolerance parameter $\beta \in (0, 1]$, find the graph $G' = (V', E'^+, E'^-)$ induced by $V' \subseteq V$ such that G' is balanced under β-Tolerance and $|E'^+ \cup E'^-|$ is maximized.*

When β becomes any value between $(0, \frac{1}{|E^+ \cup E^-|})$, Problem 6 and Problem 7 are equivalent to the Problem 4 and Problem 5 respectively. However, unlike the strictly balanced graph, even deciding whether a signed graph satisfies β-tolerance is hard. Formally, we have the following lemma.

Lemma 4.2.1. *Given a signed graph $G = (V, E^+, E^-)$, deciding whether G is balanced under β-tolerance cannot be done within polynomial time for tolerance parameter $\beta > \frac{1}{|E^+ \cup E^-|}$, unless $\boldsymbol{P = NP}$.*

Proof of lemma 4.2.1. We prove the lemma by a reduction from the calculation of the Frustration Index. If we can decide whether G is balanced under β-tolerance within the polynomial runtime, we will be able to decide whether $L(G)$ is greater than any guessing value k by setting $\beta = \frac{k}{|E^+ \cup E^-|}$. Combined with a binary search, we can compute the Frustration Index of G within the polynomial runtime, which has been proven as a **NP**-hard problem [3]. □

Therefore, it is difficult to find large balanced graphs with tolerance and we cannot adapt any previous algorithms on MBS or other related problems. Meanwhile, there is another limitation that is worth mentioning. If $\beta \geq \frac{L(G)}{|E^+ \cup E^-|}$, Problem 6 and Problem 7 has a trivial optimal solution: the largest connected component. That is to say, in such loose tolerance restriction, the optimal solution to these two problems might not reflect the polarized community. The previous showcase given in Figure 4.2 (c) is a typical example: When we solve Problem 6 and 7 with $\beta = \frac{1}{3}$ in such graph, we get the whole graph. However, vertex 6 is in fact a noise that only has negative edges connected to both V_1 and V_2. Therefore, it should not be included in any meaningful subgraph solution.

To encounter the challenges and limitations, we first propose Tolerant Balance Index, a novel metric to evaluate signed graphs under the given tolerance for the balance.

Definition 4.2.3 (Tolerant Balance Index (TBI)). *Given a signed graph $G = (V, E^+, E^-)$ and a tolerance parameter $\beta \in (0, 1]$, define its Tolerant Balance Index $\Phi(G, \beta)$ as the value of $|E^+ \cup E^-| - \frac{L(G)}{\beta}$.*

Correspondingly, we propose the following problem as the main task to solve throughout the work.

Problem 8 (β-TMBS)**.** *Given a signed graph G and a tolerance parameter $\beta \in (0, 1]$, find the graph G' induced by $V' \subseteq V$ such that G' is connected and $\Phi(G', \beta)$ is maximized.*

It is easy to show, for any balanced graph G under β-tolerance, $\Phi(G, \beta) \geq 0$. To discuss the usage of solving Problem 8, we also want to address that maximizing the number of edges (Problem 5 and 7) approximates the solution for maximizing the number of vertices (Problem 4 and 6). That is to say, these two variants of MBS problems are related.

Lemma 4.2.2. *Given a signed graph G, the solutions to Problem 5 and Problem 7 are $\frac{1}{\Delta}$-approximations for the Problem 4 and Problem 6 respectively, where Δ is the maximum degree of vertices in G.*

Proof of lemma 4.2.2. We first show that the solutions to Problem 5 is a $\frac{1}{\Delta}$-approximations for the Problem 4, and the relations between Problem 6 and Problem 7 can be proven similarly.

Given a signed graph G, let the optimal subgraph to Problem 4 be G_1 with n_1 vertices and m_1 edges, and the optimal subgraph to Problem 5 be G_2 with n_2 vertices and m_2 edges. Since G_1 is connected, we have

$$m_1 \geq n_1 \tag{4.1}$$

Since Δ is the maximum degree of vertices in G, we have

$$n_2 \geq \frac{m_2}{\Delta} \tag{4.2}$$

Combined with (4.1) and (4.2), we have $n_2 \geq \frac{m_2}{\Delta} \geq \frac{m_1}{\Delta} \geq \frac{n_1}{\Delta}$, which implies G_2 is a $\frac{1}{\Delta}$-approximations for the Problem 4. □

Considering the aforementioned interrelationship between two variants, we propose that finding the large connected subgraph with maximal $\Phi(G, \beta)$ (Problem 8) can be considered as a good approximation of the solutions to Problem 6 and Problem 7 simultaneously. In addition, since Problem 8 takes both the size and the polarity into account, it mitigates the limitation of large β. When we solve Problem 8 with $\beta = \frac{1}{3}$ on Figure 4.2 (a), the optimal subgraph will be vertices excluded vertex 6, where $V_1 = \{1, 3, 5\}$ and $V_2 = \{2, 4\}$, presented in (d). As it is shown, the noise vertex 6 is not mistakenly selected into the optimal subgraph we try to find. Therefore, the polarity is being preserved.

In Section 4.3, we will present an efficient and effective algorithm for Problem 8. Then in Section 4.4, we will demonstrate the experimental results to support our proposal.

4.3 Algorithm

We propose *Region-based Heuristic* (RH), a new algorithm that searches for large balanced subgraphs with the β-tolerance restriction (PROBLEM 8). Our algorithm runs in a given signed graph $G = (V, E^+, E^-)$, where E^+, E^- represent the positive and negative signed edge sets. The tolerance parameter β is also given as input.

4.3.1 Relaxation

It is not easy to compute $\Phi(G, \beta)$ directly. Instead, we propose an alternative method to approximate it from the lower end while guaranteeing the tolerance condition ($\Phi(G, \beta) \geq 0$) is not violated.

For each vertex in G, we assign them with a color in $\{0, 1\}$. Only vertices of the same color can be grouped in the same community. We denote any color assignment of G as $\mathcal{X}$. Such definition is aligned with the previous work [15]. Now, we are ready to give the formal definition of our newly proposed *Tolerant Balanced Count*.

Definition 4.3.1 (Tolerant Balance Count (TBC))**.** *Given a signed graph $G = (V, E^+, E^-)$ under coloring $\mathcal{X}$ and a tolerance parameter $\beta \in (0, 1]$, define its Tolerant Balance Count as*

$$\hat{\Phi}(G, \beta, \mathcal{X}) = |E^+ \cup E^-| - \frac{1}{\beta} \sum_{(i,j) \in E^+} \mathbb{1}\{x_i \neq x_j\} - \frac{1}{\beta} \sum_{(i,j) \in E^-} \mathbb{1}\{x_i = x_j\}$$

The following lemma states that $\Phi(G, \beta)$ is, in fact, the upper bound of $\hat{\Phi}(G, \beta, \mathcal{X})$ with respect to different coloring $\mathcal{X}$.

Lemma 4.3.1. *Given a signed graph $G = (V, E^+, E^-)$ and a tolerance parameter $\beta \in (0, 1]$, we have $\Phi(G, \beta) = \max_{\mathcal{X}} \hat{\Phi}(G, \beta, \mathcal{X})$.*

Proof of lemma 4.3.1. The lemma is equivalent to that:

$\sum_{(i,j) \in E^+} \mathbb{1}\{x_i \neq x_j\} + \sum_{(i,j) \in E^-} \mathbb{1}\{x_i = x_j\}$ is a tight upper bound on $L(G)$.

First, given a coloring $\mathcal{X}$, if we remove all edges that let the indicator functions be true from G, the remaining graph will be balanced for each connected component due to DEFINITION 4.2.1. This implies that $\sum_{(i,j) \in E^+} \mathbb{1}\{x_i \neq x_j\} + \sum_{(i,j) \in E^-} \mathbb{1}\{x_i = x_j\}$ is no less than the $L(G)$, which is the minimum number of edges to be removed.

We then show the bound is tight. Consider the remaining graph G' from removing all edges contributing to $L(G)$. Since all connected components of G' are balanced, we can find a partition $V = V_1 \cup V_2$ and $V_1 \neq V_2$ such that if we assign $x_i = 0$ for all $i \in V_1$ and $x_j = 1$ for all $j \in V_2$, the value of $\sum_{(i,j)\in E^+} \mathbb{1}\{x_i \neq x_j\} + \sum_{(i,j)\in E^-} \mathbb{1}\{x_i = x_j\}$ will become to zero for G'. This implies the same value for G under the coloring is equal to $L(G)$ and thus the bound is tight. □

We also have the following corollary that ensures the β-tolerance requirement is not violated.

Corollary 4.3.1. *Given a signed graph G and a tolerance parameter β, G is balanced under β-tolerance if there exists a coloring $\mathcal{X}$ such that $\hat{\Phi}(G, \beta, \mathcal{X}) \geq 0$.*

In this way, although we cannot compute $\Phi(G, \beta)$ directly, we can approximate the actual value by accumulating the non-negative $\hat{\Phi}(G, \beta, \mathcal{X})$ values of various coloring. Additionally, by Corollary 4.3.1, we can guarantee the β-tolerance during the whole computation. Such relaxation plays an important role in our algorithm, and the experimental results in Section 4.4 also validate its effectiveness.

4.3.2 Local Search

Our search process in Algorithm 32 starts from an initial vertex s instead of the whole vertices set. We will discuss how to choose s wisely in the later Section 4.3.3.

Search Operations As discussed in Section 4.3.1, we will search for a connected subgraph G' and a coloring $\mathcal{X}$ that $\hat{\Phi}(G', \beta, \mathcal{X})$ is as large as possible. In the following, if not specified, we use $\hat{\Phi}$ to denote the tolerant balanced count of the subgraph. Throughout the search process, we use a set S to store the selected vertices and the coloring simultaneously. Specifically, S stores selected tuples (x, c), where (x, c) represents a vertex x colored as c. There are three basic operations with S that will insert, delete, or change the color of a vertex x respectively. Note that these three operations well define the neighborhood of any solution we find.

- `VertexInsert`(x, c): Execute $S \leftarrow S \cup (x, c)$.
- `VertexDelete`(x): Let c be the color of x. Execute $S \leftarrow S \setminus (x, c)$.

Algorithm 32: RH: SEARCH

Input: Signed graph G; Tolerance parameter β; Non-incremental probability p; Initial vertex s; Early stop turn limit T.

Output: $\overline{H} \subseteq G$: the found balanced subgraph with β-tolerance; $\hat{\Phi}$: the lower bound of $\overline{H}$'s balance value we achieved.

1 $S \leftarrow \{(s,0)\}$, $\hat{\Phi}_{opt} \leftarrow 0$, $\hat{\Phi}_{cur} \leftarrow 0$, $t \leftarrow T$;
2 Initialize max-heaps MH_I, MH_F;
3 Insert s into MH_F;
4 **for** $v \in N(s)$ **do**
5 Insert $(v,0)$ and $(v,1)$ into MH_I;
6 **while** $t \geq 0$ *and* $|S| < |V(G)|$ **do**
7 $\{op, \Delta_{\hat{\Phi}}\} = \texttt{GetTopNode}(MH_I)$; `// ordered by` $\Delta_{\hat{\Phi}}$
8 **if** $z \sim U(0,1) < p$ **then**
9 $\{op, \Delta_{\hat{\Phi}}\} \leftarrow \max(\{op, \Delta_{\hat{\Phi}}\}, \texttt{GetTopNode}(MH_F))$;
10 **if** $z \sim U(0,1) < \frac{p \log |S|}{|S|}$ **then**
11 $\{\hat{op}, \overline{\Delta_{\hat{\Phi}}}\} \leftarrow \texttt{DelEval}(S)$;
12 $\{op, \Delta_{\hat{\Phi}}\} \leftarrow \max(\{op, \Delta_{\hat{\Phi}}\}, \{\hat{op}, \overline{\Delta_{\hat{\Phi}}}\})$;
13 $S \leftarrow$ Execute operation op on S;
14 $x \leftarrow$ the vertex of the op;
15 **for** $v \in N(x) \cup x$ **do**
16 Insert, update or delete v's operations in MH_I and MH_F;
17 $\hat{\Phi}_{cur} \leftarrow \hat{\Phi}_{cur} + \Delta_{\hat{\Phi}}$;
18 **if** $\hat{\Phi}_{cur} \leq \hat{\Phi}_{opt}$ **then** $t \leftarrow t - 1$;
19 **else** $\hat{\Phi}_{opt} \leftarrow \hat{\Phi}_{cur}$, $t \leftarrow t + T$;
20 Undo the last non-progressive operations on S;
21 **return** $\overline{H} = G[S], \hat{\Phi} = \hat{\Phi}_{opt}$;

- `ColorFlip`(x): Let c be the color of x and $\overline{c}$ be c's opposite color. Execute `VertexDelete`(x) and `VertexInsert`$(x, \overline{c})$ in order.

Since our search begins from s, we initialize S to be $\{(s,0)\}$ (line 1). We use $\hat{\Phi}_{opt}$ and $\hat{\Phi}_{cur}$ to store the maximum $\hat{\Phi}$ that we have found and the current $\hat{\Phi}$ respectively. Initially, they are initialized to be 0 (line 1). During our process, if we choose to execute `VertexInsert`(x, c) or `ColorFlip`(x) for some vertex x, we will greedily execute one with the maximal increment of $\hat{\Phi}$ that it can contribute. Therefore, we use two max-heaps MH_I, MH_F to assist. After being initialized (line 2), since s is selected, for each neighbor v of s, we calculate the corresponding contribution to $\hat{\Phi}$ when `VertexInsert`$(v, 0)$ or

`VertexInsert`$(v, 1)$ are executed and insert them to MH_I (line 4 to 5). For MH_F, since only s is selected, we calculate the contribution for `ColorFlip`(x) and insert it into MH_F (line 3).

Here, we do not use any structure to store the contribution when deleting any vertex from S. Instead, whenever we want to find the optimal deletion, we can compute for every vertex in S altogether with a total cost of $O(n)$, which we may need to use the well-known Tarjan algorithm [109] to guarantee the subgraph stays connected after the deletion. We denote such process as `DelEval`(S).

Our search method is to execute one of the three operations repeatedly. If we only consider inserting vertices from S's neighbors, the total number of `VertexInsert` is bounded by the size of the graph. However, since we also have the other two operations, which are *non-incremental*, if we do not design a proper termination strategy, the time complexity might become exponential. Specifically, we have two parameters that help to define the termination strategy of the algorithm: A float number $p \in (0, 1)$ denotes the *non-incremental probability* and an integer T to limit the number of potentially wasted operations.

Operation Selection We first discuss how to select an operation each time. Here, we will use the non-incremental probability p to help determine. We use *op* to denote the operation we select and $\Delta_{\hat{\Phi}}$ to denote the corresponding contribution value to $\hat{\Phi}$. These two variables are initialized by acquiring the best `VertexInsert` operation from MH_I (line 7), indicating choosing an insertion. Then, we generate a random float number z from uniform distribution $U(0, 1)$. If $z < p$, we also consider using a `ColorFlip` operation: We acquire the best `ColorFlip` operation from MH_F and update the two variables if the corresponding $\Delta_{\hat{\Phi}}$ is larger than the current one (line 8 to 9). Similarly, we regenerate z and if $z < \frac{p \log |S|}{|S|}$, we try choosing a `VertexDelete` operation: After calling `DelEval`(S) to recalculate for every possible vertex deletion, we choose the best among them and try updating (line 10 to 12). In this way, we choose an `VertexInsert` operation by default, and with some probability, we check if the current optimal `ColorFlip` and `VertexDelete` can contribute more. The two probabilities are by design and help to balance between accuracy and efficiency.

We execute the current *op* operation after the selection phase and also update the current $\hat{\Phi}_{cur}$. Since S is updated after the execution, we must also update MH_I and MH_F correspondingly. This is done in a similar way as the initialization (line 13 to 17).

Search Termination When the current selected operation enlarges the current $\hat{\Phi}_{cur}$, we denote such operation as a *progressive* one. Otherwise, it is *non-progressive*. We use a

parameter T to prevent too many *non-progressive* operations. Specifically, our search will terminate when the number of *non-progressive* operations exceeds T times the number of *progressive* operations. We implement such a strategy with a counter t. After we select and execute an operation (line 7 to 17), we compare the current $\hat{\Phi}_{cur}$ with $\hat{\Phi}_{opt}$. If $\hat{\Phi}_{cur}$ is smaller, we decrease t by 1. Otherwise, we update $\hat{\Phi}_{opt}$ and increase t by T (line 18 to 19). In this way, if $t < 0$, the search should terminate (line 6). In addition, we also terminate the search when S contains all vertices in G since no further insertion can be executed (line 6). After the search, we undo the last few non-progressive operations to retract the optimal subgraph we have found (line 20). We return with this subgraph as $\overline{H}$ and its corresponding $\hat{\Phi}$ (line 21).

4.3.3 Region-based Sampling

In the previous section, we propose a search process that starts from an arbitrary vertex s. It is reasonable to foresee that the choice of s might affect the result significantly. If we start only from too few vertices, our result in the end might be some local maximal solutions, which would be much worse than the global maximal one. One of the solutions is to enumerate all possible s, *i.e.*, all vertices in G. However, such pure enumeration may result in excessively high time complexity. To balance between performance and efficiency, we propose a *Region-based Sampling* strategy.

Our sampling method is mainly based on two hypotheses. The first hypothesis indicates that the probability of finding a nearly optimal subgraph is high if we are able to select some vertices in the optimal subgraph as the starting vertex. Here, the 'optimal' denotes the solution we found by enumerating all vertices as the starting vertex. We formally state such a hypothesis as follows.

Hypothesis 4.3.1. *Given a signed graph G and a tolerance parameter β, suppose the optimal subgraph found by Algorithm 32 starting with vertex x is G_x, and the optimal graph among all G_x is G_{opt}. For the given ϵ, there exists a subset $V' \subseteq V(G_{opt})$ with $\frac{|V'|}{|V(G_{opt})|} \geq \frac{1}{2}$ such that $\Phi(G_x, \beta) \geq (1-\epsilon)\Phi(G_{opt}, \beta), \forall x \in V'$.*

Another hypothesis describes the relation between $\hat{\Phi}$ values returned by two different calls of Algorithm 32, if we select different starting vertex. We argue that if the return $\hat{\Phi}$ value is larger, the found subgraph will likely be bigger. We formally state such a hypothesis as follows.

Hypothesis 4.3.2. *Given a signed graph G and a tolerance parameter β, suppose the optimal subgraph and coloring found by Algorithm 32 starting with vertex a are G_a and $\mathcal{X}_a$, starting with b are G_b and $\mathcal{X}_b$. If $\hat{\Phi}(G_a, \beta, \mathcal{X}_a) \geq \hat{\Phi}(G_b, \beta, \mathcal{X}_b)$, we have $|V(G_b)| \leq 2|V(G_a)|$.*

With these two hypotheses, we have the following lemma that describes a sampling strategy that is able to find a $(1-\epsilon)$-optimal subgraph within an acceptable number of calls of the search process.

Lemma 4.3.2. *Given a signed graph G, a tolerance parameter β, and a positive $\epsilon < 1$, suppose we run Algorithm 32 in k iterations, where the i-th iteration starts with a uniformly sampled vertex $x_i \in V(G)$, and the optimal subgraph found is G_i. If* Hypothesis *4.3.1 and* Hypothesis *4.3.2 hold, the expected number of iterations that find a $(1-\epsilon)$-optimal subgraph $\mathbb{E}[X]$ is $\Omega(\sum_{i=1}^{k} |V(G_i)|/|V(G)|)$.*

Proof of lemma 4.3.2. Suppose the optimal subgraph found by Algorithm 32 among all starting vertex is G_{opt} and the probability of Algorithm 33 finding a $(1-\epsilon)$-optimal subgraph in one iteration is Pr[Sucess]. By Hypothesis 4.3.1, we have *(i)* $\Pr[\text{Sucess}] \geq \frac{|V(G_{opt})|}{2|V(G)|}$. Let $\mathcal{C} = \frac{\sum_{i=1}^{k}|V(G_i)|}{|V(G)|}$, and then we have $\sum_{i=1}^{k}|V(G_i)| = \mathcal{C}|V(G)|$. By Hypothesis 4.3.2, $\sum_{i=1}^{k} 2|V(G_{opt})| \geq \mathcal{C}|V(G)|$, and thus *(ii)* $k \geq \frac{\mathcal{C}}{2}\frac{|V(G)|}{|V(G_{opt})|}$. Combining *(i)* and *(ii)*, we have $\mathbb{E}[X] = k\Pr[\text{Sucess}] \geq \frac{\mathcal{C}}{2}\frac{|V(G)|}{|V(G_{opt})|} \cdot \frac{|V(G_{opt})|}{2|V(G)|} = \frac{\mathcal{C}}{4} = \Omega(\mathcal{C})$. □

We provide an implementation of such sampling strategy in Algorithm 33, which is, in fact, an application of Lemma 4.3.2. We use H, $\hat{\Phi}_{opt}$ to keep track of the current optimal subgraph and its corresponding $\hat{\Phi}$ value. They are initialized to be $\emptyset$ and 0 in the beginning (line 1 to 2). We also use a variable $TotalSize$ to keep track of the total size of all return subgraphs from Algorithm 32, which is also initialized to be 0 (line 3).

For each iteration, we randomly select a vertex s (line 5) as the starting vertex and pass it into Algorithm 32 (line 6). After receiving the result from Algorithm 32, we update H, $\hat{\Phi}_{opt}$ (line 8 to 9) if the newly found subgraph is better (line 7). Before the new iteration, we accumulate the size of the newly found subgraph into $TotalSize$ (line 10. The whole process will stop when $TotalSize \geq \mathcal{C}|V(G)|$ (line 4). By Lemma 4.3.2, we can set a proper termination condition by accumulating the subgraph size from each search process. More specifically, When $TotalSize$ reaches $\mathcal{C}|V(G)|$, it is expected to find a nearly optimal solution $\mathcal{C}$ times. In the end, we return H as the main result (line 11).

Algorithm 33: RH: Sampling

Input: Signed graph G; Tolerance parameter β; Iteration constant $\mathcal{C}$; Non-incremental probability p; Early stop turn limit T.

Output: $H \subseteq G$: the found balanced subgraph with β-tolerance.

```
1  H ← ∅;
2  Φ̂_opt ← 0;
3  TotalSize ← 0;
4  while TotalSize < C|V(G)| do
5      s ← Sample a vertex from V(G);
6      H̄, Φ̂ ← Search(G, β, s, p, T);
7      if Φ̂ > Φ_opt then
8          Φ̂ ← Φ̂_opt;
9          H ← H̄;
10     TotalSize ← TotalSize + |V(H̄)|;
11 return H;
```

Time Complexity Analysis We show that the time complexity of Algorithm 33 is $O(\Delta(\mathcal{C}+1)n\log^2 n)$, where n is the number of vertices in G. We start with the time complexity of Algorithm 32.

Lemma 4.3.3. *Suppose that the loop starting on Line 6 repeats for a total of t times. Let h be the size of S on Line 21. Then we have, with high probability, $h = \Omega(t/\log n)$ and that the running time of Algorithm 32 is $O(\Delta t \log n)$ in expectation.*

To prove Lemma 4.3.3, we first need the following lemma:

Lemma 4.3.4. *With high probability, for all valid i, after the i-th iteration of the on Line 6 of Algorithm 32, $|S| = \Omega(\frac{i}{\log n})$.*

Proof. Let x denote the value of $|S|$. The initial value of x is 1. x changes according to the following rules in each iteration:

- With probability $(1-p)(1-p\frac{\log x}{x})$, x increases by 1;
- With probability $p\frac{\log x}{x}$, x increases by 1, 0, or -1;
- With probability $p(1-p\frac{\log x}{x})$, x increases by 1, or 0;
- The process ends when x reaches n.

Suppose the process lasts for t steps. It suffices to prove that with high probability, it holds simultaneously for all i from 1 to t that the value of x after the first i steps is $\Omega(\frac{i}{\log n})$.

We denote $(1-p)(1-p\frac{x}{\log x})$ by $p^+(x)$, $p(1-p\frac{x}{\log x})$ by $p^0(x)$, and $p\frac{x}{\log x}$ by $p^-(x)$. Let $next(x)$ be the value of x in the next step. Let x_0 be the smallest x such that $p^-(x) < p^+(x)/2$. Such x_0 exists because $p^-(x) \to 0$ and $p^+(x) \to 1-p$ when $x \to \infty$. For any $x \geq x_0$, $\mathbb{E}[next(x)] \geq x + \frac{1-p}{3-p}$. We have that x_0 is a constant depending on p. Let c be the maximum expected number of steps for the variable x to reach x_0 from any initial value $x_{init} < x_0$. c is a constant depending on p.

By Markov bound, we have that in every $2c$ steps, with probability at least $1/2$, there exists at least one step before which x is at least x_0. Thus, in $k = \Omega(\log n)$ steps, with high probability, there exists $\Omega(k)$ steps before which the value of x is at least x_0. In these $\Omega(k)$ steps, x increases by expectation value at least $\frac{1-p}{3-p}\Omega(k) = \Omega(k)$. The rest of the k steps can be partitioned into consecutive segments. In each segment, x increases from some value less than x_0 to x_0. The only exception is the last segment where x may decrease by at most x_0. Thus, in expectation, after the k steps, the value of x is at least $\Omega(k)$.

We may repeat the argument above for $O(\log n)$ rounds to boost the probability. After $\Omega(k \log n)$ steps, the value of x is $\Omega(k)$ with high probability. By plugging in $k = n$, we have that the process ends in $t = O(n \log n)$ steps with high probability. By a union bound, we can prove that the value of x after the first i steps being $\Omega(i/\log n)$ holds simultaneously for all i from 1 to t with high probability. □

Now we are ready to prove Lemma 4.3.3.

Proof of Lemma 4.3.3. By Lemma 4.3.4, with high probability, $h = \Omega(t/\log n)$. Let Δ be the maximum degree of G. The running time of each iteration is dominated by the cost of updating the heaps and the time for `DelEval(`S`)`. Updating the heaps cost $O(\Delta \log n)$ time. The function `DelEval(`S`)` costs $O(\Delta|S|)$ time. It is called with probability $\frac{p \log |S|}{|S|}$ in each iteration. Thus, its expected runtime in each iteration is $O(\Delta \log n)$. We have the running time of Algorithm 32 is $t \cdot O(\Delta \log n) = O(\Delta t \log n)$. □

Lemma 4.3.5. *With high probability, the expected time complexity of Algorithm 33 is* $O(\Delta(\mathcal{C}+1)n \log^2 n)$.

Proof. Suppose we call Line 6 for k times. For the i-th call, let t_i be the number of iterations of the loop on Line 6 and let h_i $(1 \leq i \leq k)$ be the size of S on Line 21. By Lemma 4.3.3, the total time complexity in expectation is $O(\sum_{i=1}^{k} \Delta t_i \log n)$, and the sum of h_i satisfies $\sum_{i=1}^{k} h_i = \Omega(\sum_{i=1}^{k} t_i/\log n)$. Since $\sum_{i=1}^{k} h_i \leq (\mathcal{C}+1)n$, we have that the total runtime of Algorithm 33 in expectation is $O(\sum_{i=1}^{k} \Delta t_i \log n) = O(\Delta(\mathcal{C}+1)n \log^2 n)$. □

4.4 Experiments

In this section, we address the following research questions to evaluate various important aspects of our algorithm:

- **RQ1 (Effectiveness - TMBS):** Given various β, what are the optimal subgraphs in terms of size, and Tolerant Index Count, found by our method and baselines?
- **RQ2 (Effectiveness - MBS):** What are our method's and baselines' performances in finding the maximal balanced subgraph?
- **RQ3 (Efficiency):** What are the runtimes for our method and the baselines, and how do they scale with very large networks?
- **RQ4 (Generalizability):** Can our model and method be adapted to other related tasks in the signed networks?

We also conduct three other experiments, which are designed to validate our hypotheses, determine optimal hyperparameters, and assess stability.

4.4.1 Experimental Setting

Baselines We compare our method with the baselines from highly related works (*e.g.,* MBS and polarized community detection), including spectral and other heuristic methods.

Note that since finding tolerant balanced subgraphs is typically more challenging than previous community detection tasks in signed networks, we also adapt our TBC-relaxation in Section 4.3.1 for all baseline methods to ensure that they can return the subgraphs satisfying the tolerance constraint. We summarize the core ideas of each baseline method as follows.

- Eigen: Based on a spectral method from [36], we first compute $\mathbf{v}$, the eigenvector of the signed adjacency matrix A corresponding to the largest eigen value λ. For each vertex i, if a Bernoulli experiment with success probability $p = |\mathbf{v}_i|$ is successful, we assign vertex i a color determined by $\text{sgn}(\mathbf{v}_i)$. The maximum connected components with non-negative $\hat{\Phi}$ serve as solutions to Problem 6 and Problem 7, while the connected component with the maximum $\hat{\Phi}$ provides the solution to Problem 8.

- TIMBAL: Based on a state-of-the-art method for MBS from [168], we start from the entire graph and repeatedly remove vertices from it, guided by an eigenvalue approximation outlined in their paper. Specifically, we compute $\hat{\Phi}$ by the optimal coloring among two non-trivial methods [36, 100] for TMBS problems.

- GRESt: This method combines a classical and effective algorithm in dense subgraph [51] and a heuristic coloring method for signed networks by spanning tree [100]. Firstly, we determine the coloring of the entire graph by a random spanning tree. We then repeatedly remove vertices maximizing $\hat{\Phi}$ of the remaining graph. A min-heap is used to speed up the process of finding vertices. To acquire the solution to different tasks, we need to store the deletion operations. Then we can restore the graph by undoing deletions one by one. Such reversal allows us to keep track of the optimal connected component efficiently, instead of scanning the whole graph after each deletion.

- RH (LS): Instead of using region-based sampling, we execute our local search in Section 4.3.2 for all starting vertices and select the optimal solutions, to investigate the effectiveness of the regional-based sampling. This method may provide a better solution than RH but is not efficient.

Datasets We select 7 publicly-available real-world signed networks[3], BITCOIN, EPINIONS, SLASHDOT, TWITTER, CONFLICT, ELECTIONS, and POLITICS, which were widely used in previous related works [168, 36, 165]. In addition, to investigate our methods' scalability, we generate GROWTH, a very large signed network induced from the real temporal network WIKIPEDIA-GROWTH[4]. Specifically, we select a threshold τ, and give positive signs for the edges with time stamp $t(e) \geq \tau$ and negative signs for the edges with $t(e) < \tau$. By using a proper τ, the ratio of negative edges of the induced graph can be 0.5. In this network, a balanced graph contains two communities, where the edges within each community are recently formed, while the crossing edges are relatively old. Detailed information for each dataset can be found in Table 4.1.

All experiments are conducted on a Ubuntu 22.04 LTS workstation, equipped with a 12th Gen Intel(R) Core(TM) i9-12900HX. We set hyperparameters $T = 20, p = 0.8, \mathcal{C} = 1.5$ for all experiments.

[3]From konect.cc and snap.stanford.edu

[4]http://konect.cc/networks/wikipedia-growth

Table 4.1: Signed network datasets used in experiments, including the number of vertices ($|V|$) and edges ($|E|$), the ratio of negative edges (ρ^-), and the ratio of non-zero elements (δ).

Dataset	$\lvert V\rvert$	$\lvert E^+ \cup E^-\rvert$	$\rho^- = \frac{\lvert E^-\rvert}{\lvert E^+ \cup E^-\rvert}$	$\delta = \frac{2\lvert E^+ \cup E^-\rvert}{\lvert V\rvert(\lvert V\rvert-1)}$
Bitcoin	5k	21k	0.15	1.2e-03
Epinions	131k	711k	0.17	8.2e-05
Slashdot	82k	500k	0.23	1.4e-04
Twitter	10k	251k	0.05	4.2e-03
Conflict	116k	2M	0.62	2.9e-04
Elections	7k	100k	0.22	3.9e-03
Politics	138k	715k	0.12	7.4e-05
Growth	1.87M	40M	0.50	2.3e-05

4.4.2 Finding Balanced Subgraphs with Tolerance

We aim to identify sizable and polarized subgraphs while taking the tolerance into account (**RQ1**). Firstly, by running algorithms in Section 4.4.1, we keep track of the optimal subgraph that we have found to Problem 6, Problem 7, and Problem 8 respectively. Noticeably, we do not modify either RH or RH (LS) for Problem 6 and Problem 7. That is to say, we directly use our result on solving Problem 8 to compare with other algorithms which are tailored for either Problem 6 or Problem 7, which demonstrates our proposed Problem 8 is a good approximation for the other two problems.

The comparison results for these three problems are shown in Figure 4.3, Figure 4.4, and Figure 4.5, respectively. In our experiments, we consider 15 different tolerance parameters ($\beta_i = 2^{-i/2}$ for all $2 \leq i \leq 16$), where $i = 2$ implies allowance for all connected subgraphs, while $i = 16$ means that the found subgraphs are almost strictly balanced. Since Problem 6 and Problem 7 become trivial when $\beta \geq \frac{L(G)}{|E^+ \cup E^-|}$, we shade the corresponding ranges[5] in Figure 4.3 and Figure 4.4. We omit the results for methods that cannot finish in 100,000 seconds.

Observing the experiment results, our proposed method RH outperforms all other baselines significantly in all three problems. This also demonstrates that our proposed tolerance model is a more realistic model for the general case of balanced signed graph models. In addition, we can see that RH produces results that are close to RH (LS). Therefore, we manage to empirically support our hypothesis in Section 4.3.3 in real-world data.

[5]As computing the exact value of $L(G)$ is **NP**-hard, we cannot calculate the exact range of β corresponding to the trivial cases. Instead, we calculate an upper bound $\overline{L}$ of $L(G)$ by a promising valid solution and shade only the subrange $\beta \geq \frac{\overline{L}}{|E^+ \cup E^-|}$.

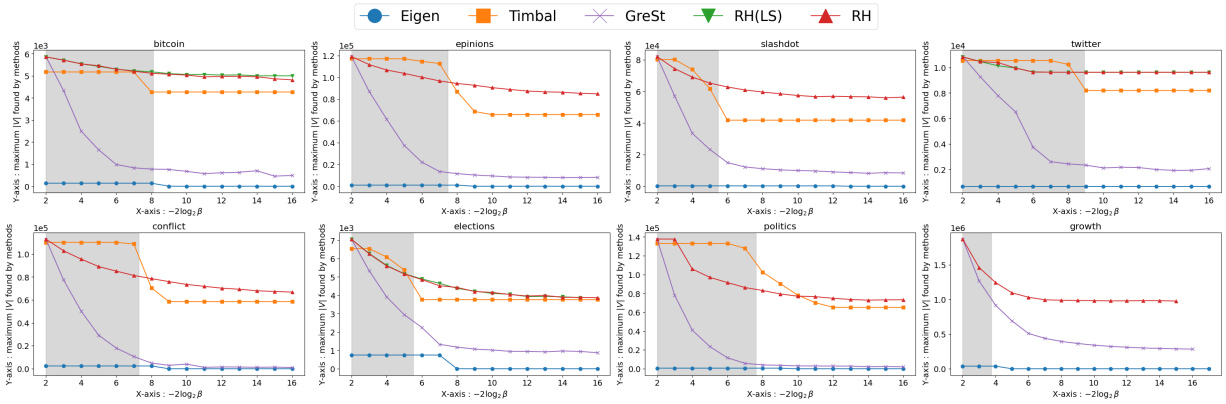


Figure 4.3: PROBLEM 6: Comparing maximum tolerantly balanced subgraphs in vertex cardinality ($|V|$) across various tolerances ($\beta = 2^{-x/2}$), where the part corresponding to trivial β values for this problem has been shaded.

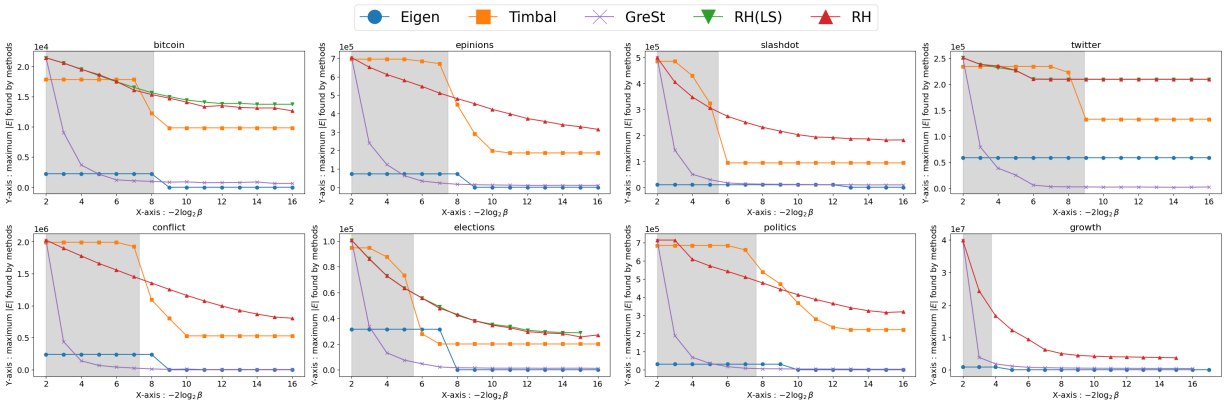


Figure 4.4: PROBELM 7: Comparing maximum tolerantly balanced subgraphs in edge cardinality ($|E^+ \cup E^-|$) across various tolerances ($\beta = 2^{-x/2}$), where the part corresponding to trivial β values for this problem has been shaded.

4.4.3 Finding Strictly Balanced Subgraphs

Finding the Maximum Balanced Subgraph (Problem 4 and 5) is an important and well-studied problem in signed networks [168] (**RQ2**). As previously discussed, MBS problems are special forms of proposed TMBS problems (Problem 6 and 7) by setting $\beta < \frac{1}{|E^+ \cup E^-|}$.

Table 4.2 shows the results of our case study on the MBS problems, where we omit the results for EIGEN since it can not return subgraphs other than a single vertex. RH produces significantly better results than the previous state-of-the-art method TIMBAL,

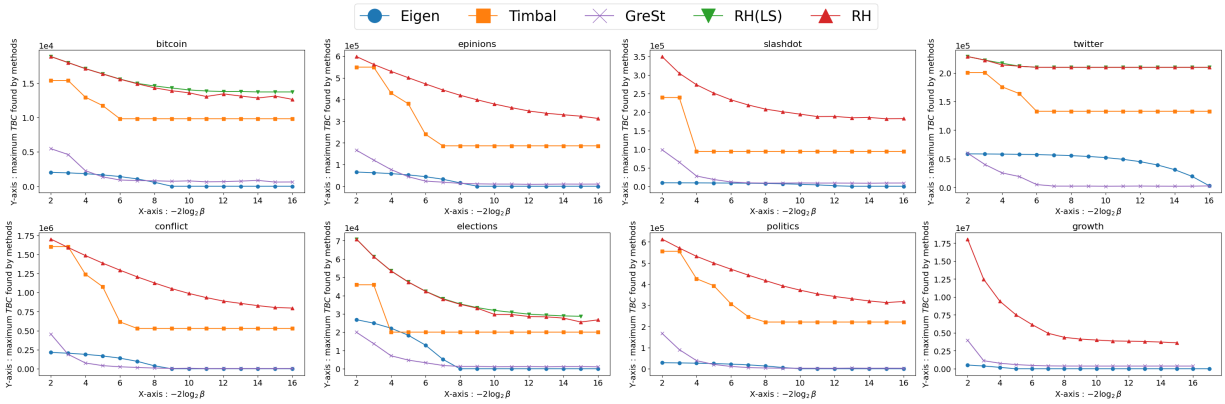


Figure 4.5: PROBELM 8: Comparing maximum tolerantly balanced subgraphs in TBC ($\hat{\Phi}$) across various tolerances ($\beta = 2^{-x/2}$).

which is specifically designed for the MBS problems while ours is not.

Table 4.2: Case study: Largest strictly balanced subgraph (in terms of $|V|$ or $|E|$) found by each method for each dataset ($\beta < \frac{1}{|E^+ \cup E^-|}$), where *NA* denotes the corresponding method cannot finish in 100,000 seconds on the respective dataset.

Method	BITCOIN		EPINIONS		SLASHDOT		TWITTER		CONFLICT		ELECTIONS		POLITICS		GROWTH	
	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
TIMBAL	4,050	9,757	65,879	190,751	43,742	101,590	8,636	129,927	51,463	361,663	3,579	17,633	65,188	230,529	*NA*	*NA*
GREST	571	713	7,078	9,229	7,923	8,465	1,895	2,277	759	4,885	935	1,232	2,371	3,155	272,621	375,603
RH (LS)	**5,002**	**13,746**	*NA*	*NA*	*NA*	*NA*	**9,628**	**209,633**	*NA*	*NA*	**3,970**	**28,453**	*NA*	*NA*	*NA*	*NA*
RH	4,935	13,050	**84,165**	**302,152**	**55,968**	**181,069**	**9,628**	**209,633**	**65,468**	**746,640**	3,926	26,478	**72,431**	**317,384**	**999,504**	**3,415,441**

4.4.4 Running Time Analysis

This experiment is designed to study the efficiency and scalability of our method (**RQ3**). As shown in Table 4.3, when the graph size is up to ($|V| = 1.87$M, $|E^+ \cup E^-| = 40$M), RH can still produce reasonable results within 1,000 seconds. On the other hand, the other four compared methods either produce much worse results while taking longer time or even fail to finish execution in 100,000 seconds. In all, the empirical results demonstrate the efficiency of our method in real-world data in addition to its asymptotic theoretical complexity.

Table 4.3: Mean running time (s) for each method on the various β in Section 4.4.2.

Dataset	EIGEN	TIMBAL	GREST	RH (LS)	RH
BITCOIN	0.979	2.670	0.018	135.040	0.057
EPINIONS	9.213	220.919	0.319	$\geq 100,000$	4.162
SLASHDOT	7.287	226.805	0.218	$\geq 100,000$	3.175
TWITTER	1.704	7.306	0.085	2,516.671	0.493
CONFLICT	12.899	310.736	0.781	$\geq 100,000$	9.031
ELECTIONS	1.022	4.542	0.039	675.484	0.288
POLITICS	10.003	143.605	0.304	$\geq 100,000$	4.782
GROWTH	297.299	$\geq 100,000$	48.915	$\geq 100,000$	813.099

4.4.5 Solving the 2PC Problem

This experiment aims to evaluate the generalizability when adapting RH and our tolerant model to different variants of polarity community detection (**RQ4**). In addition to balance-related problems, 2-POLARIZED-COMMUNITIES (2PC), proposed by Bonchi and Galimberti [36], serves as another model for polarized community detection. In their model, the measurement of polarity is penalized by the size of the solution:

Problem 9 (2PC)**.** *Given a signed graph $G = (V, E^+, E^-)$ with signed adjacency matrix A, find a vector $x \in \{-1, 0, 1\}$ that maximizes $\frac{x^T A x}{x^T x}$.*

To solve the 2PC problem by our tolerance balance model, we add an additional penalty term to the solution size in the tolerant balance count: We define $\hat{\Phi}'(G, \beta, \mathcal{X}, \rho) = \hat{\Phi}(G, \beta, \mathcal{X}) - \rho|V|$ to serve as a new object function in our RH algorithm. Throughout the algorithm, we set $\beta = \frac{1}{2}$. This is because when $\beta = \frac{1}{2}$, if $\hat{\Phi}' \geq 0$, the polarity is no less than ρ. Therefore, PROBLEM 9 can be solved by the new RH algorithm, where we apply an iterative mechanism on ρ.

To compute solutions for the 2PC problem (Problem 9), we design an iterative mechanism (Algorithm 34) for calling the search problem (Algorithm 32) properly, replacing Algorithm 33.

H is for storing the current optimal subgraph, initialized as $\emptyset$ (line 1). Our iterative strategy is related with the current optimal polarity value ρ. We initialize it as 0 (line 1) in the beginning. We pick a starting vertex s at first, sampling from all vertices in $V(G)$ (line 2). Each turn, we execute a search process with specific hyperparameters to try to find a new candidate answer (line 4). We chose the tolerance parameter β to be $\frac{1}{2}$. Since our new modified object function is $\hat{\Phi}' = \hat{\Phi} - \sigma|S|$, we also need to pass in σ as 0.9ρ. The search process will have to return the corresponding polarity $\hat{\rho}$ along with the found

Algorithm 34: RH: Iterative mechanism for 2PC

Input: Signed graph G;
Output: $H \subseteq G$: the found subgraph with high 2PC polarity; ρ : the 2PC polarity of H;

```
1  H ← ∅;  ρ ← 0;
2  s ← Sample a vertex from V(G);
3  while true do
4      H̄, ρ̂ ← Search(G, β = 1/2, σ = 0.9ρ, s);   // guided by the new object function
         Φ̂′ = Φ̂ − σ|S|.  hyperparameters T and p are omitted.
5      if ρ̂ > ρ then
6          ρ ← ρ̂;
7          H ← H̄;
8      else s ← Sample a vertex from V(G) ;
9      if meets stop-conditions then break;
10     remove all vertice with degree less than ρ from G;
11 return H, ρ;
```

subgraph. We compare $\hat{\rho}$ with the previous found ρ. If it is better, we update ρ and H correspondingly (line 5 to 7). Note that we do not sample a new starting vertex each time. Instead, we also do so when the search process does not return a better result (line 8). We regard this as a sign showing that we have reached the limit of the current chosen starting vertex. The whole process will stop when a certain condition is met (line 9). The detail is omitted and can be referred to in our provided implementation. There are a few optimizations of such iteration mechanisms that can help improve the performance. Note that if the current polarity is ρ, for any vertex with a degree less than ρ, removing them will not worsen the answer. Therefore, we remove all such vertices from G after the ρ gets updated (line 10).

We compare RH's results with the optimal result computed by the two methods in Bonchi and Galimberti's paper [36] (*i.e.,* Eigen and Greedy). The results are shown in Table 4.4. RH's slight modifications efficiently yield promising communities, demonstrating the adaptability and effectiveness of our tolerant balance model and algorithm in varied polarity community detection scenarios.

4.4.6 Statistical Hypothesis Testing

In our sampling strategy and performance analysis, we propose two hypotheses: 4.3.1 and 4.3.2. Here, we conduct some experiments on 3 datasets and 4 different β to verify if these

Table 4.4: Optimal subgraphs in the term of 2PC-polarity ($\frac{x^T Ax}{x^T x}$) found by each method, associated with running time.

Dataset	RH		Eigen		Greedy	
	polarity	time (s)	polarity	time (s)	polarity	time (s)
Bitcoin	**14.82**	**0.035**	14.76	0.706	14.50	0.999
Epinions	**85.27**	**0.873**	64.36	10.271	85.15	428.625
Slashdot	41.21	**0.584**	39.85	7.325	**41.36**	154.383
Twitter	**87.21**	**0.818**	87.04	1.717	86.97	7.853
Conflict	**94.27**	15.134	87.83	**13.144**	63.99	552.629
Elections	36.27	**0.395**	35.87	0.950	**36.34**	2.223
Politics	44.95	**0.862**	44.22	9.931	**45.01**	475.353

hypotheses are reasonable.

For Hypothesis 4.3.1, we compute the maximum $1-\epsilon$ such that there exists a subset $V' \subseteq V(G_{opt})$ with $\frac{|V'|}{|V(G_{opt})|} \geq \frac{1}{2}$ such that $\Phi(G_x, \beta) \geq (1-\epsilon)\Phi(G_{opt}, \beta), \forall x \in V'$. The result is shown in Table 4.5. The hypothesis is supported by the fact that all values are very close to 1 and not sensitive to the value of β, which indicates ϵ's lower bound approaches to 0.

Table 4.5: Verifing Hypothesis 4.3.1: For each dataset, we compute the maximum $1-\epsilon$ such that there exists a subset $V' \subseteq V(G_{opt})$ with $\frac{|V'|}{|V(G_{opt})|} \geq \frac{1}{2}$ such that $\Phi(G_x, \beta) \geq (1-\epsilon)\Phi(G_{opt}, \beta), \forall x \in V'$ under various β.

Dataset	$\beta = \frac{1}{4}$	$\beta = \frac{1}{8}$	$\beta = \frac{1}{16}$	$\beta = \frac{1}{32}$
Bitcoin	1.000	0.999	0.982	0.948
Twitter	0.987	1.000	1.000	1.000
Elections	0.999	0.997	0.987	0.946

To verify Hypothesis 4.3.2, we run Algorithm 32 from every possible starting vertex s (similar to RH (LS) in Section 4.4). For each setting, we compute max $\frac{|V(G_b)|}{|V(G_a)|}$ for any two starting vertices a, b satisfying $\hat{\Phi}(G_a, \beta, \mathcal{X}_a) \geq \hat{\Phi}(G_b, \beta, \mathcal{X}_b)$. The result is shown in Table 4.6, where all values are very close to 1 and far from the bound of 2 in the hypothesis. Therefore, Hypothesis 4.3.2 is also empirically supported.

Table 4.6: Verifing Hypothesis 4.3.2: For each dataset, we compute max $\frac{|V(G_b)|}{|V(G_a)|}$ for any two starting vertices a, b satisfying $\hat{\Phi}(G_a, \beta, \mathcal{X}_a) \geq \hat{\Phi}(G_b, \beta, \mathcal{X}_b)$ under various β.

Dataset	$\beta = \frac{1}{4}$	$\beta = \frac{1}{8}$	$\beta = \frac{1}{16}$	$\beta = \frac{1}{32}$
Bitcoin	1.018	1.014	1.020	1.037
Twitter	1.053	1.111	1.094	1.084
Elections	1.009	1.032	1.040	1.061

4.4.7 Hyperparameter Analysis

For the three hyperparameters p, C, T in Algorithm 33, we conduct a series of experiments to demonstrate that our choice is natural, shown in Figure 4.6.

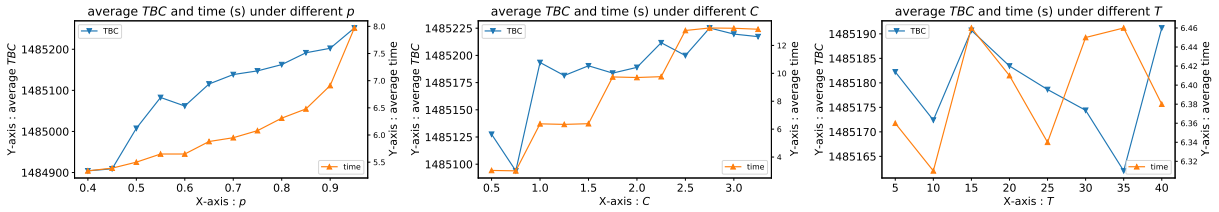


Figure 4.6: We study the influence of hyperparameters (*i.e.,* Non-incremental probability p, Iteration constant $\mathcal{C}$ and Early stop turn limit T) on CONFILICT dataset with $\beta = 1/4$: For each hyperparameter, we execute the algorithm when setting the hyperparameter as various values while keeping other hyperparameters fixed. The results include the average Tolerant Balance Count (TBC) and running time in 25 rounds.

For each hyperparameter, we fix the other two unchanged as our option in Section 4.3 and enumerate its value in a range. For each value, we compute the average runtime and TBC over 25 times the executions of solving on the CONFILICT with setting $\beta = \frac{1}{4}$. A larger p may lead to more non-incremental operations, thus a longer runtime is expected. Similarly, a larger C leads to more executions of the search process. Therefore, as p, C gets larger, the runtime and the average TBC might be getting larger as well. This can be observed from the experiment, shown in Figure 4.6.

On the other hand, although such patterns can be observed, the difference between average runtime and TBC is not significant for all three hyperparameters. This indicates that our algorithm is robust and does not tune towards the dataset.

4.4.8 Stability Study

In our search process (Algorithm 32), we will execute either vertex deletion or color flipping in a certain probability p. We conduct some experiments to study how such non-incremental operations help improve the stability of our algorithm, shown in Table 4.7. We modify the algorithm RH to a version with only insertion operations, denoted as RH (INSERTION ONLY). On all datasets, we execute both algorithms for 100 rounds. As common, we compute the variance σ^2 to measure the stability. Here, for each dataset, we also compute the maximum ($\max \hat{\Phi}$), minimum ($\min \hat{\Phi}$), and average (μ) values for tolerant

balance count (TBC). As shown in Table 4.7, we can see with these two operations, the results are more stable and consistent, which can be observed in the rightmost column.

Table 4.7: Stability study of Tolerant Balance Count (TBC) under $\beta = 1/8$ as example: We conduct 100 rounds of RH and RH (INSERTION ONLY) on each dataset. The results include maximum TBC ($\max \hat{\Phi}$), minimum TBC ($\min \hat{\Phi}$), mean TBC (μ), and the variance (σ^2) across the 100 rounds. Additionally, we present the variance reduction achieved by considering *non-incremental operations*.

Dataset	**RH (insertion only)**				**RH**				**Variance reduction**
	$\min \hat{\Phi}$	$\max \hat{\Phi}$	μ	σ^2	$\min \hat{\Phi}$	$\max \hat{\Phi}$	μ	σ^2	
BITCOIN	14,864	15,523	15,499	4,977	15,578	15,619	15,604	25	**198**×
EPINIONS	462,237	471,120	470,536	1,301,027	472,158	472,240	472,195	315	**4,129**×
SLASHDOT	223,054	232,757	232,110	1,564,480	231,988	233,149	232,957	17,521	**88**×
TWITTER	187,549	209,701	208,051	30,265,009	197,072	209,701	209,097	4,455,495	**5.7**×
CONFLICT	1,284,391	1,289,332	1,288,405	430,597	1,292,253	1,293,741	1,293,318	84,706	**4**×
ELECTIONS	40,819	42,187	41,981	33,946	42,206	42,358	42,276	1,031	**29**×
POLITICS	470,347	470,764	470,576	4,289	470,534	470,905	470,642	4,042	**0.06**×

References

[1] Robert P Abelson and Milton J Rosenberg. Symbolic psycho-logic: A model of attitudinal cognition. *Behavioral Science*, 1958. 137

[2] Gene Abrams and Roozbeh Hazrat. Connections between Abelian sandpile models and the k-theory of weighted leavitt path algebras. *European Journal of Mathematics*, 9(2):21, 2023. 7

[3] Amit Agarwal, Moses Charikar, Konstantin Makarychev, and Yury Makarychev. $O(\sqrt{\log n})$ approximation algorithms for min uncut, min 2cnf deletion, and directed cut problems. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 573–581, 2005. 137, 139

[4] William Aiello, Fan Chung, and Linyuan Lu. A random graph model for massive graphs. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 171–180, 2000. 107, 111

[5] Miklós Ajtai, János Komlós, and Endre Szemerédi. Deterministic simulation in logspace. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 132–140, 1987. 16

[6] S Akbari, F Heydari, and M Maghasedi. The intersection graph of a group. *Journal of Algebra and its Applications*, 14(05):1550065, 2015.

[7] Jin Akiyama, David Avis, Vasek Chvátal, and Hiroshi Era. Balancing signed graphs. *Discrete Applied Mathematics*, 3(4):227–233, 1981. 136

[8] Sinan G Aksoy, Tamara G Kolda, and Ali Pinar. Measuring and modeling bipartite graphs with community structure. *Journal of Complex Networks*, 5(4):581–603, 2017. 90

[9] Nour Raeef Al-Molhem, Yasser Rahal, and Mustapha Dakkak. Social network analysis in telecom data. *Journal of Big Data*, 6(1):99, 2019.

[10] Carlos A Alfaro and Ralihe R Villagran. The structure of sandpile groups of outerplanar graphs. *Applied Mathematics and Computation*, 395:125861, 2021. 7

[11] Nima Anari and Vijay V Vazirani. Planar graph perfect matching is in nc. *Journal of the ACM (JACM)*, 67(4):1–34, 2020.

[12] Pranay Anchuri and Malik Magdon-Ismail. Communities and balance in signed networks: A spectral approach. In *2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 235–242. IEEE, 2012. 136

[13] Nikolay V Antonov, Nikolay M Gulitskiy, Polina I Kakin, and German E Kochnev. Effects of turbulent environment on self-organized critical behavior: isotropy vs. anisotropy. *Universe*, 6(9):145, 2020. 7

[14] Samin Aref, Andrew J Mason, and Mark C Wilson. Computing the line index of balance using integer programming optimisation. *Optimization Problems in Graph Theory: In Honor of Gregory Z. Gutin's 60th Birthday*, pages 65–84, 2018. 137

[15] Samin Aref, Andrew J Mason, and Mark C Wilson. A modeling and computational study of the frustration index in signed networks. *Networks*, 75(1):95–110, 2020. 137, 141

[16] Markus Aschwanden. *Self-organized criticality in astrophysics: The statistics of nonlinear processes in the universe*. Springer Science & Business Media, 2011. 7

[17] V Avetisov, A Gorsky, S Maslov, Sergei Nechaev, and Olga Valba. Phase transitions in social networks inspired by the schelling model. *Physical Review E*, 98(3):032308, 2018. 7

[18] Adi Avidor and Michael Langberg. The multi-multiway cut problem. *Theoretical Computer Science*, 377(1-3):35–42, 2007. 137

[19] László Babai and Igor Gorodezky. Sandpile transience on the grid is polynomially bounded. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 627–636, 2007. 11

[20] Per Bak. *How nature works: the science of self-organized criticality*. Springer Science & Business Media, 2013. 7

[21] Per Bak, Chao Tang, and Kurt Wiesenfeld. Self-organized criticality: An explanation of the 1/f noise. *Physical review letters*, 59(4):381, 1987. 2, 7

[22] Per Bak, Chao Tang, and Kurt Wiesenfeld. Self-organized criticality. *Physical review A*, 38(1):364, 1988.

[23] Matthew Baker and Serguei Norine. Riemann–roch and abel–jacobi theory on a finite graph. *Advances in Mathematics*, 215(2):766–788, 2007. 8

[24] Matthew Baker and Farbod Shokrieh. Chip-firing games, potential theory on graphs, and spanning trees. *Journal of Combinatorial Theory, Series A*, 120(1):164–182, 2013. 8

[25] Prithu Banerjee, Wei Chen, and Laks VS Lakshmanan. Mitigating filter bubbles under a competitive diffusion model. *Proceedings of the ACM on Management of Data*, 1(2):1–26, 2023. 134

[26] Pedro H Barros, Isadora Cardoso-Pereira, Hector Allende-Cid, Osvaldo A Rosso, and Heitor S Ramos. Leveraging phase transition of topics for event detection in social media. *IEEE Access*, 8:70505–70518, 2020. 7

[27] Fabian Baumann, Philipp Lorenz-Spreen, Igor M Sokolov, and Michele Starnini. Modeling echo chambers and polarization dynamics in social networks. *Physical Review Letters*, 124(4):048301, 2020. 1, 134

[28] John M Beggs and Dietmar Plenz. Neuronal avalanches in neocortical circuits. *Journal of neuroscience*, 23(35):11167–11177, 2003. 1, 7

[29] Michael A Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[30] Alex Beutel, Leman Akoglu, and Christos Faloutsos. Fraud detection through graph-based user behavior modeling. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1696–1697, 2015. 1

[31] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *Proceedings of the 22nd international conference on World Wide Web*, pages 119–130, 2013. 65

[32] Norman Biggs. Algebraic potential theory on graphs. *Bulletin of the London Mathematical Society*, 29(6):641–682, 1997. 8

[33] Alessio Emanuele Biondo, Alessandro Pluchino, and Andrea Rapisarda. Modeling financial markets by self-organized criticality. *Physical Review E*, 92(4):042814, 2015. 7

[34] Anders Björner, László Lovász, and Peter W Shor. Chip-firing games on graphs. *European Journal of Combinatorics*, 12(4):283–291, 1991. 2, 8, 11, 13

[35] Hanjo D Boekhout, Walter A Kosters, and Frank W Takes. Efficiently counting complex multilayer temporal motifs in large-scale networks. *Computational Social Networks*, 6(1):8, 2019. 92

[36] Francesco Bonchi, Edoardo Galimberti, Aristides Gionis, Bruno Ordozgoiti, and Giancarlo Ruffo. Discovering polarized communities in signed networks. *CoRR*, abs/1910.02438, 2019. 149, 150, 154, 155

[37] Charles Bordenave and Hubert Lacoin. Cutoff at the entropic time for random walks on covered expander graphs. *Journal of the Institute of Mathematics of Jussieu*, 21(5):1571–1616, 2022.

[38] Stephen P Borgatti and Martin G Everett. Network analysis of 2-mode data. *Social networks*, 19(3):243–269, 1997. 65

[39] Stephen D Boyles, Nicholas E Lownes, and Avinash Unnikrishnan. Transportation network analysis. *Vol. I: Static and Dynamic Traffic Assignment*, 2020. 1

[40] Pawel Brach, Marek Cygan, Jakub Lkacki, and Piotr Sankowski. Algorithmic complexity of power law networks. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1306–1325. SIAM, 2016. 111

[41] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. Motif counting beyond five nodes. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 12(4):1–25, 2018. 68

[42] Gerth Stolting Brodal. Finger search trees. In *Handbook of Data Structures and Applications*, pages 171–178. Chapman and Hall/CRC, 2018. 38

[43] Coenraad Bron and Joep Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973. 80

[44] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. *Recent advances in graph partitioning*. Springer, 2016. 133

[45] Xinwei Cai, Xiangyu Ke, Kai Wang, Lu Chen, Tianming Zhang, Qing Liu, and Yunjun Gao. Efficient temporal butterfly counting and enumeration on temporal bipartite graphs. *Proc. VLDB Endow.*, 17(4):657–670, mar 2024. xv, 67, 88, 89, 92, 93, 123, 124

[46] Hannah Cairns. Rsrs explanation. https://github.com/cairnsh/fast-tree-sandpile/, 2016.

[47] Dorwin Cartwright and Frank Harary. Structural balance: a generalization of heider's theory. *Psychological review*, 63(5):277, 1956. 136

[48] Dibyayan Chakraborty, Antoine Dailly, Sandip Das, Florent Foucaud, Harmender Gahlawat, and Subir Kumar Ghosh. Complexity and algorithms for isometric path cover on chordal graphs and beyond. In *33rd International Symposium on Algorithms and Computation (ISAAC 2022)*, pages 12–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022.

[49] Ashok K Chandra, Prabhakar Raghavan, Walter L Ruzzo, and Roman Smolensky. The electrical resistance of a graph captures its commute and cover times. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 574–586, 1989. 17

[50] Ruinian Chang, Jingbang Chen, Ian Munro, Richard Peng, Qingyu Shi, and Zeyu Zheng. Sandpile prediction on undirected graphs. *arXiv preprint arXiv:2307.07711*, 2023. v, 2, 6

[51] Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. In Klaus Jansen and Samir Khuller, editors, *Approximation Algorithms for Combinatorial Optimization*, pages 84–95, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. 150

[52] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988. 95, 124

[53] Haiyan Chen and Bojan Mohar. The sandpile group of a polygon flower. *Discrete Applied Mathematics*, 270:68–82, 2019. 7

[54] Hongxu Chen, Hongzhi Yin, Tong Chen, Weiqing Wang, Xue Li, and Xia Hu. Social boosted recommendation with folded bipartite network embedding. *IEEE Transactions on Knowledge and Data Engineering*, 34(2):914–926, 2020. 65

[55] Jingbang Chen, Weinuo Li, Yingli Zhou, Hangrui Zhou, Qiuyang Mang, Can Wang, Yixiang Fang, and Chenhao Ma. Scalable approximate biclique counting over large bipartite graphs. *arXiv preprint arXiv:2505.10471*, 2025. vi, 65

[56] Jingbang Chen, Qiuyang Mang, Hangrui Zhou, Richard Peng, Yu Gao, and Chenhao Ma. Scalable algorithm for finding balanced subgraphs with tolerance in signed networks. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 278–287, 2024. vi, 3, 133

[57] Kaiyu Chen, Dong Wen, Wenjie Zhang, Ying Zhang, Xiaoyang Wang, and Xuemin Lin. Querying structural diversity in streaming graphs. *Proceedings of the VLDB Endowment*, 17(5):1034–1046, 2024. 92

[58] Xiaoshuang Chen, Kai Wang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Ying Zhang. Efficiently answering reachability and path queries on temporal bipartite graphs. *Proceedings of the VLDB Endowment*, 2021. 88

[59] Herman Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, pages 493–507, 1952. 79

[60] Dante R Chialvo. Critical brain networks. *Physica A: Statistical Mechanics and its Applications*, 340(4):756–765, 2004. 1, 7

[61] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985. 94

[62] Uthsav Chitra and Christopher Musco. Analyzing the impact of filter bubbles on social network polarization. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, pages 115–123, 2020. 134

[63] Ayush Choure and Sundar Vishwanathan. Random walks, electric networks and the transience class problem of sandpiles, 2012. 11

[64] Richard Cole. On the dynamic finger conjecture for splay trees. part ii: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000. 25, 32

[65] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. part i: Splay sorting log n-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000. 25, 32

[66] Tom Coleman, James Saunderson, and Anthony Wirth. A local-search 2-approximation for 2-correlation-clustering. In *European Symposium on Algorithms*, pages 308–319. Springer, 2008. 137

[67] Robert Crowston, Gregory Gutin, Mark Jones, and Gabriele Muciaccia. Maximum balanced subgraph problem parameterized above lower bound. *Theoretical Computer Science*, 513:53–64, 2013. 136

[68] Bhaskar DasGupta, German Andres Enciso, Eduardo Sontag, and Yi Zhang. Algorithmic and complexity results for decompositions of biological networks into monotone subsystems. *Biosystems*, 90(1):161–178, 2007. 136, 137

[69] Davide Della Giustina, Nicola Prezza, and Rossano Venturini. A new linear-time algorithm for centroid decomposition. In *String Processing and Information Retrieval: 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7–9, 2019, Proceedings*, pages 274–282. Springer, 2019.

[70] Shiyuan Deng, Shangqi Lu, and Yufei Tao. Space-query tradeoffs in range subgraph counting and listing. In *26th International Conference on Database Theory (ICDT 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023. 96

[71] Deepak Dhar. Self-organized critical state of sandpile automaton models. *Physical Review Letters*, 64(14):1613, 1990. 7

[72] Deepak Dhar. Theoretical studies of self-organized criticality. *Physica A: Statistical Mechanics and its Applications*, 369(1):29–70, 2006. 7

[73] Persi Diaconis and William Fulton. A growth model, a game, an algebra, lagrange inversion, and characteristic classes. *Rend. Sem. Mat. Univ. Pol. Torino*, 49(1):95–119, 1991. 8

[74] Andrey Dmitriev and Victor Dmitriev. Identification of self-organized critical state on twitter based on the retweets' time series analysis. *Complexity*, 2021(1):6612785, 2021. 7

[75] Patrick Doreian and Andrej Mrvar. A partitioning approach to structural balance. *Social networks*, 18(2):149–168, 1996. 136

[76] Vida Dujmovic, Gwenael Joret, Piotr Micek, Pat Morin, Torsten Ueckerdt, and David R Wood. Planar graphs have bounded queue-number. *Journal of the ACM (JACM)*, 67(4):1–38, 2020.

[77] Mark Dukes. The sandpile model on the complete split graph, motzkin words, and tiered parking functions. *Journal of Combinatorial Theory, Series A*, 180:105418, 2021. 7

[78] Mark Dukes, Thomas Selig, Jason P Smith, and Einar Steingrimsson. The Abelian sandpile model on ferrers graphs—a classification of recurrent configurations. *European Journal of Combinatorics*, 81:221–241, 2019. 7

[79] David Durfee, Matthew Fahrbach, Yu Gao, and Tao Xiao. Nearly tight bounds for sandpile transience on the grid. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 605–624. SIAM, 2018. 11

[80] Jean-Pierre Eckmann, Tatiana Nagnibeda, and Aymeric Perriard. Abelian sandpiles on cylinders. *Journal of Physics A: Mathematical and Theoretical*, 56(17):175001, 2023. 7

[81] Talya Eden, Dana Ron, and C Seshadhri. Faster sublinear approximation of the number of k-cliques in low-arboricity graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1467–1478. SIAM, 2020. 67

[82] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[83] Islam H El-Adaway, Ibrahim S Abotaleb, and Eric Vechan. Social network analysis approach for improved transportation planning. *Journal of Infrastructure Systems*, 23(2):05016004, 2017. 1

[84] P ERDdS and A R&wi. On random graphs i. *Publ. math. debrecen*, 6(290-297):18, 1959. 26

[85] Kimmo Eriksson. No polynomial bound for the chip firing game on directed graphs. *Proceedings of the American Mathematical Society*, 112(4):1203–1205, 1991. 11

[86] Stephen Eubank, Hasan Guclu, VS Anil Kumar, Madhav V Marathe, Aravind Srinivasan, Zoltan Toroczkai, and Nan Wang. Modelling disease outbreaks in realistic urban social networks. *Nature*, 429(6988):180–184, 2004. 88

[87] Stephen Eubank, Hasan Guclu, VS Anil Kumar, Madhav V Marathe, Aravind Srinivasan, Zoltan Toroczkai, and Nan Wang. Modelling disease outbreaks in realistic urban social networks. *Nature*, 429(6988):180–184, 2004. 88

[88] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. A survey of community search over big graphs. *The VLDB Journal*, 29:353–392, 2020. 88, 94

[89] Rosa Figueiredo and Yuri Frota. The maximum balanced subgraph of a signed graph: Applications and solution approaches. *European Journal of Operational Research*, 236(2):473–487, 2014. 136

[90] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010. 133

[91] Piotr Fronczak, Agata Fronczak, and Janusz A Holyst. Phase transitions in social networks. *The European Physical Journal B*, 59:133–139, 2007. 7

[92] Harold N Gabow and Robert E Tarjan. A linear-time algorithm for finding a minimum spanning pseudoforest. *Information Processing Letters*, 27(5):259–263, 1988.

[93] Anja Garber, Sarah Hallerberg, and Holger Kantz. Predicting extreme avalanches in self-organized critical sandpiles. *Physical Review E*, 80(2):026124, 2009. 8

[94] Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Conditional lower bounds for space/time tradeoffs. In Faith Ellen, Antonina Kolokolova, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures*, pages 421–436, Cham, 2017. Springer International Publishing. 96

[95] Isaac Goldstein, Moshe Lewenstein, and Ely Porat. On the Hardness of Set Disjointness and Set Intersection with Bounded Universe. In Pinyan Lu and Guochuan Zhang, editors, *30th International Symposium on Algorithms and Computation (ISAAC 2019)*, volume 149 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:22, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. 96

[96] Eric Goles and Maurice Margenstern. Sand pile as a universal computer. *International Journal of Modern Physics C*, 7(02):113–122, 1996. 8, 11

[97] Gramoz Goranci, Monika Henzinger, and Pan Peng. Improved guarantees for vertex sparsification in planar graphs. *SIAM Journal on Discrete Mathematics*, 34(1):130–162, 2020.

[98] Jean-Loup Guillaume and Matthieu Latapy. Bipartite graphs as models of complex networks. *Physica A: Statistical Mechanics and its Applications*, 371(2):795–813, 2006. 111

[99] Saket Gurukar, Sayan Ranu, and Balaraman Ravindran. Commit: A scalable approach to mining communication motifs from dynamic networks. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 475–489, 2015. 92

[100] N Gülpinar, G Gutin, G Mitra, and A Zverovitch. Extracting pure network submatrices in linear programs using signed graphs. *Discrete Applied Mathematics*, 137(3):359–372, 2004. 150

[101] Frank Harary. On the notion of balance of a signed graph. *Michigan Mathematical Journal*, 2(2):143–146, 1953. 134, 136

[102] Frank Harary. On the measurement of structural balance. *Behavioral Science*, 4(4):316–323, 1959. 137

[103] Frank Harary and Jerald A Kabell. A simple algorithm to detect balance in signed graphs. *Mathematical Social Sciences*, 1(1):131–136, 1980. 136

[104] Andor Háznagy, István Fi, András London, and Tamás Nemeth. Complex network analysis of public transportation networks: A comprehensive study. In *2015 International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*, pages 371–378. IEEE, 2015. 1

[105] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, pages 639–648, 2020. 88

[106] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017. 88

[107] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *The collected works of Wassily Hoeffding*, pages 409–426, 1994. 79

[108] Alexander E Holroyd, Lionel Levine, Karola Mészáros, Yuyal Peres, James Propp, and David B Wilson. Chip-firing and rotor-routing on directed graphs. *In and out of equilibrium 2*, pages 331–364, 2008. 9, 11, 15

[109] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, jun 1973. 144

[110] Wenzhe Hou, Xiang Zhao, and Bo Tang. Learnsc: An efficient and unified learning-based framework for subgraph counting problem. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 2625–2638. IEEE, 2024. 68

[111] Yao Ping Hou. Bounds for the least Laplacian eigenvalue of a signed graph. *Acta Mathematica Sinica*, 21(4):955–960, 2005. 136

[112] Yaoping Hou, Jiongsheng Li, and Yongliang Pan. On the Laplacian eigenvalues of signed graphs. *Linear and Multilinear Algebra*, 51(1):21–30, 2003. 136

[113] Falk Hüffner, Nadja Betzler, and Rolf Niedermeier. Optimal edge deletions for signed graph balancing. In *International Workshop on Experimental and Efficient Algorithms*, pages 297–310. Springer, 2007. 136

[114] Falk Hüffner, Nadja Betzler, and Rolf Niedermeier. Separator-based data reduction for signed graph balancing. *Journal of combinatorial optimization*, 20:335–360, 2010. 137

[115] Shweta Jain and C Seshadhri. The power of pivoting for exact clique counting. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, pages 268–276, 2020. 67

[116] Shweta Jain and C Seshadhri. Provably and efficiently approximating near-cliques using the turán shadow: Peanuts. In *Proceedings of The Web Conference 2020*, pages 1966–1976, 2020. 68

[117] Matthew Jenssen, Peter Keevash, and Will Perkins. Algorithms for# bis-hard problems on expander graphs. *SIAM Journal on Computing*, 49(4):681–710, 2020.

[118] Madhav Jha, Comandur Seshadhri, and Ali Pinar. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 589–597, 2013. 88

[119] Mazen Juma and Khaled Shaalan. Online social network analysis for cybersecurity awareness. In *Recent advances in intelligent systems and smart applications*, pages 585–614. Springer, 2020.

[120] Nikita Kalinin, Aldo Guzman-Saenz, Yulieth Prieto, Mikhail Shkolnikov, Vera Kalinina, and Ernesto Lupercio. Self-organized criticality and pattern emergence through the lens of tropical geometry. *Proceedings of the National Academy of Sciences*, 115(35):E8135–E8142, 2018. 7

[121] Mehdi Kaytoue, Sergei O Kuznetsov, Amedeo Napoli, and Sébastien Duplessis. Mining gene expression data with pattern structures in formal concept analysis. *Information Sciences*, 181(10):1989–2001, 2011. 65

[122] Subhash Khot. On the power of unique 2-prover 1-round games. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 767–775, 2002. 137

[123] Seungki Kim and Yuntao Wang. A stochastic variant of the Abelian sandpile model. *Journal of Statistical Physics*, 178(3):711–724, 2020. 7

[124] Philip N Klein, Claire Mathieu, and Hang Zhou. Correlation clustering and two-edge-connected augmentation for planar graphs. *Algorithmica*, pages 1–34, 2023.

[125] Caroline J Klivans. *The mathematics of chip-firing*. CRC Press, 2018. 7, 8, 9, 12, 13

[126] Konect. Konect. http://konect.cc/networks/, 2006. 80

[127] Stefan M Kostić, Mirjana I Simić, and Miroljub V Kostić. Social network analysis and churn prediction in telecommunications using graph theory. *Entropy*, 22(7):753, 2020. 1

[128] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(11):P11005, 2011. 92

[129] Emmanuel Kowalski. *An introduction to expander graphs*. Société mathématique de France Paris, 2019. 16

[130] Thomas Kron and Thomas Grund. Society as a self-organized critical system. *Cybernetics & Human Knowing*, 16(1-2):65–82, 2009. 7

[131] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science*, 407(1-3):458–473, 2008. 111

[132] Gregory F Lawler, Maury Bramson, and David Griffeath. Internal diffusion limited aggregation. *The Annals of Probability*, pages 2117–2140, 1992. 8

[133] Manh Ha Le, Tra An Pham, and Thi Ha Duong Phan. On the relation between chip firing games and petri nets. In *2009 IEEE-RIVF International Conference on Computing and Communication Technologies*, pages 1–8. IEEE, 2009. 8

[134] Jesus Leanos and Christophe Ndjatchi. The edge-connectivity of token graphs. *Graphs and Combinatorics*, 37:1013–1023, 2021.

[135] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[136] Ronghua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. Ordering heuristics for k-clique listing. *Proc. VLDB Endow.*, 2020. 66, 67, 69

[137] Rundong Li, Pinghui Wang, Peng Jia, Xiangliang Zhang, Junzhou Zhao, Jing Tao, Ye Yuan, and Xiaohong Guan. Approximately counting butterflies in large bipartite graph streams. *IEEE Transactions on Knowledge and Data Engineering*, 34(12):5621–5635, 2021. 92

[138] Youhuan Li, Lei Zou, M Tamer Özsu, and Dongyan Zhao. Time constrained continuous subgraph search over streaming graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1082–1093. IEEE, 2019. 92

[139] Yuchen Li, Zhengzhi Lou, Yu Shi, and Jiawei Han. Temporal motifs in heterogeneous information networks. In *MLG Workshop@ KDD*, 2018. 92

[140] Pedro G Lind, Marta C Gonzalez, and Hans J Herrmann. Cycles and clustering in bipartite networks. *Physical review E*, 72(5):056127, 2005. 90

[141] Klaus Linkenkaer-Hansen, Vadim V Nikouline, J Matias Palva, and Risto J Ilmoniemi. Long-range temporal correlations and scaling behavior in human brain oscillations. *Journal of Neuroscience*, 21(4):1370–1377, 2001. 1, 7

[142] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. Efficient (α, β)-core computation: An index-based approach. In *The World Wide Web Conference*, pages 1130–1141, 2019. 88

[143] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. Efficient (α, β)-core computation in bipartite graphs. *The VLDB Journal*, 29(5):1075–1099, 2020.

[144] Paul Liu, Austin R Benson, and Moses Charikar. Sampling methods for counting temporal motifs. In *Proceedings of the twelfth ACM international conference on web search and data mining*, pages 294–302, 2019. 92

[145] Penghang Liu, Valerio Guarrasi, and Ahmet Erdem Sarıyüce. Temporal network motifs: Models, limitations, evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 35(1):945–957, 2021. 92

[146] Giorgio Locicero, Giovanni Micale, Alfredo Pulvirenti, and Alfredo Ferro. Temporalri: a subgraph isomorphism algorithm for temporal networks. In *Complex Networks & Their Applications IX: Volume 2, Proceedings of the Ninth International Conference on Complex Networks and Their Applications COMPLEX NETWORKS 2020*, pages 675–687. Springer, 2021. 92

[147] Yaowei Long and Seth Pettie. Planar distance oracles with better time-space tradeoffs. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2517–2537. SIAM, 2021.

[148] László Lovász. Random walks on graphs. *Combinatorics, Paul erdos is eighty*, 2(1-46):4, 1993. 16

[149] Ilias Lymperopoulos and George Lekakos. Analysis of social network dynamics with models from the theory of complex adaptive systems. In *Collaborative, Trusted and Privacy-Aware e/m-Services: 12th IFIP WG 6.11 Conference on e-Business, e-Services, and e-Society, I3E 2013, Athens, Greece, April 25-26, 2013. Proceedings 12*, pages 124–140. Springer, 2013. 7

[150] Chenhao Ma, Reynold Cheng, Laks VS Lakshmanan, Tobias Grubenmann, Yixiang Fang, and Xiaodong Li. Linc: a motif counting algorithm for uncertain graphs. *Proceedings of the VLDB Endowment*, 13(2):155–168, 2019. 68

[151] Korosh Mahmoodi, Bruce J West, and Paolo Grigolini. Self-organizing complex networks: individual versus global rules. *Frontiers in physiology*, 8:478, 2017. 7

[152] Qiuyang Mang, Jingbang Chen, Hangrui Zhou, Yu Gao, Yingli Zhou, Richard Peng, Yixiang Fang, and Chenhao Ma. Efficient historical butterfly counting in large temporal bipartite networks via graph structure-aware index. *arXiv preprint arXiv:2406.00344*, 2024. v, 65, 88

[153] Roberta Martucci, Corrado Mascia, Chiara Simeoni, and Filippo Tassi. Hospital management in the COVID-19 emergency: Abelian sandpile paradigm and beyond. *arXiv preprint arXiv:2102.11974*, 2021. 7

[154] C Mejia and JA Montoya. The Abelian sandpile model, non-parsimonious simulations and unpredictability. *Proc Automata*, pages 24–32, 2019. 7

[155] Andras Meszaros. The distribution of sandpile groups of random regular graphs. *Transactions of the American Mathematical Society*, 373(9):6529–6594, 2020. 7

[156] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002. 88

[157] Michael Mitzenmacher, Jakub Pachocki, Richard Peng, Charalampos Tsourakakis, and Shen Chen Xu. Scalable large near-clique detection in large-scale networks via sampling. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 815–824, 2015. 65

[158] Clyde L Monma and Victor K Wei. Intersection graphs of paths in a tree. *Journal of Combinatorial Theory, Series B*, 41(2):141–181, 1986.

[159] J Andres Montoya and Carolina Mejia. On the complexity of sandpile prediction problems. *Electronic Notes in Theoretical Computer Science*, 252:229–245, 2009. 11

[160] J Andres Montoya and Carolina Mejia. On the predictability of the Abelian sandpile model. *Natural Computing*, 21(1):69–79, 2022. 7

[161] Juan Andres Montoya and Carolina Mejia. The computational complexity of the Abelian sandpile model, 2011. 8

[162] Cristopher Moore and Martin Nilsson. The computational complexity of sandpiles. *Journal of statistical physics*, 96(1-2):205–224, 1999. 2, 6, 8, 10, 11, 12

[163] Mark Newman. The physics of networks. *Physics today*, 61(11):33–38, 2008. 7

[164] Tien T Nguyen, Pik-Mai Hui, F Maxwell Harper, Loren Terveen, and Joseph A Konstan. Exploring the filter bubble: the effect of using recommender systems on content diversity. In *Proceedings of the 23rd international conference on World wide web*, pages 677–686, 2014. 1, 134

[165] Jason Niu and A. Erdem Sarıyüce. On cohesively polarized communities in signed networks. In *Companion Proceedings of the ACM Web Conference 2023*, WWW '23 Companion, page 1339–1347, New York, NY, USA, 2023. Association for Computing Machinery. 136, 137, 150

[166] Laboratory of Web Algorithmics. Laboratory of web algorithmics datasets. http://law.di.unimi.it/datasets.php, 2013. 80

[167] Tore Opsahl. Triadic closure in two-mode networks: Redefining the global and local clustering coefficients. *Social networks*, 35(2):159–167, 2013. 90

[168] Bruno Ordozgoiti, Antonis Matakos, and Aristides Gionis. Finding large balanced subgraphs in signed networks. In *Proceedings of The Web Conference 2020*, pages 1378–1388, 2020. 137, 150, 152

[169] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. Motifs in temporal networks. In *Proceedings of the tenth ACM international conference on web search and data mining*, pages 601–610, 2017. 92

[170] Noujan Pashanasangi and C Seshadhri. Faster and generalized temporal triangle counting, via degeneracy ordering. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 1319–1328, 2021. 92

[171] Georgios A Pavlopoulos, Panagiota I Kontou, Athanasia Pavlopoulou, Costas Bouyioukos, Evripides Markou, and Pantelis G Bagos. Bipartite graphs in systems biology and medicine: a survey of methods and applications. *GigaScience*, 7(4):giy014, 2018. 88

[172] Seth Pettie. Splay trees, davenport-schinzel sequences, and the deque conjecture. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1115–1124, 2008. 54

[173] JC Phillips. Fractals and self-organized criticality in proteins. *Physica A: Statistical Mechanics and Its Applications*, 415:440–448, 2014. 7

[174] Carlos Andre Reis Pinheiro. *Social network analysis in telecommunications*. John Wiley & Sons, 2011. 1

[175] Svatopluk Poljak and Daniel Turzík. A polynomial time heuristic for certain subgraph optimization problems with guaranteed worst case bound. *Discrete Mathematics*, 58(1):99–104, 1986. 136

[176] Stanford Network Analysis Project. SNAP. http://snap.stanford.edu/data/, 2009. 80

[177] Yuval Rabani, Alistair Sinclair, and Rolf Wanka. Local divergence of markov chains and the analysis of iterative load-balancing schemes. In *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No. 98CB36280)*, pages 694–703. IEEE, 1998. 8

[178] Akshay Ramachandran and Aaron Schild. Sandpile prediction on a tree in near linear time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, page 1115–1131, USA, 2017. Society for Industrial and Applied Mathematics. 2, 6, 10, 12, 24, 26

[179] O Ramos, Ernesto Altshuler, and KJ Maloy. Avalanche prediction in a self-organized pile of beads. *Physical review letters*, 102(7):078701, 2009. 7

[180] Ursula Redmond and Pádraig Cunningham. Temporal subgraph isomorphism. In *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 1451–1452, 2013. 92

[181] Network Repository. Network repository. https://networkrepository.com/network-data.php, 2014.

[182] Guozhen Rong, Wenjun Li, Yongjie Yang, and Jianxin Wang. Reconstruction and verification of chordal graphs with a distance oracle. *Theoretical Computer Science*, 859:48–56, 2021.

[183] Ronitt Rubinfeld. The cover time of a regular expander is $o(n \log n)$. *Information processing letters*, 35(1):49–51, 1990. 17

[184] H Saba, JGV Miranda, and MA Moret. Self-organized critical phenomenon as a q-exponential decay—avalanche epidemiology of dengue. *Physica A: Statistical Mechanics and its Applications*, 413:205–211, 2014. 1, 7

[185] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyuce, and Srikanta Tirthapura. Butterfly counting in bipartite networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2150–2159, 2018. 67, 88, 89, 92, 108

[186] Seyed-Vahid Sanei-Mehri, Yu Zhang, Ahmet Erdem Sariyüce, and Srikanta Tirthapura. Fleet: Butterfly estimation from a bipartite graph stream. In *Proceedings*

of the 28th ACM International Conference on Information and Knowledge Management, pages 1201–1210, 2019. 92

[187] Ilie Sarpe and Fabio Vandin. Oden: simultaneous approximation of multiple motif counts in large temporal networks. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pages 1568–1577, 2021. 92

[188] Thomas Sauerwald and He Sun. Spectral graph theory and applications. https://resources.mpi-inf.mpg.de/departments/d1/teaching/ws11/SGT/Lecture7.pdf, 2011. 17

[189] Jose A Scheinkman and Michael Woodford. Self-organized criticality and economic fluctuations. *The American Economic Review*, 84(2):417–421, 1994. 7

[190] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten Borgwardt. Efficient graphlet kernels for large graph comparison. In David van Dyk and Max Welling, editors, *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 488–495, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009. PMLR.

[191] Aida Sheshbolouki and M Tamer Özsu. sgrapp: Butterfly approximation in streaming graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 16(4):1–43, 2022. 66, 89, 92, 108

[192] Jessica Shi and Julian Shun. Parallel algorithms for butterfly computations. In *Massive Graph Analytics*, pages 287–330. Chapman and Hall/CRC, 2022. 92

[193] Michael Sipser and Daniel A Spielman. Expander codes. *IEEE transactions on Information Theory*, 42(6):1710–1722, 1996. 16

[194] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985. 22, 25, 30, 33, 35, 54

[195] RF Smalley Jr, Donald Lawson Turcotte, and Sara A Solla. A renormalization group approach to the stick-slip behavior of faults. *Journal of Geophysical Research: Solid Earth*, 90(B2):1894–1900, 1985. 7

[196] WD Smyth, JD Nash, and JN Moum. Self-organized criticality in geophysical turbulence. *Scientific reports*, 9(1):3747, 2019. 7

[197] Dawn Song, David Zuckerman, and JD Tygar. Expander graphs for digital stream authentication and robust overlay networks. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 258–270. IEEE, 2002. 16

[198] Erick Stattner and Nicolas Vidot. Social network analysis in epidemiology: Current trends and perspectives. In *2011 Fifth International Conference on Research Challenges in Information Science*, pages 1–11. IEEE, 2011. 1

[199] Mihail M Stoian. Concurrent link-cut trees. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2503–2505, 2022.

[200] Bosiljka Tadić, Marija Mitrovic Dankulov, and Roderick Melnik. Mechanisms of self-organized criticality in social processes of knowledge creation. *Physical Review E*, 96(3):032307, 2017. 7

[201] Gábor Tardos. Polynomial bound for a chip firing game on graphs. *SIAM journal on discrete mathematics*, 1(3):397–398, 1988. 11

[202] Véronique Van Vlasselaer, Tina Eliassi-Rad, Leman Akoglu, Monique Snoeck, and Bart Baesens. Gotcha! network-based fraud detection for social security fraud. *Management Science*, 63(9):3090–3110, 2017. 1

[203] Demival Vasques Filho and Dion RJ O'Neale. Degree distributions of bipartite networks and their projections. *Physical Review E*, 98(2):022307, 2018. 111

[204] Vasileios Vlachos, Yannis C Stamatiou, Pantelis Tzamalis, Sotiris Nikoletseas, and Kyriaki Chantzi. A social network analysis tool for uncovering cybersecurity threats. In *6th International Symposium for ICS & SCADA Cyber Security Research 2019*. BCS Learning & Development, 2019.

[205] Jia Wang, Ada Wai-Chee Fu, and James Cheng. Rectangle counting in large bipartite graphs. In *2014 IEEE International Congress on Big Data*, pages 17–24. IEEE, 2014. 65, 66, 67, 88, 89, 92, 108

[206] Jun Wang, Arjen P De Vries, and Marcel JT Reinders. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 501–508, 2006. 88

[207] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. Vertex priority based butterfly counting for large-scale bipartite networks. *PVLDB*, 2019. 65, 67, 88, 92, 93, 94, 95, 99, 100, 103, 106

[208] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. Efficient bitruss decomposition for large-scale bipartite graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 661–672. IEEE, 2020. 88

[209] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. Accelerated butterfly counting with vertex priority on bipartite graphs. *The VLDB Journal*, 32(2):257–281, 2023. 92

[210] Yafeng Wang, Huawei Fan, Weijie Lin, Ying-Cheng Lai, and Xingang Wang. Growth, collapse and self-organized criticality in complex networks. *Scientific reports*, 6(1):24445, 2016. 7

[211] Yiyuan Wang, Shaowei Cai, Jiejiang Chen, and Minghao Yin. Sccwalk: An efficient local search algorithm and its improvements for maximum weight clique problem. *Artificial Intelligence*, 280:103230, 2020.

[212] Zhibin Wang, Longbin Lai, Yixue Liu, Bing Shui, Chen Tian, and Sheng Zhong. I/o-efficient butterfly counting at scale. *Proceedings of the ACM on Management of Data*, 1(1):1–27, 2023. 67, 92, 94

[213] Nicholas W Watkins, Gunnar Pruessner, Sandra C Chapman, Norma B Crosby, and Henrik J Jensen. 25 years of self-organized criticality: concepts and controversies. *Space Science Reviews*, 198:3–44, 2016. 7

[214] Dong Wen, Bohua Yang, Ying Zhang, Lu Qin, Dawei Cheng, and Wenjie Zhang. Span-reachability querying in large temporal graphs. *The VLDB Journal*, pages 1–19, 2022. 92

[215] BJ West, GF Massari, G Culbreth, R Failla, M Bologna, RIM Dunbar, and P Grigolini. Relating size and functionality in human social networks through complexity. *Proceedings of the National Academy of Sciences*, 117(31):18355–18358, 2020. 7

[216] Maciej Woloszyn, D Stauffer, and K Kulakowski. Order-disorder phase transition in a cliquey social network. *The European Physical Journal B*, 57:331–335, 2007. 7

[217] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. Graph neural networks in recommender systems: a survey. *ACM Computing Surveys*, 55(5):1–37, 2022. 88, 134

[218] Yifei Xia, Feng Zhang, Qingyu Xu, Mingde Zhang, Zhiming Yao, Lv Lu, Xiaoyong Du, Dong Deng, Bingsheng He, and Siqi Ma. Gpu-based butterfly counting. *The VLDB Journal*, pages 1–25, 2024. 67, 92

[219] Haoxuan Xie, Yixiang Fang, Yuyang Xia, Wensheng Luo, and Chenhao Ma. On querying connected components in large temporal graphs. *Proceedings of the ACM on Management of Data*, 1(2):1–27, 2023. 92

[220] Hao Xu and Shing-Tung Yau. An explicit formula of hitting times for random walks on graphs. *arXiv preprint arXiv:1312.0065*, 2013.

[221] Yichen Xu, Chenhao Ma, Yixiang Fang, and Zhifeng Bao. Efficient and effective algorithms for generalized densest subgraph discovery. *Proc. ACM Manag. Data*, 1(2), jun 2023. 133

[222] Bo Yang, William Cheung, and Jiming Liu. Community mining from signed social networks. *IEEE transactions on knowledge and data engineering*, 19(10):1333–1348, 2007. 136

[223] Jianye Yang, Yun Peng, and Wenjie Zhang. (p, q)-biclique counting and enumeration for large sparse bipartite graphs. *Proceedings of the VLDB Endowment*, 15(2):141–153, 2021. 65, 67, 69, 76, 80

[224] Yixing Yang, Yixiang Fang, Maria E Orlowska, Wenjie Zhang, and Xuemin Lin. Efficient bi-triangle counting for large bipartite networks. *Proceedings of the VLDB Endowment*, 14(6):984–996, 2021. 111

[225] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongzhi Chen, and Guoren Wang. Lightning fast and space efficient k-clique counting. In *Proceedings of the ACM Web Conference 2022*, pages 1191–1202, 2022. 66, 68, 69

[226] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongchao Qin, and Guoren Wang. Efficient biclique counting in large bipartite graphs. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023. 65, 66, 67, 70, 80, 81

[227] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. On querying historical k-cores. *Proceedings of the VLDB Endowment*, 2021. 92

[228] Fangyuan Zhang, Dechuang Chen, Sibo Wang, Yin Yang, and Junhao Gan. Scalable approximate butterfly and bi-triangle counting for large bipartite networks. *Proceedings of the ACM on Management of Data*, 1(4):1–26, 2023. 95, 111

[229] Fangyuan Zhang, Dechuang Chen, Sibo Wang, Yin Yang, and Junhao Gan. Scalable approximate butterfly and bi-triangle counting for large bipartite networks. *Proc. ACM Manag. Data*, 1(4), dec 2023.

[230] Kangfei Zhao, Jeffrey Xu Yu, Qiyan Li, Hao Zhang, and Yu Rong. Learned sketch for subgraph counting: a holistic approach. *The VLDB Journal*, 32(5):937–962, 2023. 68

[231] Alexander Zhou, Yue Wang, and Lei Chen. Butterfly counting on uncertain bipartite graphs. *Proceedings of the VLDB Endowment*, 15(2):211–223, 2021. 67, 92

[232] Yufang Zhou and Haiyan Chen. The sandpile group of a family of nearly complete graphs. *Bulletin of the Malaysian Mathematical Sciences Society*, 44(2):625–637, 2021. 7

[233] Dmitry Zhukov, Konstantin Kunavin, and Sergey Lyamin. Online rebellion: self-organized criticality of contemporary protest movements. *Sage Open*, 10(2):2158244020923354, 2020. 7

[234] Zhaonian Zou. Bitruss decomposition of bipartite graphs. In *International conference on database systems for advanced applications*, pages 218–233. Springer, 2016. 88