Mining Large Dynamic Graphs and Tensors

Kijung Shin

CMU-CS-19-101

February 2019

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Christos Faloutsos, Chair
Tom M. Mitchell
Leman Akoglu
Philip S. Yu (University of Illinois at Chicago)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant numbers CNS-1314632 and IIS-1408924, the US Army Research Lab under grant number W911NF-09-2-0053, the Qatar National Research Fund under NRPR grant number 7-1330-2-483, the Korea Foundation for Advanced Studies, and the Siebel Scholars Foundation. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: data mining, graph mining, tensor mining, stream mining, graph stream, edge stream, tensor stream, streaming algorithms, approximation algorithms, distributed algorithms, out-of-core algorithms, external-memory algorithms, distributed streaming algorithms, MapReduce, Hadoop, triangle counting, graph summarization, graph compression, tensor decomposition, tucker decomposition, anomaly detection, fraud detection, k-cores, degeneracy, dense subtensor detection, behavior modeling, network game, sharable good game, progressive stages, WRS, TRI-FLY, COCOS, THINKD, SWEG, S-HOT, CORESCOPE, M-ZOOM, D-CUBE, DENSESTREAM, DENSEALERT, SGG, SGG-AC, SGG-NASH, SWATT, SWATTFIT

To my family and Sol-Ah, for their love and support

Abstract

Graphs are ubiquitous, representing a variety of information, ranging from *who follows whom* on online social networks to *who reviews what* on e-commerce sites. Many of these graphs are large (e.g., online social networks with over two billion active users) and dynamic (i.e., nodes and edges can be added and removed over time). Moreover, they are with rich side information (e.g., e-commerce reviews with timestamps, ratings, and text) and thus naturally modeled as tensors (i.e., multi-dimensional arrays).

Given large dynamic graphs and tensors, how can we analyze their structure? How can we detect interesting anomalies? Lastly, how can we model behaviors of individuals in the data? My thesis focuses on these closely related questions, all of which are fundamental to understand massive evolving data on user behavior. That is, we develop scalable algorithms for mining large dynamic graphs and tensors, with a focus on three tasks:

1. **Structure Analysis**: We build one-pass, sublinear-space algorithms that incrementally estimate the triangle count, which is an important connectivity measure, in large dynamic graphs. In particular, our distributed algorithm yields up to $39\times$ *more accurate* estimates faster than a baseline. We also develop distributed and out-of-core algorithms for succinctly but accurately summarizing the structure of large graphs and tensors. They summarize over $25\times$ *larger data* without quality loss than their best competitors.
2. **Anomaly Detection**: We develop near-linear time approximation algorithms for detecting unusually dense subgraphs and subtensors, which signal notable anomalies such as "edit wars" on Wikipedia and fake followers on Twitter. Especially, our tensor algorithm is up to $114\times$ *faster* without accuracy loss than the previously best heuristic. We also extend it for distributed or dynamic data with the same approximation guarantee.
3. **Behavior Modeling**: We design game-theoretic models for purchases of individuals in social networks and a fast algorithm for finding Nash equilibria of the models. In addition, we develop a stage model for the progression of individuals and a distributed optimization algorithm for fitting the model to behavior logs with *trillions* of records. Using our tools, we measure social inefficiency regarding purchase of sharable goods and discover progression patterns of LinkedIn users.

To achieve the highest performance and scalability, our algorithms for the above tasks employ mathematical techniques (e.g., approximation and sampling), use distributed computing frameworks (e.g., MAPREDUCE and MPI), and/or exploit pervasive patterns in real-world data (e.g., power-law degree distribution). We successfully apply them to massive datasets, including 20.6 *billion* social connections on LinkedIn, 1.47 *billion* follow relations on Twitter, 783 *million* hyperlinks between web pages, 483 *million* edits on Wikipedia, and a synthetic tensor with 1 *trillion* non-zero entries.

Acknowledgments

My first and biggest thanks go out to my advisor, Christos Faloutsos, for his advice and support. He has been an ideal advisor in every respect. Without his inspiration, encouragement, patience, care, and humor, my graduate study would not have been pleasurable and productive as it has been.

I also would like to thank my other thesis committee members, Leman Akoglu, Tom Mitchell, and Philip Yu, for their insightful questions and constructive feedback.

I was fortunate to be mentored by Amol Ghoting, Aastha Jain, Myunghwan Kim, Hema Raghavan, and Mahdi Shafiei, during my two internships at LinkedIn. Two chapters of this thesis are based on what I did during the internships.

I have had the opportunity to work with an amazing group of coauthors and collaborators, each of whom deserves my gratitude: Ashwin Bahulkar, Alex Beutel, Nitesh Chawla, Tina Eliassi-Rad, Dhivya Eswaran, David Hachen, Mohammad Hammoud, Insu Han, Bryan Hooi, Jinhong Jung, U Kang, Jisu Kim, Hemank Lamba, Jaeho Lee, Euiwoong Lee, Shenghua Liu, Vaishnavh Nagarajan, Aastha Nigam, Jinoh Oh, Vagelis Papalexakis, Jürgen Pfeffer, Ariel Procaccia, Lee Sael, Neil Shah, Naji Shajarisales, Jinwoo Shin, Hyun-Ah Song, Boleslaw Szymanski, Minji Yoon, Hwanjo Yu, and Jiyuan Zhang. I would especially like to thank U Kang for sparking my interest in research when I was an undergraduate, and for mentoring me ever since.

I am grateful to all the members and visitors of the CMU DB group for being great friends, coworkers, and mentors: Miguel Araùjo, Alex Beutel, Daniel Chino, Dhivya Eswaran, Sebastian Gobel, Bryan Hooi, Rohan Kumar, Srijan Kumar, Hemank Lamba, Shenghua Liu, Jinoh Oh, Vagelis Papalexakis, Namyong Park, Neil Shah, Hyun-Ah Song, Tubasa Takahashi, and Minji Yoon. Although there are too many to list individually, I have been fortunate to have wonderful friends from the CMU Korean community. I also would like to thank Deborah Cavlovich, Ann Stetser, and Marilyn Walgora for their seamless administrative support that has made my Ph.D. life run smoothly.

Last but not least, many thanks go out to my family and fiancèe, Sol-Ah Kim, for their endless love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs are simple but powerful models to describe how everything is connected to everything else. Thus, a wide range of information has been modeled as graphs: *who follows whom* on online social networks, *who reviews what* on e-commerce sites, *who searches what* on search engines, to name just a few. The low cost of storage, the rapid growth of Web applications, and the wide variety of available data have made these graphs *(a) large:* for example, an online social network has over two billion active users, *(b) dynamic:* that is, nodes and edges can be added and removed over time, and *(c) with rich side information:* for example, e-commerce reviews contain timestamps, ratings, and text. However, since most existing graph mining tools are for small static graphs without side information, the majority of these large, dynamic, and rich graphs have remained poorly understood and utilized.

In this thesis, our focus is *developing fast scalable algorithms for mining large dynamic graphs and tensors*. Tensors, or multi-dimensional arrays, are natural representations of graphs with side information. To maximize the performance and scalability of our algorithms, we employ mathematical techniques (e.g., approximation and sampling), use distributed computing frameworks (e.g., MAPREDUCE and MPI), and exploit common patterns in real-world data (e.g., power-law degree distribution). As a result, we successfully apply our algorithms to *billion-scale* and even *trillion-scale* datasets, helping understand and utilize the datasets. Specifically, our algorithms perform three tasks that are closely related to each other (see Figure 1.1): *structure analysis*, *anomaly detection*, and *behavior modeling*. Below, we provide an overview of the impact and contributions of our work on the tasks. Then, we outline the organization of the thesis.

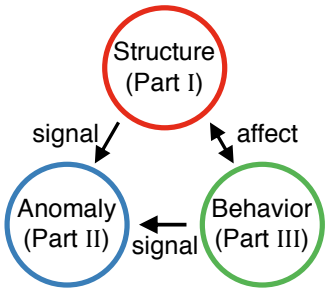


Figure 1.1: **The subtopics of the thesis (i.e., structures, anomalies, and behaviors) are closely related to each other.** The structure of data (e.g., the structure of the underlying social network) and the behavior of individuals in the data affect each other. Moreover, unusual (sub)structures and behaviors signal anomalies.

1.1 Overall Impact

This thesis has potential impact on a wide range of domains where large, dynamic, and rich information are modeled as graphs or tensors: search engines (Chapters 7 and 9), social media (Chapters 9 and 15), e-commerce (Chapter 12), computer security (Chapters 11, 12, and 13), public policy (Chapter 14), to name a few. Below, we highlight the impact of our work on academia and industry.

Impact in Academia:

- Most of the algorithms produced throughout this thesis have been *open-sourced* and *downloaded* over 350 times from 24 countries.
- Our work on patterns and anomalies in dense subgraphs [SERF18] was included in *MIT*'s course on graph analytics (MIT 6.886), and two tutorials on k-cores at *ICDM 2016* and *ECML/PKDD 2017*.

Impact in Industry:

- Our behavior model and learning algorithm for progression (SWATT and SWATTFIT [SSK+18]) were used at *LinkedIn Inc.* for analyzing user behavior.
- A patent application on our graph-summarization algorithm (SWEG [SGKR19]) was filed by *LinkedIn Inc.* in September 2018.
- Our anomaly-detection algorithm (D-CUBE [SHKF18]) was used in *production* as a part of *NAVER*, which is the dominant search engine in South Korea, to identify and filter spam sites.

Awards and Media Coverage:

- Our work on patterns and anomalies in dense subgraphs [SERF16] was selected as one among the *"Bests of ICDM 2016"* and invited to the Knowledge and Information Systems journal [SERF18].
- Our work on purchase behavior modeling [SLEP17] was featured in *New Scientist* in May 2017 (available at `https://www.newscientist.com/article/2132926`).

1.2 Contributions

We provide a brief summary of our contributions on each task considered in this thesis.

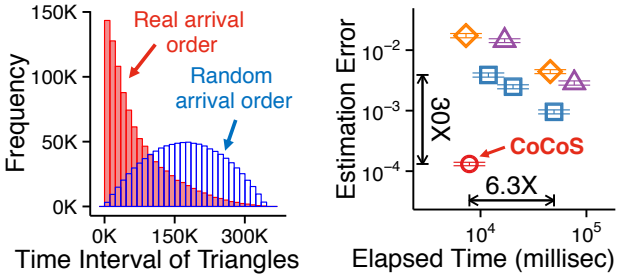
1.2.1 Part I: Structure Analysis

A greater understanding of the structure of graphs and tensors can be achieved by computing structure-related measures, and/or presenting the data in a concise and interpretable manner. In Part I, we address two tasks closely related to structure analysis: (a) counting triangles in large dynamic graphs that are modeled as graph streams, and (b) summarizing large graphs and tensors.

1.2.1.1 Counting Triangles in Graph Streams (Chapters 3-6)

"How can we rapidly and accurately keep track of the count of triangles in large dynamic graphs?"

The count of *triangles* (i.e., cliques of size three) is a key primitive in graph analysis. Many important structure-related measures, including the clustering coefficients [WF94], the transitivity ratio [New03], and the triangle connectivity [BZ07], are based on the counts of *global triangles* (i.e., all triangles in the graph) and *local triangles* (i.e., all triangles incident to each node). In addition to structure analysis, the counts of global and local triangles have been used in numerous applications, including link recommendation [TDM+11], anomaly detection [LJK18], and spam detection [BBCG10]. In Chapters 4-6, we develop fast and accurate approximate algorithms for keeping track of the counts of global and local triangles in large dynamic graphs. Specifically, we model large dynamic graphs as *graph streams*, which are a sequence of edges that (a) may not fit in the underlying storage and (b) can be examined in only one pass, and we develop *four* streaming algorithms with distinct advantages:



(a) Temporal Locality (b) Performance of CoCoS

Figure 1.2: [Chapters 4-6]
(a) Temporal locality: Real-world dynamic graphs have distinct temporal patterns.
(b) Performance of CoCoS: It rapidly and accurately estimates the count of triangles.

- Exploiting Patterns (Chapter 4): We develop WRS, which exploits temporal locality for accurate triangle counting. *Temporal locality* (see Figure 1.2(a)) is a temporal pattern found in the formation of triangles in real-world dynamic graphs. Given the same space budget, WRS is up to $1.9\times$ *more accurate* than its best competitors.
- Utilizing Multiple Machines (Chapter 5): We develop Tri-Fly and CoCoS, which are the first distributed streaming algorithms for triangle counting. Given the same space budget, CoCoS is up to $39\times$ *more accurate* than Tri-Fly, which significantly outperforms the best single-machine algorithms (see Figure 1.2(b)).
- Handling Deletions (Chapter 6): We develop ThinkD, which can handle *fully dynamic graphs*, where edges are not only added but also deleted over time. Given the same space budget, ThinkD is up to $4.3\times$ *more accurate* than its best competitors.

We show theoretically and experimentally that all the above algorithms provide unbiased estimates and scale to large-scale graphs with *100 billion edges*.

Contributions:

- Accurate Algorithms: We develop *four* algorithms with distinct advantages (WRS [Shi17], Tri-Fly [SHL+18], CoCoS [SLO+19], ThinkD [SKHF18]) for counting global and local triangles in graph streams. All of them give unbiased estimates and scale to graphs with *100 billion edges*. Given the same space budget, they are up to $39\times$ *more accurate* than their respective competitors.
- Useful Pattern in Real Data: We discover temporal locality in triangles of real graph streams. It can be exploited for a variety of applications, and WRS [Shi17] exploits it for accurate triangle counting.

Impact:

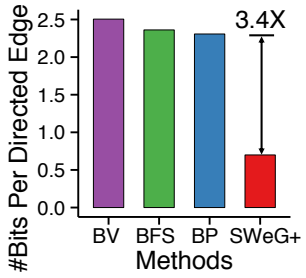
- WRS [Shi17], Tri-Fly [SHL+18], and ThinkD [SKHF18] have been open-sourced and downloaded *88* times from *13* countries.

1.2.1.2 Summarizing Large Graphs and Tensors (Chapters 7 and 8)

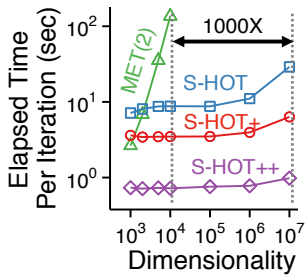
"How can we concisely describe web-scale graphs and tensors?"

Summarizing graphs and tensors (i.e., concisely representing them) is a direct way to make sense of their structure. In Chapters 7 and 8, we develop an distributed algorithm and an external-memory algorithm for summarizing web-scale graphs and tensors, respectively.

In Chapter 7, we formulate graph summarization as an optimization problem of finding the most concise representation consisting of (a) a cluster-level graph and (b) edge corrections for restoring the input graph from the cluster-level graph exactly or within error bounds. Then, we develop SWeG, a distributed optimization algorithm for web-scale graph summarization. Implemented in the MapReduce framework, SWeG summarizes a $25\times$ *larger* graph with over *20 billion edges* than its best



(a) Compression by SWEG



(b) Scalability of S-HOT

Figure 1.3: [Chapters 7-8]
(a) Compression by SWEG: It provides concise and lossless representations of graphs.
(b) Scalability of S-HOT: It provides *1000× better scalability* than its best competitor.

competitors, without quality loss. Moreover, combined with state-of-the-art compression methods, SWEG losslessly compresses a billion-scale web graph with an unprecedented compression rate (see Figure 1.3(a)).

In Chapter 8, we develop S-HOT, a fast external-memory algorithm for the Tucker decomposition [Tuc66], which is a widely-used tensor-summarization method. The Tucker decomposition has been used in many applications, including web search [SZL+05], network forensics [STF06], social network analysis [CTT06], and scientific data compression [ABK16]. By dramatically reducing the space required for intermediate data, S-HOT summarizes a high-order tensor with *1000× larger dimensionality* than the previous best algorithm for the Tucker decomposition, without quality loss (see Figure 1.3(b)). Using S-HOT, we successfully identify groups of similar publication venues from the Microsoft Academic Graph, modeled as a *4-order* tensor with *35 million* non-zero entries.

Contributions:

- Scalable Algorithms: We develop a distributed algorithm (SWEG [SGKR19]) and an external-memory algorithm (S-HOT [OSP+17]) for summarizing web-scale graphs and tensors, respectively. They summarize *25-1000× larger data* without quality loss than their respective competitors.
- Effective Compression of Real Data: By employing SWEG, we losslessly compress a billion-scale web graph with an unprecedented compression rate of *0.7 bits per directed edge*.

Impact:

- A patent application on SWEG [SGKR19] was filed by *LinkedIn Inc.* in September 2018.

1.2.2 Part II: Anomaly Detection

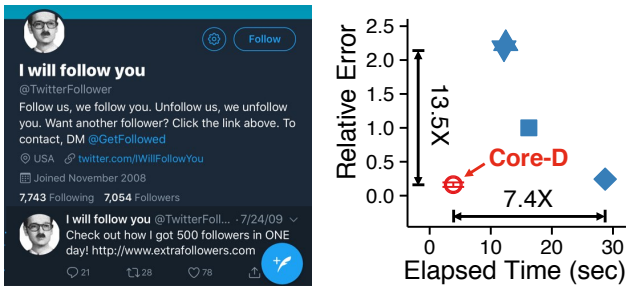
Various types of anomalies in the real world are signaled by surprising substructures, such as unusually dense subgraphs and subtensors. In Part II, we address two tasks closely related to anomaly detection: (a) finding patterns and anomalies in dense subgraphs of real-world graphs, and (b) detecting dense subtensors in large dynamic tensors.

1.2.2.1 Finding Patterns and Anomalies in Dense Subgraphs (Chapter 9)

"What are patterns and anomalies in dense subgraphs of large real-world graphs?"
"How can we exploit the patterns and anomalies to design efficient algorithms?"

Finding patterns is a necessary step for identifying anomalies that deviate from the patterns. In Chapter 9, we discover *three* empirical patterns in dense subgraphs (specifically, k-cores[1]) of various

[1]The *k-core* of a graph [Sei83] is its maximal subgraph where every node is adjacent to at least k nodes.



(a) Effectiveness of CORE-A (b) Performance of CORE-D

Figure 1.4: [Chapter 9]
(a) Effectiveness of CORE-A: It spots a 'follower booster' on Twitter.
(b) Performance of CORE-D: It rapidly and accurately estimates degeneracy.

types of real-world graphs, including social networks, web graphs, internet topology graphs, and citation networks. Then, we develop *three* algorithms that exploit the above patterns for anomaly detection and two other tasks:

- CORE-A: a near-linear time algorithm for identifying anomalies deviating from the patterns. Such anomalies include 'copy-and-paste' bibliography, a propeller-shaped web graph, and a 'follower-boosting' service in a billion-scale Twitter graph (See Figure 1.4(a)).
- CORE-D: a single-pass, sublinear-space algorithm for estimating degeneracy,[2] a classic connectivity measure. It is up to $7\times$ *faster* than its best competitors with similar accuracy (See Figure 1.4(b)).
- CORE-S: a fast algorithm for identifying influential spreaders. It is up to $17\times$ *faster* than its best competitors with similar accuracy.

Contributions:

- Useful Patterns in Real Data: We discover *three* empirical patterns in dense subgraphs of real graphs. They can be exploited for a variety of applications. Our algorithms (CORE-A, CORE-D, and CORE-S) exploit them for anomaly detection, degeneracy estimation, and influential spreader detection.
- Anomalies in Real Data: Using CORE-A, we detect many notable anomalies, including a 'follower booster' in a billion-scale Twitter graph. It has not been suspended or removed for over 9 *years*.

Impact:

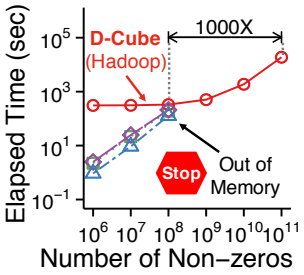
- This work [SERF16] was selected as one among the *"Bests of ICDM 2016"* and invited to the Knowledge and Information Systems journal [SSK+18].
- This work was included in *MIT*'s graduate course on graph analytics (MIT 6.886), an *ICDM 2016* tutorial on core decomposition, and an *ECML/PKDD 2017* tutorial on the same topic.
- The code used in this work has been open-sourced and downloaded 60 times from 14 countries.

1.2.2.2 Detecting Dense Subtensors in Large Tensors (Chapters 10-13)

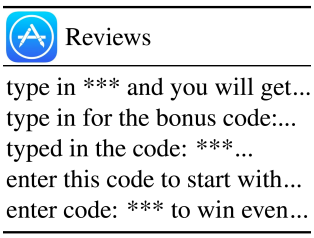
"How can we rapidly and accurately detect unusually dense subtensors in large dynamic tensors?"
"Which types of anomalies and fraud do such dense subtensors signal?"

Many types of fraud, including network intrusion [MGF11], search engine manipulation [GKT05], 'Like' boosting on Facebook [BXG+13], and 'Retweet' boosting on Weibo [JBC+16], are signaled by synchronized behavior, such as a set of accounts giving reviews with the same ratings and text to the same set of products within a short time.

[2]The *degeneracy* of a graph is the highest value k such that the k-core exists in the graph



(a) Scalability of D-CUBE



(b) Effectiveness of D-CUBE

Figure 1.5: [Chapters 11-13]
(a) Scalability of D-CUBE: It provides $1000\times$ *better scalability* than in-memory algorithms.
(b) Effectiveness of D-CUBE: It spots spam reviews on App Store.

In Chapters 11-13, we formulate identifying synchronized behavior as an optimization problem of finding the densest subtensors in the input data modeled as a tensor. Then, we develop *four* approximate algorithms with distinct advantages:

- In-memory Algorithm (Chapter 11): We develop M-ZOOM, a near-linear time algorithm for detecting dense subtensors. It is up to $114\times$ *faster* than the previous best algorithm with similar accuracy.
- External-memory Algorithm (Chapter 12): We develop D-CUBE, the first external-memory algorithm for detecting dense subtensors in web-scale tensors. Our MAPREDUCE implementation of D-CUBE scales to a $1000\times$ *larger* tensor with 100 *billion* non-zero entries than in-memory algorithms (see Figure 1.5(a)).
- Incremental Algorithms (Chapter 13): We develop DENSESTREAM and DENSEALERT, the first incremental algorithms for detecting dense subtensors in dynamic tensors. Each update by them is up to $10^6\times$ *faster* than running batch algorithms from scratch.

We demonstrate that all the above algorithms guarantee an *approximation ratio of* $1/n$ for n-order tensors, and they accurately identify many interesting anomalies, including 'edit wars' and bots on Wikipedia, spam reviews on App Store (see Figure 1.5(b)), and various types of network attacks.

Contributions:

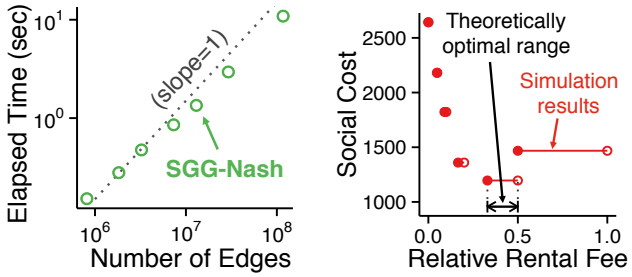
- Fast Algorithms: We develop *four* algorithms with distinct advantages (M-ZOOM [SHF18], D-CUBE [SHKF18], DENSESTREAM, and DENSEALERT [SHKF17b]) for finding dense subtensors in large dynamic tensors. They all achieve an *approximation ratio of* $1/n$ for n-order tensors. They are up to $332\times$ *faster* with $1000\times$ *better scalability* than previous methods, without accuracy loss.
- Anomalies in Real Data: Using our algorithms, we detect many notable anomalies, including 'edit wars' and bots on Wikipedia, spam reviews on App Store, and various types of network attacks.

Impact:

- D-CUBE [SHKF18] was used in *production* at *NAVER Corp.*, which handled 74.7% of web searches in South Korea in 2017, to identify and filter spam sites.
- M-ZOOM [SHF18], D-CUBE [SHKF18], DENSESTREAM, and DENSEALERT [SHKF17b] have been open-sourced and downloaded 217 times from 20 countries.

1.2.3 Part III: Behavior Modeling

In Part III, we address two tasks closely related to modeling the behaviors of individuals in graph and tensor data: (a) modeling purchases in social networks, and (b) modeling progression of users on social media. For the tasks, social networks are modeled as graphs, and behavior logs on social media are modeled as tensors.



(a) Scalability of SGG-NASH (b) Usefulness of Our Tools

Figure 1.6: [Chapter 14]
(a) Scalability of SGG-NASH: It shows linear scalability.
(b) Usefulness of Our Tools: Our analysis and simulation suggest socially optimal rental fees.

1.2.3.1 Modeling Purchases in Social Networks (Chapter 14)

"Why should we charge our friends for borrowing our stuff?"

Social networks are known to play an important role in the everyday choices people make [Mar87, Blu93, Ell93, Rog10]. In Chapter 14, we consider the purchase of *sharable goods*, which can be shared with k-hop neighbors (i.e., the set of nodes within k hops from an owner) on a social network, modeled as a graph. Examples of such goods are seldom-used ski gear and hiking equipments, which are frequently borrowed by friends or friends of friends. To examine incentives to buy shareable goods, we develop two game-theoretic models (SGG and SGG-AC) where each node (i.e., individual) decides whether to buy a good or rent a good from an owner within k hops with or without a rental fee; and we also develop a fast algorithm (SGG-NASH) for finding Nash equilibria of both games (see Figure 1.6(a)). Through theoretical analysis and simulation using SGG-NASH and real-world social networks, we show how rental fees affect the social inefficiency of Nash equilibria, and we suggest a range of rental fees for minimizing the inefficiency (see Figure 1.6(b)).

Contributions:

- Game-theoretic Models and Fast Tool: We develop SGG and SGG-AC, game-theoretic models for purchases of sharable goods on a social network; and we develop SGG-NASH, a fast algorithm for finding Nash equilibria of both games.
- Social Inefficiency on Real Data: Through theoretical analysis and simulation, we show potential social inefficiency on real-world social networks and suggest a socially optimal range of rental fees for minimizing the inefficiency.

Impact:

- This work [SLEP17] was featured in *New Scientist* in May 2017. [3]

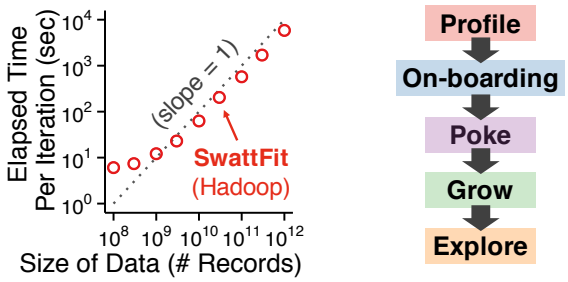
1.2.3.2 Modeling Progression of Users on Social Media (Chapter 15)

"How do the behaviors of individuals on social media progress over time?"

The behaviors of users on websites (e.g., social media) change over time due to many reasons, such as temporal trends [HM16], shift of personal interests [Liu15], and personal development [ML13]. *Progressions* of users on a website are common behavioral changes that many users go though as they become accustomed to or engaged in the website. For website operators, it is important to understand such progressions in order to provide more personalized experiences.

In Chapter 15, we develop a behavior model (SWATT) and an optimization algorithm (SWATTFIT) for discovering and summarizing progressions of users. SWATT describes progressions as transitions

[3] https://www.newscientist.com/article/2132926



(a) Scalability of SWATTFIT (b) Usefulness of our Tools

Figure 1.7: [Chapter 15]
(a) Scalability of SWATTFIT: It scales linearly to trillion-scale data.
(b) Usefulness of Our Tools: They discover meaningful progressions of LinkedIn users.

between stages, each of which is described by three probability distributions over (a) features used, (b) frequencies of usage, and (c) next stages to move. SWATTFIT fits SWATT to behavior logs, modeled as a tensor. SWATTFIT is a linear-time distributed algorithm, and its MAPREDUCE implementation scales to *trillion-scale* behavior logs (see Figure 1.7(a)). We apply SWATT and SWATTFIT to behavior logs from *LinkedIn*, a social media for businesses and professionals, and discover meaningful progressions of its users (see Figure 1.7(b)).

Contributions:

- Comprehensive Model and Scalable Tool: We develop SWATT a comprehensive behavior model for describing progressions in three aspects; and we develop SWATTFIT, a linear-time distributed algorithm for fitting SWATT to behavior logs with *trillions* of records.
- Progressions in Real Data: Using SWATT and SWATTFIT, we discover meaningful progressive stages that users of LinkedIn go through.

Impact:

- SWATT and SWATTFIT [SSK+18] were used at *LinkedIn Inc.* to analyze user behavior.

1.3 Thesis Organization

We now describe the organization of this thesis. In Chapter 2, we introduce basic concepts and notations on graphs and tensors that are used throughout the thesis. The next three parts (i.e., Parts I, II, and III) correspond to our work on structure analysis, anomaly detection, and behavior modeling, respectively. See Table 1.1 for the main problems of each part in the form of questions. In Part I, we present our work on triangle counting (Chapters 3-6), graph summarization (Chapter 7), and tensor summarization (Chapter 8). In Part II, we present our work on patterns and anomalies in dense subgraphs (Chapter 9) and dense-subtensor detection (Chapters 10-13). In Part III, we present our work on purchase behavior modeling (Chapter 14) and progressive behavior modeling (Chapter 15). Finally, in Part IV, we provide conclusions (Chapter 16) and discuss future research directions (Chapter 17).

Table 1.1: **Organization of the thesis.**

Part	Research Problem	Chapter
I: Structure Analysis	• **Counting Triangles in Graph Streams:** How can we accurately count the triangles in large dynamic graphs with fixed memory size? • **Summarizing Large Graphs and Tensors:** How can we concisely describe graphs and high-order tensors that are too large to fit in memory?	3 - 6 7 - 8
II: Anomaly Detection	• **Finding Patterns and Anomalies in Dense Subgraphs:** What are patterns and anomalies in dense subgraphs of large real-world graphs? • **Detecting Dense Subtensors in Large Tensors:** How can we rapidly and accurately detect anomalously dense subtensors in large dynamic tensors?	9 10 - 13
III: Behavior Modeling	• **Modeling Purchases in Social Networks:** What are the incentives for buying goods sharable with friends? What is the socially optimal rental fee? • **Modeling Progression of Users on Social Media:** How do the behaviors of individuals on social media evolve over time?	14 15

Chapter 2

Background

In this chapter, we introduce some key concepts and notations that are used throughout this thesis. The notations are also listed in Table 2.1, which is at the end of this chapter.

2.1 Graph-related Concepts and Notations

Graph: A *graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a pair of two sets $\mathcal{V}$ and $\mathcal{E}$ where every element of $\mathcal{E}$ is an unordered pair of elements of $\mathcal{V}$. Each element of $\mathcal{V}$ is called a *node*, and each element $e = \{u, v\}$ of $\mathcal{E}$ is called the *edge* between nodes u and v. A graph naturally represents a set of objects where some pairs of the objects are related in certain ways. Examples are as follows:

- **[Social network]**
  - The node set $\mathcal{V}$ is people.
  - The edge set $\mathcal{E}$ is the pairs of friends.
- **[E-commerce]**
  - The node set $\mathcal{V}$ is people and products.
  - The edge set $\mathcal{E}$ is the pairs of a user and a product where the user bought the product.
- **[Network security]**
  - The node set $\mathcal{V}$ is IPs.
  - The edge set $\mathcal{E}$ is the pairs of IPs where connections were made between them.

Neighbor and Degree: We say a node u is a *neighbor* of a node v (or u is *adjacent* to v) when the edge between u and v exists (i.e., $\{u, v\} \in \mathcal{E}$). For each node v, we use $\mathcal{N}_v$ to indicate the set of neighbors of v. The *degree* of a node v is defined as the number of neighbors of v.

Subgraph, Clique, and Triangle: We say a graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ is a *subgraph* of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ when $\mathcal{V}' \subset \mathcal{V}$ and $\mathcal{E}' \subset \mathcal{E}$. We say a subset $\mathcal{V}'$ of $\mathcal{V}$ is a *clique* if every pair of nodes in $\mathcal{V}'$ is connected by an edge in $\mathcal{E}$. A clique of three nodes is called a *triangle*.

k-Core, Coreness, and Degeneracy: The *k-core* of a graph is the maximal subgraph where every node has degree at least k within the subgraph. The *coreness* of a node is the largest k such that it belongs to the k-core. The *degeneracy* of a graph is the largest k such that the k-core exists.

(Fully-Dynamic) Graph Stream: A *graph stream* is an ordered sequence of edges. A *fully-dynamic graph stream* is an ordered sequence of changes, each of which is either an edge insertion or an edge deletion. The edges in a graph stream and the changes in a fully-dynamic graph streams can be accessed once in the given order unless they are explicitly stored in memory.

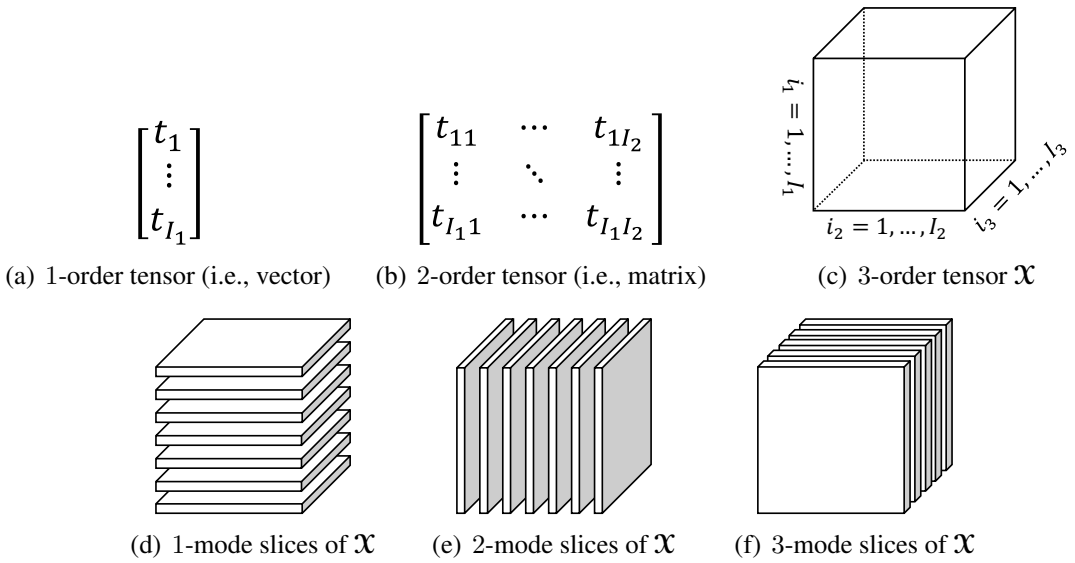
2.2 Tensor-related Concepts and Notations

Tensor: A *tensor* is a multi-dimensional array of entries. The *order* of a tensor is the number of dimensions, also known as *modes*. Consider an N-order (or N-way) tensor $\mathcal{X}$ of size $I_1 \times ... \times I_N$. We denote each $(i_1, ..., i_N)$-th entry of $\mathcal{X}$ as $x_{i_1 ... i_N}$ where each *n-th mode index* i_n ranges from 1 to I_n. For each n-th mode, we call I_n the *dimensionality* of the mode. Figures 2.1(a)-2.1(c) show illustrations of tensors. Tensors have more expressive power than graphs, as the following examples show:

- **[Social network]**
  - Consider a 3-order tensor $\mathcal{X}$ whose modes are people, people, and dates, respectively.
  - An entry $x_{i_1 i_2 i_3}$ is 1 if the i_1-th person and the i_2-th person became friends on the i_3-th date. Otherwise, $x_{i_1 i_2 i_3}$ is 0.
- **[E-commerce]**
  - Consider a 3-order tensor $\mathcal{X}$ whose modes are products, users, and dates, respectively.
  - An entry $x_{i_1 i_2 i_3}$ is the number of the i_1-th products bought by the i_2-th user on the i_3-th date.
- **[Network security]**
  - Consider a 4-order tensor $\mathcal{X}$ whose modes are IPs, IPs, protocols, and dates, respectively.
  - An entry $x_{i_1 i_2 i_3 i_4}$ is the number of connections made from the i_1-th IP to the i_2-th IP using the i_3-th protocol on the i_4-th date.

Slice, (Weighted) Degree, and Slice Sum: The *slices* of an N-order tensor are the $(N-1)$-order tensors obtained by fixing an mode index. Among the slices, the *n-mode slices* are those obtained by fixing the n-th mode index. We use $q = (n, i_n)$ to indicate the n-mode slice formed by the entries whose n-mode index is i_n. Figures 2.1(d)-2.1(f) show illustrations of the slices of a 3-order tensor. We define the *degree* of a slice as the number of non-zero entries of the slice. We define the *weighted degree* (or *slice sum*) of a slice as the sum of the entries of the slice.

Subtensor: We say an N-order tensor $\mathcal{Y}$ is a subtensor of an N-order tensor $\mathcal{X}$ when $\mathcal{Y}$ can be obtained by removing some slices from $\mathcal{X}$.



(a) 1-order tensor (i.e., vector) (b) 2-order tensor (i.e., matrix) (c) 3-order tensor $\mathcal{X}$

(d) 1-mode slices of $\mathcal{X}$ (e) 2-mode slices of $\mathcal{X}$ (f) 3-mode slices of $\mathcal{X}$

Figure 2.1: **Illustration of tensors and slices.**

Table 2.1: **Table of frequently-used symbols.**

Symbols	Definitions
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	a graph
$\mathcal{V}$	node set of $\mathcal{G}$
$\mathcal{E}$	edge set of $\mathcal{G}$
$e = \{u, v\}$	edge between nodes u and v
$\mathcal{N}_v$	set of neighbors of node v in $\mathcal{G}$
$\mathcal{X}$	a tensor of size $I_1 \times ... \times I_N$
N	order of $\mathcal{X}$
I_n	dimensionality of the n-th mode of $\mathcal{X}$
$x_{i_1 \cdots i_N}$	$(i_1, ..., i_N)$-th entry of $\mathcal{X}$
$q = (n, i_n)$	slice of $\mathcal{X}$ formed by the entries whose n-th mode index is i_n

Part I

Structure Analysis

Chapter 3

Counting Triangles in Graph Streams (0): Preliminaries

In this preliminary chapter on triangle counting, we (a) provide motivation for triangle counting, (b) review related work, and (c) introduce some concepts frequently used in the following chapters on triangle counting.

3.1 Motivation

Counting the triangles (i.e., cliques of size three) in a graph is a classical problem with numerous applications. For example, triangles in social networks have received much attention as an evidence of homophily (i.e., people choose friends similar to themselves) [MSLC01] and transitivity (i.e., people with common friends become friends) [WF94]. Thus, many structure-related concepts in social network analysis, such as the transitivity ratio [New03], local clustering coefficients [WS98], triangle connectivity [BZ07], trusses [Coh08], and social balance [WF94] are based on the counts of *global triangles* (i.e., all triangles) and *local triangles* (i.e., all triangles incident to each node). Additionally, global and local triangle counts have been used for link recommendation [TDM+11, ELM+15], anomaly detection [LJK18], spam detection [BBCG10], community detection [BHLP11], dense subgraph mining [WZTT10], web analysis [EM02], degeneracy estimation [SERF18] (see Chapter 9), and query optimization [BYKS02].

3.2 Related Work

We review previous work on triangle-counting algorithms, with a focus on streaming algorithms and distributed algorithms. See Table 3.1 for a summary.

Single-machine Streaming Algorithms for Insertion-only Streams: (1) Global Triangle Counting. Most streaming algorithms for triangle counting employ sampling for estimation with limited storage. Tsourakakis et al. [TKMF09] proposed sampling each edge independently with equal probability p and then estimating the count of *global triangles* (i.e., all triangles in a graph) from that in the sampled graph using the fact that each triangle is sampled with probability p^3. To increase the probability from p^3 to p^2, Pagh and Tsourakakis [PT12] proposed the colorful sampling scheme where each node is colored with a color chosen uniformly at random among $1/p$ colors and the edges whose endpoints have the same color are stored. Kallaugher and Price [KP17] proposed sampling each node with equal probability

Table 3.1: **Comparison of triangle-counting algorithms. Our proposed algorithms (i.e., WRS, TRI-FLY, COCOS, and THINKD) provide distinct advantages.**

	TRIEST$_{FD}$ [SERU17]	ESD [HS17]	MASCOT [LJK18]	TRIEST$_{IMPR}$ [SERU17]	Others (Streaming) [ADNK14, ADWR17, KP17] [JSP13, PTTW13, TPT13]	PATRIC [AKM13]	Others (Distributed) [PC13, PSKP14, PMK16] [Coh09, SV11, PSP+18]	WRS (Chapter 4)	TRI-FLY (Chapter 5) COCOS (Chapter 5)	THINKD (Chapter 6)
Counting Global Triangles	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Counting Local Triangles	✓		✓	✓			✓	✓	✓	✓
Handling Large Graphs*	✓		✓	✓	✓	✓		✓	✓	✓
Handling Edge Insertions	✓	✓	✓	✓	✓			✓	✓	✓
Handling Edge Deletions	✓	✓								✓
Utilizing Temporal Patterns								✓		
Utilizing Multiple Machines						✓	✓		✓	

* graphs that do not fit in memory

p and storing all edges between the sampled nodes and p of the edges between sampled nodes and unsampled nodes. This requires less samples than the colorful sampling scheme for the same accuracy guarantee [KP17]. Jha et al. [JSP13] and Pavan et al. [PTTW13] proposed sampling wedges (i.e., paths of length two) in addition to edges; and Ahmed et al. [ADNK14, ADWR17] proposed sampling edges with different probabilities, depending on the counts of adjacent sampled edges and incident triangles. Tangwongsan et al. [TPT13] proposed a shared-memory, parallel, cache-oblivious version of [PTTW13].

Single-machine Streaming Algorithms for Insertion-only Streams: (2) Local Triangle Counting. The colorful sampling scheme [PT12], described in the previous paragraph, was applied to estimate the counts of *local triangles* (i.e., all triangles incident to each node) [KP13]. Lim and Kang [LJK18] proposed MASCOT, which uses simple uniform edge sampling but updates its estimates whenever an edge arrives even if it is not sampled. De Stefani et al. [SERU17] proposed TRIEST$_{IMPR}$, which uses reservoir sampling to maintain as many sample edges as storage allows. WRS [Shi17], presented in Chapter 4, improved upon TRIEST$_{IMPR}$ in terms of accuracy under the assumption that edges are streamed in the order that they are created. In addition, Becchetti et al. [BBCG10] explored semi-streaming algorithms that require multiple passes over the stream. TRI-FLY [SHL+18] and COCOS [SLO+19], presented in Chapter 5, adapt TRIEST$_{IMPR}$ for triangle counting within each machine. However, any single-machine streaming algorithm including WRS [Shi17], can be used instead.

Single-machine Streaming Algorithms for Fully-dynamic Streams. The first algorithm for triangle counting in fully dynamic graph streams with edge deletions was proposed by Kutzkov and Pagh [KP14]. The algorithm estimates the count of global triangles by making a single pass over the input stream. However, the algorithm is inapplicable to real-time applications since it expensively computes an estimate once at the end of the stream instead of always maintaining an estimate. Moreover, in the worst case, the algorithm requires more memory than what is needed to store the entire input graph,

as pointed out in [SERU17]. Han and Sethu [HS17] proposed ESD, which maintains and updates an estimate of the global triangle count. However, its scalability is limited since it requires the entire input graph to be stored in memory. De Stefani et al. [SERU17] proposed TRIEST$_{FD}$, which maintains and updates estimates of both global and local triangle counts, and it scales better than ESD by sampling edges within a given memory budget and discarding the other edges. In Chapter 6, we present THINKD [SKHF18], which improves upon TRIEST$_{FD}$ in terms of accuracy. While TRIEST$_{FD}$ simply discards unsampled edges, THINKD utilizes unsampled edges to update estimates before discarding them. Although the idea of using unsampled edges had been considered for insertion-only streams [SERU17, Shi17, LJK18], applying the idea to fully dynamic graph streams had remained unexplored.

Distributed Batch Algorithms. Cohen [Coh09] proposed the first triangle-counting algorithm implemented in the MAPREDUCE framework. The algorithm directly parallelizes a serial algorithm. Suri and Sergei [SV11], Park et al. [PC13, PSKP14, PMK16, PSP+18], and Shaikh et al. [AKM13] proposed dividing the input graph into overlapping subgraphs and assigning them to multiple machines, which count the triangles in the assigned subgraphs in parallel, in MAPREDUCE [SV11, PC13, PSKP14, PMK16] and distributed-memory [AKM13] settings. Recently, Ko and Kim [KH18] proposed an external-memory distributed graph analytics system that supports triangle counting. These distributed methods are for exact triangle counting in static graphs, all of whose edges are given at once. They are inapplicable to graph streams, whose edges are received over time and may not fit in the underlying storage.

Distributed Streaming Algorithms. Distributed streaming algorithms for triangle counting were first discussed by Pavan et al. [PTT13] to handle multiple sources. The algorithms aim to reduce communication costs while giving the same estimation of a single-machine streaming algorithm [PTTW13]. Thus, using more machines, which are one per source, neither improves the speed nor the accuracy of the estimation. In TRI-FLY [SHL+18] and COCOS [SLO+19], presented in Chapter 5, multiple machines are used for rapid and accurate estimation. In TRI-FLY, every edge is broadcast to every worker that independently runs TRIEST$_{IMPR}$ [SERU17]. The workers send their estimates to the aggregators, which give the final estimates by averaging the received estimates. Although TRI-FLY gives unbiased estimates whose variances decrease inversely proportional to the number of workers, it incurs a highly redundant use of computational and storage resources. Specifically, in the worst case, each edge is replicated and stored in every worker, and each triangle is counted by every worker. Due to this redundancy, no matter how many workers are used, TRI-FLY cannot guarantee exact triangle counts if the input stream does not fit in each machine. COCOS significantly improves upon TRI-FLY in terms of speed and accuracy by minimizing the redundancy in storage and computation.

3.3 Concepts

We provide some concepts that are frequently used in the following chapters on triangle counting.

3.3.1 Reservoir Sampling

The reservoir sampling (RS) [Vit85] is an algorithm for choosing a sample of b items from a stream of items whose size is unknown or growing. RS keeps the first b items. When the $l(> b)$-th item arrives, RS either replaces a randomly-chosen old item with the new item with probability b/l or discards the new item with probability $1 - b/l$. Lemma 3.1 is a special case of Lemma 4.1 in [SERU17].

Lemma 3.1: Uniformity of Reservoir Sampling

Let $\mathcal{S}_l$ be the set of (at most b) items kept by RS after processing the l-th item. Then, for any $b \geq 2$, $l \geq 2$, and two items $x \neq y$ that arrived so far,

$$P[x \in \mathcal{S}_l \cap y \in \mathcal{S}_l] = \min\left(1, \frac{b(b-1)}{l(l-1)}\right).$$

3.3.2 Evaluation Metrics for Triangle Counting

We introduce the metrics that we use to measure the accuracy of global and local triangle counting in the later chapters. Let x be the count of the *global triangles* (i.e., all triangles) at the end of the input stream and $\hat{x}$ be an estimate of it. Likewise, let $x[u]$ be the count of the *local triangles* of each node $u \in \mathcal{V}$ (i.e., all triangles incident to u) at the end of the input stream and $\hat{x}[u]$ be an estimate of it. Then, the metrics are defined as follows:

- **Global Error** (the lower the better):

$$|x - \hat{x}|/(x+1).$$

- **Local Error** [LJK18] (the lower the better):

$$\frac{1}{|\mathcal{V}|}\sum\nolimits_{u \in \mathcal{V}} \frac{|x[u] - \hat{x}[u]|}{x[u]+1}.$$

- **RMSE** (the lower the better):

$$\sqrt{\frac{1}{|\mathcal{V}|}\sum_{u \in \mathcal{V}}(x[u] - \hat{x}[u])^2}.$$

- **Rank Correlation** (the higher the better):
  Spearman's rank correlation coefficient [Spe04] between $\{(u, x[u])\}_{u \in \mathcal{V}}$ and $\{(u, \hat{x}[u])\}_{u \in \mathcal{V}}$.

Chapter based on work that appeared at ICDM 2017 [Shi17].

Chapter 4

Counting Triangles in Graph Streams (1): Exploiting Temporal Patterns

Given a graph stream, how can we estimate the count of triangles in it with fixed amount of memory? If we cannot store all edges in the stream, which edges should we store to estimate the triangle count accurately?

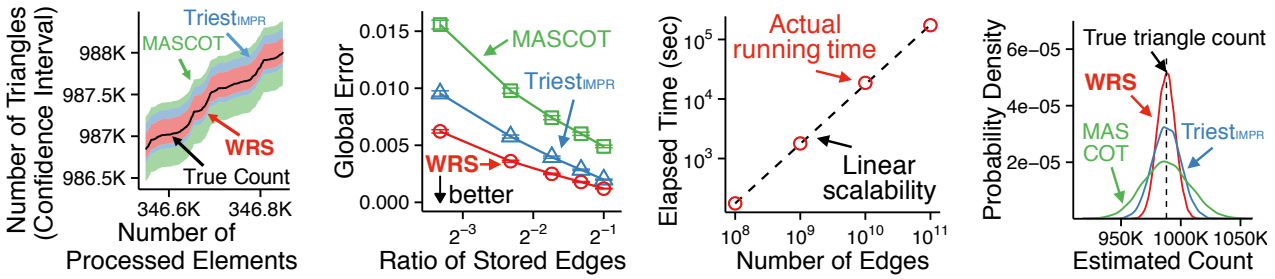
As discussed in the previous chapter, counting triangles (i.e., cliques of size three) is a fundamental graph problem with numerous applications in social network analysis, web mining, anomaly detection, etc. Recently, much effort has been made to accurately estimate the count of triangles in dynamic graphs, represented as graph streams, with limited space. However, existing streaming algorithms use sampling techniques without considering temporal dependencies in edges, while we observe *temporal locality* in real-world dynamic graphs. That is, future edges are more likely to form triangles with recent edges than with older edges.

In this chapter, we propose *Waiting-Room Sampling* (WRS), a single-pass streaming algorithm for estimating the counts of global triangles (i.e., all triangles) and local triangles incident to each node. WRS exploits the temporal locality by always storing the most recent edges, which future edges are more likely to form triangles with, in the *waiting room*, while it uses reservoir sampling for the remaining edges. We theoretically and empirically show that WRS is: *(a) Fast and 'any time':* runs in linear time, always maintaining and updating estimates while new edges arrive, *(b) Accurate*: yields up to **47% smaller estimation error** than its best competitors, and *(c) Theoretically sound*: gives unbiased estimates with small variances under the temporal locality.

4.1 Motivation

Dynamic graphs are naturally modeled as graph streams where new edges are streamed as they are created. Given such a graph stream, how can we accurately estimate the count of triangles? Especially, if we cannot store all the edges in memory, which edges should we store for accurate estimation?

As discussed in the previous chapter, many streaming algorithms have been developed for estimating the counts of *global triangles* (i.e., all triangles) and *local triangles* incident to each node in large dynamic graphs. However, all the algorithms sample edges without considering temporal dependencies in edges and thus cannot exploit *temporal locality*, i.e., the tendency that future edges are more likely to form triangles with recent edges than with older edges. We observe this temporal locality commonly



(a) 'Any time' for dynamic graphs with new edges

(b) Accurate

(c) Scalable (Theorem 4.2)

(d) Unbiased with small variance (Theorem 4.1)

Figure 4.1: **Strengths of WRS.** (a) *'Any time':* WRS always maintains the estimates of the global and local triangle counts while the input graph grows with new edges. (b) *Accurate:* WRS is more accurate than its best competitors. (c) *Scalable:* WRS scales linearly with the number of edges in the input stream. (d) *Unbiased:* WRS gives unbiased estimates (Theorem 4.1). See Section 4.6 for details.

in many realistic graph streams, where new edges are streamed as they are created. Then, how can we exploit the temporal locality for accurately estimating the counts of global and local triangles?

In this second chapter on triangle counting, we propose WRS (**W**aiting-**R**oom **S**ampling), a single-pass streaming algorithm that always stores the most recent edges in the *waiting room*, while it uses standard reservoir sampling (see Section 3.3.1) for the remaining edges. The waiting room increases the probability that, when a new edge arrives, edges forming triangles with the new edge are in memory. Reservoir sampling, on the other hand, enables WRS to yield unbiased estimates. We theoretically and empirically show that WRS has the following strengths:

- **Fast and 'any time'**: WRS runs in linear time in the number of edges, giving estimates of the global and local triangle counts at any time, not only at the end of streams (Figure 4.1(c)).
- **Accurate**: WRS produces up to *47% smaller* estimation error than its best competitors (Figure 4.1(b)).
- **Theoretically sound**: We prove the unbiasedness of the estimators provided by WRS and their small variances under the temporal locality (Theorem 4.1, Lemma 4.2, and Figure 4.1(d)).

Reproducibility: The source code and datasets used in the chapter are available at `http://www.cs.cmu.edu/~kijungs/codes/wrs/`.

The rest of the chapter is organized as follows. In Section 4.2, we introduce some preliminary concepts, notations, and a formal problem definition. In Section 4.3, we discuss temporal locality in real-world dynamic graphs. In Section 4.4, we present WRS, our proposed algorithm for triangle counting. In Section 4.5, we theoretically analyze the accuracy and complexity of WRS. After sharing some experimental results in Section 4.6, we provide a summary of this chapter in Section 4.7.

4.2 Preliminaries and Problem Definition

In this section, we first introduce some notations and concepts used throughout this chapter. Then, we define the problem of global and local triangle counting in a real-order graph stream.

Table 4.1: **Table of frequently-used symbols.**

Symbol	Definition
Notations for Graph Streams (Section 4.2)	
$\mathcal{G}^{(t)} = (\mathcal{V}^{(t)}, \mathcal{E}^{(t)})$	graph $\mathcal{G}$ at time t
$e^{(t)}$	edge arriving at time t
$\{u, v\}$	edge between nodes u and v
t_{uv}	arrival time of edge $\{u, v\}$
$\{u, v, w\}$	triangle with nodes u, v, and w
$t^{(i)}_{uvw}$	arrival time of the i-th edge in $\{u, v, w\}$
$\mathcal{T}^{(t)}$	set of triangles in $\mathcal{G}^{(t)}$
$\mathcal{T}^{(t)}[u]$	set of triangles with node u in $\mathcal{G}^{(t)}$
Notations for Our Algorithm (Section 4.4)	
$\mathcal{S}$	given storage space
$\mathcal{W}$	waiting room
$\mathcal{R}$	reservoir
b	maximum number of edges stored in $\mathcal{S}$
α	relative size of the waiting room (i.e., $\lvert\mathcal{W}\rvert/\lvert\mathcal{S}\rvert$)
$\hat{\mathcal{G}} = (\hat{\mathcal{V}}, \hat{\mathcal{E}})$	graph composed of the edges in $\mathcal{S}$
$\mathcal{N}_u$	set of neighbors of node u in $\hat{\mathcal{G}}$

4.2.1 Notations and Concepts

Symbols frequently used in the chapter are listed in Table 4.1. Consider an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with the set of nodes $\mathcal{V}$ and the set of edges $\mathcal{E}$. We use the unordered pair $\{u, v\} \in \mathcal{E}$ to indicate the edge between nodes $u \in \mathcal{V}$ and $v \in \mathcal{V}$. The graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ grows over time; and we let $e^{(t)}$ be the edge arriving at time $t \in \{1, 2, ...\}$ and t_{uv} be the arrival time of edge $\{u, v\}$ (i.e., $e^{(t)} = \{u, v\} \Leftrightarrow t_{uv} = t$). Then, we denote $\mathcal{G}$ at time t by $\mathcal{G}^{(t)} = (\mathcal{V}^{(t)}, \mathcal{E}^{(t)})$, which consists of the nodes and edges arriving at time t or earlier. Let the unordered triple $\{u, v, w\}$ be the triangle (i.e., cliques of size three) with edges $\{u, v\}$, $\{v, w\}$, and $\{w, u\}$. We use $t^{(1)}_{uvw}$:=min$\{t_{uv}, t_{vw}, t_{wu}\}$, $t^{(2)}_{uvw}$:=median$\{t_{uv}, t_{vw}, t_{wu}\}$, and $t^{(3)}_{uvw}$:=max$\{t_{uv}, t_{vw}, t_{wu}\}$ to indicate the arrival times of the first, second, and last edges, respectively, in $\{u, v, w\}$. We denote the set of triangles in $\mathcal{G}^{(t)}$ by $\mathcal{T}^{(t)}$ and the set of triangles with node u by $\mathcal{T}^{(t)}[u] \subset \mathcal{T}^{(t)}$. We call $\mathcal{T}^{(t)}$ *global triangles* and $\mathcal{T}^{(t)}[u]$ *local triangles* of node u.

4.2.2 Problem Definition

In this chapter, we consider the problem of counting the global and local triangles in a graph stream assuming the following realistic conditions:

- C1 **No Knowledge:** No information about the input stream (e.g., the node count, the edge count, etc) is available.
- C2 **Real-order:** In the input stream, new edges arrive in the order by which they are created.
- C3 **Limited Storage Budget:** We store at most b edges in the storage.
- C4 **Single Pass:** Edges are processed one by one in their order of arrival. Past edges cannot be accessed unless they are stored in the storage (within the budget stated in C3).

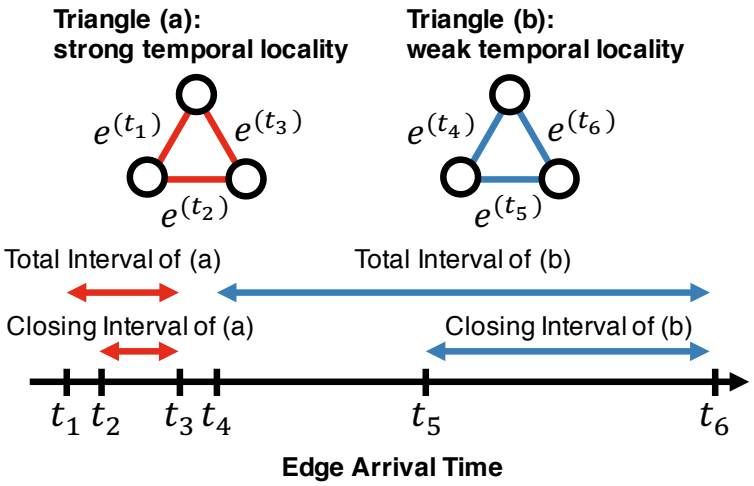


Figure 4.2: **Illustration of total intervals, closing intervals, and temporal locality.**

Based on these conditions, we define the problem of global and local triangle counting in a real-order graph stream in Problem 4.1.

Problem 4.1: Global and Local Triangle Counting in a Real-order Graph Stream

1. **Given:** a real-order graph stream $(e^{(1)}, e^{(2)}, ...)$ and a storage budget b,
2. **Maintain:** estimates of the global triangle count $|\mathcal{T}^{(t)}|$ and the local triangle counts $\{(u, |\mathcal{T}^{(t)}[u]|)\}_{u \in \mathcal{V}^{(t)}}$ for current $t \in \{1, 2, ...\}$
3. **to Minimize:** the estimation errors.

Instead of minimizing a specific measure of estimation error, we follow a general approach of reducing both bias and variance, which is robust to many measures of estimation error.

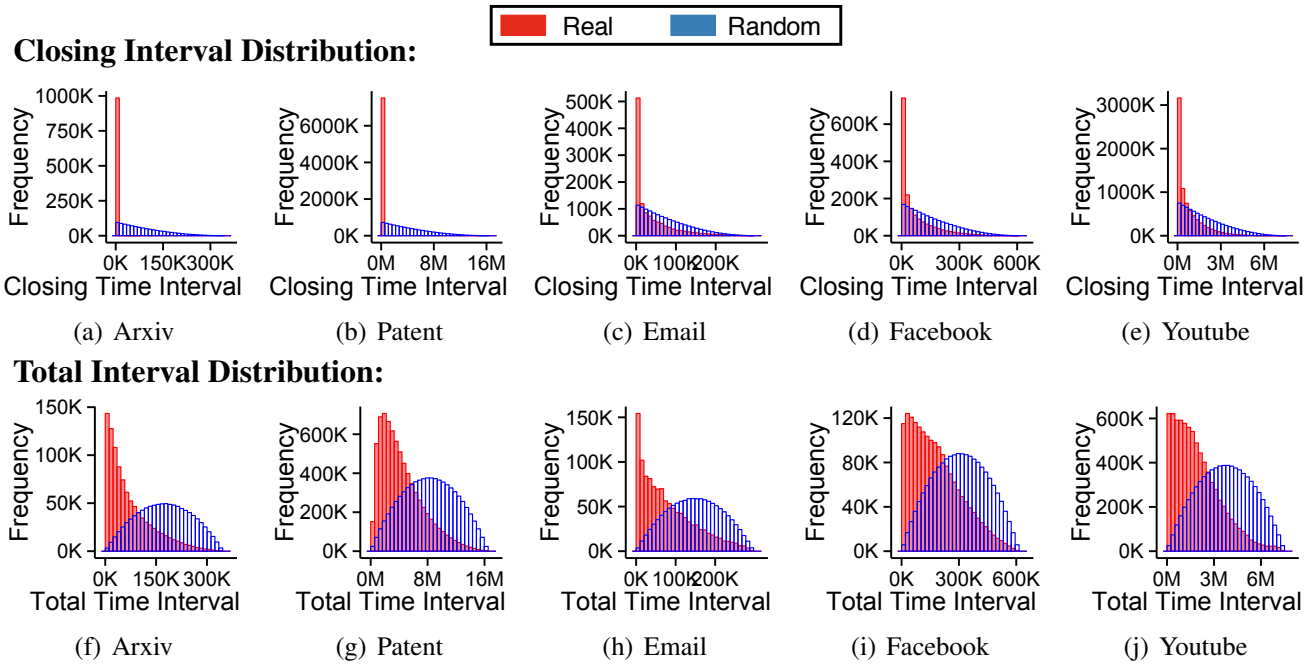
4.3 Observation: "Temporal Locality"

In this section, we discuss *temporal locality* (i.e., the tendency that future edges are more likely to form triangles with recent edges than with older edges) in real-world dynamic graphs modeled as real-order graph streams. To show the temporal locality, we investigate the distributions of *closing intervals* and *total intervals*, defined below. Figure 4.2 shows examples of closing and total intervals.

Definition 4.1: Closing Interval

The *closing interval* of a triangle is defined as the time interval between the arrivals of the second edge and the last edge. That is,

$$closing\_interval(\{u, v, w\}) := t^{(3)}_{uvw} - t^{(2)}_{uvw}.$$



(a) Arxiv (b) Patent (c) Email (d) Facebook (e) Youtube

(f) Arxiv (g) Patent (h) Email (i) Facebook (j) Youtube

Figure 4.3: **Temporal locality in triangle formation.** Closing and total intervals tend to be shorter in real-order graph streams than in random-order ones. This observation implies that future edges are more likely to form triangles with recent edges than with older edges.

Definition 4.2: Total Interval

The *total interval* of a triangle is defined as the time interval between the arrivals of the first edge and the last edge. That is,

$$total\_interval(\{u, v, w\}) := t^{(3)}_{uvw} - t^{(1)}_{uvw}.$$

Figure 4.3 shows the distributions of the closing and total intervals in real-order graph streams and those in the graph streams obtained by randomly shuffling the orders of the edges in the corresponding real-order streams. In every dataset, both intervals tend to be much shorter in the real-order stream than in the random-order one. That is, future edges do not form triangles with all previous edges with equal probability but they are more likely to form triangles with recent edges than with older edges.

Then, why does the temporal locality exist? It is related to *transitivity* [WF94], i.e., the tendency that people with common friends become friends. When an edge $\{u, v\}$ arrives, we can expect that edges connecting u and other neighbors of v or connecting v and other neighbors of u will arrive soon. Such future edges form triangles with the edge $\{u, v\}$. The temporal locality is also related to new nodes. For example, in citation networks, when a new node arrives (i.e., a paper is published), many edges incident to the node (i.e., citations of the paper), which are likely to form triangles with each other, are created almost instantly. Likewise, in social media, new users tend to make many connections within a short time by importing their friends from other social media or their address books.

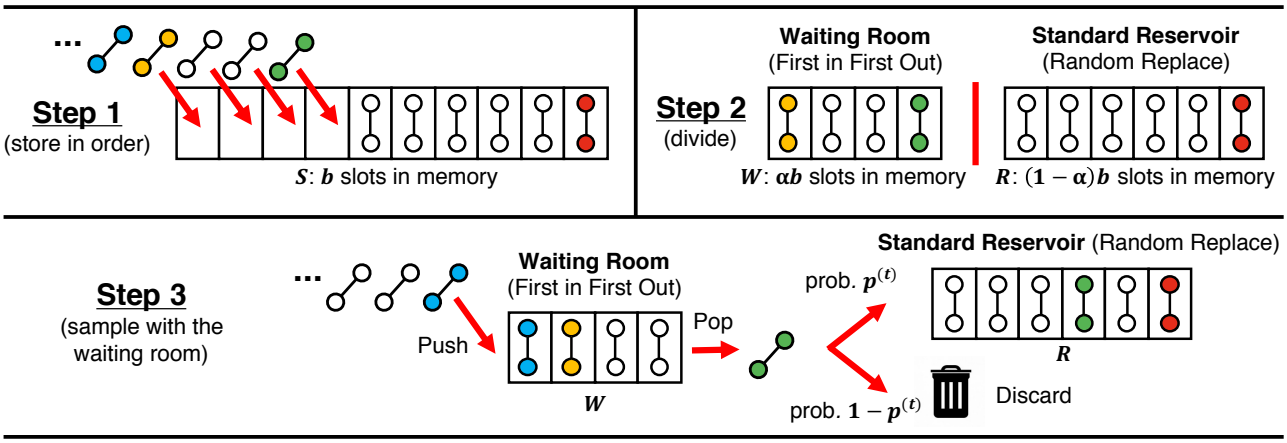


Figure 4.4: **Illustration of the sampling process in WRS.** Once the given storage space is full (by Step 1), the storage space is divided into the waiting room and the reservoir (by Step 2). In Step 3, the latest αb edges are stored in the waiting room, while the remaining older edges are uniformly sampled in the reservoir.

4.4 Proposed Algorithm: WRS

In this section, we propose WRS (**W**aiting-**R**oom **S**ampling), a single-pass streaming algorithm that exploits the temporal locality, presented in the previous section, for accurate global and local triangle counting. We first discuss the intuition behind WRS and then describe the details of WRS.

4.4.1 Overview

To minimize the estimation error, WRS minimizes both the bias and variance of its estimates. Reducing the variance is related to finding more triangles because, intuitively speaking, knowing more triangles is helpful to accurately estimate their count, as formally analyzed in Section 4.5.1. Thus, the following two goals should be considered when we decide which edges to store in the storage:

- **Goal 1.** Unbiased estimates of triangle counts should be computed from the stored edges.
- **Goal 2.** When each new edge arrives, it should form many triangles with the stored edges.

Uniform random sampling, such as reservoir sampling, achieves Goal 1 but fails to achieve Goal 2 ignoring the temporal locality, explained in Section 4.3. Storing the latest edges, while discarding the older ones, can be helpful to achieve Goal 2, as suggested by the temporal locality. However, simply discarding old edges makes unbiased estimation non-trivial.

To achieve both goals, WRS combines two policies. It divides the storage space into the *waiting room* and the *reservoir*. The most recent edges are always stored in the waiting room, while the remaining edges are uniformly sampled in the reservoir using standard reservoir sampling (see Section 3.3.1). The waiting room makes achieving Goal 2 possible since it exploits the temporal locality by storing the latest edges, which future edges are more likely to form triangles with. On the other hand, the reservoir makes achieving Goal 1 possible, as explained in detail in the following sections.

4.4.2 Detailed Description

We first describe the sampling policy of WRS. Then, we explain how to estimate the triangle counts from sampled edges. The pseudo code of WRS is given in Algorithm 4.1.

Algorithm 4.1 Waiting Room Sampling (WRS)

Input: (1) graph stream: $(e^{(1)}, e^{(2)}, ...)$,
(2) storage budget: b,
(3) relative size of the waiting room: α
Output: (1) estimated global triangle count: $\bar{c}$,
(2) estimated local triangle counts: $c[u]$ for each node u

1: **for** each new edge $e^{(t)} = \{u, v\}$ **do**
2: **for** each node w in $\hat{\mathcal{N}}_u \cap \hat{\mathcal{N}}_v$ **do**
3: initialize $\bar{c}$, $c[u]$, $c[v]$, and $c[w]$ to 0 if they have not been set
4: increase $\bar{c}$, $c[u]$, $c[v]$, and $c[w]$ by $1/p_{uvw}$
5: **if** $t \leq k$ **then** add $e^{(t)}$ to $\mathcal{S}$ ▷ (Case 1)
6: **else**
7: **if** $t = b + 1$ **then** ▷ (Case 2)
8: divide $\mathcal{S}$ into $\mathcal{W}$ and $\mathcal{R}$ as explained in Section 4.4.2.1
9: remove $e^{(t-b\alpha)}$ from $\mathcal{W}$ and add $e^{(t)}$ to $\mathcal{W}$ ▷ (Case 3)
10: **if** a random number in Bernoulli($p^{(t)}$) is 1 **then**
11: replace an edge chosen at random uniformly in $\mathcal{R}$ with $e^{(t-b\alpha)}$

4.4.2.1 Sampling Policy (Lines 5-11 of Algorithm 4.1)

Let $\mathcal{S}$ be the given storage space, where at most b edges are stored. Let $e^{(t)} = \{u, v\}$ be the edge arriving at time $t \in \{1, 2, ...\}$. The sampling method in WRS depends on t as follows (see 4.4 for a pictorial description):

- **(Case 1).** If $t \leq k$, then WRS adds $e^{(t)}$ to $\mathcal{S}$, which is not full yet.
- **(Case 2).** If $t = b+1$, then since $\mathcal{S}$ is full, WRS divides $\mathcal{S}$ into the waiting room $\mathcal{W}$ and the reservoir $\mathcal{R}$ so that the latest $b\alpha$ edges (i.e., $\{e^{(k-b\alpha+1)}, ..., e^{(k)}\}$) are in $\mathcal{W}$ and the remaining $b(1-\alpha)$ edges (i.e., $\{e^{(1)}, ..., e^{(b(1-\alpha))}\}$) are in $\mathcal{R}$. The constant α is the relative size of the waiting room. For simplicity, we assume $b\alpha$ and $b(1-\alpha)$ are integers. Then, WRS goes to (Case 3).
- **(Case 3).** If $t \geq b + 1$, then WRS replaces $e^{(t-b\alpha)}$, which is the oldest edge in $\mathcal{W}$, with $e^{(t)}$ (i.e., $\mathcal{W}$ is a queue with the "first in first out" mechanism). Then, with probability $p^{(t)}$, where

$$p^{(t)} := b(1-\alpha)/(t-b\alpha), \tag{4.1}$$

  WRS replaces an edge chosen at random uniformly in $\mathcal{R}$ with $e^{(t-b\alpha)}$. Otherwise, WRS discards $e^{(t-b\alpha)}$. That is, standard reservoir sampling (see Section 3.3.1) is used in $\mathcal{R}$, which ensures that each edge in $\{e^{(1)}, ..., e^{(t-b\alpha)}\}$ is stored in $\mathcal{R}$ with equal probability $p^{(t)}$.

In summary, when $e^{(t)}$ arrives (or after $e^{(t-1)}$ is processed), if $t \leq b+1$, then each edge in $\{e^{(1)}, ..., e^{(t-1)}\}$ is stored in $\mathcal{S}$ with probability 1. If $t > b+1$, then each edge in $\{e^{(t-b\alpha)}, ..., e^{(t-1)}\}$ is stored in $\mathcal{S}$ (specifically in $\mathcal{W}$) with probability 1, while each edge in $\{e^{(1)}, ..., e^{(t-b\alpha-1)}\}$ is stored in $\mathcal{S}$ (specifically in $\mathcal{R}$) with probability $p^{(t-1)}$.

4.4.2.2 Estimating Triangle Counts (Lines 2-4 of Algorithm 4.1)

Let $\hat{\mathcal{G}} = (\hat{\mathcal{V}}, \hat{\mathcal{E}})$ be the sampled graph composed of the edges in $\mathcal{S}$ ($\mathcal{W}$ or $\mathcal{R}$ if $\mathcal{S}$ is divided), and let $\hat{\mathcal{N}}_u$ be the set of neighbors of each node $u \in \hat{\mathcal{V}}$ in $\hat{\mathcal{G}}$. We use $\bar{c}$ and $c[u]$ to denote the estimates of the global

triangle count and the local triangle count of each node u, respectively, in the stream so far. That is, if we let $\bar{c}^{(t)}$ and $c^{(t)}[u]$ be $\bar{c}$ and $c[u]$ after processing $e^{(t)}$, they are the estimates of $|\mathcal{T}^{(t)}|$ and $|\mathcal{T}^{(t)}[u]|$, respectively.

When each edge $e^{(t)} = \{u, v\}$ arrives, WRS first finds the triangles composed of $\{u, v\}$ and two edges in $\hat{\mathcal{G}}$. The set of such triangles is $\{\{u, v, w\} : w \in \hat{\mathcal{N}}_u \cap \hat{\mathcal{N}}_v\}$. For each triangle $\{u, v, w\}$, WRS increases $\bar{c}$, $c[u]$, $c[v]$, and $c[w]$ by $1/p_{uvw}$, where p_{uvw} is the probability that WRS discovers $\{u, v, w\}$. Then, the expected increase of the counters by each triangle $\{u, v, w\}$ becomes 1, which makes the counters unbiased estimates, as formalized in Theorem 4.1 in the following section.

The only remaining task is to compute p_{uvw}, the probability that WRS discovers triangle $\{u, v, w\}$. To this end, we divide the types of triangles depending on the arrival times of their edges, as in Definition 4.3. Recall that $t^{(i)}_{uvw}$ indicates the arrival time of the edge arriving i-th among the edges forming $\{u, v, w\}$.

Definition 4.3: Types of Triangles

Given the maximum number of samples b and the relative size of the waiting room α, the *type* of each triangle $\{u, v, w\}$, denoted by $type_{uvw}$, is defined as:

$$type_{uvw} := \begin{cases} 1 & \text{if } t^{(3)}_{uvw} \leq b + 1 \\ 2 & \text{else if } t^{(3)}_{uvw} - t^{(1)}_{uvw} \leq b\alpha \\ 3 & \text{else if } t^{(3)}_{uvw} - t^{(2)}_{uvw} \leq b\alpha \\ 4 & \text{otherwise.} \end{cases}$$

That is, a triangle has Type 1 if all its edges arrive early, Type 2 if its total interval is short, Type 3 if its closing interval is short, and Type 4 otherwise. The probability that each triangle is discovered by WRS (i.e., considered in line 2 of Algorithm 4.1) depends on its type, as formalized in Lemma 4.1.

Lemma 4.1: Triangle Discovering Probability

Given the maximum number of samples b and the relative size of the waiting room α, the probability p_{uvw} that WRS discovers each triangle $\{u, v, w\}$ is

$$p_{uvw} = \begin{cases} 1 & \text{if } type_{uvw} \leq 2 \\ \frac{b(1-\alpha)}{t^{(3)}_{uvw}-1-b\alpha} & \text{if } type_{uvw} = 3 \\ \frac{b(1-\alpha)}{t^{(3)}_{uvw}-1-b\alpha} \times \frac{b(1-\alpha)-1}{t^{(3)}_{uvw}-2-b\alpha} & \text{if } type_{uvw} = 4 \end{cases} \tag{4.2}$$

Proof. Without loss of generality, we assume $t^{(1)}_{uvw} = t_{vw}$, $t^{(2)}_{uvw} = t_{wu}$, and $t^{(3)}_{uvw} = t_{uv}$. That is, $\{v, w\}$ arrives earlier than $\{w, u\}$, and $\{w, u\}$ arrives earlier than $\{u, v\}$.

If $type_{uvw} = 1$, $\{u, v\}$ arrives at time $b + 1$ or earlier. When $\{u, v\}$ arrives, $\{v, w\}$ and $\{w, u\}$ are always stored in $\mathcal{S}$. Thus, WRS discovers $\{u, v, w\}$ with probability 1.

If $type_{uvw} = 2$, we have $t_{uv} - t_{wu} < t_{uv} - t_{vw} \leq b\alpha$. When $\{u, v\}$ arrives, $\{v, w\}$ and $\{w, u\}$ are always stored in $\mathcal{W}$. Thus, WRS discovers $\{u, v, w\}$ with probability 1.

If $type_{uvw} = 3$, we have $t_{uv} - t_{wu} \leq b\alpha$ but $t_{uv} - t_{vw} > b\alpha$. When $\{u, v\}$ arrives, $\{w, u\}$ is always stored in $\mathcal{W}$, while $\{v, w\}$ cannot be in $\mathcal{W}$ but can be in $\mathcal{R}$ with probability $p^{(t_{uv}-1)}$ (see Eq. (4.1)). For WRS to discover $\{u, v, w\}$, $\{v, w\}$ should be in $\mathcal{R}$, thus the probability is $p^{(t_{uv}-1)} = b(1-\alpha)/(t_{uv} - 1 - b\alpha) = b(1-\alpha)/(t^{(3)}_{uvw} - 1 - b\alpha)$.

If $type_{uvw} = 4$, we have $t_{uv} - t_{vw} > t_{uv} - t_{wu} > b\alpha$. Thus, when $\{u, v\}$ arrives, $\{v, w\}$ and $\{w, u\}$ cannot be in $\mathcal{W}$. For WRS to discover $\{u, v, w\}$, both $\{v, w\}$ and $\{w, u\}$ should be in $\mathcal{R}$ when $\{u, v\}$ arrives. The probability of the event is

$$\begin{aligned}\mathbb{P}[\{v, w\} \in \mathcal{R} \text{ and } \{w, u\} \in \mathcal{R}] &= \mathbb{P}[\{v, w\} \in \mathcal{R}] \times \mathbb{P}[\{w, u\} \in \mathcal{R} | \{v, w\} \in \mathcal{R}] \\ &= \frac{b(1-\alpha)}{t_{uv} - 1 - b\alpha} \times \frac{b(1-\alpha) - 1}{t_{uv} - 2 - b\alpha},\end{aligned}$$

which is equal to the last case of Eq. (4.2). ■

Notice that no additional space is required to store the arrival times of sampled edges. This is because the type of each triangle $\{u, v, w\}$ and its discovering probability p_{uvw} can be computed from current time t and whether each edge is stored in $\mathcal{W}$ or $\mathcal{R}$ at time t, as explained in the proof of Lemma 4.1.

4.5 Theoretical Analysis

We theoretically analyze the accuracy, time complexity, and space complexity of WRS.

4.5.1 Accuracy Analysis

We analyze the bias and variance of the estimates provided by WRS. To this end, we define x_{uvw} as the increase in $\bar{c}$ by triangle $\{u, v, w\}$. By lines 2-4 of Algorithm 4.1, x_{uvw} is $1/p_{uvw}$ with its discovering probability p_{uvw}, and 0 with probability $1 - p_{uvw}$. Based on this concept, the unbiasedness of the estimates given by WRS is shown in Theorem 4.1.

Theorem 4.1: 'Any time' unbiasedness of WRS

If $b(1-\alpha) \geq 2$, WRS gives unbiased estimates of the global and local triangle counts at any time. That is, if we let $\bar{c}^{(t)}$ and $c^{(t)}[u]$ be $\bar{c}$ and $c[u]$ after processing $e^{(t)}$, respectively, the followings hold:

$$\mathbb{E}[\bar{c}^{(t)}] = |\mathcal{T}^{(t)}|, \ \forall t \in \{1, 2, ...\}, \tag{4.3}$$

$$\mathbb{E}[c^{(t)}[u]] = |\mathcal{T}^{(t)}[u]|, \ \forall u \in \mathcal{V}^{(t)}, \ \forall t \in \{1, 2, ...\}. \tag{4.4}$$

Proof. From Eq. (4.2), if $b(1-\alpha) \geq 2$, we have $\mathbb{E}[x_{uvw}] = 1/p_{uvw} \times p_{uvw} + 0 \times (1 - p_{uvw}) = 1$. Combining this and $\bar{c}^{(t)} = \sum_{\{u,v,w\} \in \mathcal{T}^{(t)}} x_{uvw}$ gives

$$\mathbb{E}[\bar{c}^{(t)}] = \mathbb{E}\left[\sum_{\{u,v,w\} \in \mathcal{T}^{(t)}} x_{uvw}\right] = \sum_{\{u,v,w\} \in \mathcal{T}^{(t)}} \mathbb{E}[x_{uvw}] = |\mathcal{T}^{(t)}|,$$

which proves Eq. (4.3) for every $t \in \{1, 2, ...\}$. Likewise, $\mathbb{E}[x_{uvw}] = 1$ and $c^{(t)}[u] = \sum_{\{u,v,w\} \in \mathcal{T}^{(t)}[u]} x_{uvw}$ imply

$$\mathbb{E}[c^{(t)}[u]] = \mathbb{E}\left[\sum_{\{u,v,w\} \in \mathcal{T}^{(t)}[u]} x_{uvw}\right] = \sum_{\{u,v,w\} \in \mathcal{T}^{(t)}[u]} \mathbb{E}[x_{uvw}] = |\mathcal{T}^{(t)}[u]|,$$

which proves Eq. (4.4) for every $u \in \mathcal{V}^{(t)}$ and $t \in \{1, 2, ...\}$. ■

In our variance analysis, to give a simple intuition, we focuses on

$$\tilde{\mathrm{Var}}[\bar{c}^{(t)}] = \sum\nolimits_{\{u,v,w\} \in \mathcal{T}^{(t)}} \mathrm{Var}[x_{uvw}], \tag{4.5}$$

$$\tilde{\mathrm{Var}}[c^{(t)}[u]] = \sum\nolimits_{\{u,v,w\} \in \mathcal{T}^{(t)}[u]} \mathrm{Var}[x_{uvw}], \tag{4.6}$$

which are the variances when the dependencies in $\{x_{uvw}\}_{\{u,v,w\} \in \mathcal{T}^{(t)}}$ are ignored. Specifically, we show how the temporal locality, described in Section 4.3, is related to reducing $\tilde{\mathrm{Var}}[\bar{c}^{(t)}]$ and $\tilde{\mathrm{Var}}[c^{(t)}[u]]$.

From $\mathrm{Var}[x_{uvw}] = \mathbb{E}[x_{uvw}^2] - (\mathbb{E}[x_{uvw}])^2$, we have $\mathrm{Var}[x_{uvw}] = (1/p_{uvw}) - 1$. From Eq. (4.2), if $b(1-\alpha) \geq 2$,

$$\mathrm{Var}[x_{uvw}] = \begin{cases} 0 & \text{if } type_{uvw} \leq 2 \\ \frac{t^{(3)}_{uvw} - 1 - b\alpha}{b(1-\alpha)} - 1 & \text{if } type_{uvw} = 3 \\ \frac{t^{(3)}_{uvw} - 1 - b\alpha}{b(1-\alpha)} \times \frac{t^{(3)}_{uvw} - 2 - b\alpha}{b(1-\alpha)-1} - 1 & \text{if } type_{uvw} = 4. \end{cases} \tag{4.7}$$

Compared to TRIEST$_{\text{IMPR}}$ [SERU17], where $\mathrm{Var}[x_{uvw}] = 0$ if $type_{uvw} = 1$ and $\mathrm{Var}[x_{uvw}] = \frac{t^{(3)}_{uvw}-1}{b} \times \frac{t^{(3)}_{uvw}-2}{b-1} - 1$ otherwise, WRS reduces the variance regarding the triangles of Type 2 or 3, as formalized in Lemma 4.2, while WRS increases the variance regarding the triangles of Type 4.

Lemma 4.2: Comparison of Variances

For each triangle $\{u, v, w\}$, $\mathrm{Var}[x_{uvw}]$ is smaller in WRS than in TRIEST$_{\text{IMPR}}$ [SERU17], i.e.,

$$\mathrm{Var}[x_{uvw}] < \frac{t^{(3)}_{uvw} - 1}{b} \times \frac{t^{(3)}_{uvw} - 2}{b-1} - 1 \tag{4.8}$$

if **any** of the following conditions are satisfied:

- $type_{uvw} = 2$
- $type_{uvw} = 3$ and $t^{(3)}_{uvw} > 1 + \frac{\alpha}{1-\alpha} b$
- $type_{uvw} = 3$ and $\alpha < 0.5$.

Proof. From the definition of $type_{uvw}$, $type_{uvw} \geq 2$ implies

$$t^{(3)}_{uvw} > b + 1. \tag{4.9}$$

First, we show the case when $type_{uvw} = 2$. Eq. (4.7) and Eq. (4.9) give

$$\frac{t^{(3)}_{uvw} - 1}{b} \times \frac{t^{(3)}_{uvw} - 2}{b-1} - 1 > 0 = \mathrm{Var}[x_{uvw}].$$

Second, we show the case when $type_{uvw} = 3$ and $t^{(3)}_{uvw} > 1 + \frac{\alpha}{1-\alpha}b$. From $t^{(3)}_{uvw} > 1 + \frac{\alpha}{1-\alpha}b$, $\left(1 + \frac{b}{t^{(3)}_{uvw}-1}\right)\alpha < 1$ holds. This and Eq. (4.9) imply $\left(1 + \frac{b}{t^{(3)}_{uvw}-1}\right)\left(1 - \frac{b-1}{t^{(3)}_{uvw}-2}\right)\alpha < 1 - \frac{b-1}{t^{(3)}_{uvw}-2}$. Again, this and Eq. (4.9) give $\left(1 - \frac{b(b-1)}{(t^{(3)}_{uvw}-1)(t^{(3)}_{uvw}-2)}\right)\alpha < \left(1 + \frac{b}{t^{(3)}_{uvw}-1} - \frac{b-1}{t^{(3)}_{uvw}-2} - \frac{b(b-1)}{(t^{(3)}_{uvw}-1)(t^{(3)}_{uvw}-2)}\right)\alpha < 1 - \frac{b-1}{t^{(3)}_{uvw}-2}$. This is equivalent to $(b-1)(t^{(3)}_{uvw} - 1 - b\alpha) < (t^{(3)}_{uvw} - 1)(t^{(3)}_{uvw} - 2)(1-\alpha)$, which is again equivalent to $\frac{t^{(3)}_{uvw}-1-b\alpha}{b(1-\alpha)} - 1 < \frac{t^{(3)}_{uvw}-1}{b} \times \frac{t^{(3)}_{uvw}-2}{b-1} - 1$. Combining this and Eq. (4.7) gives

$$\mathrm{Var}[x_{uvw}] = \frac{t^{(3)}_{uvw} - 1 - b\alpha}{b(1-\alpha)} - 1 < \frac{t^{(3)}_{uvw} - 1}{b} \times \frac{t^{(3)}_{uvw} - 2}{b-1} - 1.$$

Finally, the same conclusion holds when $type_{uvw} = 3$ and $\alpha < 0.5$. Eq. (4.9) and $\alpha < 0.5$ imply $t^{(3)}_{uvw} > 1 + \frac{\alpha}{1-\alpha}b$. This and $type_{uvw} = 3$ give the second case, which is proven above.

Thus, Eq. (4.8) holds under any of the given conditions. ■

Therefore, the superiority of WRS in terms of small $\tilde{\mathrm{Var}}[\bar{c}^{(t)}]$ and $\tilde{\mathrm{Var}}[c^{(t)}[u]]$ depends on the distribution of the types of triangles in the input graph stream. In the experiment section, we show that the triangles of Type 2 or 3 are abundant enough in real-world dynamic graphs, as suggested by the temporal locality, so that WRS is more accurate than TRIEST$_{\text{IMPR}}$.

4.5.2 Complexity Analysis

We prove the time and space complexity of WRS. Especially, we show that WRS has the same time and space complexity as the state-of-the-art algorithms [LJK18, SERU17], leading to the conclusion that WRS yields higher accuracy than these methods without increasing time or space complexity. We assume that sampled edges are stored in the adjacency list format in memory, as in our implementation used for experiments. However, storing them sequentially, as in Figure 4.4, does not change the results below.

The worst-case time complexity of WRS is linear in the memory budget and the number of edges in the input stream, as formalized in Theorem 4.2.

Theorem 4.2: Worst-Case Time Complexity of WRS

Processing an incoming edge in Algorithm 4.1 takes $O(b)$, and thus processing t edges in the input stream takes $O(bt)$.

Proof. The most expensive step of processing each incoming edge $\{u, v\}$ in Algorithm 4.1 is to find their common neighbors $\hat{\mathcal{N}}_u \cap \hat{\mathcal{N}}_v$ in line 2. Computing $\hat{\mathcal{N}}_u \cap \hat{\mathcal{N}}_v$ requires accessing $|\hat{\mathcal{N}}_u| + |\hat{\mathcal{N}}_v| = O(b)$ edges. ■

However, this analysis of the worst-case graph stream is too pessimistic for real-world graphs, since $|\hat{\mathcal{N}}_u| + |\hat{\mathcal{N}}_v|$ is usually much smaller than k in real-world graphs.

Theorem 4.3 gives the space complexity of WRS. Note that, except the space for outputs (specifically, local triangle counts) WRS only requires $O(b)$ additional space.

Theorem 4.3: Space Complexity of WRS

Let $\mathcal{V}^{(t)}$ be the set of nodes in the graph consisting of the first t edges in the input stream. Processing t edges in the input stream by Algorithm 4.1 requires $O(b)$ space in case of global triangle counting and $O(b + |\mathcal{V}^{(t)}|)$ space in case of local triangle counting.

Proof. Algorithm 4.1 uses $O(b)$ space for sampling edges and $O(|\mathcal{V}^{(t)}|)$ space for maintaining local triangle counts, which need not be maintained in case of global triangle counting. ■

4.6 Experiments

We review our experiments to answer the following questions:

- **Q1. Illustration of Theorems**: Does WRS give unbiased estimates with smaller variances than its best competitors?
- **Q2. Accuracy**: How accurately does WRS estimate global and local triangle counts?
- **Q3. Scalability**: How does WRS scale with the number of edges in the input stream?
- **Q4. Effects of Parameters on Accuracy**: How does the relative size α of the waiting room affect the accuracy of WRS? What is the optimal value of α?

4.6.1 Experimental Settings

Machine: We ran all experiments on a PC with a 3.60GHz Intel i7-4790 CPU and 32GB memory.

Datasets: Table 4.2 lists the graph streams used in our experiments, including the following real-order graph streams:

- Arxiv [GGK03]: A citation network between papers in the ArXiv's High Energy Physics. Each edge $\{u, v\}$ represents that paper u cited paper v. We used the submission time of u as the creation time of $\{u, v\}$.
- Facebook [VMCG09]: A friendship network between users of Facebook. Each edge $\{u, v\}$ represents that user v appeared in the friend list of user u. Edges whose creation time is unknown were ignored.
- Email [KY04]: An email network from Enron Corporation. Each edge $\{u, v\}$ represents that employee u sent to or received from person v (who may not be an employee) at least one email. We used the creation time of the first email between u and v as the creation time of $\{u, v\}$.
- Youtube [Mis09]: A friend network between users of Youtube. Each edge $\{u, v\}$ represents that user u and user v became friends. Edges created before 12/10/2006 were ignored since their exact creation times are unknown.
- Patent [HJT01]: A citation network between patents. Each edge $\{u, v\}$ indicates that patent u cited patent v. We used the time when u was granted as the creation time of $\{u, v\}$.

All self loops, duplicated edges, and directions of the edges were ignored in all the above graph streams. In them, edges were streamed in the order by which they are created.

Table 4.2: **Summary of the graph streams used in our experiments.** B: billion, M: million, K: thousand.

Name	# Nodes	# Edges	Summary
Arxiv [GGK03]	30.5K	347K	Citation network
Facebook [VMCG09]	61.1K	615K	Friendship network
Email [KY04]	87.0K	297K	Email network
Youtube [Mis09]	3.18M	7.51M	Friendship network
Patent [HJT01]	3.77M	16.5M	Citation network
Random (800GB)	1M	0.1B - 100B	Synthetic graph

Implementations: We compared WRS to TRIEST$_{\text{IMPR}}$ [SERU17] and MASCOT [LJK18], which are single-pass streaming algorithms estimating both global and local triangle counts within a limited storage budget. We implemented all the methods in Java, and in all of them, we stored sampled edges in the adjacency list format in main memory. In WRS, the relative size α of the waiting room was set to 0.1 unless otherwise stated (see Section 4.6.5 for the effect of α on the accuracy).

Evaluation measures: We measured the accuracies of the considered algorithms using *global error*, *local error*, and *rank correlation*, all of which are defined in Section 3.3.2.

4.6.2 Q1. Illustration of Theorems

We ran experiments illustrating our analyses in Section 4.5. Figure 4.1(d) shows the distributions of $10,000$ estimates of the global triangle count in the ArXiv dataset obtained by each method. We set b to the 10% of the number of edges in the dataset. The average of the estimates of WRS was close to the true triangle count. Moreover, the estimates of WRS had smaller variances than those of the competitors. These results are consistent with Theorem 4.1 and Lemma 4.2.

4.6.3 Q2. Accuracy

Figure 4.5 shows the accuracies of the considered methods in the real-world graph streams with different storage budgets b. Each evaluation metric was computed $1,000$ times for each method, and the average was reported with an error bar indicating the estimated standard error. In all the datasets, WRS was most accurate in global and local triangle counting, regardless of storage budgets. The accuracy gain was especially high in the Arxiv and Patent datasets, which showed the strongest temporal locality. In the Arxiv dataset, for example, WRS gave up to *47% smaller local error* and *40% smaller global error* than the second best method.

WRS was more accurate since its estimates were based on more triangles. Due to its effective sampling scheme, WRS discovered up to $2.9\times$ more triangles than its competitors while processing the same streams, as shown in Figure 4.6(a).

4.6.4 Q3. Scalability

We measured how the running time of WRS scales with the number of edges in the input stream. To measure the scalability independently of the speed of the input stream, we measured the time taken by WRS to process all the edges ignoring the time taken to wait for the arrivals of edges. Figure 4.1(c)

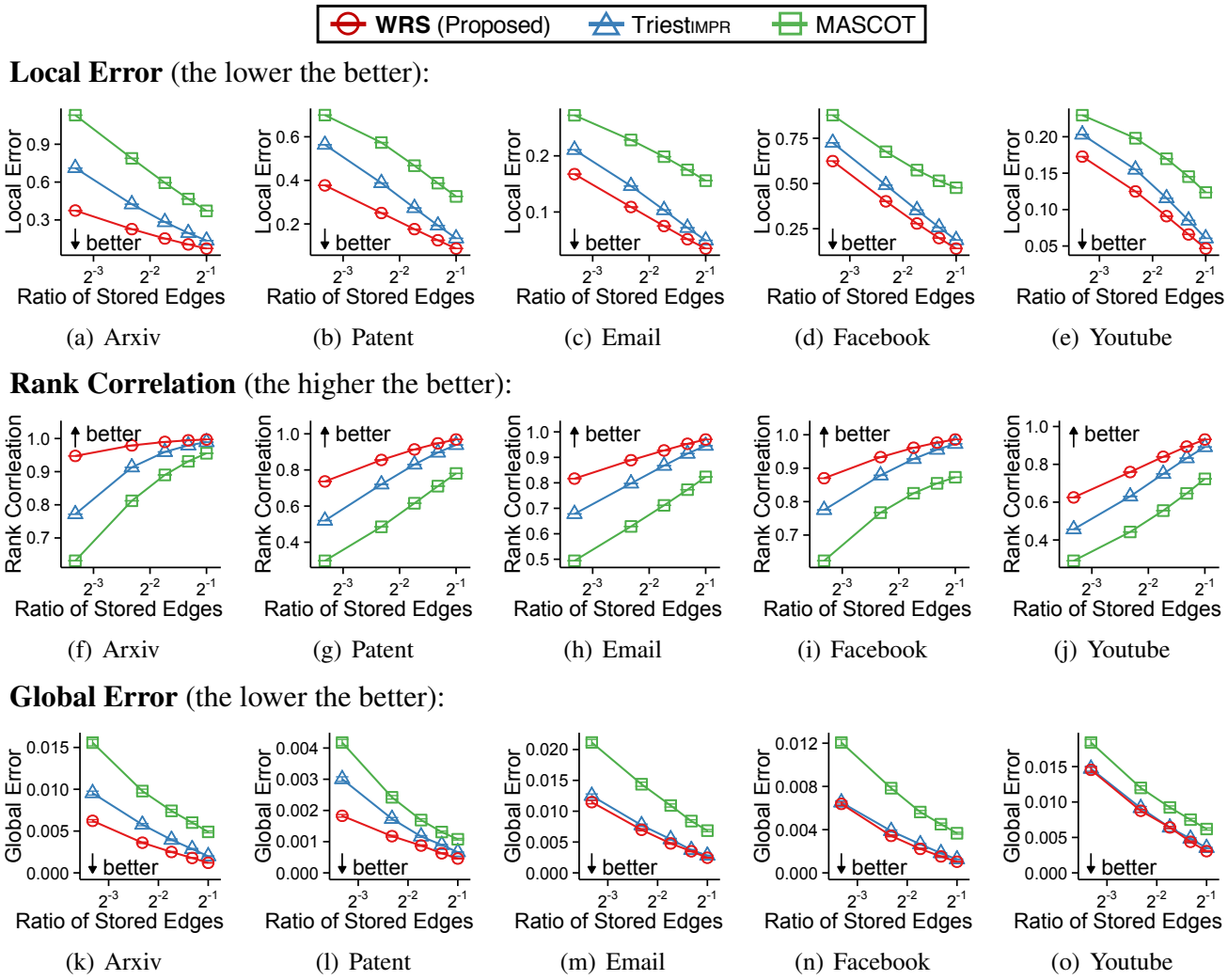
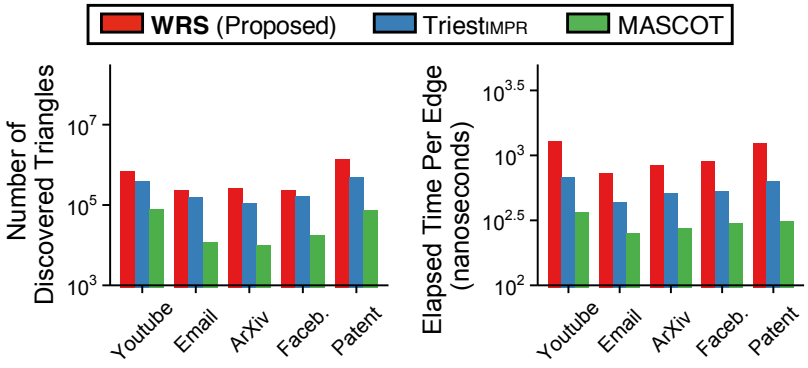


Figure 4.5: **WRS is accurate.** In all the datasets, WRS is most accurate in global and local triangle counting regardless of storage budget b. The relative size of waiting room (i.e., α) is fixed to 0.1.

shows the results in the Random datasets, which were created by the Erdös-Rényi model. The running time of WRS scaled linearly with the number of edges. That is, the time taken by WRS to process each edge was almost constant regardless of the number of edges arriving so far. Despite its linear scalability, WRS was slower than its competitors since it discovered and processed more triangles, as shown in Figure 4.6. However, it took only about a microsecond for WRS to process an edge, which means that WRS can handle dynamic graphs where about one million new edges are created per second.

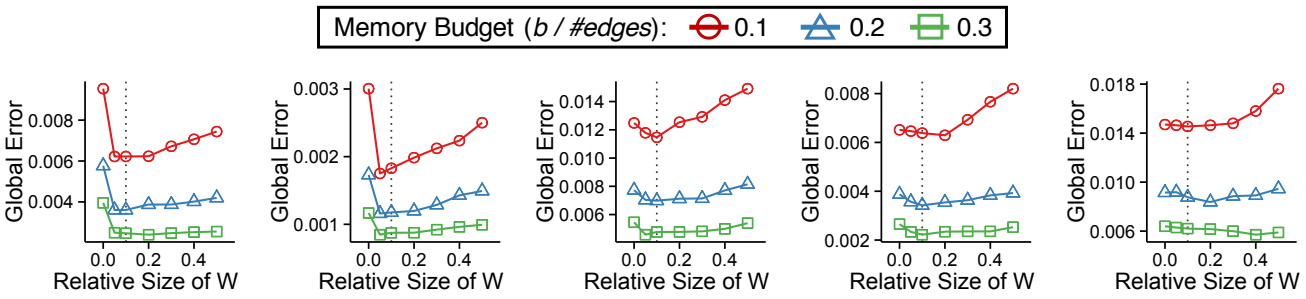
4.6.5 Q4. Effects of Parameters on Accuracy

We measured how the accuracy of WRS changes depending on α, the relative size of the waiting room. Figure 4.7 shows the results with different storage budgets. Here, we used global error as the accuracy metric, and the average values over $1,000$ runs are reported. In all the datasets and regardless of storage budgets, using proper amount of memory space for the waiting room gave better accuracy than using no space for the waiting room ($\alpha = 0$) and using half the space for the waiting room ($\alpha = 0.5$). Although proper α values depended on datasets and storage budgets, the accuracy was maximized when α was about 0.1 in most of the cases.



(a) Number of Discovered Triangles (b) Running Time

Figure 4.6: **The sampling scheme of WRS is effective.** b is 10% of the number of the edges in each dataset and α is 0.1. (a) WRS discovers up to $2.9\times$ more triangles than the second best method, in the same streams. (b) The running times and the numbers of discovered triangles show similar trends.



(a) Arxiv (b) Patent (c) Email (d) Facebook (e) Youtube

Figure 4.7: **Effects of α on the accuracy of WRS.** Using about 10% of memory space for the waiting room ($\alpha = 0.1$) gives higher accuracy than using no space for the waiting room ($\alpha = 0$) or using half the space for the waiting room ($\alpha = 0.5$).

4.7 Summary

In this chapter, we propose WRS, a single-pass streaming algorithm for global and local triangle counting. WRS divides available memory space into the waiting room, where the latest edges are stored, and the reservoir, where the remaining edges are uniformly sampled. By doing so, WRS exploits the temporal locality in real-world dynamic graphs while giving unbiased estimates. We show that WRS has the following advantages:

- **Fast and 'any time'**: WRS scales linearly with the number of edges in the input graph stream, and it gives estimates at any time while the input stream grows (Figure 4.1(a)).
- **Accurate**: Estimation error in WRS is up to 47% *smaller* than those in its best competitors (Figures 4.1(b) and 4.5).
- **Theoretically sound**: WRS gives unbiased estimates with small variances under the temporal locality (Theorem 4.1, Lemma 4.2 and Figure 4.1(c)).

Reproducibility: The source code and datasets used in this chapter are available at `http://www.cs.cmu.edu/~kijungs/codes/wrs/`.

Chapter based on work that appeared at PAKDD 2018 [SHL+18] and its extension [SLO+19].

Chapter 5

Counting Triangles in Graph Streams (2): Utilizing Multiple Machines

Given a graph stream, how can we estimate the number of triangles in it using multiple machines with limited storage? Specifically, how should edges be processed and sampled across the machines for rapid and accurate estimation?

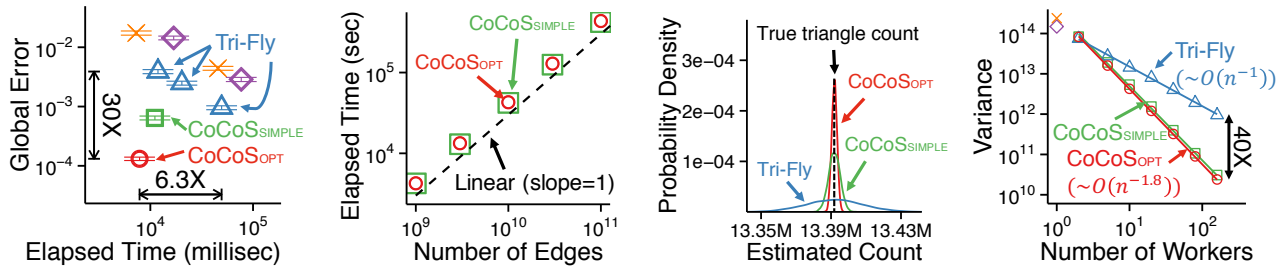
As discussed in the previous chapters, the count of triangles (i.e., cliques of size three) has proven useful in numerous applications, including anomaly detection, community detection, and link recommendation. For triangle counting in large and dynamic graphs, recent work has focused largely on streaming algorithms and distributed algorithms but little on their combinations for "the best of both worlds".

In this chapter, we propose CoCoS, a fast and accurate distributed streaming algorithm for estimating the counts of global triangles (i.e., all triangles) and local triangles incident to each node. Making one pass over the input stream, CoCoS carefully processes and stores the edges across multiple machines so that the redundant use of computational and storage resources is minimized. Compared to baselines, CoCoS is *(a) Accurate:* giving up to $\mathbf{39}\times$ **smaller estimation error**, *(b) Fast*: up to $\mathbf{10.4}\times$ **faster**, scaling linearly with the size of the input stream, and *(c) Theoretically sound*: yielding unbiased estimates with variances dropping faster as the number of machines is scaled up.

5.1 Motivation

Given a graph stream, how can we utilize multiple machines for rapidly and accurately estimating the count of triangles in it? How should we process and sample the edges across the machines to minimize the redundant use of computational and storage resources?

As discussed in Chapter 3, for triangle counting in real-world graphs, many of which are large and evolving with new edges, recent work has focused largely on streaming algorithms [KP13, LJK18, SERU17, PTT13, ADNK14, ADWR17, PTTW13, Shi17, SHL+18, KP17, PT12]. Given a graph stream, which is a sequence of edges that may not fit in the underlying storage, these algorithms estimate the count of triangles while making a single pass over the stream. Especially, these algorithms maintain and gradually update their estimates as each edge is received rather than operating on the entire graph. Thus, they are appropriate for dynamic graphs, whose edges are received over time.



(a) Fast and accurate (b) Scalable (c) Unbaised with small variance (Theorems 5.2 and 5.4)

Figure 5.1: **Strengths of CoCoS.** (a) *Fast and accurate:* CoCoS is faster and more accurate than its best competitors. (b) *Scalable:* The running time of CoCoS is linear in the number of edges in the input stream. (c-d) *Unbiased with small variance:* CoCoS gives unbiased estimates with variances dropping rapidly as we use more machines (Theorems 5.2 and 5.4). See Section 5.5 for details.

Another popular approach is to extend triangle counting algorithms to distributed settings, including distributed-memory [AKM13] and MapReduce [Coh09, SV11, PC13, PSKP14, PMK16, PSP+18] settings. These distributed algorithms utilize computational and storage resources of multiple machines for speed and scalability. However, unlike streaming algorithms, they require all edges to be given at once. Thus, they are not applicable to dynamic graphs, whose edges are received over time, or graphs that are too large to fit in the underlying storage.

Can we have the best of both worlds? In other words, can we utilize multiple machines for rapid and accurate triangle counting in a graph stream? A promising approach is Tri-Fly (see Section 5.3.2), where edges are broadcast to every machine that independently runs a state-of-the-art streaming algorithm called $\text{TRIEST}_{\text{IMPR}}$ [SERU17]. The final estimates are the averages of the estimates provided by all the machines. Although Tri-Fly successfully reduces estimation error inversely proportional to the number of machines, Tri-Fly incurs a redundant use of computational and storage resources.

In this third chapter on triangle counting, we propose CoCoS (**Co**nditional **Co**unting and **S**ampling), a fast and accurate distributed streaming algorithm that estimates the counts of global and local triangles. CoCoS gives the advantages of both streaming and distributed algorithms, significantly outperforming Tri-Fly, as shown in Figure 5.1. CoCoS minimizes the redundant use of computational and storage resources by carefully processing and sampling edges across distributed machines so that each edge is stored in at most two machines and each triangle is counted by at most one machine. We theoretically and empirically demonstrate that CoCoS has the following advantages:

- **Accurate**: CoCoS yields up to $30\times$ and $39\times$ *smaller estimation errors* for global and local triangle counts, respectively, than baselines with similar speeds (Figure 5.1(a)).
- **Fast**: CoCoS scales linearly with the number of edges in the input stream (Figure 5.1(b)), and it is up to $10.4\times$ *faster* than baselines while giving more accurate estimates (Figure 5.1(a)).
- **Theoretically Sound:** CoCoS gives unbiased estimates with variances dropping rapidly as the number of machines is scaled up (Theorems 5.2 and 5.4; and Figure 5.1(c)).

Reproducibility: The source code and datasets used in this chapter are available at `http://www.cs.cmu.edu/~kijungs/codes/trifly/`.

The rest of this chapter is organized as follows. In Section 5.2, we we introduce some preliminary concepts, notations, and a formal problem definition. In Section 5.3, we present our proposed algorithm, namely CoCoS, and a baseline algorithm, namely Tri-Fly. In Section 5.4, we theoretically

Table 5.1: **Table of frequently-used symbols.**

Symbol	Definition
Notations for Graph Streams (Section 5.2)	
$(e^{(1)}, e^{(2)}, ...)$	input graph stream
$e^{(t)}$	edge that arrives at time $t \in \{1, 2, ...\}$
$\{u, v\}$	edge between nodes u and v
t_{uv}	arrival time of edge $\{u, v\}$
$\{u, v, w\}$	triangle composed of nodes u, v, and w
$\mathcal{G}^{(t)} = (\mathcal{V}^{(t)}, \mathcal{E}^{(t)})$	graph at time t
$\mathcal{T}^{(t)}$	set of global triangles in $\mathcal{G}^{(t)}$
$\mathcal{T}^{(t)}[u]$	set of local triangles associated with node u in $\mathcal{G}^{(t)}$
Notations for Algorithms (Section 5.3)	
k	number of workers
b	maximum number of edges stored in each worker
$\bar{c}$	estimate of the global triangle count
$c[u]$	estimate of the local triangle count of node u
$f : \mathcal{V} \to \{1, ..., k\}$	function assigning nodes to workers
l_i	load of the i-th worker
θ	tolerance for load difference
Notations for Analysis (Section 5.4)	
$p^{(t)}$	number of Type 1 triangle pairs in $\mathcal{G}^{(t)}$
$q^{(t)}$	number of Type 2 triangle pairs in $\mathcal{G}^{(t)}$

analyze the accuracy and complexity of them. After sharing some experimental results in Section 5.5, we provide a summary of this chapter in Section 5.6.

5.2 Preliminaries and Problem Definition

In this section, we first introduce some notations and concepts used throughout this chapter. Then, we define the problem of distributed global and local triangle counting in a graph stream.

5.2.1 Notations and Concepts

We list the frequently-used symbols in Table 5.1. Consider a graph stream $(e^{(1)}, e^{(2)}, ...)$, where $e^{(t)}$ denotes the undirected edge that arrives at time $t \in \{1, 2, ...\}$. Then, let $\mathcal{G}^{(t)} = (\mathcal{V}^{(t)}, \mathcal{E}^{(t)})$ be the graph composed of the nodes and edges arriving at time t or earlier. We use the unordered pair $\{u, v\} \in \mathcal{E}^{(t)}$ to indicate the edge between two distinct nodes $u, v \in \mathcal{V}^{(t)}$. We denote the arrival time of each edge $\{u, v\}$ by t_{uv}. We use the unordered triple $\{u, v, w\}$ to indicate the triangle (i.e., three nodes every pair of which is connected by an edge) composed of three distinct nodes $u, v, w \in \mathcal{V}^{(t)}$. We let $\mathcal{T}^{(t)}$ be the set of *global triangles* in $\mathcal{G}^{(t)}$ (i.e., all triangles in $\mathcal{G}^{(t)}$), and for each node $u \in \mathcal{V}^{(t)}$, let $\mathcal{T}^{(t)}[u] \subset \mathcal{T}^{(t)}$ be the set of *local triangles of* u in $\mathcal{G}^{(t)}$ (i.e., all triangles associated with u).

5.2.2 Problem Definition

In this chapter, we consider Problem 5.1, where we use a general approach of simultaneously reducing bias and variance, instead of minimizing a specific measure of estimation error, to reduce different measures of estimation error robustly.

Problem 5.1: Distributed Global and Local Triangle Counting in a Graph Stream

- **Given:**
  - a graph stream $(e^{(1)}, e^{(2)}, ...)$,
  - k distributed storages in each of which up to b (≥ 2) edges can be stored
- **Maintain:** estimates of the global triangle count $|\mathcal{T}^{(t)}|$ and the local triangle counts $\{(u, |\mathcal{T}^{(t)}[u]|)\}_{u \in \mathcal{V}^{(t)}}$ for current time $t \in \{1, 2, ...\}$,
- **to Minimize:** the estimation errors
- **Subject to:** the following realistic conditions:

C1 **Knowledge free:** No prior knowledge of the input graph stream (e.g., the counts of nodes and edges) is available.

C2 **Shared nothing environment:** Data stored in the storage of a machine is not accessible by the other machines.

C3 **One pass:** Edges are accessed one by one in their arrival order. Past edges are not accessible by a machine unless they are stored in the given storage of the machine.

5.3 Proposed Algorithms: TRI-FLY and COCOS

In this section, we present two distributed streaming algorithms for Problem 5.1. First, we provide an overview with the common structure and notations in Section 5.3.1. Then, we present a baseline algorithm TRI-FLY and our proposed algorithm COCOS (**Co**nditional **Co**unting and **S**ampling) in Sections 5.3.2 and 5.3.3, respectively. After that, we discuss lazy aggregation in Section 5.3.4. Lastly, we discuss extensions of the algorithms with multiple sources, masters, and aggregators in Section 5.3.5

5.3.1 Overview

Figure 5.2 describes the roles of machines and the flow of data in the algorithms described in the following subsections. For simplicity, we assume one source, one master and one aggregator although extending the algorithms with multiple of them is trivial, as discussed in Section 5.3.5. Edges are streamed from the source to the master, which unicasts or broadcasts the edges to the workers. Each worker counts the global and local triangles from the received edges using its local storage, and it sends the counts to the aggregator. Since we assume a shared-nothing environment in Problem 5.1, each worker cannot access data stored in the other worrkers. The counts are aggregated in the aggregator, which gives the final estimates of the global and local triangle counts.

Before describing the algorithms, we define the notations used in them. We use k to denote the number of workers and use b to denote the storage budget per worker (i.e., the maximum number of

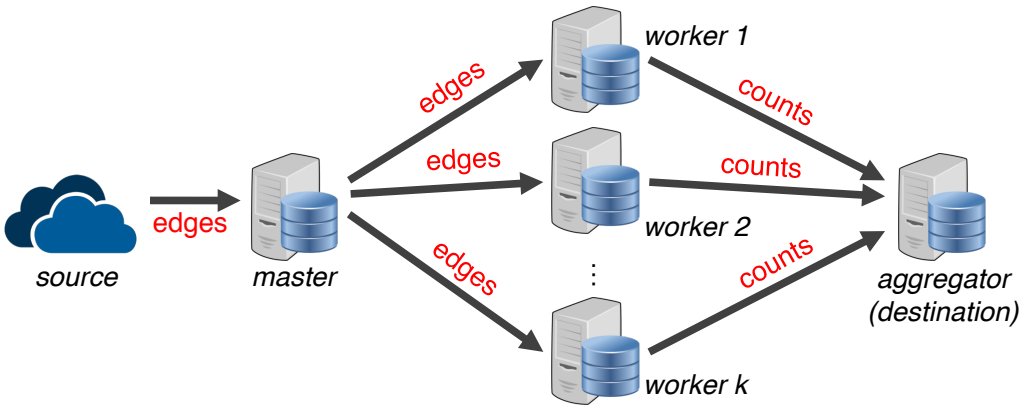


Figure 5.2: **Roles of machines and the flow of data in TRI-FLY and COCOS.** Extensions of them with multiple sources, masters, and aggregators are discussed in Section 5.3.5.

edges that we store in each worker). For each $i \in \{1, ..., k\}$, we let $\mathcal{E}_i$ be the edges currently stored in the i-th worker and let $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i)$ be the graph composed of the edges in $\mathcal{E}_i$. For each node $u \in \mathcal{V}_i$, $\mathcal{N}_i[u]$ denotes the neighboring nodes of u in $\mathcal{G}_i$. Since its storage is limited, each worker uses sampling to decide which edges to store. We use l_i to denote the number of edges that the i-th worker has considered for sampling so far. Lastly, $\bar{c}$ indicates the estimate of the global triangle count, and for each node u, $c[u]$ indicates the estimate of the local triangle count of u.

5.3.2 Baseline Algorithm: TRI-FLY

We present TRI-FLY, a baseline algorithm for Problem 5.1. The pseudo code of TRI-FLY is given in Algorithm 5.1. We first describe the master, the workers, and the aggregator in TRI-FLY. Then, we discuss its advantages and disadvantages.

Master (lines 1-2): The master simply broadcasts every edge from the source to every worker.

Workers (lines 3-17): Each worker independently estimates the global and local triangle counts using TRIEST$_{\text{IMPR}}$, a state-of-the-art streaming algorithm based on reservoir sampling. Note that the workers use different random seeds and thus give different results. Each worker $i \in \{1, ..., k\}$ starts with an empty storage (i.e., $\mathcal{E}_i = \emptyset$) (line 3 of Algorithm 5.1). Whenever it receives an edge $\{u, v\}$ (line 4) from the master, the worker first counts the triangles with $\{u, v\}$ in its local storage by calling the procedure COUNT (line 5), Then, the worker calls procedure SAMPLE (line 6) to store $\{u, v\}$ in its local storage with non-zero probability. We describe the procedures SAMPLE and COUNT below.

In the procedure SAMPLE (lines 14-17), each worker $i \in \{1, ..., k\}$ first increases l_i, the number of edges considered for sampling, by one since the new edge $\{u, v\}$ is being considered. If its local storage is not full (i.e., $|\mathcal{E}_i| < b$), the worker stores $\{u, v\}$ by adding $\{u, v\}$ to $\mathcal{E}_i$ (line 15). If the local storage is full (i.e., $|\mathcal{E}_i| = b$), the worker stores $\{u, v\}$ with probability b/l_i by replacing an edge chosen at random uniformly in $\mathcal{E}_i$ with $\{u, v\}$ (lines 16-17). This is the standard reservoir sampling, which guarantees that each of the l_i edges is sampled and included in $\mathcal{E}_i$ with the equal probability $\min(1, b/l_i)$.

In the procedure COUNT (lines 8-12), each worker $i \in \{1, ..., k\}$ finds the common neighbors of nodes u and v in graph $\mathcal{G}_i$, a graph consisting of the edges $\mathcal{E}_i$ in its local storage (line 9). Each common neighbor w indicates the existence of triangle $\{u, v, w\}$. Thus, for each common neighbor w, the worker increases the global triangle count, and the local triangle counts of nodes u, v, and w by sending the increases to the aggregator (lines 10 and 12). The amount of increase in the counts is $1/(p_i[uvw])$ for

Algorithm 5.1 TRI-FLY: Baseline Algorithm (not recommended)

Input: (1) input graph stream: $(e^{(1)}, e^{(2)}, ...)$
(2) storage budget in each worker: b (≥ 2)
Output: (1) estimated global triangle count: $\bar{c}$
(2) estimated local triangle counts: $c[u]$ for each node u

Master:

for each edge $\{u, v\}$ from the source **do**
    broadcast $\{u, v\}$ to every worker

Worker (each worker with index i):

$\mathcal{E}_i \leftarrow \emptyset$; $l_i \leftarrow 0$
for each edge $\{u, v\}$ from the master **do**
    COUNT($\{u, v\}$)
    SAMPLE($\{u, v\}$)
procedure COUNT($\{u, v\}$):
    $sum \leftarrow 0$
    for each node $w \in \mathcal{N}_i[u] \cap \mathcal{N}_i[v]$ **do**
        send $(w, 1/(p_i[uvw]))$ to the aggregator
        $sum \leftarrow sum + 1/(p_i[uvw])$ ▷ see Eq. (5.1) for $p_i[uvw]$
    send $(*, sum)$, (u, sum) and (v, sum) to the aggregator ▷ '$*$' indicates the global triangle count
procedure SAMPLE($\{u, v\}$):
    $l_i \leftarrow l_i + 1$.
    if $|\mathcal{E}_i| < b$ **then** $\mathcal{E}_i \leftarrow \mathcal{E}_i \cup \{\{u, v\}\}$
    else if a random number in $\mathbf{Bernoulli}(b/l_i)$ is 1
        replace an edge chosen at random uniformly in $\mathcal{E}_i$ with $\{u, v\}$

Aggregator:

$\bar{c} \leftarrow 0$
initialize an empty map c with default value 0
for each pair (u, δ) from the workers **do**
    if $u = *$ **then** $\bar{c} \leftarrow \bar{c} + \delta/k$
    else $c[u] \leftarrow c[u] + \delta/k$

each triangle $\{u, v, w\}$, where

$$p_i[uvw] := \min\left(1, \frac{b(b-1)}{l_i(l_i-1)}\right) \tag{5.1}$$

is the probability that triangle $\{u, v, w\}$ is discovered by worker i. In other words, $p_i[uvw]$ is the probability that both $\{v, w\}$ and $\{w, u\}$ are in $\mathcal{E}_i$ when $\{u, v\}$ arrives at worker i.[1] Increasing counts by $1/(p_i[uvw])$ guarantees that the expected amount of the increase sent from each worker is exactly $1(= p_i[uvw] \times 1/(p_i[uvw]) + (1 - p_i[uvw]) \times 0)$ for each triangle, enabling TRI-FLY to give unbiased estimates. See Theorem 5.1 in Section 5.4.1 for a detailed proof.

Aggregator (lines 18-22): The aggregator maintains and updates the estimate $\bar{c}$ of the global triangle count and the estimate $c[u]$ of the local triangle count of each node u. Specifically, it increases the estimates by $1/k$ of what it receives, averaging the increases sent from the workers (lines 21 and 22).

Advantages and Disadvantages of TRI-FLY: Our theoretical and empirical analyses in the following sections show the advantages of TRI-FLY. Specifically, TRI-FLY gives unbiased estimates, and the variances of the estimates decrease inversely proportional to the number of workers (see Theorems 5.1 and 5.3 in Section 5.4.1). Moreover, TRI-FLY gives the same results as $\text{TRIEST}_{\text{IMPR}}$ [SERU17], a state-of-the-art streaming algorithm, when a single worker is used.

However, TRI-FLY incurs a redundant use of computational and storage resources. Specifically, each edge can be replicated and stored in up to k workers, and each triangle can be counted repeatedly by up to k workers. Due to its redundant use of storage, no matter how many workers are used, TRI-FLY cannot guarantee exact triangle counts if the number of edges so far (i.e., t) is greater than $b + 1$.

5.3.3 Proposed Algorithm: CoCoS

To address the drawbacks of TRI-FLY, we propose CoCoS, an improved algorithm for Problem 5.1. The pseudo code of CoCoS is given in Algorithm 5.2. We first describe the master, the workers, and the aggregator in CoCoS. Then, we prove its properties. Lastly, we discuss adaptive node mapping.

5.3.3.1 Algorithm Description

Master (lines 1-3): The master requires a function f that maps each node to a worker. We assume that f is given and discuss it later in Section 5.3.3.3. The master sends each edge $\{u, v\}$ to the workers depending on $f(u)$ and $f(v)$ as follows:

- Case LUCKY (line 2): If nodes u and v are assigned to the same worker by f (i.e., $f(u) = f(v)$), then the master sends $\{u, v\}$ only to the worker (i.e., the $f(u)$-th worker).
- Case UNLUCKY (line 3): Otherwise (i.e., if $f(u) \neq f(v)$), the master sends $\{u, v\}$ to every worker.

Workers (lines 4-8): The workers start with an empty storage (line 4). Whenever they receive an edge $\{u, v\}$ from the master (line 5), they count the triangles with $\{u, v\}$ in its local storage by calling the procedure COUNT (line 6), as in TRI-FLY. However, the procedure SAMPLE is called selectively depending on $f(u)$ and $f(v)$ as follows:

[1] For $\{v, w\}$ to be in $\mathcal{E}_i$, $\{v, w\}$ should be one among b edges sampled from l_i edges, i.e., $p[\{v, w\} \in \mathcal{E}_i] = \min(1, b/l_i)$. For $\{w, u\}$ to be in $\mathcal{E}_i$, given $\{v, w\}$ is in $\mathcal{E}_i$, $\{w, u\}$ should be one among $b - 1$ edges sampled from $l_i - 1$ edges, i.e., $p[\{w, u\} \in \mathcal{E}_i | \{v, w\} \in \mathcal{E}_i] = \min(1, (b-1)/(l_i - 1))$. Eq. (5.1) follows from $p_i[uvw] = p[\{w, u\} \in \mathcal{E}_i, \{v, w\} \in \mathcal{E}_i] = p[\{v, w\} \in \mathcal{E}_i] \times p[\{w, u\} \in \mathcal{E}_i | \{v, w\} \in \mathcal{E}_i]$.

Algorithm 5.2 CoCoS: Proposed Algorithm

Input: (1) input graph stream: $(e^{(1)}, e^{(2)}, ...)$
(2) storage budget in each worker: b (≥ 2)
Output: (1) estimated global triangle count: $\bar{c}$
(2) estimated local triangle counts: $c[u]$ for each node u

Master:

1: **for** each edge $\{u, v\}$ from the source **do**
2: **if** $f(u) = f(v)$ **then** send $\{u, v\}$ to worker $f(u)$ ▷ Case LUCKY
3: **else**send $\{u, v\}$ to every worker ▷ Case UNLUCKY

Worker (each worker with index i):

4: $\mathcal{E}_i \leftarrow \emptyset$; $l_i \leftarrow 0$
5: **for** each edge $\{u, v\}$ from the master **do**
6: COUNT($\{u, v\}$) ▷ see Algorithm 5.1 for COUNT()
7: **if** $f(u) = i$ or $f(v) = i$ **then** ▷ Case ASSIGNED
8: SAMPLE($\{u, v\}$) ▷ see Algorithm 5.1 for SAMPLE()

Aggregator:

9: $\bar{c} \leftarrow 0$
10: initialize an empty map c with default value 0
11: **for** each pair (u, δ) from the workers **do**
12: **if** $u = *$ **then** $\bar{c} \leftarrow \bar{c} + \delta$
13: **else** $c[u] \leftarrow c[u] + \delta$

- Case ASSIGNED (line 7): If $f(u) = i$ or $f(v) = i$, the i-th worker considers storing $\{u, v\}$ in its local storage by calling SAMPLE.
- Case UNASSIGNED: Otherwise (i.e., if $f(u) \neq i \neq f(v)$), the i-th worker simply discards $\{u, v\}$ without considering storing it.

Aggregator (lines 9-13): The aggregator applies each received update to the corresponding estimate.

5.3.3.2 Basic Properties

The properties of CoCoS that we use for the theoretical analysis in Section 5.4 are stated in Lemma 5.1.

Lemma 5.1:

Algorithm 5.2 has the following properties:

P1 **Limited redundancy in storage:** Each edge is stored in at most two workers.

P2 **No redundancy in computation:** Each triangle is counted by at most one worker.

P3 **No definitely missing triangles:** Each triangle is counted with non-zero probability.

Proof. First, we prove P1. Each edge $\{u, v\}$ can be stored in a worker only when case ASSIGNED happens. Since case ASSIGNED happens in at most two workers (i.e., the $f(u)$-th worker and the $f(v)$-

Algorithm 5.3 Master in CoCoS$_{\text{OPT}}$

Input: (1) input graph stream: $(e^{(1)}, e^{(2)}, ...)$
(2) tolerance for load difference: θ (≥ 0) .
Output: edges sent to workers

$l_i \leftarrow 0, \forall i \in \{1, ..., k\}$
for each edge $\{u, v\}$ from the source **do**
  $i^* \leftarrow \arg\min_{i \in \{1,...,k\}} l_i$
  if u and v have not been assigned to a worker by f **then**
    $f(u) \leftarrow i^*$; $f(v) \leftarrow i^*$
  else if u has not been assigned to a worker by f **then**
    if $l_{f(v)} \leq (1+\theta) l_{i^*}$ **then** $f(u) \leftarrow f(v)$ **else** $f(u) \leftarrow i^*$
  else if v has not been assigned to a worker by f **then**
    if $l_{f(u)} \leq (1+\theta) l_{i^*}$ **then** $f(v) \leftarrow f(u)$ **else** $f(v) \leftarrow i^*$
  if $f(u) = f(v)$ **then** ▷ Case LUCKY
    send $\{u, v\}$ to worker $f(u)$
    $l_{f(u)} \leftarrow l_{f(u)} + 1$
  else ▷ Case UNLUCKY
    send $\{u, v\}$ to every worker
    $l_{f(u)} \leftarrow l_{f(u)} + 1$; $l_{f(v)} \leftarrow l_{f(v)} + 1$

th worker), $\{u, v\}$ can be stored in at most two workers. Then, we prove P2 and P3 by showing that, for each triangle, there exists exactly one worker that counts it with non-zero probability. Consider a triangle $\{u, v, w\}$ and assume $\{u, v\}$ is the last edge (i.e., $t_{vw} < t_{uv}$ and $t_{wu} < t_{uv}$) without loss of generality. If $f(u) = f(v)$ (case LUCKY), all the workers except the $f(u)(= f(v))$-th worker cannot count $\{u, v, w\}$ since $\{u, v\}$ is sent only to the $f(u)$-th worker. Since the $f(u)(= f(v))$-th worker stores $\{v, w\}$ and $\{w, u\}$ with non-zero probability (case ASSIGNED happens for both edges), it counts $\{u, v, w\}$ with non-zero probability. If $f(u) \neq f(v)$ (case UNLUCKY), although $\{u, v\}$ is sent to every worker, all the workers except the $f(w)$-th worker cannot count $\{u, v, w\}$ since they cannot store both $\{v, w\}$ and $\{w, u\}$ (case UNASSIGNED happens for at least one of the edges). Since the $f(w)$-th worker stores $\{v, w\}$ and $\{w, u\}$ with non-zero probability (case ASSIGNED happens for both edges), it counts $\{u, v, w\}$ with non-zero probability. Therefore, in both cases, there exists exactly one worker that counts $\{u, v, w\}$ with non-zero probability, satisfying P2 and P3. ■

P1 is desirable for accuracy. Less redundancy in storage enables us to store more unique edges, which we can estimate triangle counts more accurately from. P2 is desirable for speed. P3 enables CoCoS to give unbiased estimates of triangle counts, as we explain in Section 5.4. P3 is what we should not compromise while reducing the redundancy in storage and computation. For example, further reducing redundancy in storage by storing each edge in at most one worker compromises P3 unless k equals 1.

5.3.3.3 Adaptive Node Mapping Function

So far we have assumed that the function f, which assigns each node to a worker, is given. We discuss how to design f and propose CoCoS$_{\text{OPT}}$, which is CoCoS with our proposed function as f.

Design Goals: We say an edge $\{u, v\}$ is assigned to the i-th worker if $f(u) = i$ or $f(v) = i$ and thus $\{u, v\}$ can possibly be stored in the i-th worker. In Algorithm 5.2, the load l_i of each i-th worker

Table 5.2: **Advantages of Case LUCKY.** Case LUCKY saves storage, communication, and computation costs, compared to case UNLUCKY.

Cases	LUCKY	UNLUCKY
storage (edge is stored in at most)	1 worker	2 workers
communication (edge is sent to)		k workers
computation (COUNT() is called in)		k workers

denotes the number of edges assigned to the worker. Then, two goals that a desirable f function should meet are as follows:

G1 **Storage:** The redundant use of storage (i.e., the number of edges stored in multiple workers) should be minimized.

G2 **Load Balancing:** A similar number of edges should be assigned to every worker, i.e., $l_i \approx l_j$, $\forall i, j \in \{1, ..., k\}$.

However, achieving both goals is non-trivial because the goals compete with each other. For example, if we assign every node to the same worker, then the first goal is achieved since every edge is stored only in the machine. However, this maximizes load imbalance, conflicting with the second goal. Moreover, f should be decided without additional passes or any prior knowledge of the input stream, due to our conditions in Problem 5.1.

CoCoS$_{\text{OPT}}$ with Adaptive f. We propose CoCoS$_{\text{OPT}}$, where the master, described in Algorithm 5.3, adaptively decides the function f based on current loads of the workers so that the redundancy of storage is minimized within a specified level of load difference.

Recall that, in CoCoS, case LUCKY is preferred over case UNLUCKY for reducing the redundancy in storage. This is because each edge $\{u, v\}$ is stored in at most one worker in case LUCKY (i.e., $f(u) = f(v)$), while it is stored in at most two workers in case UNLUCKY (i.e., $f(u) \neq f(v)$). Let the i^*-th worker be the worker with least assigned edges so far (line 3). If an edge $\{u, v\}$ with two new nodes u and v arrives, the master assigns both nodes to the i^*-th worker (lines 4-5) for pursuing case LUCKY and balancing loads. If an edge $\{u, v\}$ with one new node u (without loss of generality) arrives, the master assigns u to the $f(v)$-th worker, for case LUCKY to happen, as long as the load of the $f(v)$-th worker is not higher than $(1 + \theta)$ times of the load of the i^*-th worker. Otherwise, load balancing is prioritized, and u is assigned to the i^*-th worker (lines 6-9). Once $f(u)$ and $f(v)$ are determined, each edge $\{u, v\}$ is sent to the worker(s) depending on $f(u)$ and $f(v)$ as in Algorithm 5.2, and the load of the corresponding worker(s) is updated (lines 10-15). Note that $f(u)$ and $f(v)$ are never changed once they are determined, Since the assignments by f are only in the master, along each edge to each worker, one bit indicating whether the edge is assigned to the worker or not should be sent to be used in line 7 of Algorithm 5.2.

Advantages of CoCoS$_{\text{OPT}}$: By co-optimizing storage and load balancing, CoCoS$_{\text{OPT}}$ stores more uniques edges and thus produces more accurate estimates than CoCoS$_{\text{SIMPLE}}$, which is CoCoS using the simple modulo function as f. Although our explanation so far has focused on storage and load balancing, CoCoS$_{\text{OPT}}$ also improves upon CoCoS$_{\text{SIMPLE}}$ in terms of speed by increasing the chance of case LUCKY, which saves not only storage but also communication and computation costs, as summarized in Table 5.2.

5.3.4 Lazy Aggregation

In the procedure COUNT of Algorithm 5.1, which is commonly used by TRI-FLY and COCOS, each worker sends the update of the local triangle count of node w to the aggregator whenever it discovers each triangle $\{u, v, w\}$ (line 10). Likewise, each worker sends the updates of the global triangle count and the local triangle counts of nodes u and v to the aggregator whenever it processes each edge $\{u, v\}$ (line 12). In cases where this eager aggregation is not needed, we reduce the amount of communication by employing lazy aggregation. Specifically, counts aggregated locally in each worker are sent to and aggregated in the aggregator (and removed from the workers) when they are queried.

5.3.5 Multiple Sources, Masters and Aggregators

Although our experiments in Section 5.5.3 show that the performance bottlenecks of proposed algorithms are workers rather than the master, multiple masters can be considered for handling multiple sources or for fault tolerance. Consider the case when edges are streamed from one or more sources to multiple masters without duplication. By simply using the same non-adaptive node mapping function f [2] (e.g., the modulo function) in every master, we can run masters independently without affecting the accuracy of TRI-FLY or COCOS. This is because, in such cases, masters do not have any state and thus have nothing to share with each other.

Multiple aggregators are required when outputs (i.e., 1 global triangle count and $|\mathcal{V}^{(t)}|$ local triangle counts) do not fit one machine or aggregation is a performance bottleneck. In TRI-FLY and COCOS, workers send key-value pairs, whose key is either '$*$' or a node id, to the aggregator (line 12 of Algorithm 5.1). The computation and storage required for aggregation are distributed across multiple aggregators if workers use the same hash function (that maps each key to an aggregator) to decide where to send each key-value pair.

5.4 Theoretical Analysis

We theoretically analyze the accuracy, time complexity, and space complexity of COCOS and TRI-FLY.

5.4.1 Accuracy Analysis

We analyze the biases and variances of the estimates given by COCOS and TRI-FLY. The biases and variances determine the estimation error of the algorithms. We first prove that both COCOS and TRI-FLY give estimates with no bias. Then, we analyze the variances of the estimates to give an intuition why COCOS is more accurate than TRI-FLY.

5.4.1.1 Bias Analysis

We prove the unbiasedness of TRI-FLY and COCOS. That is, we show that TRI-FLY and COCOS give estimates whose expected values are equal to the true triangle counts. For proofs, consider $\mathcal{G}^{(t)} = (\mathcal{V}^{(t)}, \mathcal{E}^{(t)})$, which is the graph consisting of the edges arriving at time t or earlier. We define $\bar{c}^{(t)}$ as $\bar{c}$ in the aggregator after edge $e^{(t)}$ is processed. Then, $\bar{c}^{(t)}$ is an estimate of $|\mathcal{T}^{(t)}|$, the count of global

[2] A node mapping function f is non-adaptive if its mapping does not depend on any states. Algorithm 5.3 is adaptive since its mapping depends on the loads of workers.

triangles in $\mathcal{G}^{(t)}$. Likewise, for each node $u \in \mathcal{V}^{(t)}$, we define $c^{(t)}[u]$ as $c[u]$ in the aggregator after $e^{(t)}$ is processed. Then, each $c^{(t)}[u]$ is an estimate of $|\mathcal{T}^{(t)}[u]|$, the count of local triangles of u in $\mathcal{G}^{(t)}$.

Theorem 5.1: Unbiasedness of TRI-FLY

At any time, the expected values of the estimates given by TRI-FLY are equal to the true global and local triangle counts. That is, in Algorithm 5.1,

$$\mathbb{E}[\bar{c}^{(t)}] = |\mathcal{T}^{(t)}|, \qquad \forall t \in \{1, 2, ...\}.$$
$$\mathbb{E}[c^{(t)}[u]] = |\mathcal{T}^{(t)}[u]|, \qquad \forall u \in \mathcal{V}^{(t)},\ \forall t \in \{1, 2, ...\}.$$

Proof. The unbiasedness of TRI-FLY follows from that of TRIEST$_{\text{IMPR}}$ [SERU17], which each worker in TRI-FLY runs independently. Let $\bar{c}_i^{(t)}$ be the global triangle count sent from each worker i by time t. By line 21 of Algorithm 5.1, $\bar{c}^{(t)} = \sum_{i=1}^{k} \bar{c}_i^{(t)}/k$. From $\mathbb{E}[\bar{c}_i^{(t)}] = |\mathcal{T}^{(t)}|$ (Theorem 4.12 of [SERU17]),

$$\mathbb{E}[\bar{c}^{(t)}] = \sum_{i=1}^{k} \mathbb{E}[\bar{c}_i^{(t)}]/k = |\mathcal{T}^{(t)}|.$$

Likewise, for each node $u \in \mathcal{V}^{(t)}$, let $c_i^{(t)}[u]$ be the local triangle count of u sent from each worker i by time t. By line 22 of Algorithm 5.1, $c^{(t)}[u] = \sum_{i=1}^{k} c_i^{(t)}[u]/k$. From $\mathbb{E}[c_i^{(t)}[u]] = |\mathcal{T}^{(t)}[u]|$ (Theorem 4.12 of [SERU17]),

$$\mathbb{E}[c^{(t)}[u]] = \sum_{i=1}^{k} \mathbb{E}[c_i^{(t)}[u]]/k = |\mathcal{T}^{(t)}[u]|.$$

■

Theorem 5.2: Unbiasedness of CoCoS

At any time, the expected values of the estimates given by CoCoS are equal to the true global and local triangle counts. That is, in Algorithm 5.2,

$$\mathbb{E}[\bar{c}^{(t)}] = |\mathcal{T}^{(t)}|, \qquad \forall t \in \{1, 2, ...\}.$$
$$\mathbb{E}[c^{(t)}[u]] = |\mathcal{T}^{(t)}[u]|, \qquad \forall u \in \mathcal{V}^{(t)},\ \forall t \in \{1, 2, ...\}.$$

Proof. Consider a triangle $\{u, v, w\} \in \mathcal{T}^{(t)}$ and assume without loss of generality that $t_{vw} < t_{wu} < t_{uv} \leq t$. By Lemma 5.1, there is exactly one worker that can count $\{u, v, w\}$. Let $f(uvw) \in \{1, ..., k\}$ denote the worker. Let $d_i[uvw]$ be the contribution of $\{u, v, w\}$ to each of $\bar{c}^{(t)}$, $c^{(t)}[u]$, $c^{(t)}[v]$, and $c^{(t)}[w]$ by each i-th worker. Then, $d_i[uvw] = 0$ if $i \neq f(uvw)$. If we let $\mathcal{E}_{f(uvw)}^{(t_{uv})}$ be the set of edges stored in the $f(uvw)$-th worker when $\{u, v\}$ arrives, then by lines 10-12 of Algorithm 5.1 and lines 12-13 of Algorithm 5.2,

$$d_{f(uvw)}[uvw] = \begin{cases} 1/(p_{f(uvw)}[uvw]) & \text{if } \{v, w\}, \{w, u\} \in \mathcal{E}_{f(uvw)}^{(t_{uv})} \\ 0 & \text{otherwise.} \end{cases}$$

By definition, $p_{f(uvw)}[uvw]$ is the probability that both $\{v, w\}$ and $\{w, u\}$ are in $\mathcal{E}^{(t_{uv})}_{f(uvw)}$. Therefore, $\mathbb{E}[d_{f(uvw)}[uvw]] = 1$. By linearity of expectation, the following equations hold:

$$\mathbb{E}[\bar{c}^{(t)}] = \mathbb{E}\left[\sum_{i=1}^{k} \sum_{\{u,v,w\}\in\mathcal{T}^{(t)}} d_i[uvw]\right] = \sum_{i=1}^{k} \sum_{\{u,v,w\}\in\mathcal{T}^{(t)}} \mathbb{E}[d_i[uvw]]$$
$$= \sum_{\{u,v,w\}\in\mathcal{T}^{(t)}} \mathbb{E}[d_{f(uvw)}[uvw]] = \sum_{\{u,v,w\}\in\mathcal{T}^{(t)}} 1 = |\mathcal{T}^{(t)}|, \qquad \forall t \in \{1, 2, ...\}.$$

$$\mathbb{E}[c^{(t)}[u]] = \mathbb{E}\left[\sum_{i=1}^{k} \sum_{\{u,v,w\}\in\mathcal{T}^{(t)}[u]} d_i[uvw]\right] = \sum_{i=1}^{k} \sum_{\{u,v,w\}\in\mathcal{T}^{(t)}[u]} \mathbb{E}[d_i[uvw]]$$
$$= \sum_{\{u,v,w\}\in\mathcal{T}^{(t)}[u]} \mathbb{E}[d_{f(uvw)}[uvw]] = \sum_{\{u,v,w\}\in\mathcal{T}^{(t)}[u]} 1 = |\mathcal{T}^{(t)}[u]|, \qquad \forall t \in \{1, 2, ...\}, \forall u \in \mathcal{V}^{(t)}.$$

Hence, the estimates given by Algorithm 5.2 are unbiased. ■

5.4.1.2 Variance Analysis

Having shown that the estimate $\bar{c}^{(t)}$ is an unbiased estimate of the global triangle count $|\mathcal{T}^{(t)}|$, we analyze its variance in TRI-FLY and COCOS to give an intuition why the variance is smaller in COCOS than in TRI-FLY. The variance of each $c^{(t)}[u]$ can be analyzed in the same manner considering only the local triangles with node u. We first define the two types of triangle pairs illustrated in Figure 5.3.

Definition 5.1: Type 1 Triangle Pair

A Type 1 triangle pair is two different triangles $\{u, v, w\}$ and $\{u, v, x\}$ sharing an edge $\{u, v\}$ satisfying $t_{wu} = \max(t_{uv}, t_{vw}, t_{wu})$ and $t_{xu} = \max(t_{uv}, t_{vx}, t_{xu})$.

Definition 5.2: Type 2 Triangle Pair

A Type 2 triangle pair is two different triangles $\{u, v, w\}$ and $\{u, v, x\}$ sharing an edge $\{u, v\}$ satisfying $t_{vw} = \max(t_{uv}, t_{vw}, t_{wu})$ and $t_{xu} = \max(t_{uv}, t_{vx}, t_{xu})$.

Let $p^{(t)}$ and $q^{(t)}$ be the numbers of Type 1 pairs and Type 2 pairs, respectively, in $\mathcal{G}^{(t)}$, which is the graph composed of the edges arriving at time t or earlier. Then, we define $z^{(t)}$ as

$$z^{(t)} := \max\left(0, |\mathcal{T}^{(t)}|\left(\frac{(t-1)(t-2)}{b(b-1)} - 1\right) + \left(p^{(t)} + q^{(t)}\right)\frac{t-1-b}{b}\right),$$

Our analysis in this section is largely based on Lemma 5.2, where $z^{(t)}$ upper bounds the variance of the estimate $\bar{c}^{(t)}$ in $\text{TRIEST}_{\text{IMPR}}$, which is equivalent to TRI-FLY and COCOS with a single worker. Notice that $z^{(t)}$ decreases as the storage budget (i.e., b) increases, while $z^{(t)}$ increases as the numbers of edges (i.e., t), triangles (i.e., $|\mathcal{T}^{(t)}|$), and Type 1 or 2 triangle pairs (i.e., $p^{(t)}$ and $q^{(t)}$) increase.

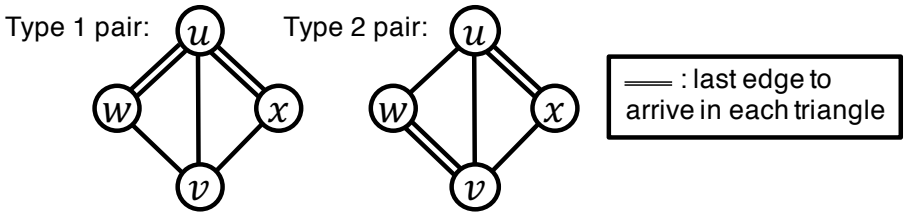


Figure 5.3: **Illustrations of Type 1 and Type 2 triangle pairs.**

Lemma 5.2: Variance of TRIEST$_{\text{IMPR}}$ [SERU17]

Assume that a single worker is used (i.e., $k = 1$) in Algorithm 5.1 or Algorithm 5.2. At any time t, the variance of the estimate $\bar{c}^{(t)}$ of the global triangle count $|\mathcal{T}^{(t)}|$ is upper bounded by $z^{(t)}$. That is,

$$Var[\bar{c}^{(t)}] \leq z^{(t)}, \ \forall t \in \{1, 2, ...\}.$$

The upper bound of the variance of the estimate $\bar{c}^{(t)}$ in TRI-FLY decreases inversely proportional to the number of workers, as formalized in Theorem 5.3. This follows from the fact that $\bar{c}^{(t)}$ in TRI-FLY is the simple average of k estimates obtained by running TRIEST$_{\text{IMPR}}$ independently in k workers.

Theorem 5.3: Variance of TRI-FLY

In Algorithm 5.1, the upper bound of the variance of the estimate $\bar{c}^{(t)}$, given in Lemma 5.2, decreases inversely proportional to the number of workers k. That is,

$$Var[\bar{c}^{(t)}] \leq z^{(t)}/k, \ \forall t \in \{1, 2, ...\}. \tag{5.2}$$

Proof. Let $\bar{c}_i^{(t)}$ be the global triangle count sent from each worker i by time t. Then, by line 21 of Algorithm 5.1, $\bar{c}^{(t)} = \sum_{i=1}^{k} \bar{c}_i^{(t)}/k$. Since $\bar{c}_i^{(t)}$ of each worker $i \in \{1, ..., k\}$ is independent from that of the other workers,

$$Var[\bar{c}^{(t)}] = \sum\nolimits_{i=1}^{k} Var[\bar{c}_i^{(t)}/k] = \sum\nolimits_{i=1}^{k} Var[\bar{c}_i^{(t)}]/k^2 \leq k \cdot z^{(t)}/k^2 = z^{(t)}/k,$$

where the inequality follows from Theorem 4.13 in [SERU17], which states that $Var[\bar{c}_i^{(t)}] \leq z^{(t)}$ for each worker $i \in \{1, ..., k\}$. ■

The variance of the estimate $\bar{c}^{(t)}$ in COCOS depends on how the triangles in $\mathcal{T}^{(t)}$ are distributed across workers. By Lemma 5.1, there is exactly one worker that can count each triangle. Thus, for each $i \in \{1, ..., k\}$, let $\mathcal{T}_i^{(t)} \subset \mathcal{T}^{(t)}$ be the set of triangles that can be counted by the i-th worker. Likewise, let $p_i^{(t)}$ and $q_i^{(t)}$ be the numbers of Type 1 pairs and Type 2 pairs, respectively, among the triangles in $\mathcal{T}_i^{(t)}$. Then, for each i-th worker, we define $z_i^{(t)}$ as

$$z_i^{(t)} := \max\left(0, |\mathcal{T}_i^{(t)}|\left(\frac{(l_i^{(t)} - 1)(l_i^{(t)} - 2)}{b(b-1)} - 1\right) + \left(p_i^{(t)} + q_i^{(t)}\right)\frac{l_i^{(t)} - 1 - b}{b}\right),$$

where $l_i^{(t)}$ is the load l_i of each i-th worker when $e^{(t)}$ arrives. This term is used to upper bound the variance of $\bar{c}^{(t)}$ in Theorem 5.4. According to the theorem, each worker's contribution to the variance decreases as the storage budget b increases, while the contribution increases as more edges, triangles, and Type 1 or 2 triangle pairs (whose discovering probabilities are positively correlated) are assigned to the worker, which matches our intuition.

Theorem 5.4: Variance of CoCoS

At any time t, the variance of the estimate $\bar{c}^{(t)}$ of the global triangle count $|\mathcal{T}^{(t)}|$ in Algorithm 5.2 is upper bounded by the sum of $z_i^{(t)}$ in each i-th worker. That is

$$Var[\bar{c}^{(t)}] \leq \sum\nolimits_{i=1}^{k} z_i^{(t)}, \ \forall t \in \{1, 2, ...\} \tag{5.3}$$

Sketch of Proof. Let $\bar{c}_i^{(t)}$ be the global triangle count sent from each i-th worker to the corresponding aggregator by time t. Then, by line 12, $\bar{c}^{(t)} = \sum_{i=1}^{k} \bar{c}_i^{(t)}$. Since $\bar{c}_i^{(t)}$ of each i-th worker is independent from that of the other workers,

$$Var[\bar{c}^{(t)}] = \sum\nolimits_{i=1}^{k} Var[\bar{c}_i^{(t)}]. \tag{5.4}$$

Then, Theorem 4.13 in [SERU17] is generalized for each $\bar{c}_i^{(t)}$ to $Var[\bar{c}_i^{(t)}] \leq z_i^{(t)}$. This generalization and Eq. (5.4) imply Eq. (5.3). ■

We compare the variance of $\bar{c}^{(t)}$ in CoCoS (i.e., Eq. (5.3)) to that in Tri-Fly Eq. (5.2) when a uniform random function is used as f. Lemma 5.3 states how rapidly each random variable in Eq. (5.3) decreases depending on the number of workers (i.e., k). Note that $\mathbb{E}_f[q_i^{(t)}]$ decreases faster than $O(q^{(t)}/k)$ since more workers result in more Type 2 triangle pairs not assigned to any worker.[3]

Lemma 5.3

Assume $f : \mathcal{V} \to \{1, ..., k\}$ is a random function where $\mathbb{P}[f(u) = i] = 1/k$ for each node $u \in \mathcal{V}$ and each i-th worker. Let $p^{(t)}$ and $q^{(t)}$ be the counts of Type 1 and Type 2 triangle pairs in $\mathcal{G}^{(t)}$. Then, the following equations hold for any time $t \in \{1, 2, ...\}$:

$$\mathbb{E}_f[|\mathcal{T}_i^{(t)}|] = O\left(\frac{|\mathcal{T}^{(t)}|}{k}\right), \ \mathbb{E}_f[l_i^{(t)}] = O\left(\frac{t}{k}\right),$$
$$\mathbb{E}_f[p_i^{(t)}] = O\left(\frac{p^{(t)}}{k}\right), \ \mathbb{E}_f[q_i^{(t)}] = O\left(\frac{q^{(t)}}{k^2}\right).$$

Proof. See Section 5.7. ■

[3] A Type 2 triangle pair is not assigned to any worker if the two triangles are assigned to different workers.

Table 5.3: **Time and space complexities of processing first t edges in the input stream.** $S := \min(t, bk) \leq L := \min(tk, bk)$.

Time Complexity

Methods	Master	Workers (Total)	Aggregator
CoCoS (both)	$O(tk)$*	$O(tS)$	$O(\min(tS, \lvert\mathcal{T}^{(t)}\rvert))$*
Tri-Fly		$O(tL)$	$O(\min(tL, \lvert\mathcal{T}^{(t)}\rvert \cdot k))$*

Space Complexity

Methods	Master	Workers (Total)	Aggregator
CoCoS$_{\text{SIMPLE}}$	$O(k)$	$O(S)$	$O(\lvert\mathcal{V}^{(t)}\rvert)$*
CoCoS$_{\text{OPT}}$	$O(\lvert\mathcal{V}^{(t)}\rvert + k))$	$O(S)$	
Tri-Fly	$O(k)$	$O(L)$	

*can be distributed across multiple masters or aggregators (see Section 5.3.5)

If we assume that there is not much positive correlation between random variables, then for each $z_i^{(t)}$ in Eq. (5.3),

$$\mathbb{E}_f[z_i^{(t)}] \approx \mathbb{E}_f[|\mathcal{T}_i^{(t)}|]\left(\frac{(\mathbb{E}_f[l_i^{(t)}]-1)(\mathbb{E}_f[l_i^{(t)}]-2)}{b(b-1)} - 1\right)$$
$$+ \left(\mathbb{E}_f[p_i^{(t)}] + \mathbb{E}_f[q_i^{(t)}]\right)\frac{\mathbb{E}_f[l_i^{(t)}]-1-b}{b} = O\left(\frac{|\mathcal{T}^{(t)}|t^2}{k^3b^2} + \frac{p^{(t)}t}{k^2b} + \frac{q^{(t)}t}{k^3b}\right).$$

Then, by Theorem 5.4, the expected variance of $\bar{c}^{(t)}$ is

$$\mathbb{E}_f[Var[\bar{c}^{(t)}]] \approx \boxed{O\left(\frac{|\mathcal{T}^{(t)}|t^2}{k^2b^2} + \frac{p^{(t)}t}{kb} + \frac{q^{(t)}t}{k^2b}\right),} \tag{5.5}$$

On the other hand, by Theorem 5.3, the variance of the estimate in Tri-Fly is

$$Var[\bar{c}^{(t)}] = O\left(\frac{|\mathcal{T}^{(t)}|t^2}{kb^2} + \frac{p^{(t)}t}{kb} + \frac{q^{(t)}t}{kb}\right). \tag{5.6}$$

Notice how rapidly the variances in CoCoS (Eq. (5.5)) and Tri-Fly (Eq. (5.6)) decrease depending on the number of workers (i.e., k). In Eq. (5.5), only the second term is $O(1/k)$ while the other terms are $O(1/k^2)$. In Eq. (5.6), however, all the terms are $O(1/k)$. This analysis gives an intuition why we can expect smaller variance of $\bar{c}^{(t)}$ in CoCoS than in Tri-Fly, especially when many workers are used. See Section 5.5.2 for empirical comparison of the variances.

5.4.2 Complexity Analysis

We discuss the time and space complexities of Tri-Fly, CoCoS$_{\text{SIMPLE}}$ (CoCoS with the simple modulo function as f) and CoCoS$_{\text{OPT}}$ (CoCoS with Algorithm 5.3 as f). We assume that sampled edges are stored in the adjacency list format in memory, as in our implementation used for our experiments.

5.4.2.1 Time Complexity Analysis

The time complexities of the considered algorithms for processing t edges in the input stream are summarized in Table 5.3. The master commonly takes $O(t \cdot k)$ since, in the worst case, every edge is broadcast.

The workers in TRI-FLY take $O(t \cdot \min(tk, bk))$ in total, while the workers in COCOS take only $O(t \cdot \min(t, bk))$ in total, as shown in Theorems 5.5 and 5.6, which are based on Lemma 5.4.

Lemma 5.4

Let $l_i^{(s)}$ be the load l_i of the i-th worker when $e^{(s)}$ arrives. If the i-th worker receives $e^{(s)}$, then it takes $O(\min(l_i^{(s)}, b))$ to process $e^{(s)}$ (i.e., to run lines 5-6 of Algorithm 5.1 and lines 6-8 of Algorithm 5.2).

Proof. The most expensive step of processing $e^{(s)} = \{u, v\}$ in both Algorithms 5.1 and 5.2 is to find the common neighbors of nodes u and v (line 9 of Algorithm 5.1). Computing $\mathcal{N}_i[u] \cap \mathcal{N}_i[v]$ requires accessing $|\mathcal{N}_i[u]| + |\mathcal{N}_i[v]| = O(|\mathcal{E}_i^{(s)}|) = O(\min(l_i^{(s)}, b))$ edges, where $\mathcal{E}_i^{(s)}$ is the set of edges stored in the i-th worker when $e^{(s)}$ arrives. ■

Theorem 5.5: Time Complexity of Workers in TRI-FLY

In Algorithm 5.1, the total time complexity of the workers for processing the first t edges in the input stream is $O(t \cdot \min(tk, bk))$.

Proof. From Lemma 5.4, processing an edge $e^{(s)}$ by the workers takes $O(\sum_{i=1}^{k} \min(l_i^{(s)}, b))$ in total. Thus, processing the first t edges takes $O\left(\sum_{s=1}^{t} \sum_{i=1}^{k} \min(l_i^{(s)}, b)\right)$. Since $l_i^{(s)} = s - 1$ in Algorithm 5.1,

$$\sum_{s=1}^{t} \sum_{i=1}^{k} \min(l_i^{(s)}, b) = \sum_{s=1}^{t} \sum_{i=1}^{k} \min(s-1, b) = \sum_{s=1}^{t} \min((s-1)k, bk) \leq t \cdot \min(tk, bk).$$

Hence, the workers take $O(t \cdot \min(tk, bk))$ in total to process the first t edges in the input stream. ■

Theorem 5.6: Time Complexity of Workers in COCOS

In Algorithm 5.2, the total time complexity of the workers for processing the first t edges in the input stream is $O(t \cdot \min(t, bk))$.

Proof. From Lemma 5.4, processing an edge $e^{(s)}$ by the workers takes $O(\sum_{i=1}^{k} \min(l_i^{(s)}, b))$ in total. Thus, processing the first t edges takes $O\left(\sum_{s=1}^{t} \sum_{i=1}^{k} \min(l_i^{(s)}, b)\right)$. Since each edge is assigned to at most two workers (i.e., P1 in Lemma 5.1), $\sum_{i=1}^{k} l_i^{(s)} \leq 2(s-1)$ holds, and it implies

$$\sum_{s=1}^{t} \sum_{i=1}^{k} \min(l_i^{(s)}, b) \leq \sum_{s=1}^{t} \min(\sum_{i=1}^{k} l_i^{(s)}, \sum_{i=1}^{k} b)) \leq \sum_{s=1}^{t} \min(2(s-1), bk)) \leq t \cdot \min(2t, bk).$$

Hence, the workers take $O(t \cdot \min(t, bk))$ in total to process the first t edges in the input stream. ■

The aggregator takes $O(|\mathcal{T}^{(t)}| \cdot k)$ in TRI-FLY since, in the worst case, each triangle is counted by every worker and thus the increases in counts by each triangle are sent to the aggregator k times. In COCOS$_{\text{SIMPLE}}$ and COCOS$_{\text{OPT}}$, however, the aggregator takes $O(\min(|\mathcal{T}^{(t)}|, t \cdot \min(t, bk)))$. Since the aggregator takes $O(1)$ for each update that it receives, its time complexity is proportional to the number of triangles counted by the workers. The number of counted triangles is $O(t \cdot \min(t, bk))$ by Theorem 5.6, and it is $O(|\mathcal{T}^{(t)}|)$ since each triangle is counted by at most one worker (i.e., P2 in Lemma 5.1). However, the computational cost of the aggregator can be easily distributed across multiple aggregators, as discussed in Section 5.3.5.

Notice that, with a fixed storage budget b, the time complexities of COCOS$_{\text{SIMPLE}}$ and COCOS$_{\text{OPT}}$ are linear in the number of edges in the input stream, as also shown empirically in Section 5.5.4.

5.4.2.2 Space Complexity Analysis

The space complexities of the considered algorithms for processing t edges in the input stream are summarized in Table 5.3. In TRI-FLY and COCOS$_{\text{SIMPLE}}$, the master requires $O(k)$ space to maintain the addresses of all the workers. In COCOS$_{\text{OPT}}$, the master requires additional $O(k + |\mathcal{V}^{(t)}|)$ space to store the loads of the workers and the mapping between the nodes and the workers (i.e., function f) while processing the first t edges in the input stream.

In all the algorithms, the workers require $O(\sum_{i=1}^{k} \min(l_i^{(t+1)}, b))$ space in total, to store sampled edges, where $l_i^{(t)}$ is the load l_i of the i-th worker when $e^{(t)}$ arrives. In TRI-FLY, since $l_i^{(t+1)} = t$, the space complexity of the workers is $O(\min(tk, bk))$ in total. In COCOS$_{\text{SIMPLE}}$ ad COCOS$_{\text{OPT}}$, since each edge is stored in at most two workers (i.e., P1 in Lemma 5.1), $\sum_{i=1}^{k} l_i^{(t+1)} \leq 2t$ holds, and it implies

$$\sum_{i=1}^{k} \min(l_i^{(t+1)}, b) \leq \min(\sum_{i=1}^{k} l_i^{(t+1)}, \sum_{i=1}^{k} b) \leq \min(2t, bk).$$

Hence, the total space complexity of the workers is $O(\min(t, bk))$.

In all the algorithms, the aggregator maintains one estimate of the global triangle count and $O(|\mathcal{V}^{(t)}|)$ estimates of the local triangle counts. However, this requirement can be easily distributed across multiple aggregators, as discussed in Section 5.3.5.

5.5 Experiments

We review our experiments for answering the following questions:

- **Q1. Illustration of Theorems**: Does COCOS give unbiased estimates? How do their variances scale with the number of workers?
- **Q2. Speed and Accuracy**: Is COCOS faster and more accurate than baselines?
- **Q3. Scalability**: Does COCOS scale linearly with the number of edges in the input stream?
- **Q4. Effects of Parameters**: How do the number of workers, storage budget, and parameter θ affect the accuracy of COCOS?

5.5.1 Experimental Settings

Machines: All experiments were conducted on a cluster of 40 machines with 3.47GHz Intel Xeon X5690 CPUs and 32GB RAM.

Table 5.4: **Summary of the graph streams used in our experiments.** B: billion, M: million, K: thousand.

Name	# Nodes	# Edges	Summary
Arxiv [GGK03]	34.5K	421K	Citation network
Facebook [VMCG09]	63.7K	817K	Friendship network
Google [LLDM09]	875K	4.32M	Web graph
BerkStan [LLDM09]	685K	6.65M	Web graph
Youtube [MMG+07]	3.22M	9.38M	Friendship network
Flickr [MMG+07]	2.30M	22.8M	Friendship network
LiveJournal [MMG+07]	4.00M	34.7M	Friendship network
Friendster [YL15]	65.6M	1.81B	Friendship network
Random (800GB)	1M	0.1B-100B	Synthetic graph

Datasets: We used the graphs listed in Table 5.4. We ignored all self loops, parallel edges, and directions of edges. We simulated graph streams by streaming the edges of the corresponding graph in a random order from the disk of the machine hosting the master.

Implementations: We implemented the following algorithms, which are the state-of-the-art algorithms for estimating both global and local triangle counts, commonly in C++ and MPICH 3.1:

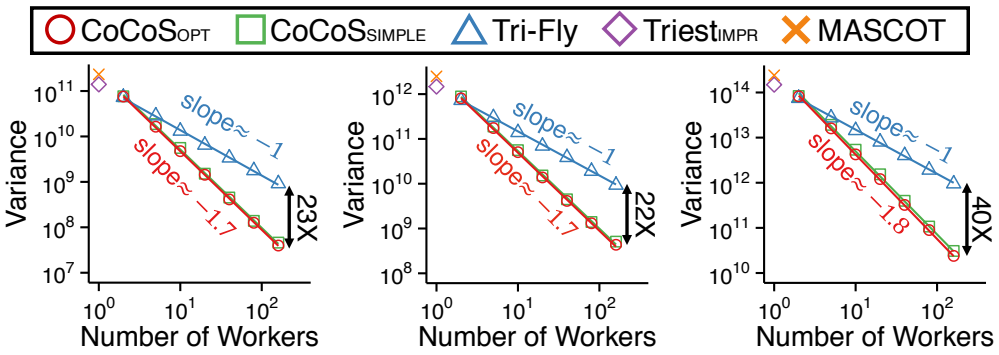
- **CoCoS$_{\text{SIMPLE}}$** (Section 5.3.3): proposed distributed streaming algorithms using the modulo function as the node mapping function f.
- **CoCoS$_{\text{OPT}}$** (Section 5.3.3.3): proposed distributed streaming algorithms using Algorithm 5.3 as the node mapping function f.
- **Tri-Fly** (Section 5.3.2): baseline distributed streaming algorithm.
- **Mascot** [LJK18] and **Triest$_{\text{IMPR}}$** [SERU17]: state-of-the-art single-machine streaming algorithms.

For the distributed algorithms, we used one master and one aggregator hosted by the same machine. Workers were hosted by different machines (unless their number was greater than that of machines). They used a part of the main memory of hosting machines as their local storage. In every algorithm, sampled edges were stored in the adjacency list format, and lazy aggregation, explained in Section 5.3.4, was used so that all estimates were aggregated once at the end of the input stream. We fixed θ in CoCoS$_{\text{OPT}}$ to 0.2, which gave the best accuracy (see Section 5.5.5).

Evaluation Metrics: We measured the accuracy of the considered algorithms using *global error*, *local error*, *RMSE*, and *rank correlation*, all of which are defined in Section 3.3.2.

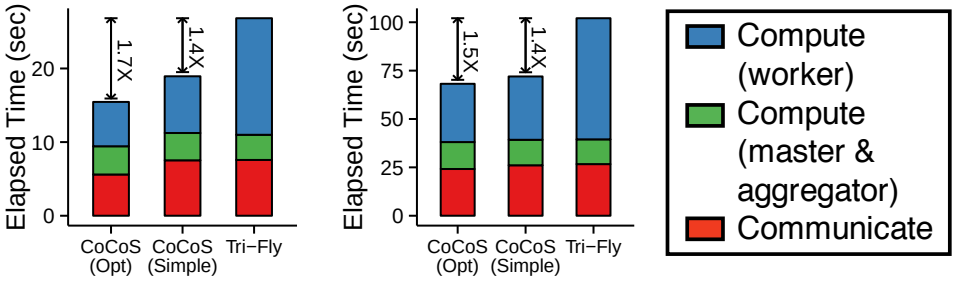
5.5.2 Q1. Illustration of Our Theorems

CoCoS gave unbiased estimates with small variances. Figure 5.1(c) in Section 5.1 illustrates Theorems 5.1 and 5.2, the unbiasedness of Tri-Fly and CoCoS. We obtained $10,000$ estimates of the global triangle count in the Google dataset using each distributed algorithm. We used 30 workers, and set b so that each worker stored up to 5% of the edges. As expected from Theorems 5.1 and 5.2, Tri-Fly, CoCoS$_{\text{OPT}}$, and CoCoS$_{\text{SIMPLE}}$ gave estimates whose averages were close to the true triangle



(a) Arxiv Dataset (b) Facebook Dataset (c) Google Dataset

Figure 5.4: **The variance of estimates drops faster in CoCoS$_{\text{OPT}}$ and CoCoS$_{\text{SIMPLE}}$ than in Tri-Fly**, as we use more workers.



(a) Youtube Dataset (b) LiveJournal Dataset

Figure 5.5: **CoCoS$_{\text{OPT}}$ reduces both computation and communication overhead**, compared to CoCoS$_{\text{SIMPLE}}$ and Tri-Fly. CoCoS$_{\text{OPT}}$ is also more accurate than the others, as seen in Figure 5.6.

count. The variance was the smallest in CoCoS$_{\text{OPT}}$, and the variance in CoCoS$_{\text{SIMPLE}}$ was smaller than that in Tri-Fly.

The variance in CoCoS dropped fast with the number of workers. Figure 5.4 illustrates Theorems 5.3 and 5.4, the variances of the estimates of the global triangle count in Tri-Fly and CoCoS. As we scaled up the number of workers, the variance decreased faster in CoCoS$_{\text{OPT}}$ and CoCoS$_{\text{SIMPLE}}$ ($\approx k^{-1.7}$) than in Tri-Fly ($\approx k^{-1}$), as expected in Eq. (5.5) and Eq. (5.6) in Section 5.4.1.2. In each setting, b was set to $1,000$, and the variance was estimated from $1,000$ trials.

5.5.3 Q2. Speed and Accuracy

We measured the speed and accuracy of the considered algorithms with different storage budgets.[4] We used 30 workers for each distributed streaming algorithm. To compare their speeds independently of the speed of the input stream, we measured the time taken by each algorithm to process edges, ignoring the time taken to wait for the arrival of edges in the input stream. In Figure 5.6, we report the evaluation metrics and elapsed times averaged over 10 trials in the Friendster dataset and over 100 trials in the other large datasets.

[4] $b = 5\%$ of the number of edges in each dataset in CoCoS$_{\text{SIMPLE}}$ and CoCoS$_{\text{OPT}}$. $b = \{2\%,5\%,20\%\}$ in Tri-Fly. $b = \{5\%,40\%\}$ in Triest$_{\text{IMPR}}$ and Mascot. See Section 5.5.5 for the effects of b values on the accuracies of the algorithms.

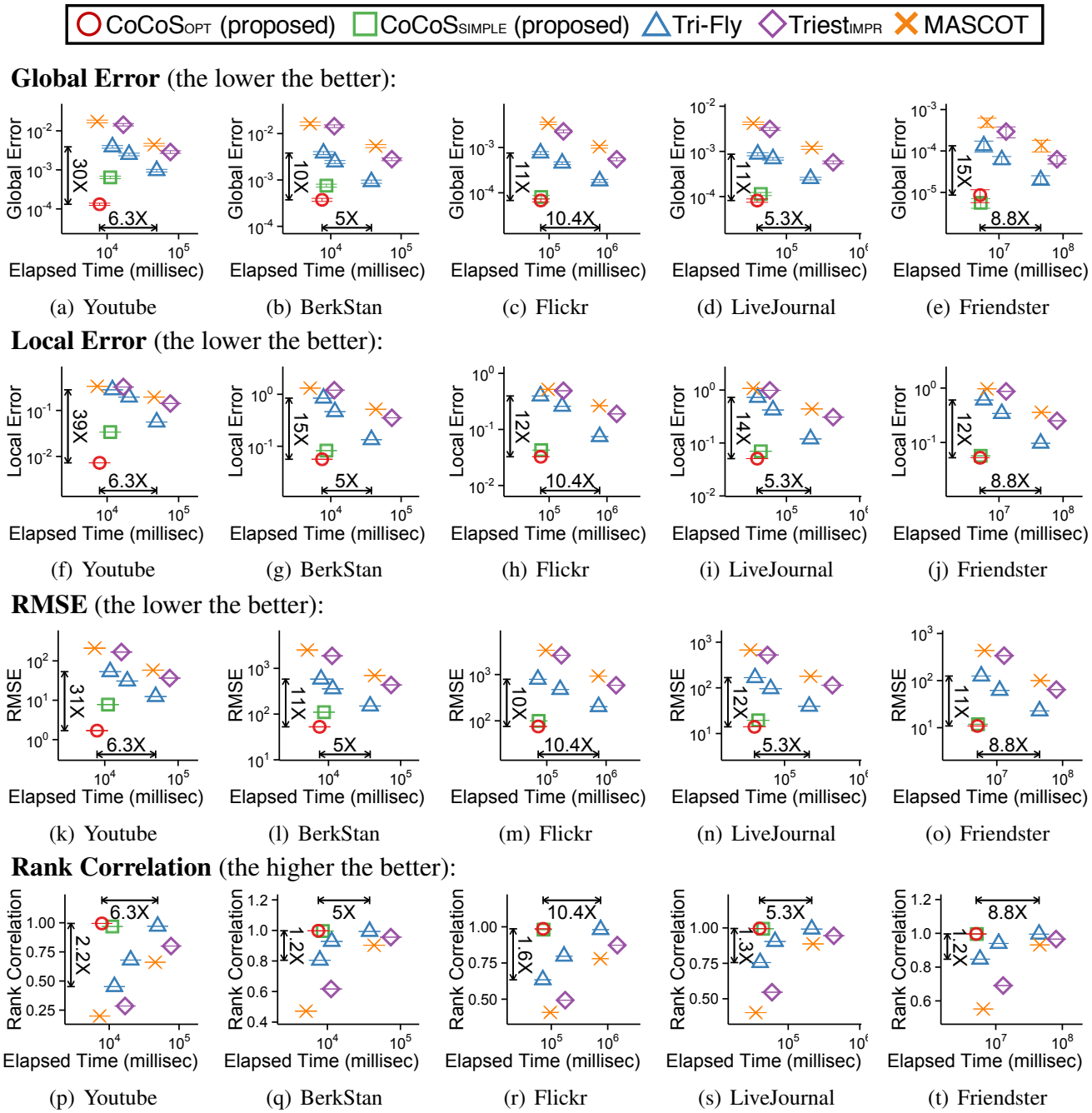
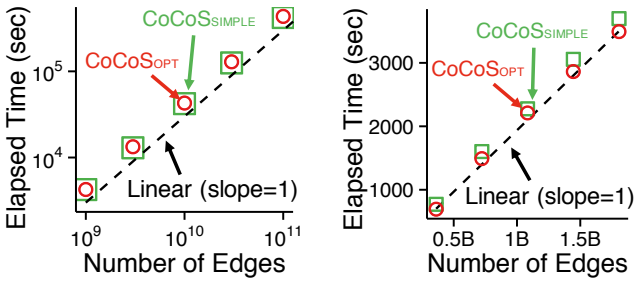


Figure 5.6: **CoCoS is fast and accurate.** CoCoS$_{\text{OPT}}$ (with θ fixed to 0.2) is up to 39× more accurate than the baselines with similar speeds, and it is up to 10.4× faster than the baselines while offering higher accuracy. Error bars show sample standard errors.



(a) Random Dataset (b) Friendster Dataset

Figure 5.7: **CoCoS$_{\text{OPT}}$ and CoCoS$_{\text{SIMPLE}}$ scale to terabyte-scale streams linearly with the size of the input stream.**

CoCoS gave the best trade-off between speed and accuracy. Specifically, CoCoS was up to $10.4\times$ *faster* than the baselines while giving more accurate estimates. Moreover, CoCoS was up to $30\times$ and $39\times$ *more accurate* than the baselines with similar speeds in terms of global error and local error, respectively. Between the proposed algorithms, CoCoS$_{\text{OPT}}$ was up to $1.4\times$ faster and $4.9\times$ more accurate than CoCoS$_{\text{SIMPLE}}$.

CoCoS$_{\text{OPT}}$ reduced computation and communication overhead. Figure 5.5 shows elapsed times for (a) computation in the master and aggregator, (b) computation in the slowest worker, and (c) communication between machines in CoCoS$_{\text{OPT}}$, CoCoS$_{\text{SIMPLE}}$, and TRI-FLY. The storage budget b was set to 5% of the number of edges in each dataset. CoCoS$_{\text{OPT}}$ reduced computation and communication costs, compared to CoCoS$_{\text{SIMPLE}}$ and TRI-FLY, as we expect in Section 5.3.3.3. Recall that CoCoS$_{\text{OPT}}$ was also more accurate than CoCoS$_{\text{SIMPLE}}$ and TRI-FLY.

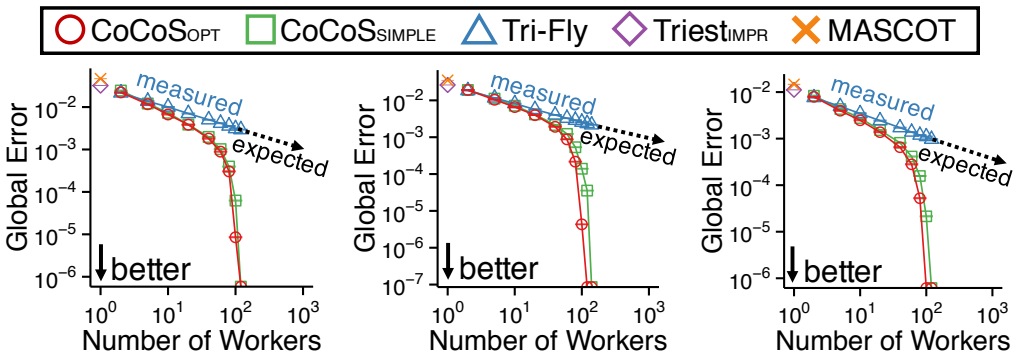
5.5.4 Q3. Scalability

We measured how the running times of CoCoS$_{\text{OPT}}$ and CoCoS$_{\text{SIMPLE}}$ scale with the number of edges in the input stream. We used 30 workers with b fixed to 10^7, and we measured their running times independently of the speed of the input stream, as in Section 5.5.3.

CoCoS scaled linearly and handled terabyte-scale graphs. Figure 5.7(a) shows the results in Erdős-Rényi random graph streams with 1 million nodes and different numbers of edges, and Figure 5.7(b) shows the results in graph streams with realistic structures created by sampling different numbers of edges from the Friendster dataset. Note that the largest stream has *100 billion edges*, which are *800GB*. CoCoS$_{\text{OPT}}$ and CoCoS$_{\text{SIMPLE}}$ scaled linearly with the size of the input stream, as we expect in Section 5.4.2.1.

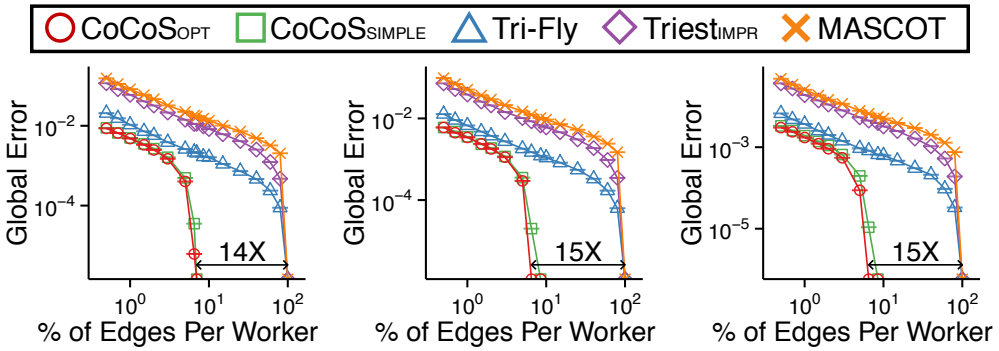
5.5.5 Q4. Effects of Parameters on Accuracy

We explored the effects of the parameters on the accuracies of the considered algorithms. As a default setting, we used 30 workers for the distributed streaming algorithms and set b to 2% of the number of edges for each dataset and θ to 0.2. When the effect of a parameter was analyzed, the others were fixed to their default values. We reported results with global error as the evaluation metric but obtained consistent results with the other metrics. We measured it $1,000$ times in each setting and reported the average. In Figures 5.8-5.10, the error bars denote sample standard errors.



(a) Arxiv Dataset (b) Facebook Dataset (c) Google Dataset

Figure 5.8: **Estimation error decreases faster in CoCoS than in Tri-Fly**, as we use more workers.



(a) Arxiv Dataset (b) Facebook Dataset (c) Google Dataset

Figure 5.9: **Estimation error decreases faster in CoCoS than in the baselines**, as we increase storage budget b. For exact estimation, CoCoS requires $14\times$ smaller b than the others.

As the workers were added, the estimation error dropped faster in CoCoS than in the baselines. As seen in Figure 5.8, the estimation errors of $\text{CoCoS}_{\text{OPT}}$ and $\text{CoCoS}_{\text{SIMPLE}}$ became zero with about 100 workers. However, that of Tri-Fly dropped slowly with expectation that it never becomes zero with a finite number of workers (see Theorem 5.3).

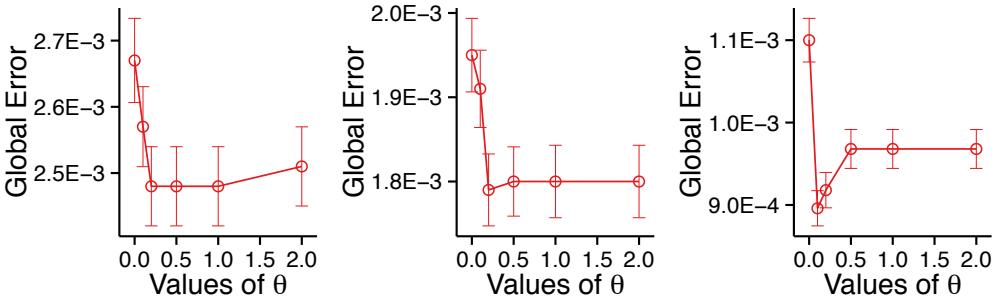
As storage budget increased, the estimation error dropped faster in CoCoS than in the baselines. As seen in Figure 5.9, the estimation errors of $\text{CoCoS}_{\text{OPT}}$ and $\text{CoCoS}_{\text{SIMPLE}}$ became 0 when each worker could store about 7% of the edges in each dataset. However, the estimation errors of the baselines became zero only when each worker could store all the edges in each dataset.

$\text{CoCoS}_{\text{OPT}}$ was most accurate when θ was around 0.2, as seen in Figure 5.10. The estimation error, however, was not very sensitive to the value of θ as long as θ was at least 0.2.

5.6 Summary

We propose CoCoS, a fast and accurate distributed streaming algorithm for the counts of global and local triangles. By minimizing the redundant use of distributed computational and storage resources (P1-P3 in Lemma 5.1), CoCoS offers the following advantages:

- **Accurate**: CoCoS is up to *39× more accurate* than its similarly fast competitors (Figure 5.6). It gives exact estimates within *14× smaller storage budgets* than its competitors (Figure 5.9).



(a) Arxiv Dataset (b) Facebook Dataset (c) Google Dataset

Figure 5.10: **Estimation error in CoCoS$_{\text{OPT}}$ is smallest when θ is around** 0.2**,** while the error is not very sensitive to θ.

- **Fast**: CoCoS is up to $10.4\times$ *faster* than its competitors while giving more accurate estimates (Figure 5.6). CoCoS scales linearly with the size of the input stream (Figure 5.7).
- **Theoretically Sound**: CoCoS gives unbiased estimates (Theorem 5.2). Their variances drop faster than its competitors' as the number of machines is scaled up (Theorem 5.4 and Figure 5.4).

Reproducibility: The source code and datasets used in this chapter are available at `http://www.cs.cmu.edu/~kijungs/codes/trifly/`.

5.7 Appendix: Proof of Lemma 5.3

Proof. For each triangle $\{u, v, w\} \in \mathcal{T}^{(t)}$ with $t_{vw} < t_{wu} < t_{uv} \leq t$, let $f(uvw) \in \{1, ..., k\}$ be the worker that can possibly count $\{u, v, w\}$. That is, $f(uvw) = f(w)$ if $f(u) \neq f(v)$, and $f(uvw) = f(u) = f(v)$ otherwise.

For the first claim, note that for each triangle $\{u, v, w\}$, each worker has the equal probability to be $f(uvw)$. Therefore,

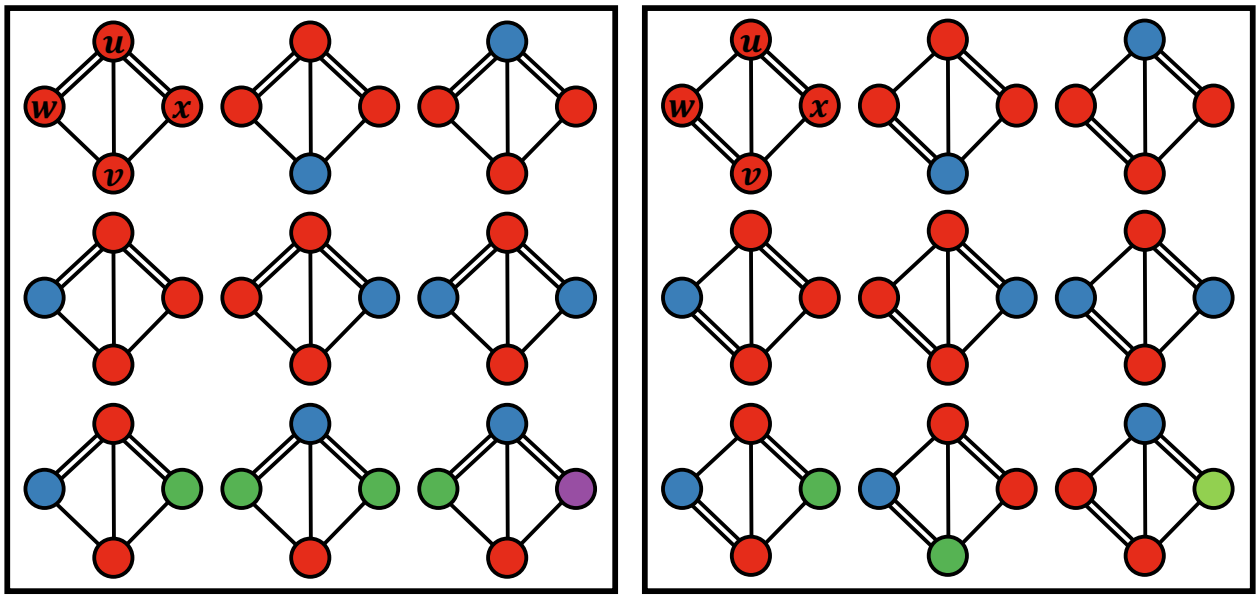
$$\mathbb{E}[|\mathcal{T}_i^{(t)}|] = \frac{|\mathcal{T}^{(t)}|}{k} = O(\frac{|\mathcal{T}^{(t)}|}{k}).$$

For the second claim, for each edge $\{u, v\}$, the probability that it is assigned to each i-th worker is equal to the probability that $f(u) = i$ or $f(v) = i$, which is $1 - \left(1 - \frac{1}{k}\right)^2 = \frac{2k-1}{k^2}$. Therefore,

$$\mathbb{E}[l_i^{(t)}] = \frac{(2k-1)t}{k^2} = O\left(\frac{t}{k}\right).$$

For the third claim, consider a Type 1 triangle pair $\{u, v, w\}$ and $\{u, v, x\}$. By considering $f : \mathcal{V} \to \{1, ..., k\}$ as a coloring of nodes $\mathcal{V}$ with k colors, Figure 5.11(a) represents all the nine ways where $f(uvw) = f(uvx)$. Note that $f(uvw) = f(uvx)$ is colored red in all of them. Fix a worker $i \in \{1, ..., k\}$. Then,

$$\begin{aligned} P[f(uvw) = f(uvx) = i] =& \frac{1}{k^4} + \frac{6}{k^3}(1 - \frac{1}{k}) + \frac{2}{k^2}(1 - \frac{1}{k})(1 - \frac{2}{k}) \\ &+ \frac{1}{k}(1 - \frac{1}{k})(1 - \frac{2}{k})(1 - \frac{3}{k}), \end{aligned}$$



(a) Type 1 Pair (b) Type 2 Pair

Figure 5.11: **Coloring of (a) Type 1 and (b) Type 2 triangle pairs where $\mathbf{f(uvw) = f(uvx)}$.** Nodes assigned to worker $f(uvw)$ $(= f(uvx))$ by f are colored red. Nodes with different colors are assigned to different workers by f.

where each term from left to right in the right hand side corresponds to the 1st case, 2nd-6th cases, 7th-8th cases, and 9th case, respectively, in Figure 5.11(a). Therefore,

$$\mathbb{E}[p_i^{(t)}] = P[f(uvw) = f(uvx) = i]p^{(t)} = \frac{k^3 - 4k^2 + 10k - 6}{k^4} p^{(t)} = O\left(\frac{p^{(t)}}{k}\right).$$

For the fourth claim, consider a Type 2 triangle pair $\{u, v, w\}$ and $\{u, v, x\}$. By considering $f : \mathcal{V} \to \{1, ..., k\}$ as a coloring of nodes $\mathcal{V}$ with k colors, Figure 5.11(b) represents all the nine ways where $f(uvw) = f(uvx)$. Note that $f(uvw) = f(uvx)$ is colored red in all of them. Fix a worker $i \in \{1, ..., k\}$. Then,

$$P[f(uvw) = f(uvx) = i] = \frac{1}{k^4} + \frac{1}{k^3}(1 - \frac{1}{k}) + \frac{1}{k^2}(1 - \frac{1}{k})(1 - \frac{2}{k}),$$

where each term from left to right in the right hand side corresponds to the 1st case, 2nd-6th cases, and 7th-9th cases, respectively, in Figure 5.11(b). Therefore,

$$\mathbb{E}[q_i^{(t)}] = P[f(uvw) = f(uvx) = i]q^{(t)} = \frac{3k^2 - 4k + 2}{k^4} q^{(t)} = O\left(\frac{q^{(t)}}{k^2}\right).$$

■

Chapter based on work that appeared at ECML/PKDD 2018 [SKHF18] and its extension [SKHF19].

Chapter 6

Counting Triangles in Graph Streams (3): Handling Deletions

Given a fully dynamic graph where edges are added and removed over time, how can we estimate the count of triangles in it? If we can store only a subset of the edges, how can we obtain unbiased estimates with small variances?

As discussed in the previous chapters, counting triangles (i.e., cliques of size three) in a graph is a classical problem with applications in a vast range of research areas, including social network analysis, data mining, and databases. Recently, streaming algorithms for triangle counting have been extensively studied since they can naturally be used for large dynamic graphs. However, existing algorithms cannot handle edge deletions or suffer from low accuracy.

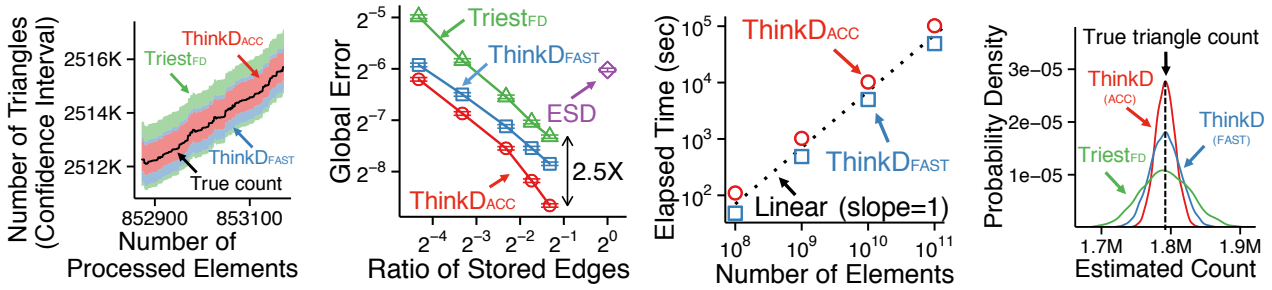
Can we handle edge deletions while achieving high accuracy? In this chapter, we propose THINKD, which accurately estimates the counts of global triangles (i.e., all triangles) and local triangles associated with each node in a fully dynamic graph stream with edge additions and deletions. Compared to its best competitors, THINKD is (a) *Accurate:* up to $\mathbf{4.3}\times$ **more accurate** within the same memory budget, (b) *Fast:* up to $\mathbf{2.2}\times$ **faster** for the same accuracy requirements, and (c) *Theoretically sound:* always maintaining *unbiased* estimates with small variances.

6.1 Motivation

Given a fully dynamic graph stream with edge additions and deletions, how can we accurately estimate the count of triangles in it with fixed memory size?

There has been great interest in graph stream algorithms, which gradually update their outputs as each edge insertion or deletion is received rather than operating on the entire graph at once. However, as discussed in Chapter 3, existing streaming algorithms for triangle counting focus on insertion-only streams [JSP13, PTTW13, TPT13, ADWR17, Shi17, LJK18, SHL+18] or greatly sacrifice accuracy to support edge deletions [KP14, SERU17, HS17].

In this last chapter on triangle counting, we propose THINKD (**Think** before you **D**iscard), an accurate streaming algorithm for triangle counting in a fully dynamic graph stream with both edge additions and deletions. THINKD maintains and updates estimates of the counts of global triangles (i.e., all triangles) and local triangles incident to each node. THINKD is named after the fact that, upon receiving each edge addition or deletion, THINKD uses it to improve its estimates even if the edge is



(a) 'Any time' for fully-dynamic graphs (b) Accurate (c) Scalable (Theorem 6.4) (d) Unbiased with small variance (Theorem 6.1)

Figure 6.1: **Strengths of THINKD.** (a) *'Any time':* THINKD always maintains the estimates of the global and local triangle counts while the input *fully-dynamic* graph evolves with edges insertions and deletions. (b) *Accurate:* THINKD is significantly more accurate than its best competitors. (c) *Scalable:* THINKD scales linearly with the number of elements in the input stream (Theorem 6.4). (d) *Unbiased:* THINKD gives unbiased estimates (Theorem 6.1). See Section 6.5 for details.

about to be discarded without being stored. This allows THINKD to achieve higher accuracy than if it were to only use edges in memory for estimation. As a result, THINKD has the following strengths:

- **Accurate**: THINKD gives up to $4\times$ and $4.3\times$ *smaller estimation errors* for global and local triangle counts, respectively, than its best competitors within the same memory budget (Figure 6.1(b)).
- **Fast**: THINKD *scales linearly* with the size of the input stream (Figure 6.1(c), Corollary 6.1, and Theorem 6.4). THINKD is also up to $2.2\times$ *faster* than its best competitors with similar accuracies.
- **Theoretically Sound**: We prove the formulas for the bias and variance of the estimates given by THINKD (Theorems 6.1 and 6.2). In particular, we show that THINKD always maintains *unbiased* estimates with small variances (Figures 6.1(a) and 6.1(d)).

Reproducibility: The source code and datasets used in this chapter are available at `http://www.cs.cmu.edu/~kijungs/thinkd/`.

The rest of this chapter is organized as follows. In Section 6.2, we introduce some preliminary concepts, notations, and a formal problem definition. In Section 6.3, we present THINKD, our proposed algorithm for triangle counting. In Section 6.4, we theoretically analyze the accuracy and complexity of THINKD. After sharing some experimental results in Section 6.5, we provide a summary of this chapter in Section 6.6.

6.2 Preliminaries and Problem Definition

In this section, we first introduce some notations and concepts used throughout this chapter. Then, we define the problem of global and local triangle counting in a fully dynamic graph stream.

6.2.1 Notations and Concepts

Table 6.1 lists the symbols frequently used in the chapter. Consider an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with nodes $\mathcal{V}$ and edges $\mathcal{E}$. Each edge $\{u, v\} \in \mathcal{E}$ connects two distinct nodes $u \neq v \in \mathcal{V}$. We say a subset $\{u, v, w\} \subset \mathcal{V}$ of size 3 is a *triangle* if every pair of distinct nodes u, v, and w is connected by

Table 6.1: **Table of frequently-used symbols.**

Symbol	Definition
Notations for Fully Dynamic Graph Streams (Section 6.2)	
$e^{(t)} = (\{u, v\}, \delta)$	change in the input graph $\mathcal{G}$ at time t
$\mathcal{G}^{(t)} = (\mathcal{V}^{(t)}, \mathcal{E}^{(t)})$	graph $\mathcal{G}$ at time t
$\{u, v\}$	edge between nodes u and v
$\{u, v, w\}$	triangle with nodes u, v, and w
$\mathcal{T}^{(t)}$	set of global triangles in $\mathcal{G}^{(t)}$
$\mathcal{T}^{(t)}[u]$	set of local triangles of node u in $\mathcal{G}^{(t)}$
Notations for Algorithms (Sections 6.3)	
$\mathcal{S}$	set of sampled edges
$\hat{\mathcal{N}}_u$	set of neighbors of node u in $\mathcal{S}$
$\bar{c}$	estimate of the count of global triangles
$c[u]$	estimate of the count of local triangles of node u
r	sampling probability in THINKD$_{\text{FAST}}$
b	maximum number of sampled edges in THINKD$_{\text{ACC}}$
Notations for Analysis (Sections 6.4)	
$\mathcal{A}^{(t)}$	set of added triangles at time t
$\mathcal{D}^{(t)}$	set of deleted triangles at time t

an edge in $\mathcal{E}$. We denote the set of *global triangles* (i.e., all triangles) in $\mathcal{G}$ by $\mathcal{T}$ and the set of *local triangles* of each node $u \in \mathcal{V}$ (i.e., all triangles containing u) by $\mathcal{T}[u] \subset \mathcal{T}$.

Assume the graph $\mathcal{G}$ evolves from the empty graph. We consider the *fully dynamic graph stream* representing the sequence of changes in $\mathcal{G}$, and denote the stream by $(e^{(1)}, e^{(2)}, ...)$. For each $t \in \{1, 2, ...\}$, the pair $e^{(t)} = (\{u, v\}, \delta)$ of an edge $\{u, v\}$ and a sign $\delta \in \{+, -\}$ denotes the change in $\mathcal{G}$ at time t. Specifically, $(\{u, v\}, +)$ indicates the addition of a new edge $\{u, v\} \notin \mathcal{E}$, and $(\{u, v\}, -)$ indicates the deletion of an existing edge $\{u, v\} \in \mathcal{E}$. We use $\mathcal{G}^{(t)} = (\mathcal{V}^{(t)}, \mathcal{E}^{(t)})$ to indicate $\mathcal{G}$ at time t. That is,

$$\mathcal{E}^{(0)} = \emptyset \qquad \text{and} \qquad \mathcal{E}^{(t)} = \begin{cases} \mathcal{E}^{(t-1)} \cup \{\{u, v\}\}, & \text{if } e^{(t)} = (\{u, v\}, +), \\ \mathcal{E}^{(t-1)} \setminus \{\{u, v\}\}, & \text{if } e^{(t)} = (\{u, v\}, -). \end{cases}$$

Lastly, we let $\mathcal{T}^{(t)}$ denote the set of global triangles in $\mathcal{G}^{(t)}$ and $\mathcal{T}^{(t)}[u] \subset \mathcal{T}^{(t)}$ denote the set of local triangles of each node $u \in \mathcal{V}^{(t)}$ in $\mathcal{G}^{(t)}$.

6.2.2 Problem Definition

In this chapter, we address the problem of estimating the counts of global and local triangles in a fully dynamic graph stream. We assume the standard data stream model where the elements in the input stream, which may not fit in memory, can be accessed once in the given order unless they are explicitly stored in memory.

Problem 6.1: Global and Local Triangle Counting in a Fully Dynamic Graph Stream

1. **Given:** a fully dynamic graph stream $(e^{(1)}, e^{(2)}, ...)$
   (i.e., sequence of edge additions and deletions in graph $\mathcal{G}$)
2. **Maintain:** estimates of the global triangle count $|\mathcal{T}^{(t)}|$ and the local triangle counts $\{(u, |\mathcal{T}^{(t)}[u]|)\}_{u \in \mathcal{V}^{(t)}}$ for current $t \in \{1, 2, ...\}$
3. **to Minimize:** the estimation errors.

We follow a general approach of reducing the biases and variances of estimates simultaneously rather than minimizing a specific measure of estimation error.

6.3 Proposed Algorithm: THINKD

In this section, we propose THINKD (**Think** before you **D**iscard), a fast and accurate algorithm for estimating the counts of global and local triangles in a fully dynamic graph stream. We first provide an overview and then describe two versions of THINKD with distinct advantages in detail.

6.3.1 Overview

For estimation with limited memory, THINKD samples edges and maintains those sampled edges, while discarding the other edges. The main idea of THINKD is to fully utilize unsampled edges before they are discarded. Specifically, whenever each element in the input stream arrives, THINKD first updates its estimates using the element. After that, if the element is an addition of an edge, THINKD decides whether to sample the edge or not.

In the following subsections, we present two versions of THINKD: THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$. To this end, we use $\bar{c}$ to denote the maintained estimate of the count of global triangles. Likewise, for each node u, we use $c[u]$ to denote the maintained estimate of the count of local triangles of node u. In addition, we let $\mathcal{S}$ be the set of currently sampled edges, and for each node u, we let $\hat{\mathcal{N}}_u$ be the set of neighbors of u in the graph composed of the edges in $\mathcal{S}$.

6.3.2 Simple and Fast Version: THINKD$_{\text{FAST}}$

THINKD$_{\text{FAST}}$, which is a simple and fast version of THINKD, is described in Algorithm 6.1. THINKD$_{\text{FAST}}$ initially has no sampled edges (line 1). Whenever each element $(\{u, v\}, \delta)$ of the input stream arrives (line 2), THINKD$_{\text{FAST}}$ first updates its estimates by calling the procedure UPDATE (line 3). Then, if the element is an addition (i.e., $\delta = +$), THINKD$_{\text{FAST}}$ samples the edge $\{u, v\}$ with a given sampling probability r (line 11) by calling the procedure INSERT (line 4). If the element is a deletion (i.e., $\delta = -$), THINKD$_{\text{FAST}}$ removes the edge $\{u, v\}$ from the existing samples (line 13) by calling the procedure DELETE (line 5).

In the procedure UPDATE, THINKD$_{\text{FAST}}$ finds the triangles connected by the arrived edge $\{u, v\}$ and two edges from the existing samples $\mathcal{S}$ (line 7). To this end, THINKD uses the fact that each common neighbor w of the nodes u and v in the graph composed of the sampled edges in $\mathcal{S}$ indicates the existence of such a triangle $\{u, v, w\}$. In the case of additions (i.e., $\delta = +$), since such triangles are new triangles added to the input stream, THINKD$_{\text{FAST}}$ increases the estimates of the global count and the corresponding local counts (line 8). In the case of deletions (i.e., $\delta = -$), since such triangles are

Algorithm 6.1 THINKD$_{\text{FAST}}$: Simple and Fast Version of THINKD

Input: (1) fully dynamic graph stream: $(e^{(1)}, e^{(2)}, ...)$,
(2) sampling probability: r
Output: (1) estimated global triangle count: $\bar{c}$,
(2) estimated local triangle counts: $c[u]$ for each node u

1: $\mathcal{S} \leftarrow \emptyset$
2: **for each** element $e^{(t)} = (\{u, v\}, \delta)$ in the input stream **do**
3: UPDATE$(\{u, v\}, \delta)$
4: **if** $\delta = +$ **then** INSERT$(\{u, v\})$
5: **else if** $\delta = -$ **then** DELETE$(\{u, v\})$
6: **procedure** UPDATE$(\{u, v\}, \delta)$
7: **for each** common neighbor $w \in \hat{\mathcal{N}}_u \cap \hat{\mathcal{N}}_v$ **do**
8: **if** $\delta = +$ **then** increase $\bar{c}$, $c[u]$, $c[v]$, and $c[w]$ by $1/r^2$
9: **else if** $\delta = -$ **then** decrease $\bar{c}$, $c[u]$, $c[v]$, and $c[w]$ by $1/r^2$
10: **procedure** INSERT$(\{u, v\})$
11: **if** a random number in Bernoulli(r) is 1 **then** $\mathcal{S} \leftarrow \mathcal{S} \cup \{\{u, v\}\}$
12: **procedure** DELETE$(\{u, v\})$
13: **if** $\{u, v\} \in \mathcal{S}$ **then** $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\{u, v\}\}$

those removed from the input stream, THINKD$_{\text{FAST}}$ decreases the estimates of the global count and the corresponding local counts (line 9). Notice that the amount of change per triangle is $1/r^2$, which is the reciprocal of the probability that each added or deleted triangle is discovered by THINKD$_{\text{FAST}}$. Note that each such triangle $\{u, v, w\}$ is discovered if and only if $\{v, w\}$ and $\{w, u\}$ are in $\mathcal{S}$, whose probability is r^2, as formalized in Lemma 6.1. This makes the expected amount of changes in the corresponding estimates for each such triangle be exactly one and thus makes THINKD$_{\text{FAST}}$ give unbiased estimates, as explained in detail in Section 6.4.1.

Lemma 6.1: Discovery Probability of Triangles in THINKD$_{\text{FAST}}$

In THINKD$_{\text{FAST}}$, any two distinct edges in graph $\mathcal{G}^{(t)} = (\mathcal{V}^{(t)}, \mathcal{E}^{(t)})$ are sampled with probability r^2. That is, if we let $\mathcal{S}^{(t)}$ be $\mathcal{S}$ in Algorithm 6.1 after the t-th element $e^{(t)}$ is processed, then

$$Pr[\{u, v\} \in \mathcal{S}^{(t)} \cap \{w, x\} \in \mathcal{S}^{(t)}] = r^2, \ \forall t \geq 1, \ \forall \{u, v\} \neq \{w, x\} \in \mathcal{E}^{(t)}. \tag{6.1}$$

Proof. Eq. (6.1) holds since each edge is sampled independently with probability r. See Section 6.7.1 for a formal proof. ■

Advantages and disadvantages of THINKD$_{\text{FAST}}$: Due to its simplicity, THINKD$_{\text{FAST}}$ is faster than its competitors, as shown empirically in Section 6.5.4. However, it is less accurate than THINKD$_{\text{ACC}}$, described in the following subsection, since it may discard edges even when memory is not full, leading to avoidable loss of information.

Algorithm 6.2 THINKD$_{\text{ACC}}$: Accurate Version of THINKD

Input: (1) fully dynamic graph stream: $(e^{(1)}, e^{(2)}, ...)$,
(2) memory budget: b (≥ 2)
Output: (1) estimated global triangle count: $\bar{c}$,
(2) estimated local triangle counts: $c[u]$ for each node u

1: $\mathcal{S} \leftarrow \emptyset$
2: $|\mathcal{E}| \leftarrow 0$, $n_b \leftarrow 0$, $n_g \leftarrow 0$
3: **for each** element $e^{(t)} = (\{u, v\}, \delta)$ in the input stream **do**
4: UPDATE($\{u, v\}, \delta$)
5: **if** $\delta = +$ **then** INSERT($\{u, v\}$)
6: **else if** $\delta = -$ **then** DELETE($\{u, v\}$)
7: **procedure** UPDATE($\{u, v\}, \delta$)
8: **for each** common neighbor $w \in \hat{\mathcal{N}}_u \cap \hat{\mathcal{N}}_v$ **do**
9: **if** $\delta = +$ **then** increase $\bar{c}$, $c[u]$, $c[v]$, and $c[w]$ by $1/p(|\mathcal{E}|, n_b, n_g)$
10: **else if** $\delta = -$ **then** decrease $\bar{c}$, $c[u]$, $c[v]$, and $c[w]$ by $1/p(|\mathcal{E}|, n_b, n_g)$
11: **procedure** INSERT({u,v})
12: $|\mathcal{E}| \leftarrow |\mathcal{E}| + 1$
13: **if** $n_b + n_g = 0$ **then**
14: **if** $|\mathcal{S}| < b$ **then**
15: $\mathcal{S} \leftarrow \mathcal{S} \cup \{\{u, v\}\}$
16: **else if** a random number in Bernoulli$(b/|\mathcal{E}|)$ is 1 **then**
17: replace an edge chosen at random uniformly in $\mathcal{S}$ with $\{u, v\}$
18: **else if** a random number in Bernoulli$(n_b/(n_b + n_g))$ is 1 **then**
19: $\mathcal{S} \leftarrow \mathcal{S} \cup \{\{u, v\}\}$, $n_b \leftarrow n_b - 1$
else $n_g \leftarrow n_g - 1$
20: **procedure** DELETE({u,v})
21: $|\mathcal{E}| \leftarrow |\mathcal{E}| - 1$
22: **if** $\{u, v\} \in \mathcal{S}$ **then** $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\{u, v\}\}$, $n_b \leftarrow n_b + 1$
23: **else** $n_g \leftarrow n_g + 1$

6.3.3 Accurate Version: THINKD$_{\text{ACC}}$

THINKD$_{\text{ACC}}$, which is an accurate version of THINKD, is described in Algorithm 6.2. Different from THINKD$_{\text{FAST}}$, which may discard edges even when memory is not full, THINKD$_{\text{ACC}}$ maintains as many samples as possible within a given memory budget b (≥ 2) to minimize information loss.

To this end, THINKD$_{\text{ACC}}$ uses a sampling method called Random Pairing (RP) [GLH08]. Given a fully dynamic stream with deletions, and a memory budget b, RP maintains at most b samples while satisfying the uniformity of the samples. Specifically, if we let $\mathcal{E}$ be the set of edges remaining (without being deleted) in the input stream so far and $\mathcal{S} \subset \mathcal{E}$ be the set of samples being maintained by RP, then the following equations hold:

$$|\mathcal{S}| \leq k \qquad \text{and} \qquad Pr[\mathcal{S} = \mathcal{A}] = Pr[\mathcal{S} = \mathcal{B}],\ \forall \mathcal{A} \neq \mathcal{B} \subset \mathcal{E} \text{ s.t. } |\mathcal{A}| = |\mathcal{B}|.$$

Updating the set $\mathcal{S}$ of samples using RP is described in lines 11-23. Whenever a deletion of an edge arrives, RP increases n_b or n_g depending on whether the edge is in $\mathcal{S}$ or not (lines 22 and 23). Roughly speaking, n_b and n_g denote the number of deletions that need to be "compensated" by additions

(lines 18-19). If there is no deletion to compensate, RP processes each addition of an edge as in Reservoir Sampling [Vit85]. That is, if memory is not full (i.e., $|\mathcal{S}| < b$), RP adds the new edge to $\mathcal{S}$ (lines 14 and 15), while otherwise, RP replaces an edge chosen at random uniformly in $\mathcal{S}$ with the new edge with a certain probability (lines 16-17). We refer to [GLH08] for the intuition behind the compensation and the details of RP; and we focus on how to use RP for triangle counting in the rest of this section.

Updating the estimates in THINKD$_{\text{ACC}}$ is the same as that in THINKD$_{\text{FAST}}$ except for the amount of change per triangle (lines 9 and 10), which is the reciprocal of the probability that each added or deleted triangle is discovered. When each element $e^{(t)} = (\{u, v\}, \delta)$ arrives, each added or deleted triangle $\{u, v, w\}$ is discovered if and only if $\{v, w\}$ and $\{w, u\}$ are in $\mathcal{S}$. As shown in Lemma 6.2, if we let $y = \min(b, |\mathcal{E}| + n_b + n_g)$, then the probability of such an event is

$$p(|\mathcal{E}|, n_b, n_g) := \frac{y}{|\mathcal{E}| + n_b + n_g} \times \frac{y - 1}{|\mathcal{E}| + n_b + n_g - 1}. \tag{6.2}$$

Lemma 6.2: Discovery Probability of Triangles in THINKD$_{\text{ACC}}$

In THINKD$_{\text{ACC}}$, any two distinct edges in graph $\mathcal{G}^{(t)} = (\mathcal{V}^{(t)}, \mathcal{E}^{(t)})$ are sampled with probability as in Eq. (6.2). That is, if we let $p^{(t)}$ and $\mathcal{S}^{(t)}$ be the values of Eq. (6.2) and $\mathcal{S}$, respectively, in Algorithm 6.2 after the t-th element $e^{(t)}$ is processed, then

$$Pr[\{u, v\} \in \mathcal{S}^{(t)} \cap \{w, x\} \in \mathcal{S}^{(t)}] = p^{(t)}, \ \forall t \geq 1, \ \forall \{u, v\} \neq \{w, x\} \in \mathcal{E}^{(t)}. \tag{6.3}$$

Proof. See Section 6.7.2 for a proof. ■

Advantages and disadvantages of THINKD$_{\text{ACC}}$: Within the same memory budget, THINKD$_{\text{ACC}}$ is slower than THINKD$_{\text{FAST}}$ since THINKD$_{\text{ACC}}$ maintains and processes more samples on average. However, THINKD$_{\text{ACC}}$ is more accurate than THINKD$_{\text{FAST}}$ by utilizing more samples. These are shown empirically in Sections 6.5.3 and 6.5.4.

Reducing estimation errors by sacrificing unbiasedness: The estimates (i.e., $\bar{c}$ and $c[u]$ for each node u) in Algorithms 6.1 and 6.2 can have negative values. Since true triangle counts are always non-negative, lower bounding the estimates by zero always reduces the estimation errors. However, the estimates become biased, and Theorem 6.1 in the following section does not hold anymore.

6.4 Theoretical Analysis

We theoretically analyze the accuracy, time complexity, and space complexity of THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$.

6.4.1 Accuracy Analysis

We prove that THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$ maintain unbiased estimates with the expected values equal to the true global and local triangle counts. Then, we analyze the variances of the estimates

that THINKD$_{\text{FAST}}$ maintains. To this end, for each variable (e.g., $\bar{c}$) in Algorithms 6.1 and 6.2, we use superscript (t) (e.g., $\bar{c}^{(t)}$) to denote the value of the variable after the t-th element $e^{(t)}$ is processed.

We first define *added triangles* and *deleted triangles* in Definitions 6.1 and 6.2. Note that triangles composed of the same nodes can be added multiple times (and thus can be removed multiple times) only if deleted edges are added again.

Definition 6.1: Added Triangles

Let $\mathcal{A}^{(t)}$ be the set of triangles that have been added to graph $\mathcal{G}$ at time t or earlier. Formally,

$$\mathcal{A}^{(t)} := \{(\{u, v, w\}, s) : 1 \leq s \leq t \text{ and } \{u, v, w\} \notin \mathcal{T}^{(s-1)} \text{ and } \{u, v, w\} \in \mathcal{T}^{(s)}\},$$

where addition time s is for distinguishing triangles composed of the same nodes but added at different times.

Definition 6.2: Deleted Triangles

Let $\mathcal{D}^{(t)}$ be the set of triangles that have been removed from graph $\mathcal{G}$ at time t or earlier. Formally,

$$\mathcal{D}^{(t)} := \{(\{u, v, w\}, s) : 1 \leq s \leq t \text{ and } \{u, v, w\} \in \mathcal{T}^{(s-1)} \text{ and } \{u, v, w\} \notin \mathcal{T}^{(s)}\},$$

where deletion time s is for distinguishing triangles composed of the same nodes but deleted at different times.

Similarly, for each node $u \in \mathcal{V}^{(t)}$, we use $\mathcal{A}^{(t)}[u] \subset \mathcal{A}^{(t)}$ and $\mathcal{D}^{(t)}[u] \subset \mathcal{D}^{(t)}$ to denote the added and deleted triangles with node u, respectively. Lemma 6.3 formalizes the relationship between these concepts and the number of triangles.

Lemma 6.3: Count of Triangles in the Current Graph

The count of triangles in the current graph equals to the count of added triangles subtracted by the count of deleted triangles. Formally,

$$|\mathcal{T}^{(t)}| = |\mathcal{A}^{(t)}| - |\mathcal{D}^{(t)}|, \ \forall t \in \{1, 2, ...\}, \tag{6.4}$$

$$|\mathcal{T}^{(t)}[u]| = |\mathcal{A}^{(t)}[u]| - |\mathcal{D}^{(t)}[u]|, \ \forall t \in \{1, 2, ...\}, \ \forall u \in \mathcal{V}^{(t)}. \tag{6.5}$$

Proof. Eq. (6.4) and Eq. (6.5) follow from Definitions 6.1 and 6.2. See Section 6.7.3 for a formal proof. ■

Based on these concepts, we prove that THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$ maintain unbiased estimates in Theorem 6.1. For the unbiasedness of the estimate $\bar{c}$ of the global count, we show that the expected amount of change in $\bar{c}$ for each added triangle is $+1$, while that for each deleted triangle is -1. Then, by Lemma 6.3, the expected value of $\bar{c}$ equals to the true global count. Likewise, we show the unbiasedness

of the estimate of the local triangle count of each node by considering only the added and deleted triangles incident to the node.

Theorem 6.1: 'Any Time' Unbiasedness of THINKD

THINKD gives unbiased estimates at any time. Formally, in Algorithms 6.1 and 6.2,

$$\mathbb{E}[\bar{c}^{(t)}] = |\mathcal{T}^{(t)}|, \ \forall t \in \{1, 2, ...\}, \tag{6.6}$$

$$\mathbb{E}[c^{(t)}[u]] = |\mathcal{T}^{(t)}[u]|, \ \forall t \in \{1, 2, ...\}, \ \forall u \in \mathcal{V}^{(t)}. \tag{6.7}$$

Proof. Consider a triangle $(\{u, v, w\}, s) \in \mathcal{A}^{(t)}$, and let $e^{(s)} = (\{u, v\}, +)$ without loss of generality. The amount $\alpha^{(s)}_{uvw}$ of change in each of $\bar{c}$, $c[u]$, $c[v]$, and $c[w]$ due to the discovery of $(\{u, v, w\}, s)$ in line 8 of Algorithm 6.1 or Algorithm 6.2 is

$$\alpha^{(s)}_{uvw} = \begin{cases} 1/r^2 & if \{w, u\} \in \mathcal{S}^{(s-1)} \text{ and } \{v, w\} \in \mathcal{S}^{(s-1)} \text{ in Algorithm 6.1} \\ 1/p^{(s-1)} & if \{w, u\} \in \mathcal{S}^{(s-1)} \text{ and } \{v, w\} \in \mathcal{S}^{(s-1)} \text{ in Algorithm 6.2} \\ 0 & otherwise. \end{cases}$$

Then, from Eq. (6.1) and Eq. (6.3), the following equation holds:

$$\alpha^{(s)}_{uvw} = \begin{cases} \frac{1}{Pr[\{w,u\} \in \mathcal{S}^{(s-1)} \cap \{v,w\} \in \mathcal{S}^{(s-1)}]} & if \{w, u\} \in \mathcal{S}^{(s-1)} \text{ and } \{v, w\} \in \mathcal{S}^{(s-1)} \\ 0 & otherwise. \end{cases}$$

Hence,

$$\mathbb{E}[\alpha^{(s)}_{uvw}] = 1. \tag{6.8}$$

Consider a triangle $(\{u, v, w\}, s) \in \mathcal{D}^{(t)}$, and let $e^{(s)} = (\{u, v\}, -)$ without loss of generality. The amount $\beta^{(s)}_{uvw}$ of change in each of $\bar{c}$, $c[u]$, $c[v]$, and $c[w]$ due to the discovery of $(\{u, v, w\}, s)$ in line 9 of Algorithm 6.1 or Algorithm 6.2 is

$$\beta^{(s)}_{uvw} = \begin{cases} -1/r^2 & if \{w, u\} \in \mathcal{S}^{(s-1)} \text{ and } \{v, w\} \in \mathcal{S}^{(s-1)} \text{ in Algorithm 6.1} \\ -1/p^{(s-1)} & if \{w, u\} \in \mathcal{S}^{(s-1)} \text{ and } \{v, w\} \in \mathcal{S}^{(s-1)} \text{ in Algorithm 6.2} \\ 0 & otherwise. \end{cases}$$

Then, from Eq. (6.1) and Eq. (6.3), the following equation holds:

$$\beta^{(s)}_{uvw} = \begin{cases} \frac{-1}{Pr[\{w,u\} \in \mathcal{S}^{(s-1)} \cap \{v,w\} \in \mathcal{S}^{(s-1)}]} & if \{w, u\} \in \mathcal{S}^{(s-1)} \text{ and } \{v, w\} \in \mathcal{S}^{(s-1)} \\ 0 & otherwise. \end{cases}$$

Hence,

$$\mathbb{E}[\beta^{(s)}_{uvw}] = -1. \tag{6.9}$$

By definition, the following holds:

$$\bar{c}^{(t)} = \sum_{(\{u,v,w\},s) \in \mathcal{A}^{(t)}} \alpha^{(s)}_{uvw} + \sum_{(\{u,v,w\},s) \in \mathcal{D}^{(t)}} \beta^{(s)}_{uvw}.$$

By linearity of expectation, Eq. (6.8), Eq. (6.9), and Lemma 6.3, the following holds:

$$\begin{aligned}\mathbb{E}[\bar{c}^{(t)}] &= \sum_{(\{u,v,w\},s)\in\mathcal{A}^{(t)}} \mathbb{E}[\alpha^{(s)}_{uvw}] + \sum_{(\{u,v,w\},s)\in\mathcal{D}^{(t)}} \mathbb{E}[\beta^{(s)}_{uvw}] \\ &= \sum_{(\{u,v,w\},s)\in\mathcal{A}^{(t)}} 1 + \sum_{(\{u,v,w\},s)\in\mathcal{D}^{(t)}} (-1) = |\mathcal{A}^{(t)}| - |\mathcal{D}^{(t)}| = |\mathcal{T}^{(t)}|.\end{aligned}$$

Likewise, for each node $u \in \mathcal{V}^{(t)}$, the following holds:

$$c^{(t)}[u] = \sum_{(\{u,v,w\},s)\in\mathcal{A}^{(t)}[u]} \alpha^{(s)}_{uvw} + \sum_{(\{u,v,w\},s)\in\mathcal{D}^{(t)}[u]} \beta^{(s)}_{uvw}.$$

By linearity of expectation, Eq. (6.8), Eq. (6.9), and Lemma 6.3, the following holds:

$$\begin{aligned}\mathbb{E}[c^{(t)}[u]] &= \sum_{(\{u,v,w\},s)\in\mathcal{A}^{(t)}[u]} \mathbb{E}[\alpha^{(s)}_{uvw}] + \sum_{(\{u,v,w\},s)\in\mathcal{D}^{(t)}[u]} \mathbb{E}[\beta^{(s)}_{uvw}] \\ &= \sum_{(\{u,v,w\},s)\in\mathcal{A}^{(t)}[u]} 1 + \sum_{(\{u,v,w\},s)\in\mathcal{D}^{(t)}[u]} (-1) = |\mathcal{A}^{(t)}[u]| - |\mathcal{D}^{(t)}[u]| = |\mathcal{T}^{(t)}[u]|.\end{aligned}$$

■

In Section 6.8, we prove the formulas for the variances of estimates given by THINKD$_{\text{FAST}}$. Theorem 6.2 is implied by the formulas.

Theorem 6.2: Variance of THINKD$_{\text{FAST}}$

Given an input graph stream, the variances of estimates maintained by THINKD$_{\text{FAST}}$ with the sampling probability r is proportional to $1/r^2$. Formally, in Algorithm 6.1,

$Var[\bar{c}^{(t)}] = O(1/r^2),\ \forall t \in \{1, 2, ...\},$ and $Var[c^{(t)}[u]] = O(1/r^2),\ \forall t \in \{1, 2, ...\},\ \forall u \in \mathcal{V}^{(t)}.$

Proof. See Theorem 6.5 in Section 6.8. ■

6.4.2 Complexity Analysis

We analyze the time and space complexities of THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$. To this end, we let $\bar{\mathcal{V}}^{(t)} := \bigcup_{s=1}^{t} \mathcal{V}^{(s)}$ be the set of nodes that appear in the t-th or earlier elements in the input stream.

Space Complexity: While processing the first t elements in the input graph stream, THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$ maintain one estimate for the global triangle count and at most $|\bar{\mathcal{V}}^{(t)}|$ estimates for the local triangle counts. In addition, THINKD$_{\text{FAST}}$ maintains $|\mathcal{E}^{(t)}| \cdot r$ edges on average, while THINKD$_{\text{ACC}}$ maintains up to b edges. Thus, the average space complexities of THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$ are $O(|\mathcal{E}^{(t)}| \cdot r + |\bar{\mathcal{V}}^{(t)}|)$ and $O(b + |\bar{\mathcal{V}}^{(t)}|)$, respectively. The complexities become $O(|\mathcal{E}^{(t)}| \cdot r)$ and $O(b)$ when only the global triangle count needs to be estimated.

Time Complexity: We prove the average time complexity of THINKD$_{\text{FAST}}$ in Theorem 6.3, which implies Corollary 6.1, and the worst-case time complexity of THINKD$_{\text{ACC}}$ in Theorem 6.4. Corollary 6.1

and Theorem 6.4 state that, given a fixed memory budget b, THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$ scale linearly with the number of elements in the input stream.

Theorem 6.3: Time Complexity of THINKD$_{\text{FAST}}$

Algorithm 6.1 takes $O(t + t^2r)$ on average to process the first t elements in the input stream.

Proof. In Algorithm 6.1, the most expensive step in processing each element $e^{(s)} = (\{u, v\}, \delta)$ is to intersect $\hat{\mathcal{N}}_u$ and $\hat{\mathcal{N}}_v$ (line 8), which takes $O(1 + \mathbb{E}[|\hat{\mathcal{N}}_u| + |\hat{\mathcal{N}}_v|]) = O(1 + \mathbb{E}[|\mathcal{S}|]) = O(1 + sr)$ on average. Hence, processing the first t elements takes $\sum_{s=1}^{t} O(1 + sr) = O(t + t^2r)$ on average. ■

Corollary 6.1: Time Complexity of THINKD$_{\text{FAST}}$ with Fixed Memory b

If $r = O(b/t)$ for a constant b (≥ 1), then Algorithm 6.1 takes $O(tb)$ on average to process the first t elements in the input stream.

Theorem 6.4: Time Complexity of THINKD$_{\text{ACC}}$

Algorithm 6.2 takes $O(tb)$ to process the first t elements in the input stream.

Proof. In Algorithm 6.2, the most expensive step in processing each element $e^{(s)} = (\{u, v\}, \delta)$ is to intersect $\hat{\mathcal{N}}_u$ and $\hat{\mathcal{N}}_v$ (line 8), which takes $O(1 + |\hat{\mathcal{N}}_u| + |\hat{\mathcal{N}}_v|) = O(b)$. Thus, processing the first t elements takes $O(tb)$. ■

6.5 Experiments

In this section, we review our experiments for answering the following questions:

- **Q1. Illustration of Theorems**: Does THINKD give unbiased estimates?
- **Q2. Accuracy**: Is THINKD more accurate than its best competitors?
- **Q3. Speed**: Is THINKD faster than its best competitors?
- **Q4. Scalability** Does THINKD scale linearly with the size of the input stream?
- **Q5. Effects of Deletions**: Is THINKD consistently accurate regardless of the ratio of deleted edges?

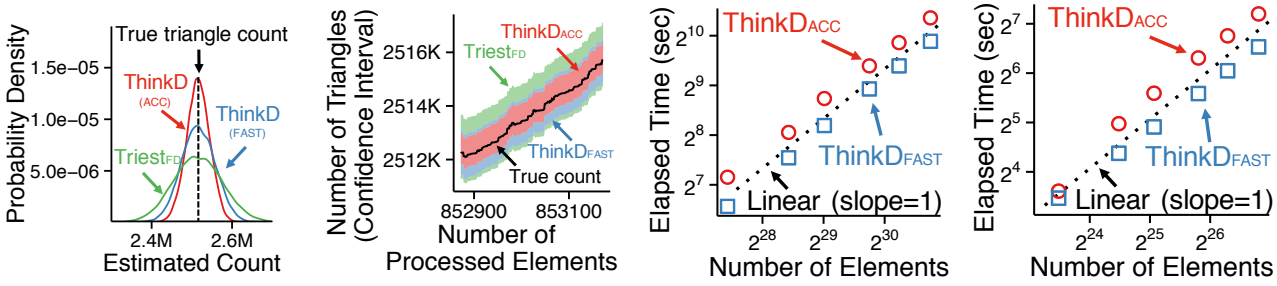
6.5.1 Experimental Settings

Machines: We used a PC with a 3.60GHz Intel i7-4790 CPU and 32GB RAM unless otherwise stated.

Datasets: We created fully dynamic graph streams with deletions using the real-world graphs listed in Table 6.2 as follows: (a) create the additions of the edges in the input graph and shuffle them, (b)

Table 6.2: **Summary of the graph streams used in our experiments.** B: billion, M: million, K: thousand.

Name	#Nodes	#Edges	Type
Friendster [YL15]	65.6M	1.81B	Friendship network
Orkut [MMG+07]	3.07M	117M	Friendship network
Flickr [MMG+07]	2.30M	22.8M	Friendship network
Patent [HJT01]	3.77M	16.5M	Citation network
Youtube [MMG+07]	3.22M	9.38M	Friendship network
BerkStan [LLDM09]	685K	6.65M	Web graph
Facebook [VMCG09]	63.7K	817K	Friendship network
Epinion [MA05]	132K	711K	Trust network
Random (800GB)	1M	0.1B-100B	Synthetic graph



(a) Distribution of estimates at the end (b) 95% confidence intervals over time (c) Scalability (Friendster) (d) Scalability (Orkut)

Figure 6.2: **THINKD is provably accurate and scalable.** (a) THINKD gives unbiased estimates. (b) THINKD maintains more accurate estimates with smaller confidence intervals than its best competitor. (c-d) THINKD scales linearly with the size of the input stream.

choose $\alpha\%$ of the edges and create the deletions of them, (c) locate each deletion in a random position after the corresponding addition. We set α to 20% unless otherwise stated (see Section 6.5.6 for its effect on accuracy). The created streams were streamed from the disk.

Implementations: We implemented THINKD$_{\text{FAST}}$ (Section 6.3.2), THINKD$_{\text{ACC}}$ (Section 6.3.3), TRIEST$_{\text{FD}}$ [SERU17], TRIEST$_{\text{IMPR}}$ [SERU17], ESD [HS17], and MASCOT [LJK18] commonly in Java. In all of them, sampled edges are stored in the adjacency list format, and as described in the last paragraph of Section 6.3.3, estimates are lower bounded by zero.

Evaluation Metrics: We measured the accuracy of the considered algorithms using *global error*, *local error*, and *rank correlation*, all of which are defined in Section 3.3.2.

6.5.2 Q1. Illustration of Theorems

THINKD gives unbiased estimates (Theorem 6.1). We compared $10,000$ estimates of the global triangle count obtained by THINKD$_{\text{FAST}}$, THINKD$_{\text{ACC}}$, and TRIEST$_{\text{FD}}$, whose parameters were set so that on average 10% of the edges are stored at the end of each graph stream. Figures 6.1(d) (in Section 6.1)

and 6.2(a) show the distributions of the estimates at the end of the Facebook and Epinion datasets, respectively. The means of the estimates were close to the true triangle count, consistently with Theorem 6.1 (i.e., unbiasedness of THINKD). Moreover, THINKD$_{\text{ACC}}$ and THINKD$_{\text{FAST}}$ gave estimates with smaller variances than TRIEST$_{\text{FD}}$. Figures 6.1(a) (in Section 6.1) and 6.2(b) show how the 95% confidence intervals, estimated from $10,000$ trials, changed over time in the Facebook and Epinion datasets, respectively. THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$ maintained more accurate estimates with smaller confidence intervals than TRIEST$_{\text{FD}}$. Between THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$, THINKD$_{\text{ACC}}$ was more accurate.

6.5.3 Q2. Accuracy

THINKD is more accurate than its competitors. We compared the accuracies of four algorithms that support edge deletions. As we changed the ratio of stored edges at the end of each input stream from 5% to 40%, we measured the accuracies of THINKD$_{\text{FAST}}$, THINKD$_{\text{ACC}}$, and TRIEST$_{\text{FD}}$. ESD always stores the entire input stream in memory, and we set its parameter to 1.0 to maximize its accuracy. Each evaluation metric was averaged over 100 trials in the Friendster and Orkut datasets and $1,000$ trials in the others.[1] As seen in Figure 6.3, THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$ consistently gave the best trade-off between space and accuracy. Specifically, within the same memory budget, THINKD$_{\text{ACC}}$ was up to $4\times$ and $4.3\times$ *more accurate* than TRIEST$_{\text{FD}}$ in terms of global error and RMSE, respectively. Between our algorithms, THINKD$_{\text{ACC}}$ consistently outperformed THINKD$_{\text{FAST}}$ in terms of accuracy.

6.5.4 Q3. Speed

THINKD is faster than its competitors. We compared the speeds and accuracies of four algorithms that support edge deletions. The detailed settings were the same as those in Section 6.5.3 except that we measured the performance of ESD as we changed its parameter from 0.2 to 1.0. To measure the speeds of the algorithms independently of the speed of the input stream, we ignored time taken to wait for the arrival of elements. As seen in Figure 6.4, THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$ consistently gave the best trade-off between speed and accuracy. Specifically, for the same global error and RMSE, THINKD$_{\text{FAST}}$ was up to $2.2\times$ *faster* than TRIEST$_{\text{FD}}$. Between our algorithms, THINKD$_{\text{FAST}}$ consistently outperformed THINKD$_{\text{ACC}}$ in terms of speed.

6.5.5 Q4. Scalability

THINKD scales linearly (Corollary 6.1 and Theorem 6.4). We measured the elapsed times taken by THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$ to process all elements in graph streams with different numbers of elements. To measure their speeds independently of the speed of the input stream, we ignored time taken to wait for the arrival of elements. In both algorithms, we set b and r so that on average 10^7 edges are stored at the end of each input stream. Figure 6.1(c) in Section 6.1 shows the results in the Random datasets, which were created by the Erdös-Rényi model. Both THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$ scaled linearly with the number of elements, as expected in Corollary 6.1 and Theorem 6.4. Notice that the largest dataset is *800GB with 100 billion elements*. As seen in Figures 6.2(c)-(d), we obtained similar trends in graph streams with realistic structures created by sampling different numbers of elements from the Friendster and Orkut datasets.

[1] We used a workstation with 2.67GHz Intel Xeon E7-8837 CPUs and 1TB memory for the Friendster dataset.

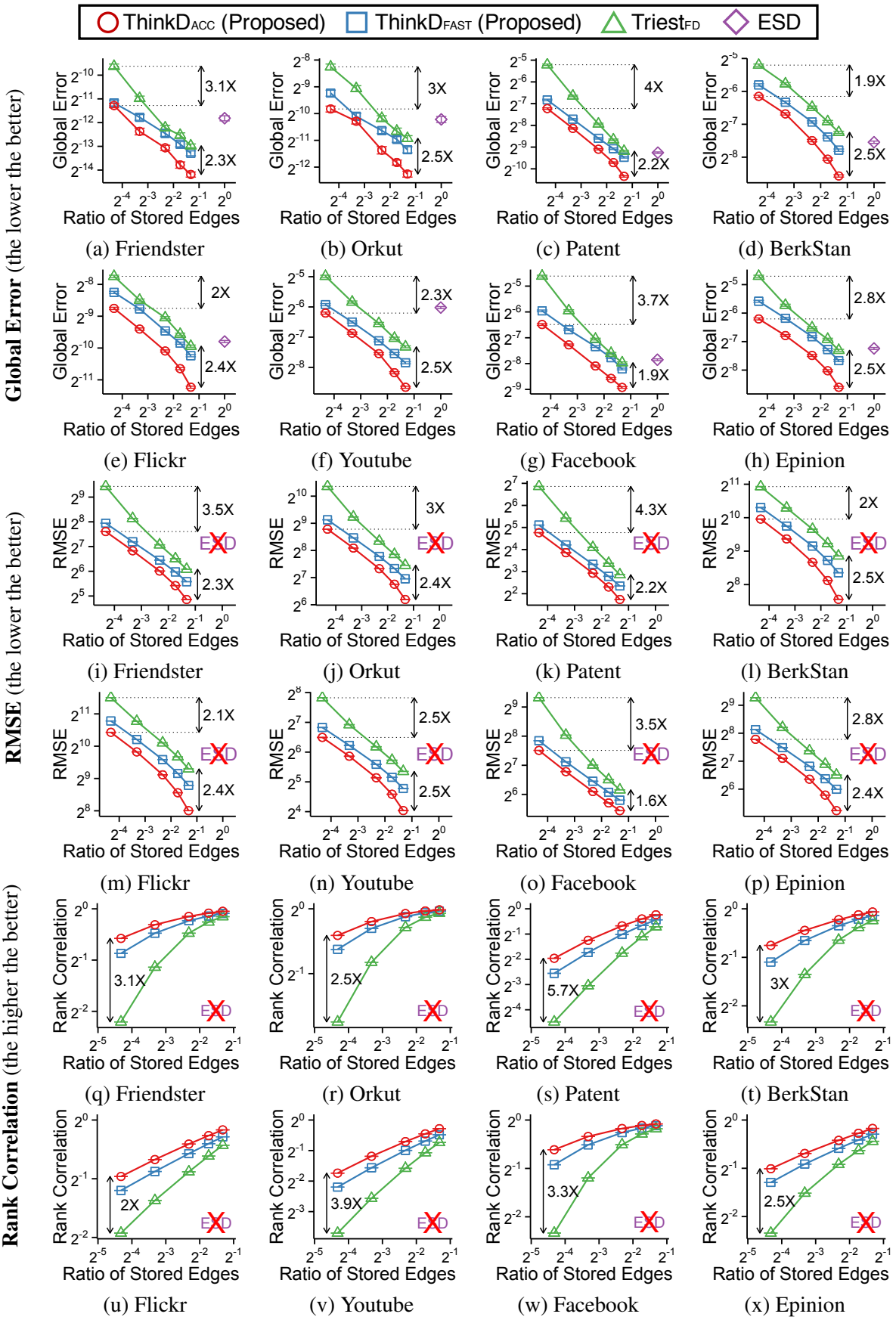


Figure 6.3: **THINKD is accurate**. THINKD provides the best trade-off between space and accuracy. In particular, THINKD$_{\text{ACC}}$ is up to $4.3\times$ *more accurate* than TRIEST$_{\text{FD}}$ within the same memory budget. Error bars denote ± 1 standard error. ESD is inapplicable to local triangle counting.

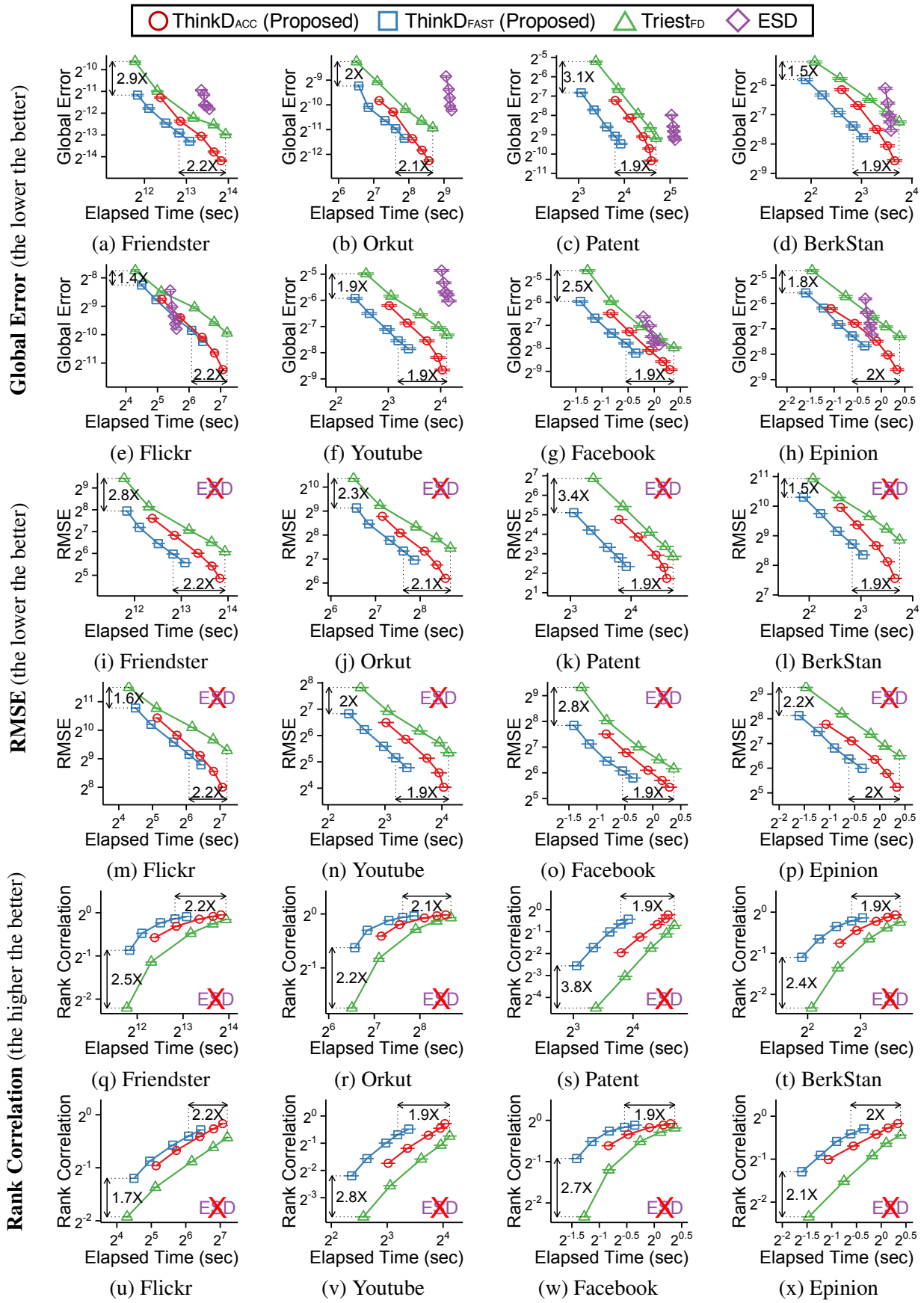


Figure 6.4: **THINKD is fast.** THINKD provides the best trade-off between speed and accuracy. In particular, THINKD$_{\text{FAST}}$ is up to $2.2\times$ *faster* than TRIEST$_{\text{FD}}$ when they are similarly accurate. Error bars denote ± 1 standard error. ESD is inapplicable to local triangle counting.

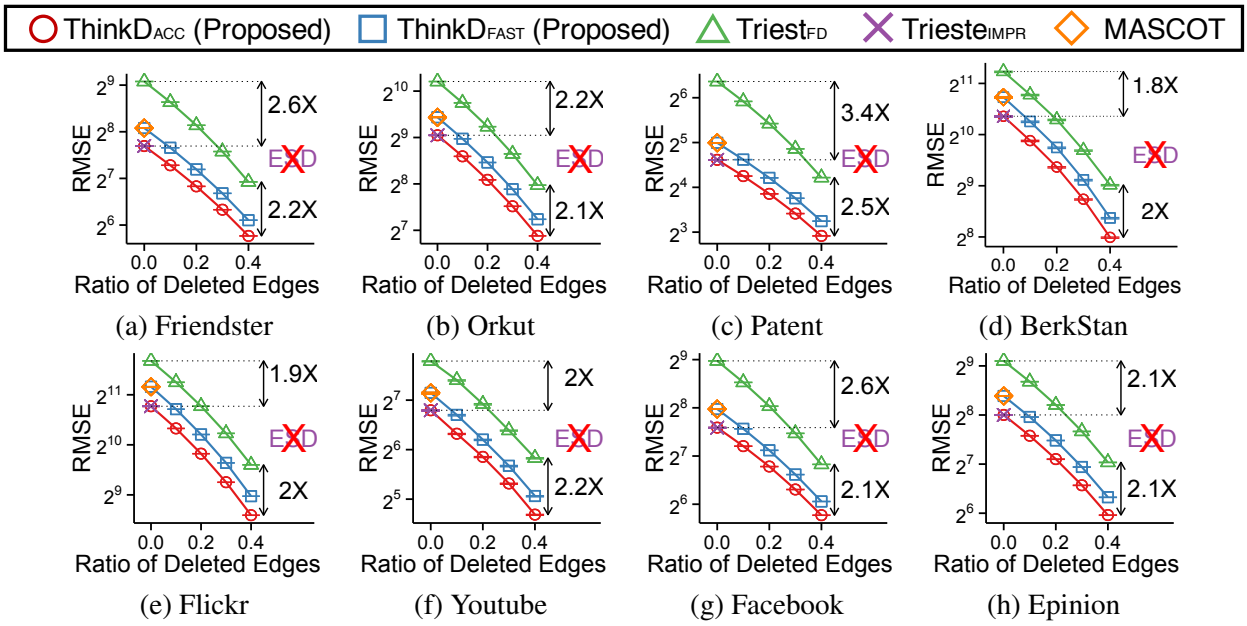


Figure 6.5: **THINKD is consistently accurate regardless of the ratio of deleted edges.** Error bars denote ± 1 standard error. TRIEST$_{\text{IMPR}}$ and MASCOT are inapplicable when there are deletions. ESD is inapplicable to local triangle counting.

6.5.6 Q5. Effects of Deletions on Accuracy

THINKD is consistently accurate. We measured how the ratio of deleted edges (i.e., α in Section 6.5.1) in input graph streams affects the accuracies of the considered algorithms. In every algorithm, we set the ratio of stored edges at the end of each input stream to 10%. As seen in Figure 6.5, all algorithms that support edge deletions became more accurate as input graphs became smaller with more deletions. THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$ were similarly accurate with MASCOT and TRIEST$_{\text{IMPR}}$, respectively, in the streams without deletions. In the streams with deletions, which MASCOT and TRIEST$_{\text{IMPR}}$ cannot handle, THINKD$_{\text{FAST}}$ and THINKD$_{\text{ACC}}$ were $1.8 - 3.4\times$ *more accurate* than TRIEST$_{\text{FD}}$ regardless of the ratio of deleted edges.

6.6 Summary

We propose THINKD, which estimates the counts of global and local triangles in a fully dynamic graph stream with edge additions and deletions. We theoretically and empirically show that THINKD has the following advantages:

- **Accurate**: THINKD is up to $4.3\times$ *more accurate* than its best competitors within the same memory budget (Figure 6.3).
- **Fast**: THINKD is up to $2.2\times$ *faster* than its best competitors with similar accuracies (Figure 6.4). It processes *terabyte*-scale streams with linear scalability (Figure 6.2, Corollary 6.1, and Theorem 6.4).
- **Theoretically Sound**: THINKD maintains *unbiased* estimates (Theorem 6.1) with small variances (Theorem 6.2) *at any time* while the input graph evolves.

Reproducibility: The source code and datasets used in this chapter are available at `http://www.cs.cmu.edu/~kijungs/thinkd/`.

6.7 Appendix: Proofs

For each variable (e.g., $\bar{c}$) in Algorithms 6.1 and 6.2, we use superscript (t) (e.g., $\bar{c}^{(t)}$) to denote the value of the variable after the t-th element $e^{(t)}$ is processed. For any time $t \in \{1, 2, ...\}$, let $X^{(t)}$ be the random number in $Bernoulli(r)$ drawn in line 11 of Algorithm 6.1 while the t-th element $e^{(t)}$ is processed, and for each edge $\{u, v\}$, let $l_{uv}^{(t)}$ be the last time that $\{u, v\}$ is added to or removed from $\mathcal{G}$ at time t or earlier. That is,

$$l_{uv}^{(t)} := \max(\{1 \le s \le t : e^{(s)} = (\{u, v\}, +) \text{ or } e^{(s)} = (\{u, v\}, -)\}). \tag{6.10}$$

Lemma 6.4

In Algorithm 6.1, for each time $t \in \{1, 2, ...\}$ and any edge $\{u, v\} \in \mathcal{E}^{(t)}$, $\{u, v\} \in \mathcal{S}^{(t)}$ if and only if $X^{(l_{uv}^{(t)})} = 1$. That is,

$$\{u, v\} \in \mathcal{S}^{(t)} \iff X^{(l_{uv}^{(t)})} = 1, \ \forall t \in \{1, 2, ...\}, \ \forall \{u, v\} \in \mathcal{E}^{(t)} \tag{6.11}$$

Proof. Note that $\{u, v\} \in \mathcal{E}^{(t)}$ implies that $e^{(l_{uv}^{(t)})} = (\{u, v\}, +)$, i.e. the edge $\{u, v\}$ is added at time $l_{uv}^{(t)}$. Then $\{u, v\} \notin \mathcal{E}^{(l_{uv}^{(t)}-1)}$, and since $\mathcal{S}^{(s)} \subset \mathcal{E}^{(s)}$ for all $s \in \{1, 2, ...\}$, $\{u, v\} \notin \mathcal{S}^{(l_{uv}^{(t)}-1)}$ as well. Therefore,

$$\{u, v\} \in \mathcal{S}^{(l_{uv}^{(t)})} \iff X^{(l_{uv}^{(t)})} = 1. \tag{6.12}$$

Also, from Eq. (6.10), $e^{(s)} \neq (\{u, v\}, \delta)$ if $l_{uv}^{(t)} < s \le t$, and hence $\{u, v\}$ is not added after time $l_{uv}^{(t)}$ in Algorithm 1. Hence for all $s \in [l_{uv}^{(t)}, t]$,

$$\{u, v\} \in \mathcal{S}^{(s)} \iff \{u, v\} \in \mathcal{S}^{(l_{uv}^{(t)})}. \tag{6.13}$$

Combining Eq. (6.12) and Eq. (6.13) with $s = t$ gives Eq. (6.11). ■

6.7.1 Proof of Lemma 6.1

Proof. Applying Lemma 6.4 to the edges $\{u, v\}$ and $\{w, x\}$ gives

$$\{u, v\} \in \mathcal{S}^{(t)} \iff X^{(l_{uv}^{(t)})} = 1 \quad \text{and} \quad \{w, x\} \in \mathcal{S}^{(t)} \iff X^{(l_{wx}^{(t)})} = 1. \tag{6.14}$$

Then, since $X^{(s)}$'s are independent $Bernoulli(r)$ and $l_{uv}^{(t)} \neq l_{wx}^{(t)}$, applying Eq. (6.14) with independence of $X^{(l_{uv}^{(t)})}$ and $X^{(l_{wx}^{(t)})}$ gives

$$\begin{aligned} Pr\left[\{u, v\} \in \mathcal{S}^{(t)} \cap \{w, x\} \in \mathcal{S}^{(t)}\right] &= Pr\left[X^{(l_{uv}^{(t)})} = 1 \cap X^{(l_{wx}^{(t)})} = 1\right] \\ &= Pr\left[X^{(l_{uv}^{(t)})} = 1\right] Pr\left[X^{(l_{wx}^{(t)})} = 1\right] = r^2. \end{aligned}$$

■

6.7.2 Proof of Lemma 6.2

As in Section 6.3.3, for each time $t \in \{1, 2, ...\}$, let $\mathcal{E}^{(t)}$ be the set of edges remaining (without being deleted) in the input graph stream and $\mathcal{S}^{(t)} \subset \mathcal{E}^{(t)}$ be the set of samples maintained by Algorithm 6.2 after the t-th element is processed. Also let $y^{(t)} = \min(b, |\mathcal{E}^{(t)}| + n_b^{(t)} + n_g^{(t)})$.

Lemma 6.5: Properties in Random Pairing [GLH08]

In Algorithm 6.2, the expected value and variance of the size of the samples are as follows:

$$\mathbb{E}[|\mathcal{S}^{(t)}|] = \frac{|\mathcal{E}^{(t)}| \cdot y^{(t)}}{|\mathcal{E}^{(t)}| + n_b^{(t)} + n_g^{(t)}}, \ \forall t \in \{1, 2, ...\}. \tag{6.15}$$

$$Var[|\mathcal{S}^{(t)}|] = \frac{(n_b^{(t)} + n_g^{(t)}) \cdot y^{(t)} \cdot (|\mathcal{E}^{(t)}| + n_b^{(t)} + n_g^{(t)} - y^{(t)}) \cdot |\mathcal{E}^{(t)}|}{(|\mathcal{E}^{(t)}| + n_b^{(t)} + n_g^{(t)})^2 \cdot (|\mathcal{E}^{(t)}| + n_b^{(t)} + n_g^{(t)} - 1)}, \forall t \in \{1, 2, ...\}. \tag{6.16}$$

At each fixed time, all equal-sized subsets of the remaining elements in the input graph stream have the same probability to be the set of samples maintained in Algorithm 6.2. Formally,

$$Pr[\mathcal{S}^{(t)} = \mathcal{A}] = Pr[\mathcal{S}^{(t)} = \mathcal{B}], \ \forall t \in \{1, 2, ...\}, \ \forall \mathcal{A} \neq \mathcal{B} \subset \mathcal{E}^{(t)} \text{ s.t. } |\mathcal{A}| = |\mathcal{B}|. \tag{6.17}$$

Lemma 6.6: Uniformity in Random Pairing

At each fixed time, all equal-sized subsets of the remaining elements in the input graph stream have the same probability to be a subset of the samples maintained in Algorithm 6.2. Formally,

$$Pr[\mathcal{A} \subset \mathcal{S}^{(t)}] = Pr[\mathcal{B} \subset \mathcal{S}^{(t)}], \ \forall t \in \{1, 2, ...\}, \ \forall \mathcal{A} \neq \mathcal{B} \subset \mathcal{E}^{(t)} \text{ s.t. } |\mathcal{A}| = |\mathcal{B}|. \tag{6.18}$$

Proof. Let $e_i^{\mathcal{A}}$ be the family of size-i subsets of $\mathcal{E}^{(t)}$ including $\mathcal{A}$, and let $e_i^{\mathcal{B}}$ be the family of size-i subsets of $\mathcal{E}^{(t)}$ including $\mathcal{B}$. Then, Eq. (6.18) is obtained as follows:

$$\begin{aligned} Pr[\mathcal{A} \subset \mathcal{S}^{(t)}] &= \sum_i \sum_{\mathcal{C} \in e_i^{\mathcal{A}}} Pr[\mathcal{C} = \mathcal{S}^{(t)}] \\ &= \sum_i \sum_{\mathcal{C} \in e_i^{\mathcal{B}}} Pr[\mathcal{C} = \mathcal{S}^{(t)}] = Pr[\mathcal{B} \subset \mathcal{S}^{(t)}], \end{aligned}$$

where the second equality is from Eq. (6.17) and $|e_i^{\mathcal{A}}| = |e_i^{\mathcal{B}}|$. ■

Lemma 6.7: Sampling Probability of Each Edge

The probability that each edge is sampled in Algorithm 6.2 is as follows:

$$Pr[\{u,v\} \in \mathcal{S}^{(t)}] = \frac{y^{(t)}}{|\mathcal{E}^{(t)}| + n_b^{(t)} + n_g^{(t)}}, \ \forall t \in \{1,2,...\}, \ \forall \{u,v\} \in \mathcal{E}^{(t)}. \tag{6.19}$$

Proof. Let $\mathbf{1}(\{u,v\} \in \mathcal{S}^{(t)})$ be a random variable which is 1 if $\{u,v\} \in \mathcal{S}^{(t)}$ and 0 otherwise. By definition,

$$|\mathcal{S}^{(t)}| = \sum_{\{u,v\} \in \mathcal{E}^{(t)}} \mathbf{1}(\{u,v\} \in \mathcal{S}^{(t)}). \tag{6.20}$$

Then, by linearity of expectation and Eq. (6.20),

$$\mathbb{E}[|\mathcal{S}^{(t)}|] = \sum_{\{u,v\} \in \mathcal{E}^{(t)}} \mathbb{E}[\mathbf{1}(\{u,v\} \in \mathcal{S}^{(t)})] = \sum_{\{u,v\} \in \mathcal{E}^{(t)}} Pr[\{u,v\} \in \mathcal{S}^{(t)}]. \tag{6.21}$$

Then, Eq. (6.19) is obtained as follows:

$$\begin{aligned} Pr[\{u,v\} \in \mathcal{S}^{(t)}] &= \frac{1}{|\mathcal{E}^{(t)}|} \sum_{\{w,x\} \in \mathcal{E}^{(t)}} Pr[\{w,x\} \in \mathcal{S}^{(t)}] \\ &= \frac{\mathbb{E}[|\mathcal{S}^{(t)}|]}{|\mathcal{E}^{(t)}|} = \frac{y^{(t)}}{|\mathcal{E}^{(t)}| + n_b^{(t)} + n_g^{(t)}}, \end{aligned}$$

where the first, second, and last equalities are from Eq. (6.18), Eq. (6.21), and Eq. (6.15), respectively. ■

Proof of Lemma 6.2:

Proof. Let $\mathbf{1}(\{u,v\} \in \mathcal{S}^{(t)})$ be a random variable which is 1 if $\{u,v\} \in \mathcal{S}^{(t)}$ and 0 otherwise. For calculating $Pr[\{u,v\} \in \mathcal{S}^{(t)} \cap \{w,x\} \in \mathcal{S}^{(t)}]$, we expand the covariance sum $\sum_{\{u,v\} \neq \{w,x\}} Cov(\mathbf{1}(\{u,v\} \in \mathcal{S}^{(t)}), \mathbf{1}(\{w,x\} \in \mathcal{S}^{(t)}))$ in two ways and compare them.

First, we use the expansion of the variance of $\mathcal{S}^{(t)}$. From Eq. (6.20),

$$\begin{aligned} Var[|\mathcal{S}^{(t)}|] = &\sum_{\{u,v\} \in \mathcal{E}^{(t)}} Var[\mathbf{1}(\{u,v\} \in \mathcal{S}^{(t)})] \\ &+ \sum_{\{u,v\} \neq \{w,x\}} Cov(\mathbf{1}(\{u,v\} \in \mathcal{S}^{(t)}), \mathbf{1}(\{w,x\} \in \mathcal{S}^{(t)})), \end{aligned}$$

and hence the covariance sum can be expanded as

$$\sum_{\{u,v\} \neq \{w,x\}} Cov(\mathbf{1}(\{u,v\} \in \mathcal{S}^{(t)}), \mathbf{1}(\{w,x\} \in \mathcal{S}^{(t)})) = Var[|\mathcal{S}^{(t)}|] - \sum_{\{u,v\} \in \mathcal{E}^{(t)}} Var[\mathbf{1}(\{u,v\} \in \mathcal{S}^{(t)})]. \tag{6.22}$$

For the second term of Eq. (6.22), $Var[x] = \mathbb{E}[x^2] - (\mathbb{E}[x])^2$ implies

$$Var[\mathbf{1}(\{u,v\} \in \mathcal{S}^{(t)})] = Pr[\{u,v\} \in \mathcal{S}^{(t)}] - Pr[\{u,v\} \in \mathcal{S}^{(t)}]^2. \tag{6.23}$$

Hence applying Eq. (6.19) and Eq. (6.23) to Eq. (6.22) gives the covariance sum as

$$\begin{aligned}
&\sum_{\{u,v\}\neq\{w,x\}} Cov(\mathbf{1}(\{u,v\}\in\mathcal{S}^{(t)}),\mathbf{1}(\{w,x\}\in\mathcal{S}^{(t)}))\\
&= Var[|\mathcal{S}^{(t)}|] - \sum_{\{u,v\}\in\mathcal{E}^{(t)}} Var[\mathbf{1}(\{u,v\}\in\mathcal{S}^{(t)})]\\
&= Var[|\mathcal{S}^{(t)}|] - \sum_{\{u,v\}\in\mathcal{E}^{(t)}} \left(Pr[\{u,v\}\in\mathcal{S}^{(t)}] - Pr[\{u,v\}\in\mathcal{S}^{(t)}]^2\right)\\
&= Var[|\mathcal{S}^{(t)}|] - |\mathcal{E}^{(t)}|\cdot\frac{y^{(t)}\cdot(|\mathcal{E}^{(t)}|+n_b^{(t)}+n_g^{(t)}-y^{(t)})}{(|\mathcal{E}^{(t)}|+n_b^{(t)}+n_g^{(t)})^2}.
\end{aligned} \tag{6.24}$$

Second, we directly expand the covariance sum. Expanding the covariance sum with $Cov(x,y) = \mathbb{E}[xy] - \mathbb{E}[x]\cdot\mathbb{E}[y]$ and applying Eq. (6.19) give

$$\begin{aligned}
&\sum_{\{u,v\}\neq\{w,x\}} Cov(\mathbf{1}(\{u,v\}\in\mathcal{S}^{(t)}),\mathbf{1}(\{w,x\}\in\mathcal{S}^{(t)}))\\
&= \sum_{\{u,v\}\neq\{w,x\}} \left(Pr[\{u,v\}\in\mathcal{S}^{(t)}\cap\{w,x\}\in\mathcal{S}^{(t)}] - Pr[\{u,v\}\in\mathcal{S}^{(t)}]\cdot Pr[\{w,x\}\in\mathcal{S}^{(t)}]\right)\\
&= \sum_{\{u,v\}\neq\{w,x\}} \left(Pr[\{u,v\}\in\mathcal{S}^{(t)}\cap\{w,x\}\in\mathcal{S}^{(t)}] - \left(\frac{y^{(t)}}{|\mathcal{E}^{(t)}|+n_b^{(t)}+n_g^{(t)}}\right)^2\right)\\
&= \sum_{\{u,v\}\neq\{w,x\}} \left(Pr[\{u,v\}\in\mathcal{S}^{(t)}\cap\{w,x\}\in\mathcal{S}^{(t)}]\right) - \frac{y^{(t)}\cdot y^{(t)}\cdot|\mathcal{E}^{(t)}|\cdot(|\mathcal{E}^{(t)}|-1)}{(|\mathcal{E}^{(t)}|+n_b^{(t)}+n_g^{(t)})^2}.
\end{aligned} \tag{6.25}$$

Now, the probability sum $\sum_{\{u,v\}\neq\{w,x\}} Pr[\{u,v\}\in\mathcal{S}^{(t)}\cap\{w,x\}\in\mathcal{S}^{(t)}]$ can be obtained by comparing two expansions Eq. (6.24) and Eq. (6.25) of the covariance sum and applying Eq. (6.16) as

$$\begin{aligned}
&\sum_{\{u,v\}\neq\{w,x\}} Pr[\{u,v\}\in\mathcal{S}^{(t)}\cap\{w,x\}\in\mathcal{S}^{(t)}]\\
&= \sum_{\{u,v\}\neq\{w,x\}} Cov(\mathbf{1}(\{u,v\}\in\mathcal{S}^{(t)}),\mathbf{1}(\{w,x\}\in\mathcal{S}^{(t)})) + \frac{y^{(t)}\cdot y^{(t)}\cdot|\mathcal{E}^{(t)}|\cdot(|\mathcal{E}^{(t)}|-1)}{(|\mathcal{E}^{(t)}|+n_b^{(t)}+n_g^{(t)})^2}\\
&= Var[|\mathcal{S}^{(t)}|] - |\mathcal{E}^{(t)}|\cdot\frac{y^{(t)}\cdot(|\mathcal{E}^{(t)}|+n_b^{(t)}+n_g^{(t)}-y^{(t)})}{(|\mathcal{E}^{(t)}|+n_b^{(t)}+n_g^{(t)})^2} + \frac{y^{(t)}\cdot y^{(t)}\cdot|\mathcal{E}^{(t)}|\cdot(|\mathcal{E}^{(t)}|-1)}{(|\mathcal{E}^{(t)}|+n_b^{(t)}+n_g^{(t)})^2}\\
&= \frac{y^{(t)}\cdot(y^{(t)}-1)\cdot|\mathcal{E}^{(t)}|\cdot(|\mathcal{E}^{(t)}|-1)}{(|\mathcal{E}^{(t)}|+n_b^{(t)}+n_g^{(t)})\cdot(|\mathcal{E}^{(t)}|+n_b^{(t)}+n_g^{(t)}-1)}.
\end{aligned} \tag{6.26}$$

Then, Eq. (6.3) is obtained by Eq. (6.26) as follows:

$$\begin{aligned}
&Pr[\{u,v\} \in \mathcal{S}^{(t)} \cap \{w,x\} \in \mathcal{S}^{(t)}] \\
&= \frac{1}{|\mathcal{E}^{(t)}| \cdot (|\mathcal{E}^{(t)}| - 1)} \left(\sum_{\{u,v\} \neq \{w,x\}} Pr[\{u,v\} \in \mathcal{S}^{(t)} \cap \{w,x\} \in \mathcal{S}^{(t)}] \right) \\
&= \frac{1}{|\mathcal{E}^{(t)}| \cdot (|\mathcal{E}^{(t)}| - 1)} \cdot \frac{y^{(t)} \cdot (y^{(t)} - 1) \cdot |\mathcal{E}^{(t)}| \cdot (|\mathcal{E}^{(t)}| - 1)}{(|\mathcal{E}^{(t)}| + n_b^{(t)} + n_g^{(t)}) \cdot (|\mathcal{E}^{(t)}| + n_b^{(t)} + n_g^{(t)} - 1)} \\
&= \frac{y^{(t)}}{|\mathcal{E}^{(t)}| + n_b^{(t)} + n_g^{(t)}} \cdot \frac{y^{(t)} - 1}{|\mathcal{E}^{(t)}| + n_b^{(t)} + n_g^{(t)} - 1} = p^{(t)},
\end{aligned}$$

where the first equality is from Eq. (6.18). ■

6.7.3 Proof of Lemma 6.3

Proof. When $t = 1$, then $\mathcal{T}^{(1)} = \mathcal{A}^{(1)} = \mathcal{D}^{(1)} = \emptyset$ holds, and hence Eq. (6.4) trivially holds. Hence we assume that $t \in \{2, 3, ...\}$ from now on. First, we show that for each time $s \in \{2, 3, ...\}$,

$$|\mathcal{T}^{(s)}| - |\mathcal{T}^{(s-1)}| = |\mathcal{A}^{(s)} \backslash \mathcal{A}^{(s-1)}| - |\mathcal{D}^{(s)} \backslash \mathcal{D}^{(s-1)}|. \tag{6.27}$$

To show this, we show the following relations,

$$|\mathcal{A}^{(s)} \backslash \mathcal{A}^{(s-1)}| = |\mathcal{T}^{(s)} \backslash \mathcal{T}^{(s-1)}|, \tag{6.28}$$

$$|\mathcal{D}^{(s)} \backslash \mathcal{D}^{(s-1)}| = |\mathcal{T}^{(s-1)} \backslash \mathcal{T}^{(s)}|. \tag{6.29}$$

For Eq. (6.28), note that

$$\mathcal{A}^{(s)} \backslash \mathcal{A}^{(s-1)} = \left\{ (\{u,v,w\}, s) : \{u,v,w\} \notin \mathcal{T}^{(s-1)} \text{ and } \{u,v,w\} \in \mathcal{T}^{(s)} \right\}$$

and

$$\mathcal{T}^{(s)} \backslash \mathcal{T}^{(s-1)} = \left\{ \{u,v,w\} : \{u,v,w\} \notin \mathcal{T}^{(s-1)} \text{ and } \{u,v,w\} \in \mathcal{T}^{(s)} \right\}$$

hold, and hence Eq. (6.28) holds. Similarly,

$$\mathcal{D}^{(s)} \backslash \mathcal{D}^{(s-1)} = \left\{ (\{u,v,w\}, s) : \{u,v,w\} \in \mathcal{T}^{(s-1)} \text{ and } \{u,v,w\} \notin \mathcal{T}^{(s)} \right\}$$

and

$$\mathcal{T}^{(s-1)} \backslash \mathcal{T}^{(s)} = \left\{ \{u,v,w\} : \{u,v,w\} \in \mathcal{T}^{(s-1)} \text{ and } \{u,v,w\} \notin \mathcal{T}^{(s)} \right\}$$

hold, and hence Eq. (6.29) holds. Then, Eq. (6.28) and Eq. (6.29) imply

$$\begin{aligned}
|\mathcal{A}^{(s)} \backslash \mathcal{A}^{(s-1)}| + |\mathcal{T}^{(s-1)}| &= |\mathcal{T}^{(s)} \backslash \mathcal{T}^{(s-1)}| + |\mathcal{T}^{(s-1)}| = |\mathcal{T}^{(s-1)} \cup \mathcal{T}^{(s)}| \\
&= |\mathcal{T}^{(s-1)} \backslash \mathcal{T}^{(s)}| + |\mathcal{T}^{(s)}| = |\mathcal{D}^{(s)} \backslash \mathcal{D}^{(s-1)}| + |\mathcal{T}^{(s)}|,
\end{aligned}$$

and hence Eq. (6.27) holds. Then, summing up Eq. (6.27) from $s = 2$ to t yields

$$|\mathcal{T}^{(t)}| - |\mathcal{T}^{(1)}| = \sum_{s=2}^{t} |\mathcal{A}^{(s)} \backslash \mathcal{A}^{(s-1)}| - \sum_{s=2}^{t} |\mathcal{D}^{(s)} \backslash \mathcal{D}^{(s-1)}|. \tag{6.30}$$

Then, $\left\{\mathcal{A}^{(s)} \backslash \mathcal{A}^{(s-1)}\right\}_{s=2}^{t}$ being disjoint over s implies

$$\sum_{s=2}^{t} |\mathcal{A}^{(s)} \backslash \mathcal{A}^{(s-1)}| = \left| \bigcup_{s=2}^{t} (\mathcal{A}^{(s)} \backslash \mathcal{A}^{(s-1)}) \right| = |\mathcal{A}^{(t)} \backslash \mathcal{A}^{(1)}|, \tag{6.31}$$

and similarly,

$$\sum_{s=2}^{t} |\mathcal{D}^{(s)} \backslash \mathcal{D}^{(s-1)}| = |\mathcal{D}^{(t)} \backslash \mathcal{D}^{(1)}|. \tag{6.32}$$

holds. Then, applying Eq. (6.31), Eq. (6.32), and $\mathcal{T}^{(1)} = \mathcal{A}^{(1)} = \mathcal{D}^{(1)} = \emptyset$ to Eq. (6.30) yields that for all $t \in \{2, 3, ...\}$,

$$|\mathcal{T}^{(t)}| = |\mathcal{A}^{(t)}| - |\mathcal{D}^{(t)}|,$$

which completes the proof of Eq. (6.4).

For Eq. (6.5), replacing $\mathcal{T}^{(s)}$ by $\mathcal{T}^{(s)}[u]$, $\mathcal{A}^{(s)}$ by $\mathcal{A}^{(s)}[u]$, and $\mathcal{D}^{(s)}$ by $\mathcal{D}^{(s)}[u]$ and repeating above give a proof. ■

6.8 Appendix: Detailed Variance Analysis

As in Section. 6.7, let $l_{uv}^{(t)}$ be the last time that edge $\{u, v\}$ is added to or removed from $\mathcal{G}$ at time t or earlier. And for each added or deleted triangle $(\{u, v, w\}, s) \in \mathcal{A}^{(t)} \cup \mathcal{D}^{(t)}$, we use $\mathbf{1}_{(\{u,v,w\},s)}$ to denote the time when its first edge has arrived and $\mathbf{2}_{(\{u,v,w\},s)}$ to denote the time when its second edge has arrived. Formally,

$$\mathbf{1}_{(\{u,v,w\},s)} := \min(l_{uv}^{(s)}, l_{vw}^{(s)}, l_{wu}^{(s)}), \quad \mathbf{2}_{(\{u,v,w\},s)} := \text{median}(l_{uv}^{(s)}, l_{vw}^{(s)}, l_{wu}^{(s)}).$$

Then, we define the type of each triangle pair in Definition 6.3.

Definition 6.3: Types of Triangle Pairs

The type of each ordered pair of two distinct triangles $\tau \neq \omega \in \mathcal{A}^{(t)} \cup \mathcal{D}^{(t)}$ is defined as follows:

$$Type_{(\tau,\omega)} = \begin{cases} 1, & \text{if } \tau \in \mathcal{A}^{(t)} \text{ and } \omega \in \mathcal{A}^{(t)} \text{ and } |\{\mathbf{1}_\tau, \mathbf{2}_\tau\} \cap \{\mathbf{1}_\omega, \mathbf{2}_\omega\}| = 1, \\ 2, & \text{if } \tau \in \mathcal{D}^{(t)} \text{ and } \omega \in \mathcal{D}^{(t)} \text{ and } |\{\mathbf{1}_\tau, \mathbf{2}_\tau\} \cap \{\mathbf{1}_\omega, \mathbf{2}_\omega\}| = 1, \\ 3, & \text{if } \tau \in \mathcal{A}^{(t)} \text{ and } \omega \in \mathcal{D}^{(t)} \text{ and } |\{\mathbf{1}_\tau, \mathbf{2}_\tau\} \cap \{\mathbf{1}_\omega, \mathbf{2}_\omega\}| = 1, \\ 4, & \text{if } \tau \in \mathcal{D}^{(t)} \text{ and } \omega \in \mathcal{A}^{(t)} \text{ and } |\{\mathbf{1}_\tau, \mathbf{2}_\tau\} \cap \{\mathbf{1}_\omega, \mathbf{2}_\omega\}| = 1, \\ 5, & \text{if } \tau \in \mathcal{A}^{(t)} \text{ and } \omega \in \mathcal{A}^{(t)} \text{ and } |\{\mathbf{1}_\tau, \mathbf{2}_\tau\} \cap \{\mathbf{1}_\omega, \mathbf{2}_\omega\}| = 2, \\ 6, & \text{if } \tau \in \mathcal{D}^{(t)} \text{ and } \omega \in \mathcal{D}^{(t)} \text{ and } |\{\mathbf{1}_\tau, \mathbf{2}_\tau\} \cap \{\mathbf{1}_\omega, \mathbf{2}_\omega\}| = 2, \\ 7, & \text{if } \tau \in \mathcal{A}^{(t)} \text{ and } \omega \in \mathcal{D}^{(t)} \text{ and } |\{\mathbf{1}_\tau, \mathbf{2}_\tau\} \cap \{\mathbf{1}_\omega, \mathbf{2}_\omega\}| = 2, \\ 8, & \text{if } \tau \in \mathcal{D}^{(t)} \text{ and } \omega \in \mathcal{A}^{(t)} \text{ and } |\{\mathbf{1}_\tau, \mathbf{2}_\tau\} \cap \{\mathbf{1}_\omega, \mathbf{2}_\omega\}| = 2, \\ 9, & \text{otherwise (i.e., } |\{\mathbf{1}_\tau, \mathbf{2}_\tau\} \cap \{\mathbf{1}_\omega, \mathbf{2}_\omega\}| = 0). \end{cases} \tag{6.33}$$

Theorem 6.5: Variance of THINKD$_{\text{FAST}}$

Let $n_i^{(t)}$ be the number of Type-i triangle pairs in $\mathcal{A}^{(t)} \cup \mathcal{D}^{(t)}$. Likewise, Let $n_i^{(t)}[u]$ be the number of Type-i triangle pairs in $\mathcal{A}^{(t)}[u] \cup \mathcal{D}^{(t)}[u]$. Then,

$$Var[\bar{c}^{(t)}] = (|\mathcal{A}^{(t)}| + |\mathcal{D}^{(t)}|) \cdot \frac{1-r^2}{r^2} + (n_1^{(t)} + n_2^{(t)} - n_3^{(t)} - n_4^{(t)}) \cdot \frac{1-r}{r} + (n_5^{(t)} + n_6^{(t)} - n_7^{(t)} - n_8^{(t)}) \cdot \frac{1-r^2}{r^2}, \ \forall t \in \{1, 2, ...\}. \tag{6.34}$$

Likewise,

$$Var[c^{(t)}[u]] = (|\mathcal{A}^{(t)}[u]| + |\mathcal{D}^{(t)}[u]|) \cdot \frac{1-r^2}{r^2} + (n_1^{(t)}[u] + n_2^{(t)}[u] - n_3^{(t)}[u] - n_4^{(t)}[u]) \cdot \frac{1-r}{r} + (n_5^{(t)}[u] + n_6^{(t)}[u] - n_7^{(t)}[u] - n_8^{(t)}[u]) \cdot \frac{1-r^2}{r^2}, \ \forall t \in \{1, 2, ...\}, \ \forall u \in \mathcal{V}^{(t)}. \tag{6.35}$$

Proof. As in Section 6.7, for each time $t \in \{1, 2, ...\}$, let $X^{(t)}$ be the random number in $Bernoulli(r)$ drawn in line 11 of Algorithm 6.1 while the t-th element $e^{(t)}$ is processed. Then, from Lemma 6.4,

$$\{u, v\} \in \mathcal{S}^{(t)} \iff X^{(l_{uv}^{(t)})} = 1. \tag{6.36}$$

Now, for each $\tau = (\{u, v, w\}, s) \in \mathcal{A}^{(t)} \cup \mathcal{D}^{(t)}$, let γ_τ be the amount of change in each of $\bar{c}$, $c[u]$, $c[v]$, and $c[w]$ due to the discovery of τ in line 8 or line 9 of Algorithm 6.1. Let $\delta_\tau = +1$ when $\tau \in \mathcal{A}^{(t)}$, i.e. when the last edge is added, and let $\delta_\tau = -1$ when $\tau \in \mathcal{D}^{(t)}$, i.e. when the last edge is deleted. Let $\{u, v\}$ be the edge added or deleted at time s without loss of generality. Then, $\gamma_\tau = \frac{\delta_\tau}{r^2}$ if both $\{w, u\}, \{v, w\} \in \mathcal{S}^{(s)}$ and 0 otherwise. Hence, combined with Eq. (6.36),

$$\gamma_\tau = \frac{\delta_\tau}{r^2} X^{(\mathbf{1}_\tau)} X^{(\mathbf{2}_\tau)}. \tag{6.37}$$

Then from the definitions of γ_τ, $\bar{c}^{(t)} = \sum_{\tau \in \mathcal{A}^{(t)} \cup \mathcal{D}^{(t)}} \gamma_\tau$, and its variance is

$$Var[\bar{c}^{(t)}] = \sum_{\tau \in \mathcal{A}^{(t)} \cup \mathcal{D}^{(t)}} Var[\gamma_\tau] + \sum_{\tau \neq \omega \in \mathcal{A}^{(t)} \cup \mathcal{D}^{(t)}} Cov[\gamma_\tau, \gamma_\omega]. \tag{6.38}$$

For the variance term in Eq. (6.38), note first that applying that $X^{(\mathbf{1}_\tau)}$ and $X^{(\mathbf{2}_\tau)}$ are independent $Bernoulli(r)$ to Eq. (6.37) gives $\mathbb{E}[\gamma_\tau]$ as

$$\mathbb{E}[\gamma_\tau] = \mathbb{E}\left[\frac{\delta_\tau}{r^2} X^{(\mathbf{1}_\tau)} X^{(\mathbf{2}_\tau)}\right] = \frac{\delta_\tau}{r^2} \mathbb{E}\left[X^{(\mathbf{1}_\tau)}\right] \mathbb{E}\left[X^{(\mathbf{2}_\tau)}\right] = \delta_\tau. \tag{6.39}$$

Then, further applying $\delta_\tau^2 = 1$ and $(X^{(s)})^2 = X^{(s)}$ to Eq. (6.37) and again applying that $X^{(\mathbf{1}_\tau)}$ and $X^{(\mathbf{2}_\tau)}$ are independent $Bernoulli(r)$ give $Var[\gamma_\tau]$ as

$$Var[\gamma_\tau] = \mathbb{E}[\gamma_\tau^2] - (\mathbb{E}[\gamma_\tau])^2 = \mathbb{E}\left[\frac{\delta_\tau^2}{r^4} X^{(\mathbf{1}_\tau)} X^{(\mathbf{2}_\tau)}\right] - \delta_\tau^2 = \frac{1}{r^4} \mathbb{E}\left[X^{(\mathbf{1}_\tau)}\right] \mathbb{E}\left[X^{(\mathbf{2}_\tau)}\right] - 1 = \frac{1-r^2}{r^2}.$$

Hence the variance term in Eq. (6.38) is computed as

$$\sum_{\tau\in\mathcal{A}^{(t)}\cup\mathcal{D}^{(t)}} Var[\gamma_\tau] = \sum_{\tau\in\mathcal{A}^{(t)}\cup\mathcal{D}^{(t)}} \frac{1-r^2}{r^2} = (|\mathcal{A}^{(t)}| + |\mathcal{D}^{(t)}|)\cdot\frac{1-r^2}{r^2}. \tag{6.40}$$

For the covariance term in Eq. (6.38), applying Eq. (6.37) and Eq. (6.39) and using the fact that all the $X^{(s)}$'s are independent and identically distributed as $Bernoulli(r)$ and $(X^{(s)})^2 = X^{(s)}$ yield the $Cov[\gamma_\tau, \gamma_\omega]$ as

$$\begin{aligned} Cov[\gamma_\tau,\gamma_\omega] &= \mathbb{E}[\gamma_\tau\gamma_\omega] - \mathbb{E}[\gamma_\tau]\mathbb{E}[\gamma_\omega] = \mathbb{E}\left[\frac{\delta_\tau\delta_\omega}{r^4}X^{(\mathbf{1}_\tau)}X^{(\mathbf{2}_\tau)}X^{(\mathbf{1}_\omega)}X^{(\mathbf{2}_\omega)}\right] - \delta_\tau\delta_\omega \\ &= \delta_\tau\delta_\omega\left(\frac{1}{r^4}\mathbb{E}\left[\prod_{i\in\{\mathbf{1}_\gamma,\mathbf{2}_\gamma\}\cup\{\mathbf{1}_\omega,\mathbf{2}_\omega\}} X^{(i)}\right] - 1\right) \\ &= \delta_\tau\delta_\omega\left(\frac{1}{r^4}\prod_{i\in\{\mathbf{1}_\gamma,\mathbf{2}_\gamma\}\cup\{\mathbf{1}_\omega,\mathbf{2}_\omega\}} \mathbb{E}\left[X^{(i)}\right] - 1\right) = \delta_\tau\delta_\omega\left(\frac{r^{|\{\mathbf{1}_\gamma,\mathbf{2}_\gamma\}\cup\{\mathbf{1}_\omega,\mathbf{2}_\omega\}|}}{r^4} - 1\right). \end{aligned}$$

Then $\delta_\tau\delta_\omega = 1$ if $\tau,\omega\in\mathcal{A}^{(t)}$ or $\tau,\omega\in\mathcal{D}^{(t)}$, and $\delta_\tau\delta_\omega = -1$ if $\tau\in\mathcal{A}^{(t)}, \omega\in\mathcal{D}^{(t)}$ or $\tau\in\mathcal{D}^{(t)}, \omega\in\mathcal{A}^{(t)}$. Hence $Cov[\gamma_\tau,\gamma_\omega]$ can be calculated as

$$Cov[\gamma_\tau,\gamma_\omega] = \begin{cases} \frac{1-r}{r}, & \text{if } Type_{(\tau,\omega)} = 1 \text{ or } 2, \\ -\frac{1-r}{r}, & \text{if } Type_{(\tau,\omega)} = 3 \text{ or } 4, \\ \frac{1-r^2}{r^2}, & \text{if } Type_{(\tau,\omega)} = 5 \text{ or } 6, \\ -\frac{1-r^2}{r^2}, & \text{if } Type_{(\tau,\omega)} = 7 \text{ or } 8, \\ 0, & \text{if } Type_{(\tau,\omega)} = 9. \end{cases}$$

Hence the covariance term in Eq. (6.38) is computed as

$$\sum_{\tau\neq\omega\in\mathcal{A}^{(t)}\cup\mathcal{D}^{(t)}} Cov[\gamma_\tau,\gamma_\omega] = (n_1^{(t)}+n_2^{(t)}-n_3^{(t)}-n_4^{(t)})\cdot\frac{1-r}{r} + (n_5^{(t)}+n_6^{(t)}-n_7^{(t)}-n_8^{(t)})\cdot\frac{1-r^2}{r^2}. \tag{6.41}$$

Hence applying Eq. (6.40) and Eq. (6.41) to Eq. (6.38) gives

$$\begin{aligned} Var[\bar{c}^{(t)}] &= \sum_{\tau\in\mathcal{A}^{(t)}\cup\mathcal{D}^{(t)}} Var[\gamma_\tau] + \sum_{\tau\neq\omega\in\mathcal{A}^{(t)}\cup\mathcal{D}^{(t)}} Cov[\gamma_\tau,\gamma_\omega] \\ &= (|\mathcal{A}^{(t)}| + |\mathcal{D}^{(t)}|)\cdot\frac{1-r^2}{r^2} + (n_1^{(t)}+n_2^{(t)}-n_3^{(t)}-n_4^{(t)})\cdot\frac{1-r}{r} + (n_5^{(t)}+n_6^{(t)}-n_7^{(t)}-n_8^{(t)})\cdot\frac{1-r^2}{r^2}, \end{aligned}$$

which completes the proof of Eq. (6.34).

For Eq. (6.35), replacing $\bar{c}^{(t)}$ by $c^{(t)}[u]$, $\mathcal{A}^{(t)}$ by $\mathcal{A}^{(t)}[u]$, and $\mathcal{D}^{(s)}$ by $\mathcal{D}^{(s)}[u]$ and repeating above give a proof. ■

Chapter based on work that will appear at WWW 2019 [SGKR19].

Chapter 7

Summarizing Large Graphs

Given a terabyte-scale or larger graph distributed across multiple machines, how can we summarize it, with much fewer nodes and edges, so that we can restore the original graph exactly or within error bounds?

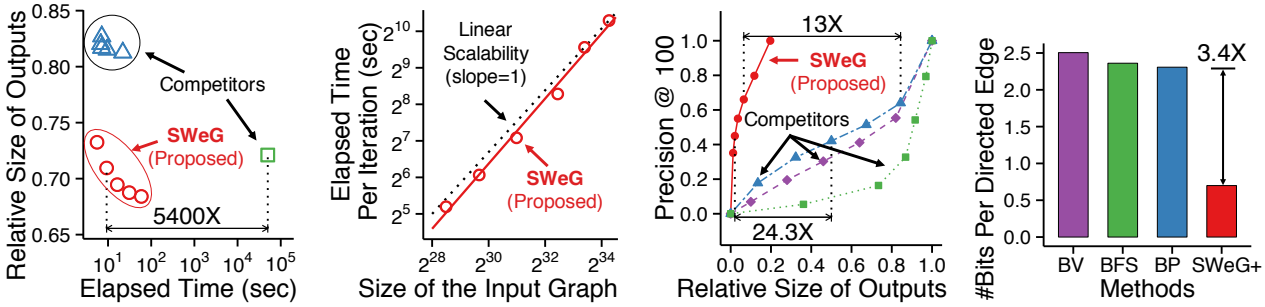
As large-scale graphs are ubiquitous, ranging from web graphs to online social networks, compactly representing graphs becomes important to efficiently store and process them. Given a graph, *graph summarization* aims to find its compact representation consisting of (a) a *summary graph* where the nodes are disjoint sets of nodes in the input graph, and each edge indicates the edges between all pairs of nodes in the two sets; and (b) *edge corrections* for restoring the input graph from the summary graph exactly or within error bounds. Although graph summarization is a widely-used graph-compression technique readily combinable with other techniques, existing algorithms for graph summarization are not satisfactory in terms of speed or compactness of outputs. More importantly, they assume that the input graph is small enough to fit in main memory.

We propose SWeG, a fast parallel algorithm for summarizing graphs with compact representations. SWeG is designed for not only shared-memory but also MapReduce settings to summarize graphs that are too large to fit in main memory. We demonstrate that SWeG is *(a) Fast:* SWeG is up to $\mathbf{5400}\times$ **faster** than its competitors that give similarly compact representations, *(b) Scalable:* SWeG scales to graphs with **tens of billions** of edges, and *(c) Compact:* combined with state-of-the-art graph-compression techniques, SWeG achieves up to $\mathbf{3.4}\times$ **better compression** than them.

7.1 Motivation

Large-scale graphs are everywhere. Graphs are natural representations of the web, social networks, collaboration networks, internet topologies, citation networks, to name just a few. The rapid growth of the web and its applications has led to large-scale graphs of unprecedented size, such as 3.5 billion web pages connected by 129 billion hyperlinks [MVLB14], online social networks with 300 billion connections [DKK+16], and 1.15 billion query-URL pairs [LGF09].

Consequently, representing graphs in a storage-efficient manner has become important. In addition to the reduction of hardware costs, compact representation may allow large-scale graphs to fit in main memory of one machine, eliminating expensive I/O over the network or to disk. Moreover, it can lead to performance gains by allowing large fractions of the graphs to reside in cache [BC08, SDB15].



(a) Fast (lossless summarization)

(b) Scalable (lossless summarization)

(c) Accurate (lossy summarization)

(d) Compact (lossless compression)

Figure 7.1: **Strengths of SWeG.** (a) *Fast:* SWeG is faster with more compact representations than state-of-the-art lossless-summarization methods. (b) *Scalable:* SWeG scales linearly with the size of the input graph, successfully scaling to graphs with over 20 billions edges. (c) *Accurate:* The lossy version of SWeG yields more compact and accurate representations than other lossy-summarization methods. (d) *Compact:* Combining SWeG and an advanced compression technique yields up to $3.4\times$ more compact representations than using the technique alone. See Section 7.5 for details.

To compactly represent graphs, a variety of graph-compression techniques have been developed, including relabeling nodes [BV04, AD09, DKK+16, CKL+09], employing integer-sequence encoding schemes (e.g., reference, gap, and interval encodings) [BV04], and encoding common structures (e.g., cliques, bipartite-cores, and stars) with fewer bits [KKVF14, BC08, RZ18]. The techniques are either *lossless* and *lossy* depending on whether the input graph can be reconstructed exactly from their outputs.

Graph summarization [NRS08] is a widely-used graph-compression technique. It aims to find a compact representation of a given graph $\mathcal{G}$ consisting of a summary graph and edge corrections (i.e., edges to be inserted, and edges to be deleted). The summary graph $\overline{\mathcal{G}}$ is a graph where the nodes are disjoint sets of nodes in $\mathcal{G}$ and each edge indicates the edges between all pairs of nodes in the two sets. The edge corrections are for restoring $\mathcal{G}$ from $\overline{\mathcal{G}}$ exactly (in cases of *lossless summarization*) or within given error bounds (in cases of *lossy summarization*). The outputs can be considered as three graphs: (a) the summary graph, (b) the graph consisting of the edges to be inserted, and (c) the graph consisting of the edges to be deleted. While we use the term "graph summarization" as defined above in this chapter, it also has been used for a wider range of problems, as discussed in Section 7.6.

As a graph-compression technique, graph summarization has the following desirable properties: **(a) Combinable with other compression techniques**: since graph summarization produces three graphs, as explained in the previous paragraph, each of them can be further compressed by any other graph compression technique, **(b) Adjustable**: the trade-off between compression rates and information loss can be adjusted by given error bounds, and **(c) Queryable**: neighbor queries (i.e., returning the neighboring nodes of a given node) can be processed efficiently on the outputs of graph summarization, as discussed in Section 7.8.

However, existing algorithms for graph summarization are not satisfactory in terms of speed [NRS08, SPH+18] or compactness of output representations [KNL15]. These serial algorithms [NRS08, KNL15, SPH+18] either have high computational complexity or significantly sacrifice compactness of outputs for lower complexity. More significantly, they assume that the input graph is small enough to fit in main memory. The largest graph used in the previous studies has just about 10 million edges [NRS08, KNL15, SPH+18]. As large-scale graphs often do not fit in main memory, improving the scalability of graph summarization remains an important challenge.

To address this challenge, we propose SWEG (**S**ummarizing **We**b-Scale **G**raphs), a fast parallel algorithm for summarizing large-scale graphs with compact representations. SWEG is designed for both shared-memory and MAPREDUCE settings. In a nutshell, SWEG repeatedly divides a given graph into small subgraphs and processes the subgraphs in parallel without having to load the entire graph in main memory. SWEG also adjusts, in each iteration, how aggressively it merges nodes within subgraphs, and this turns out to be crucial for obtaining compact representations.

Extensive experiments with thirteen real-world graphs show that SWEG significantly outperforms existing graph-summarization methods and enhances the best compression techniques, as shown in Figure 7.1. Specifically, SWEG provides the following advantages:

- **Fast**: SWEG is up to $5,400\times$ *faster* than its competitors that give similarly compact representations (Figure 7.1(a)).
- **Scalable**: SWEG scales to graphs with *tens of billions* of edges, showing near-linear data and machine scalability (Figure 7.1(b)).
- **Compact**: Combined with advanced compression methods, SWEG yields up to $3.4\times$ *better* compression than them, and it represents a web graph using about 0.7 bits per directed edge (Figure 7.1(d)).

The rest of this chapter is organized as follows. In Section 7.2, we introduce some preliminary concepts, notations, and a formal problem definition. In Section 7.3, we present SWEG, our proposed algorithm for graph summarization. In Section 7.4, we analyze the time and space complexity of SWEG. In Section 7.5, we share some experimental results. After discussing related work in Section 7.6, we provide a summary of this chapter in Section 7.7.

7.2 Preliminaries and Problem Definition

In this section, we first introduce some notations and concepts used throughout this chapter. Then, we define the problem of large-scale graph summarization.

Table 7.1: **Table of frequently-used symbols.**

Symbol	Definition
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	input graph with nodes $\mathcal{V}$ and edges $\mathcal{E}$
$\mathcal{N}_v$	set of neighbors of node v in $\mathcal{G}$
$\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$	edge corrections (i.e., edges insertions and deletions)
$\mathcal{C}^+$	set of edges to be inserted
$\mathcal{C}^-$	set of edges to be deleted
$\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$	summary graph with supernodes $\mathcal{S}$ and superedges $\mathcal{P}$
$\mathcal{P}^*$	non-loop superedges in $\mathcal{P}$
$\hat{\mathcal{G}} = (\mathcal{V}, \hat{\mathcal{E}})$	graph restored from $\overline{\mathcal{G}}$ and $\mathcal{C}$
$\hat{\mathcal{N}}_v$	set of neighbors of node v in $\hat{\mathcal{G}}$
ϵ	error bound
T	number of iterations
$\theta(t)$	merging threshold in the t-th iteration

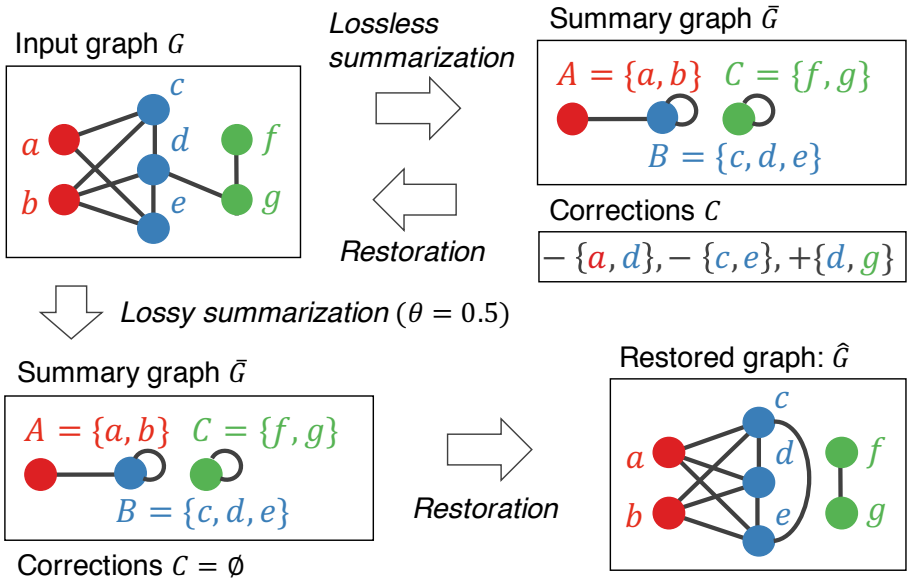


Figure 7.2: **Illustration of graph summarization.** The lossless summarization of the input graph (upper left) yields a summary graph and corrections (upper right) from which the input graph is restored exactly. The lossy summarization of the input graph (upper left) yields a summary graph and corrections (lower left). The restored graph (lower right) satisfies Eq. (7.2). Note that the outputs of lossless and lossy graph summarization have fewer edges than the input graph.

7.2.1 Notations and Concepts

See Table 7.1 for frequently-used symbols and Figure 7.2 for an illustration of concepts. Consider a simple undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with nodes $\mathcal{V}$ and edges $\mathcal{E}$. We denote each node in $\mathcal{V}$ by a lowercase (e.g., v) and each edge in $\mathcal{E}$ by an unordered pair (e.g., $\{u, v\}$). We denote the set of neighbors of a node $v \in \mathcal{V}$ in $\mathcal{G}$ by $\mathcal{N}_v \subset \mathcal{V}$.

A *summary graph* of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, denoted by $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, is a graph whose nodes are a partition of $\mathcal{V}$ (i.e., disjoint subsets of $\mathcal{V}$ whose union is $\mathcal{V}$). That is, each node $v \in \mathcal{V}$ belongs to exactly one node in $\mathcal{S}$. We call nodes and edges in $\overline{\mathcal{G}}$ *supernodes* and *superedges*; and we denote each supernode by an uppercase (i.e., A). Summary graphs may have self-loops, and we use $\mathcal{P}^* \subset \mathcal{P}$ to denote the set of non-loop superedges. See Figure 7.2 for example summary graphs.

Given a summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and *corrections* $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$, where $\mathcal{C}^+$ denotes the set of edges to be inserted and $\mathcal{C}^-$ denotes the set of edges to be deleted, a graph $\hat{\mathcal{G}} = (\mathcal{V}, \hat{\mathcal{E}})$, which we call a *restored graph*, is recreated by the following steps:

1. For each superedge $\{A, B\} \in \mathcal{P}$, all pairs of distinct nodes in A and B (i.e.,. $\{\{u, v\} : u \in A, v \in B, u \neq v\}$) are added to $\hat{\mathcal{E}}$,
2. Each edge in $\mathcal{C}^+$ is added to $\hat{\mathcal{E}}$,
3. Each edge in $\mathcal{C}^-$ is removed from $\hat{\mathcal{E}}$.

See Figure 7.2 for example restored graphs. We denote the set of neighbors of a node $v \in \mathcal{V}$ in $\hat{\mathcal{G}}$ by $\hat{\mathcal{N}}_v \subset \mathcal{V}$. Given $\overline{\mathcal{G}}$ and $\mathcal{C}$, neighbor queries (i.e., returning $\hat{\mathcal{N}}_v$ for a given node $v \in \mathcal{V}$) can be processed efficiently without restoring entire $\hat{\mathcal{G}}$, as described in Section 7.8.

7.2.2 Problem Definition

The large-scale graph summarization problem, which we address in this chapter, is defined in Problem 7.1.

Problem 7.1: Large-scale Graph Summarization

1. **Given**:
   - a large-scale graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, which may or may not fit in main memory
   - an error bound ϵ (≥ 0)
2. **Find:** a summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and corrections $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$
3. **to Minimize:**
$$|\mathcal{P}^*| + |\mathcal{C}^+| + |\mathcal{C}^-| \tag{7.1}$$
4. **Subject to:** the restored graph $\hat{\mathcal{G}} = (\mathcal{V}, \hat{\mathcal{E}})$ satisfies
$$|\mathcal{N}_v - \hat{\mathcal{N}}_v| + |\hat{\mathcal{N}}_v - \mathcal{N}_v| \leq \epsilon|\mathcal{N}_v|, \ \forall v \in \mathcal{V}. \tag{7.2}$$

The objective (Eq. (7.1)), which we aim to minimize, measures the size of the output representation by the count of non-loop (super) edges. We exclude the self-loops in $\mathcal{P}$ from the objective since they can be encoded concisely using 1 bit per supernode regardless of their count. The constraints (Eq. (7.2)) [NRS08] states that the neighbors $\mathcal{N}_v$ of each node $v \in \mathcal{V}$ in the input graph and the node's neighbors $\hat{\mathcal{N}}_v$ in the restored graph $\hat{\mathcal{G}}$ should be similar enough so that the size of their symmetric difference (i.e., $(\mathcal{N}_v \cup \hat{\mathcal{N}}_v) - (\hat{\mathcal{N}}_v \cap \mathcal{N}_v)$) is at most a certain proportion of the size of the node's neighbors $\mathcal{N}_v$ in the input graph. The proportion is given as a parameter ϵ, which we call an *error bound*. The error bound ϵ controls the trade-off between compression rates and the amount of information loss. If $\epsilon = 0$, then $\mathcal{N}_v = \hat{\mathcal{N}}_v$ holds for every node $v \in \mathcal{V}$ and thus the restored graph $\hat{\mathcal{G}}$ equals to the input graph $\mathcal{G}$. Thus, we call Problem 7.1 *large-scale lossless graph summarization problem* if $\epsilon = 0$ and *large-scale lossy graph summarization problem* if $\epsilon > 0$. See Figure 7.2 for the lossless and lossy summarization of a toy graph.

7.3 Proposed Algorithm: SWEG

We present our proposed algorithm, SWEG (**S**ummarizing **We**b-Scale **G**raphs). SWEG provides approximate solutions to the lossless and lossy graph summarization problems (i.e., Problems 7.1). In Section 7.3.1, we provide an overview of SWEG. In Section 7.3.2, we describe each step of SWEG in detail. In Sections 7.3.3 and 7.3.4, we discuss parallelizing SWEG in shared-memory and MAPREDUCE settings. In Section 7.3.5, we present SWEG+, an algorithm for further compression.

7.3.1 Overview

We give an overview of SWEG. SWEG, described in Algorithm 7.1, requires an input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the number of iterations T, and an error bound ϵ, as in Problem 7.1. SWEG first initializes the set $\mathcal{S}$ of supernodes so that (a) each supernode consists of a node in $\mathcal{V}$ and (b) every node in $\mathcal{V}$ belongs to one supernode (line 1). Then, SWEG updates $\mathcal{S}$ by repeating the following steps T times (line 2):

Algorithm 7.1 Overview of SWEG

Input: input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of iterations T, error bound ϵ
Output: summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, corrections $\mathcal{C}$

1: initialize supernodes $\mathcal{S}$ to $\{\{v\} : v \in \mathcal{V}\}$
2: **for** $t = 1...T$ **do**
3: divide $\mathcal{S}$ into disjoint groups ▷ Algorithm 7.2
4: merge some supernodes within each group ▷ Algorithm 7.3
5: encode edges $\mathcal{E}$ into superedges $\mathcal{P}$ and corrections $\mathcal{C}$ ▷ Algorithm 7.4
6: **if** $\epsilon > 0$ **then** drop some (super) edges from $\mathcal{P}$ and $\mathcal{C}$ ▷ Algorithm 7.5
7: **return** $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and $\mathcal{C}$

- **Dividing step (line 3, Section 7.3.2.1)**: divides $\mathcal{S}$ into disjoint groups, each of which is composed of supernodes with similar connectivity. Different groups are obtained in each iteration.
- **Merging step (line 4, Section 7.3.2.2)**: merges some supernodes within each group in a greedy manner.

Dividing $\mathcal{S}$ into small groups makes SWEG faster and more memory efficient by allowing it to process different groups in parallel without having to load entire $\mathcal{G}$ in memory (see Sections 7.3.3 and 7.3.4). After updating $\mathcal{S}$, SWEG performs the following steps:

- **Encoding step (line 5, Section 7.3.2.3)**: encodes the edges $\mathcal{E}$ into superedges $\mathcal{P}$ and corrections $\mathcal{C}$ so that the count of non-loop (super) edges (i.e., Eq. (7.1)) is minimized given supernodes $\mathcal{S}$.
- **Dropping step (line 6, Section 7.3.2.4)**: makes $\mathcal{P}$ and $\mathcal{C}$ more compact by dropping some (super) edges from them within the error bounds (i.e., Eq. (7.2)) in cases of lossy summarization.

Lastly, SWEG returns the summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and the corrections $\mathcal{C}$ as its outputs (line 7).

7.3.2 Detailed Description

We describe each step of SWEG in detail. For simplicity, in this subsection, we assume that SWEG is executed serially and the input graph is small enough to fit in main memory of one machine. Parallelization and distributed processing are discussed in later subsections.

7.3.2.1 Dividing Step (Algorithm 7.2)

The dividing step aims to divide the supernodes $\mathcal{S}$ into disjoint groups of supernodes with similar connectivity. To this end, we adapt a shingle-based method [BGMZ97] since it is efficiently parallelized in shared-memory and MAPREDUCE settings (see Sections 7.3.3 and 7.3.4), while any node clustering methods, such as node-embedding [WCW+17], spectral [PSS+10], and cut-based [KK98] methods, can be used instead. Specifically, we extend the shingle of nodes, for which it is known that two nodes have the same shingle with probability equal to the jaccard similarity of their neighbor sets [BCFM00], to supernodes. We define the *shingle* $F(A)$ of each supernode $A \in \mathcal{S}$ as the smallest shingle of the nodes in A. Then, two supernodes $A \neq B \in \mathcal{S}$ are more likely to have the same shingle if the nodes in A and those in B have similar connectivity. Formally, given a random bijective hash function $h : \mathcal{V} \to \{1, ..., |\mathcal{V}|\}$,

$$F(A) := \min_{v \in A}(f(v)),$$

Algorithm 7.2 Dividing Step of SWEG

Input: input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, current supernodes $\mathcal{S}$
Output: disjoint groups of supernodes: $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(k)}\}$
1: generate a random bijective hash function $h : \mathcal{V} \to \{1, ..., |\mathcal{V}|\}$
2: **for each** supernode $A \in \mathcal{S}$ **do**
3: **for each** node $v \in A$ **do**
4: $f(v) \leftarrow \min(\{h(u) : u \in N_v \text{ or } u = v\})$ ▷ shingle of node v
5: $F(A) \leftarrow \min(\{f(v) : v \in A\})$ ▷ shingle of supernode A
6: divide the supernodes in $\mathcal{S}$ into $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(k)}\}$ by their $F(\cdot)$ value
7: **return** $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(k)}\}$

where $f(v) := \min_{u \in N_v \text{ or } u=v} h(u)$ is the shingle of node $v \in \mathcal{V}$, and N_v is the set of neighbors of node $v \in \mathcal{V}$ in the input graph $\mathcal{G}$. SWEG generates such a hash function h by shuffling the order of the nodes in $\mathcal{V}$ and mapping each i-th node to i (line 1 of Algorithm 7.2) and computes the shingle of every supernode in $\mathcal{S}$ (lines 2-5). Lastly, SWEG divides the supernodes $\mathcal{S}$ into disjoint groups $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(k)}\}$ by their shingles and returns the groups (lines 6-7). In cases where large groups exist, Algorithm 7.2 can be applied recursively to each of the large groups with a new hash function h.

7.3.2.2 Merging Step (Algorithm 7.3)

To describe this step, we first define several concepts. We define the *cost* of a supernode $A \in \mathcal{S}$ given supernodes $\mathcal{S}$, denoted by $Cost(A, \mathcal{S})$, as the amount of increase in Eq. (7.1) (i.e., the count of non-loop (super) edges in outputs) due to the edges adjacent to any node in A (see the encoding step in Section 7.3.2.3). Then, we define the *saving* due to the merger between supernodes $A \neq B \in \mathcal{S}$ given supernodes $\mathcal{S}$ as

$$Saving(A, B, \mathcal{S}) := 1 - \frac{Cost(A \cup B, (\mathcal{S} - \{A, B\}) \cup \{A \cup B\})}{Cost(A, \mathcal{S}) + Cost(B, \mathcal{S})}, \tag{7.3}$$

where $Cost(A, \mathcal{S}) + Cost(B, \mathcal{S})$ is the cost of A and B before their merger, and $Cost(A \cup B, (\mathcal{S} - \{A, B\}) \cup \{A \cup B\})$ is their cost after their merger. That is, $Saving(A, B, \mathcal{S})$ is the ratio of the cost reduction due to the merger and the cost before the merger. Lastly, we define the *supernode jaccard similarity* between supernodes $A \neq B \in \mathcal{S}$ as

$$SuperJaccard(A, B) := \frac{\sum_{v \in N_A \cup N_B} \min(w(A, v), w(B, v))}{\sum_{v \in N_A \cup N_B} \max(w(A, v), w(B, v))}, \tag{7.4}$$

where $N_A := \bigcup_{v \in A} N_v$ is the set of nodes adjacent to any node in supernode $A \in \mathcal{S}$, and $w(A, v) := |\{u \in A : \{u, v\} \in \mathcal{E}\}|$ is the number of nodes in supernode $A \in \mathcal{S}$ adjacent to node $v \in \mathcal{V}$. $SuperJaccard(A, B)$ measures the similarity of A and B in terms of their connectivity. Notice that it is 1 if A and B have the same connectivity (i.e., $w(A, v) = w(B, v)$ for every $v \in N_A \cup N_B$) and it is 0 if their nodes have no common neighbors (i.e., $N_A \cap N_B = \emptyset$).

Given disjoint groups of supernodes $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(k)}\}$, the merging step merges some pairs of supernodes within each group $\mathcal{S}^{(i)}$ in a greedy manner, as described in Algorithm 7.3. Notice that, in line 5, SWEG uses $SuperJaccard(A, B)$ instead of $Saving(A, B, \mathcal{S})$, which is a more straightforward choice, to find a candidate pair of supernodes $A \neq B \in \mathcal{S}^{(i)}$. This is because (a) $SuperJaccard(A, B)$ is

Algorithm 7.3 Merging Step of SWEG

Input: input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, current supernodes $\mathcal{S}$,
current iteration t, disjoint groups of supernodes $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(k)}\}$
Output: updated supernodes $\mathcal{S}$

1: **for each** group $\mathcal{S}^{(i)} \in \{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(k)}\}$ **do**
2: $\quad\mathcal{Q} \leftarrow \mathcal{S}^{(i)}$
3: $\quad$**while** $|\mathcal{Q}| > 1$ **do**
4: $\quad\quad$pick and remove a random supernode A from $\mathcal{Q}$
5: $\quad\quad B \leftarrow \arg\max_{C \in \mathcal{Q}} \mathit{SuperJaccard}(A, C)$ ▷ Eq. (7.4)
6: $\quad\quad$**if** $\mathit{Saving}(A, B, \mathcal{S}) \geq \theta(t)$ **then** ▷ Eq. (7.3) and Eq. (7.5)
7: $\quad\quad\quad\mathcal{S} \leftarrow (\mathcal{S} - \{A, B\}) \cup \{A \cup B\}$ ▷ merge A and B
8: $\quad\quad\quad\mathcal{S}^{(i)} \leftarrow (\mathcal{S}^{(i)} - \{A, B\}) \cup \{A \cup B\}$
9: $\quad\quad\quad\mathcal{Q} \leftarrow (\mathcal{Q} - \{B\}) \cup \{A \cup B\}$ ▷ replace B with $A \cup B$
10: **return** $\mathcal{S}$

Algorithm 7.4 Encoding Step of SWEG

Input: input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, supernodes $\mathcal{S}$
Output: summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, corrections $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$

1: $\mathcal{P} \leftarrow \emptyset$; $\mathcal{C}^+ \leftarrow \emptyset$; $\mathcal{C}^- \leftarrow \emptyset$;
2: **for each** supernode $A \in \mathcal{S}$ **do**
3: $\quad$**for each** supernode B $(\neq A)$ where $\mathcal{E}_{AB} \neq \emptyset$ **do** ▷ Eq. (7.6)
4: $\quad\quad$**if** $\mathcal{E}_{AB} \leq \frac{|A| \cdot |B|}{2}$ **then** $\mathcal{C}^+ \leftarrow \mathcal{C}^+ \cup \mathcal{E}_{AB}$
5: $\quad\quad$**else** $\mathcal{P} \leftarrow \mathcal{P} \cup \{\{A, B\}\}$; $\mathcal{C}^- \leftarrow \mathcal{C}^- \cup (\pi_{AB} - \mathcal{E}_{AB})$ ▷ Eq. (7.7)
6: $\quad$**if** $\mathcal{E}_{AA} \leq \frac{|A| \cdot (|A|-1)}{4}$ **then** $\mathcal{C}^+ \leftarrow \mathcal{C}^+ \cup \mathcal{E}_{AA}$ ▷ Eq. (7.8)
7: $\quad$**else** $\mathcal{P} \leftarrow \mathcal{P} \cup \{\{A, A\}\}$; $\mathcal{C}^- \leftarrow \mathcal{C}^- \cup (\pi_{AA} - \mathcal{E}_{AA})$ ▷ Eq. (7.9)
8: **return** $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$

cheaper to compute than $\mathit{Saving}(A, B, \mathcal{S})$ and (b) intuitively, $\mathit{Saving}(A, B, \mathcal{S})$ tends to be high when A and B have similar connectivity. Computing $\mathit{SuperJaccard}(A, C)$ instead of $\mathit{Saving}(A, C, \mathcal{S})$ for every supernode $C \in \mathcal{Q}$ in line 5 leads to a significant improvement in speed with small loss in the compactness of outputs, as shown empirically in Section 7.5.2. In line 6, the *merging threshold* $\theta(t)$ is set as in Eq. (7.5) so that SWEG gradually shifts from exploration (of supernodes in the other groups) to exploitation (of supernodes in the same group).

$$\theta(t) := \begin{cases} (1+t)^{-1} & \text{if } t < T, \\ 0 & \text{if } t = T. \end{cases} \tag{7.5}$$

This decreasing threshold is crucial for obtaining compact output representations, as shown empirically in Section 7.5.2.

7.3.2.3 Encoding Step (Algorithm 7.4)

Given the supernodes $\mathcal{S}$ from the previous steps, the encoding step encodes the edges $\mathcal{E}$ of the input graph into superedges $\mathcal{P}$ and corrections $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$. We first describe how to encode edges that connect different supernodes (lines 3-5 of Algorithm 7.4). For each supernode pair $A \neq B \in \mathcal{S}$, we let

Algorithm 7.5 Dropping Step of SWEG (Optional)

Input: input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, error bound ϵ
summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, corrections $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$
Output: updated summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$,
updated corrections $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$

1: $c_v \leftarrow \epsilon \cdot |N_v|$ for each node $v \in \mathcal{V}$ ▷ change limits of nodes
2: **for each** edge $\{u, v\} \in \mathcal{C}^+$ **do**
3: **if** $c_u \geq 1$ and $c_v \geq 1$ **then**
4: $\mathcal{C}^+ \leftarrow \mathcal{C}^+ - \{\{u, v\}\}$; $c_u \leftarrow c_u - 1$; $c_v \leftarrow c_v - 1$
5: **for each** edge $\{u, v\} \in \mathcal{C}^-$ **do**
6: **if** $c_u \geq 1$ and $c_v \geq 1$ **then**
7: $\mathcal{C}^- \leftarrow \mathcal{C}^- - \{\{u, v\}\}$; $c_u \leftarrow c_u - 1$; $c_v \leftarrow c_v - 1$
8: **for each** superedge $\{A, B\} \in \mathcal{P}$ in the increasing order of $|A| \cdot |B|$ **do**
9: **if** $A \neq B$ and $(\forall v \in A, c_v \geq |B|)$ and $(\forall v \in B, c_v \geq |A|)$ **then**
10: $\mathcal{P} \leftarrow \mathcal{P} - \{\{A, B\}\}$;
11: **for each** $v \in A$ **do** $c_v \leftarrow c_v - |B|$
12: **for each** $v \in B$ **do** $c_v \leftarrow c_v - |A|$
13: **return** $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$

$\mathcal{E}_{AB} \subset \mathcal{E}$ be the set of edges connecting A and B, and π_{AB} be the set of all pairs of nodes in A and B. That is,

$$\mathcal{E}_{AB} := \{\{u, v\} \subset \mathcal{V} : u \in A, v \in B, \{u, v\} \in \mathcal{E}\}, \tag{7.6}$$
$$\pi_{AB} := \{\{u, v\} \subset \mathcal{V} : u \in A, v \in B\}. \tag{7.7}$$

Recall how a graph is restored from $\mathcal{P}$ and $\mathcal{C}$ in Section 7.2. Then, the two options to encode the edges in $\mathcal{E}_{AB}$ are as follows:

(a) *without superedges*: merge $\mathcal{E}_{AB}$ into $\mathcal{C}^+$,

(b) *with a superedge*: add $\{A, B\}$ to $\mathcal{P}$; merge $(\pi_{AB} - \mathcal{E}_{AB})$ into $\mathcal{C}^-$.

Since (a) and (b) increase our objective (i.e., Eq. (7.1)) by $|\mathcal{E}_{AB}|$ and $(1 + |\pi_{AB}| - |\mathcal{E}_{AB}|)$, respectively, SWEG chooses (a) if $|\mathcal{E}_{AB}| \leq |\pi_{AB}|/2 = |A| \cdot |B|/2$ (line 4). Otherwise, it chooses (b) (line 5).

SWEG encodes edges within each supernode in a similar manner (lines 6-7). For each supernode $A \in \mathcal{S}$, we let $\mathcal{E}_{AA} \subset \mathcal{E}$ be the set of edges between nodes within A, and π_{AA} be the set of all pairs of distinct nodes within A. That is,

$$\mathcal{E}_{AA} := \{\{u, v\} \subset \mathcal{V} : u \neq v \in A, \{u, v\} \in \mathcal{E}\}, \tag{7.8}$$
$$\pi_{AA} := \{\{u, v\} \subset \mathcal{V} : u \neq v \in A\}. \tag{7.9}$$

Then, the two options to encode the edges in $\mathcal{E}_{AA}$ are as follows:

(c) *without superloops*: merge $\mathcal{E}_{AA}$ into $\mathcal{C}^+$,

(d) *with a superloop*: add $\{A, A\}$ to $\mathcal{P}$; merge $(\pi_{AA} - \mathcal{E}_{AA})$ into $\mathcal{C}^-$.

Since (c) and (d) increase our objective (i.e., Eq. (7.1)) by $|\mathcal{E}_{AA}|$ and $(|\pi_{AA}| - |\mathcal{E}_{AA}|)$, respectively, SWEG chooses (c) if $|\mathcal{E}_{AA}| \leq |\pi_{AA}|/2 = |A| \cdot (|A| - 1)/4$ (line 6). Otherwise, it chooses (d) (line 7).

7.3.2.4 Dropping Step (Algorithm 7.5)

The dropping step is an optional step for lossy summarization (i.e., when $\epsilon > 0$). This step is skipped if lossless summarization is needed (i.e., when $\epsilon = 0$). Given the summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and corrections $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$ from the previous step, the dropping step drops some (super) edges from $\mathcal{P}$ and $\mathcal{C}$ to make the output representation more compact (i.e., to further reduce our objective Eq. (7.1)) without changing more than ϵ of the neighbors of each node (i.e., within the error bounds given in Eq. (7.2)). SWEG first initializes the *change limit* c_v of each node $v \in \mathcal{V}$ to $\epsilon \cdot |N_v|$, which is the right-hand side of Eq. (7.2) (line 1 of Algorithm 7.5). Then, within the change limit of each node, SWEG drops some adjacent edges from $\mathcal{C}^+$, $\mathcal{C}^-$, and $\mathcal{P}$, as described in lines 2-4, lines 5-7, and lines 8-12, respectively. Notice that, when a superedge $\{A, B\}$ is dropped, the total decrement in the change limits is proportional to $|A| \cdot |B|$, which is used to sort the superedges in line 8. Lastly, SWEG returns the updated summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and corrections $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$, which satisfy the error bounds given in Eq. (7.2), as shown in Theorem 7.1 (line 13).

Theorem 7.1: Error Bounds

Given an input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, and corrections $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$ satisfying that the restored graph $\hat{\mathcal{G}}$ is equal to $\mathcal{G}$, Algorithm 7.5 returns $\overline{\mathcal{G}}$ and $\mathcal{C}$ that satisfy Eq. (7.2).

Proof. From $\hat{\mathcal{G}} = \mathcal{G}$, $\hat{N}_v = N_v$ holds for every node $v \in \mathcal{V}$ at the beginning of the algorithm. Thus, after the change limit c_v of each node $v \in \mathcal{V}$ is initialized to $\epsilon \cdot |N_v|$ in line 1, Eq. (7.10) holds.

$$c_v \leq \epsilon \cdot |N_v| - |\hat{N}_v - N_v| - |N_v - \hat{N}_v|, \forall v \in \mathcal{V}. \tag{7.10}$$

Recall how $\hat{\mathcal{G}}$ is constructed from $\overline{\mathcal{G}}$ and $\mathcal{C}$ in Section 7.2. Eq. (7.10) still holds after $\mathcal{C}^+$ is processed in lines 2-4 since dropping an edge from $\mathcal{C}^+$ decreases c_v by 1, increases $|N_v - \hat{N}_v|$ by at most 1, and keeps $|\hat{N}_v - N_v|$ the same for each adjacent node $v \in \mathcal{V}$. Likewise, Eq. (7.10) still holds after $\mathcal{C}^-$ is processed in lines 5-7 since dropping an edge in $\mathcal{C}^-$ decreases c_v by 1, increases $|\hat{N}_v - N_v|$ by at most 1, and keeps $|N_v - \hat{N}_v|$ the same for each adjacent node $v \in \mathcal{V}$. Similarly, we can show that Eq. (7.10) still holds after $\mathcal{P}$ is processed in lines 8-12. Eq. (7.11) is enforced by lines 3, 6, and 9.

$$c_v \geq 0, \forall v \in \mathcal{V}. \tag{7.11}$$

Eq. (7.10) and Eq. (7.11) imply Eq. (7.2). ■

7.3.3 Parallelization in Shared Memory

We describe how each step of SWEG (Algorithm 7.1) is parallelized in shared-memory environments. In the dividing step (Algorithm 7.2), the supernodes in line 2 are processed independently in parallel. In the merging step (Algorithm 7.3), the groups of supernodes in line 1 are processed in parallel. In lines 6 and 7, the accesses and updates of $\mathcal{S}$ are synchronized. In the encoding step (Algorithm 7.4), the supernodes in line 2 are processed independently in parallel. Specifically, each thread has its own copies of $\mathcal{P}$, $\mathcal{C}^+$, and $\mathcal{C}^-$; and the copies are merged once, after all supernodes are processed. Lastly, in the dropping step (line 7.5), parallel merge sort [Lea00] is used when sorting $\mathcal{P}$ in line 8. Although the other parts of the dropping step are executed serially, they take a negligible portion of the total

execution time because they are executed only once, while the dividing and merging steps are repeated multiple times.

7.3.4 Distributed Processing with MAPREDUCE

We describe how each step of SWEG is implemented in the MAPREDUCE framework for large-scale graphs not fitting in main memory. We assume that the input graph $\mathcal{G}$ is stored in a file in a distributed file system where each record describes the nodes in a supernode and the neighbors of the nodes in $\mathcal{G}$. Specifically, the record $R(A)$ for a supernode $A \in \mathcal{S}$ is in the following format:

$$R(A) := (\text{id of } A, |A|, \underbrace{(u, |N_u|, (\overbrace{v, ..., w}^{|N_u|})), ..., (x, |N_x|, (\overbrace{y, ..., z}^{|N_x|}))}_{|A|}), \tag{7.12}$$

where $(u, |N_u|, (v, ..., w))$ describes the degree and the neighbors of node $u \in A$.

Dividing and Merging Steps: First, each iteration of the dividing and merging steps (Algorithms 7.2 and 7.3) is performed by the following MAPREDUCE job:

- **Map-1**: The hash function h is broadcast to the mappers. Each mapper repeats taking a record $R(A)$, computing $F(A)$ (lines 3-5 of Algorithm 7.2), and emitting $< F(A), R(A) >$.
- **Reduce-1:** The supernodes $\mathcal{S}$ are broadcast to the reducers. Each reducer repeats taking $\{R(A)|A \in \mathcal{S}^{(i)}\}$ for a group $\mathcal{S}^{(i)}$, updating $\mathcal{S}^{(i)}$ (lines 2-9 of Algorithm 7.3), and emitting $R(A)$ for each supernode A in the updated $\mathcal{S}^{(i)}$. In the end, each reducer writes the updates in $\mathcal{S}$ to the distributed file system.

Note that updates in $\mathcal{S}$ are not shared among the reducers during the reduce stage. However, in our experiments, the effect of this lazy synchronization on the compactness of output representations was negligible regardless of the number of reducers.

Encoding Step: Next, the following map-only job performs the encoding step (Algorithm 7.4):

- **Map-2:** The supernodes $\mathcal{S}$ are broadcast to the mappers. Each mapper repeats taking a record $R(A)$, encoding the edges adjacent to any node in A (lines 3-7 of Algorithm 7.4), and emitting the new (super) edges in $\mathcal{P}$, $\mathcal{C}^+$, and $\mathcal{C}^-$. Different output paths are used for $\mathcal{P}$, $\mathcal{C}^+$, and $\mathcal{C}^-$.

Dropping Step: Lastly, for the dropping step (Algorithm 7.5), Map-3 initializes the change limit of each node (line 1); and Map-4 and Reduce-4 sort the superedges in $\mathcal{P}$ (line 9).

- **Map-3**: Each mapper repeats taking a record $R(A)$ and emitting $< v, \epsilon \cdot |N_v| >$ for each node $v \in A$.
- **Map-4**: The input file, which is an output of Reduce-2, lists the superedges in $\mathcal{P}$. Each mapper repeats taking a superedge $\{A, B\}$ and emitting $< (|A| \cdot |B|, \{A, B\}), \emptyset >$.
- **Reduce-4**: A single reducer repeats taking a superedge $\{A, B\} \in \mathcal{P}$ and emitting it. The superedges are sorted in the shuffle stage.

The other parts of the dropping step are processed serially. Specifically, after loading the nodes' change limits (the output of Map-3) in memory, our implementation repeats reading a (super) edge in $\mathcal{C}^+$ (an output of Map-2), $\mathcal{C}^-$ (an output of Map-2), and $\mathcal{P}$ (the output of Reduce-4) and writing the (super) edge to the output file if it is not dropped. Note that entire $\mathcal{C}^+$, $\mathcal{C}^-$, or $\mathcal{P}$ is not loaded in memory at once. These serial parts take a small portion of the total execution time because they are executed only once, while the dividing and merging steps are repeated multiple times.

7.3.5 Further Compression: SWEG+

We propose SWEG+, an algorithm for further compression. The outputs of SWEG (i.e., $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$) can be represented as three graphs: $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, $\mathcal{G}^+ = (\mathcal{V}, \mathcal{C}^+)$, and $\mathcal{G}^- = (\mathcal{V}, \mathcal{C}^-)$. Thus, as described in Algorithm 7.6, SWEG+ further compresses each of the graphs using a given graph-compression algorithm ALG. If ALG relabels nodes, to keep the labels of $\mathcal{V}$ in $\mathcal{G}^+$ and $\mathcal{G}^-$ the same, the labels of $\mathcal{V}$ obtained when compressing $\mathcal{G}^+$ are also used for $\mathcal{G}^-$, which is usually much smaller than $\mathcal{G}^+$.

Any graph-compression method, such as [DKK+16, BV04, AD09, CKL+09, BC08, RZ18], can be used as ALG depending on the objectives of compression. In Section 7.5.6, we empirically show that for many graph-compression methods, SWEG+ gives significantly more compact representations than directly compressing the input graph using the methods.

Algorithm 7.6 SWEG+: Algorithm for Further Compression

Input: input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, number of iterations T, error bound ϵ
graph-compression method ALG
Output: compressed $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, $\mathcal{G}^+ = (\mathcal{V}, \mathcal{C}^+)$, and $\mathcal{G}^- = (\mathcal{V}, \mathcal{C}^-)$
1: run SWEG to get $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$ ▷ Algorithm 7.1
2: run ALG on each of $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, $\mathcal{G}^+ = (\mathcal{V}, \mathcal{C}^+)$, and $\mathcal{G}^- = (\mathcal{V}, \mathcal{C}^-)$
3: **return** the compressed $\overline{\mathcal{G}}$, $\mathcal{G}^+$, and $\mathcal{G}^-$

7.4 Theoretical Analysis

We analyze the time complexity and memory requirements of SWEG. To this end, we let $\mathcal{E}_A := \{\{u, v\} \in \mathcal{E} : u \in A\}$ be the set of edges adjacent to any node in supernode $A \in \mathcal{S}$ and $\mathcal{E}^{(i)} := \bigcup_{A \in \mathcal{S}^{(i)}} \mathcal{E}_A$ be the set of edges adjacent to any node in any supernode in group $\mathcal{S}^{(i)}$. We also let k be the number of groups of supernodes from the dividing step. Since the groups are disjoint,

$$\sum_{i=1}^{k} |\mathcal{E}^{(i)}| \leq \sum_{A \in \mathcal{S}} |\mathcal{E}_A| \leq 2|\mathcal{E}|. \tag{7.13}$$

Since the encoding step (Algorithm 7.4) yields outputs with no more (super) edges than ($\mathcal{P} \leftarrow \emptyset, \mathcal{C}^+ \leftarrow \mathcal{E}, \mathcal{C}^- \leftarrow \emptyset$),

$$|\mathcal{C}^+| + |\mathcal{C}^-| + |\mathcal{P}| \leq |\mathcal{E}|. \tag{7.14}$$

Lastly, we assume that $|\mathcal{V}| \leq |\mathcal{E}|$, for simplicity.

7.4.1 Time Complexity Analysis

The dividing step (Algorithm 7.2) takes $O(|\mathcal{E}|)$ because its computational bottleneck is to compute the shingles of all supernodes (lines 2-5), which requires accessing every edge in $\mathcal{E}$ twice. The merging step (Algorithm 7.3) takes $O(\sum_{i=1}^{k} |\mathcal{S}^{(i)}| \, |\mathcal{E}^{(i)}|)$, because when each group $\mathcal{S}^{(i)}$ is processed (lines 2-9), the number of iterations is $|\mathcal{S}^{(i)}| - 1$ and each iteration takes $O(|\mathcal{E}^{(i)}|)$. The encoding step (Algorithm 7.4) takes $O(\sum_{A \in \mathcal{S}}^{k} |\mathcal{E}_A|) = O(|\mathcal{E}|)$ because to process each supernode A (lines 3-7) takes $O(|\mathcal{E}_{AA}| + \sum_{B(\neq A): \mathcal{E}_{AB} \neq \emptyset} |\mathcal{E}_{AB}|) = O(|\mathcal{E}_A|)$. This follows from $|\pi_{AB}| < 2 \cdot |\mathcal{E}_{AB}|$ and $|\pi_{AA}| < 2 \cdot |\mathcal{E}_{AA}|$ in lines 5 and 7, which are due to the conditions in lines 4 and 6. Lastly, the dropping step (Algorithm 7.5) takes

$O(|\mathcal{E}|+|\mathcal{C}^+|+|\mathcal{C}^-|+|\mathcal{P}|) = O(|\mathcal{E}|)$ (see Eq. (7.14)). Note that $\mathcal{P}$ (lines 8) can be sorted in $O(|\mathcal{P}|+|\mathcal{E}|)$ using any linear-time integer sorting algorithm (e.g., counting sort) since $|A| \cdot |B| \in \{1, ..., 2|\mathcal{E}|\}$ for every $\{A, B\} \in \mathcal{P}$ due to the conditions in lines 4 and 6 of Algorithm 7.4. Thus, all the steps except for the merging step take $O(|\mathcal{E}|)$. The merging step, whose time complexity is $O(\sum_{i=1}^{k} |\mathcal{S}^{(i)}||\mathcal{E}^{(i)}|)$, also takes $O(|\mathcal{E}|)$ if $\mathcal{S}$ is divided finely in the dividing step so that the size of each group is less than a constant (see Eq. (7.13)). In such cases, the overall time complexity of SWEG (Algorithm 7.1) is $O(T \cdot |\mathcal{E}|)$ since the dividing and merging steps are repeated T times. This linear scalability is shown experimentally in Section 7.5.4.

7.4.2 Memory Requirement Analysis

The space required for storing $\mathcal{S}$, $\{\mathcal{S}^{(1)}, ..., \mathcal{S}^{(k)}\}$, $\{h(v) : v \in \mathcal{V}\}$, and $\{c_v : v \in \mathcal{V}\}$ is $O(|\mathcal{V}|)$, and the space required for storing $\mathcal{G}$, $\overline{\mathcal{G}}$, and $\mathcal{C}$ is $O(|\mathcal{E}|)$ (see Eq. (7.14)). Thus, in shared-memory settings, where all of them are stored in memory, the memory requirements of SWEG (Algorithm 7.1) are $O(|\mathcal{V}| + |\mathcal{E}|) = O(|\mathcal{E}|)$. In MAPREDUCE settings, as described in Section 7.3.4, each mapper or reducer requires $O(|\mathcal{V}|)$ memory, which is used to store $\mathcal{S}$, $\{h(v) : v \in \mathcal{V}\}$, or $\{c_v : v \in \mathcal{V}\}$, in all the stages except for Reduce-1. In Reduce-1, in addition to $O(|\mathcal{V}|)$ memory for $\mathcal{S}$, each reducer requires $O(\max_{1\le i\le k} |\mathcal{E}^{(i)}|)$ memory where $\{R(A) : A \in \mathcal{S}^{(i)}\}$ for each group $\mathcal{S}^{(i)}$ is loaded at once. Thus, the memory requirements are $O(|\mathcal{V}| + \max_{1\le i\le k} |\mathcal{E}^{(i)}|)$ per reducer. In real-world graphs, $\max_{1\le i\le k} |\mathcal{E}^{(i)}|$ is much smaller than $|\mathcal{E}|$, as shown experimentally in Figure 7.3.

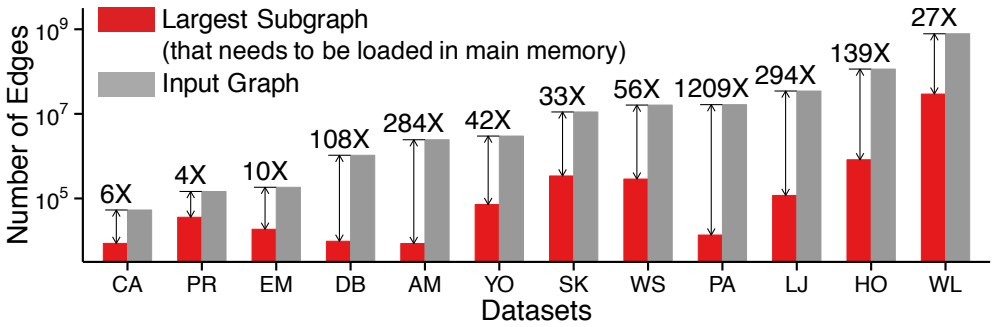


Figure 7.3: **SWEG is memory efficient.** Subgraphs that SWEG loads in memory at once are significantly smaller than input graphs. See Section 7.5.1 for the detailed experimental settings.

7.5 Experiments

We review our experiments for answering the following questions:

- **Q1. Lossless Summarization**: Does the lossless version of SWEG yield more compact representations faster than its competitors?
- **Q2. Lossy Summarization**: Does the lossy version SWEG yield more compact and accurate representations than baselines?
- **Q3. Scalability**: Does SWEG scale linearly with the size of the input graph? Does SWEG scale up and scale out?
- **Q4. Effects of Parameters**: How do the number of iterations T and the error bound ϵ affect the compactness of outputs?

Table 7.2: **Summary of the graphs used in our experiments.** B: billion, M: million, K: thousand.

Name	# Nodes	# Edges	Summary
Caida (CA) [LKF07]	26.5K	53.4K	Internet graph
Protein (PR) [JTGV+05]	6,229	146K	Protein interaction graph
Email (EM) [KY04]	36.7K	184K	Email network
DBLP (DB) [YL15]	317K	1.05M	Collaboration network
Amazon (AM) [LAH07]	403K	2.44M	Co-purchase network
Youtube (YO) [MMG+07]	1.13M	2.99M	Social network
Skitter (SK) [LKF07]	1.70M	11.1M	Internet graph
Web-Small (WS) [BV04]	863K	16.1M	Web graph
Patent (PA) [HJT01]	3.77M	16.5M	Citation network
LiveJournal (LJ) [YL15]	4.00M	34.7M	Social network
Hollywood (HO) [BV04]	1.99M	114M	Collaboration network
Web-Large (WL) [BV04]	39.5M	783M	Web graph
LinkedIn (LI)	$>$ 600M	$>$ 20B	Social

- **Q5. Further Compression**: How much does SWEG+ improve the compression rates of combined compression methods?

7.5.1 Experimental Settings

Machines: We ran single instance experiments on a machine with 2.10GHz Intel Xeon E6-2620 CPUs (with 6 cores) and 64GB memory. We ran MapReduce experiments on a private Hadoop cluster.

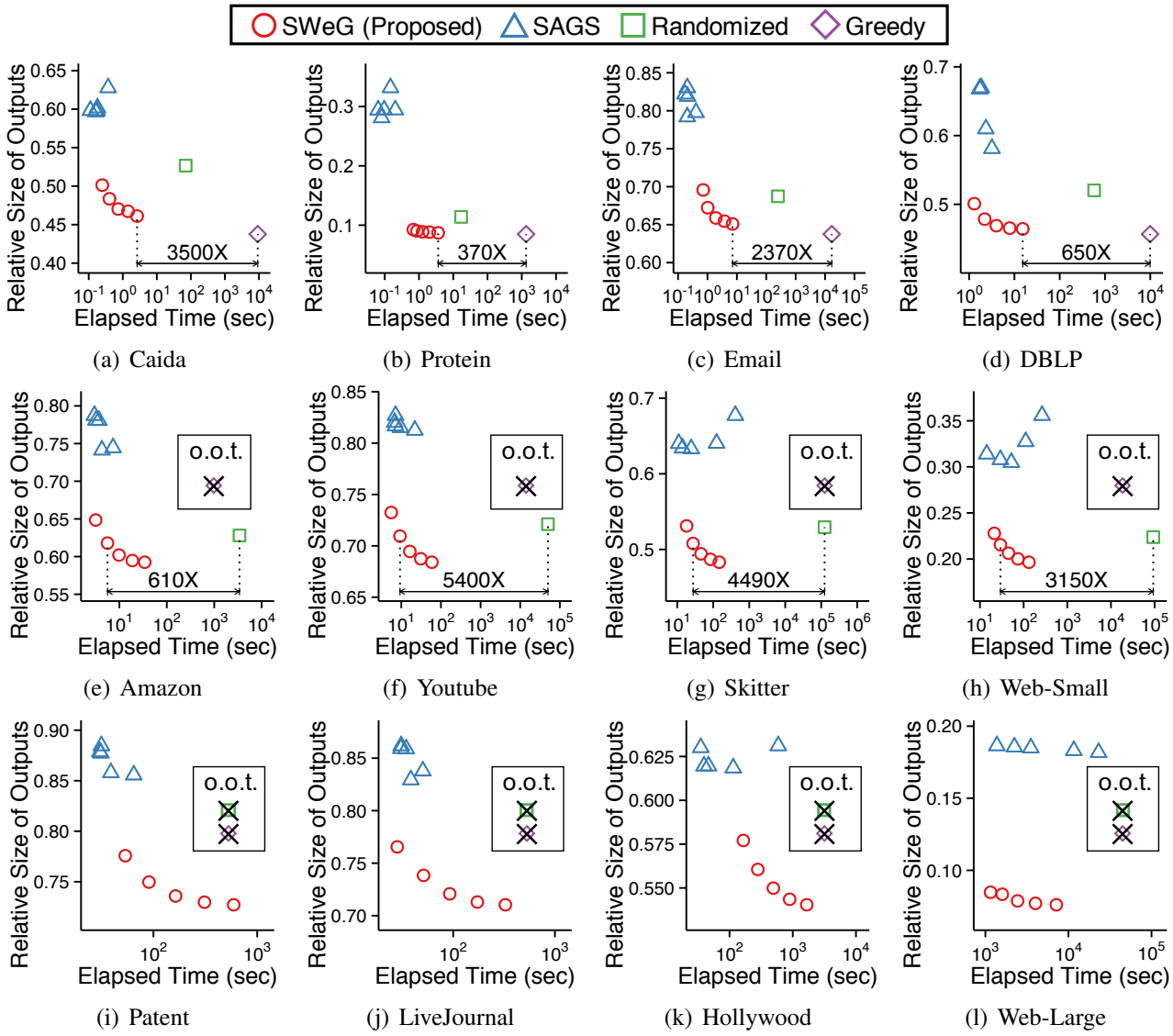
Datasets: We used 13 real-world graphs summarized in Table 7.2. We ignored the direction of edges in all the datasets. All the datasets are publicly available except for the LinkedIn dataset, which we used only for scalability tests.

Implementations: We implemented all the considered algorithms in Java 1.8. We implemented the shared-memory version of SWEG using standard Java multithreading, and we set the number of threads to 8 unless otherwise stated. We implemented the MAPREDUCE version of SWEG using Hadoop 2.6.1, and we set the numbers of mappers and reducers to 40 unless otherwise stated. In both implementations, we ran the dividing step recursively so that each group had at most 500 supernodes, as described in Section 7.3.2.1.

Evaluation Metric: To measure the compactness of the outputs of summarization (i.e., $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$) of a given graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we used *relative size of outputs*, defined as

$$\frac{(|\mathcal{P}^*| + |\mathcal{C}^+| + |\mathcal{C}^-|)}{|\mathcal{E}|}, \tag{7.15}$$

where the numerator is our objective function (i.e., Eq. (7.1)) and the denominator is a constant for a given input graph.



(a) Caida (b) Protein (c) Email (d) DBLP

(e) Amazon (f) Youtube (g) Skitter (h) Web-Small

(i) Patent (j) LiveJournal (k) Hollywood (l) Web-Large

Figure 7.4: **SWEG (lossless and shared-memory) significantly outperforms existing lossless summarization methods.** o.o.t.: out of time (> 48 hours). Specifically, SWEG was up to $5,400\times$ *faster* than the others that give similarly compact outputs.

7.5.2 Q1. Lossless Summarization

We compared lossless graph-summarization methods in terms of speed and compactness of representations. To this end, we measured the elapsed time and the relative size of the outputs (i.e., Eq. (7.15)) of the following methods:

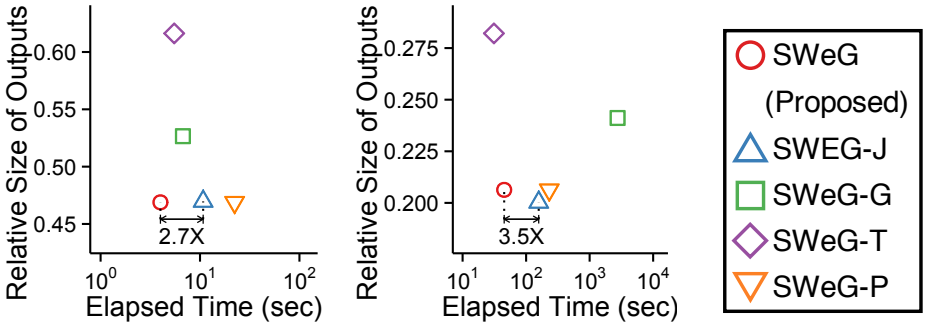
(a) SWEG (Proposed): the shared-memory and lossless version of SWEG with $\epsilon = 0$ and $T = \{5, 10, 20, 40, 80\}$.

(b) SAGS [KNL15]: SAGS with five different parameter settings. [1]

(c) RANDOMIZED [NRS08].

(d) GREEDY [NRS08].

[1] $\{h = 28,\ b = 7,\ OverlapRatio = 0.3\}$, $\{h = 28,\ b = 7,\ OverlapRatio = 0.1\}$, $\{h = 28,\ b = 7,\ OverlapRatio = 0.2\}$, $\{h = 30,\ b = 10,\ OverlapRatio = 0.3\}$, and $\{h = 30,\ b = 15,\ OverlapRatio = 0.3\}$.



(a) DBLP (b) Web-Small

Figure 7.5: **SWEG (lossless and shared-memory) significantly outperforms its variants.** This justifies our design choices.

(e) SWEG-P: a variant of (a) without **p**arallelization with $T = 20$.

(f) SWEG-G: a variant of (a) that does not **g**roup the supernodes with $T = 20$.

(g) SWEG-J: a variant of (a) that chooses supernodes to merge based on the exact savings instead of **j**accard similarity with $T = 20$.

(h) SWEG-T: a variant of (a) where the merging **t**hresohld $\theta(t) = 0$ and $T = 20$.

Among them, (b)-(e) are serial methods and the others are parallel methods. We slightly modified (b)-(d) so that they aim to minimize the same objective (i.e., Eq. (7.1)) of SWEG.

SWEG provided the best trade-off between speed and compactness of outputs. As seen in Figure 7.4, SWEG significantly outperformed SAGS, RANDOMIZED, and GREEDY on all the datasets. For example, on the Youtube dataset, SWEG was $\mathbf{5,400\times}$ **faster** with 2% smaller outputs than RANDOMIZED and 18% faster with 10% smaller outputs than SAGS. GREEDY did not terminate within a reasonable time period ($\geq$ 48 hours) on the dataset. As seen in Figure 7.5, SWEG (with $T = 20$) also outperformed its variants. Although we only reported the results on the DBLP and Web-Small datasets, we obtained consistent results on all other datasets. These results justify our design choices in Section 7.3.

7.5.3 Q2. Lossy Summarization

We compared the following lossy graph-summarization methods in terms of the compactness and accuracy of output representations:

(a) SWEG (Proposed): the shared-memory and lossy version of SWEG with $T = 80$ and $\epsilon = \{0, 0.18, 0.36, 0.54, 0.72, 0.9\}$.

(b) BAZI [BAZK18]: BAZI with $s = \log^2 |\mathcal{S}|$, $w = 50$, and $k = \{0.1 \cdot |\mathcal{V}|, 0.28 \cdot |\mathcal{V}|, 0.46 \cdot |\mathcal{V}|, 0.64 \cdot |\mathcal{V}|, 0.82 \cdot |\mathcal{V}|, |\mathcal{V}|\}$.

(c) BOUNDED: a baseline that performs only the dropping step of SWEG, i.e., SWEG with $T = 0$ and $\epsilon = \{0.18, 0.36, 0.54, 0.72, 0.9\}$.

(d) RANDOM: a baseline that randomly drops $\epsilon = \{0.18, 0.36, 0.54, 0.72, 0.9\}$ of the edges from the input graph.

We measured the relative size of outputs (e.g., Eq. (7.15)) to evaluate the compactness of outputs. Then, to evaluate the accuracy of outputs, we measured how accurately the outputs preserve the relevances between nodes as follows:

S1. randomly choose $100,000$ seed nodes in the input graph.

S2. compute the **true relevances** between each seed node and the other nodes in the input graph.

S3. compute the **approximate relevances** between each seed node and the other nodes in the graph restored from the outputs.

S4. measure how accurate the approximate relevances from S3 are.

In **S2** and **S3**, we used one of the following relevance scores:

- *Random Walk with Restart (RWR)* [TFP06]: each node's RWR score with respect to a seed node is defined as the stationary probability that a random surfer is at the node. The random surfer either moves to a neighboring node of the current node (with probability 0.8) or restarts at the seed node (with probability 0.2).
- *Number of Common Neighbors (NCN)*: each node's NCN score with respect to a seed node is defined as the number of common neighbors of the node and the seed node.

In **S4**, we computed one of the following accuracy measures for every seed node and averaged them.

- *Precision*@100: Precision@100 is defined as the fraction of the 100 most relevant nodes in terms of the true relevances among those in terms of the approximate relevances.
- *NDCG*@100 [JK02]: Let $r(i)$ be the true relevance of the i-th most relevant node in terms of the approximate relevances. Then,

$$\text{NDCG@100} := \frac{1}{Z}\sum_{i=1}^{100}\frac{2^{r(i)}}{\log_2(1+i)},$$

where Z is a constant normalizing NDCG@100 to be within $[0, 1]$.

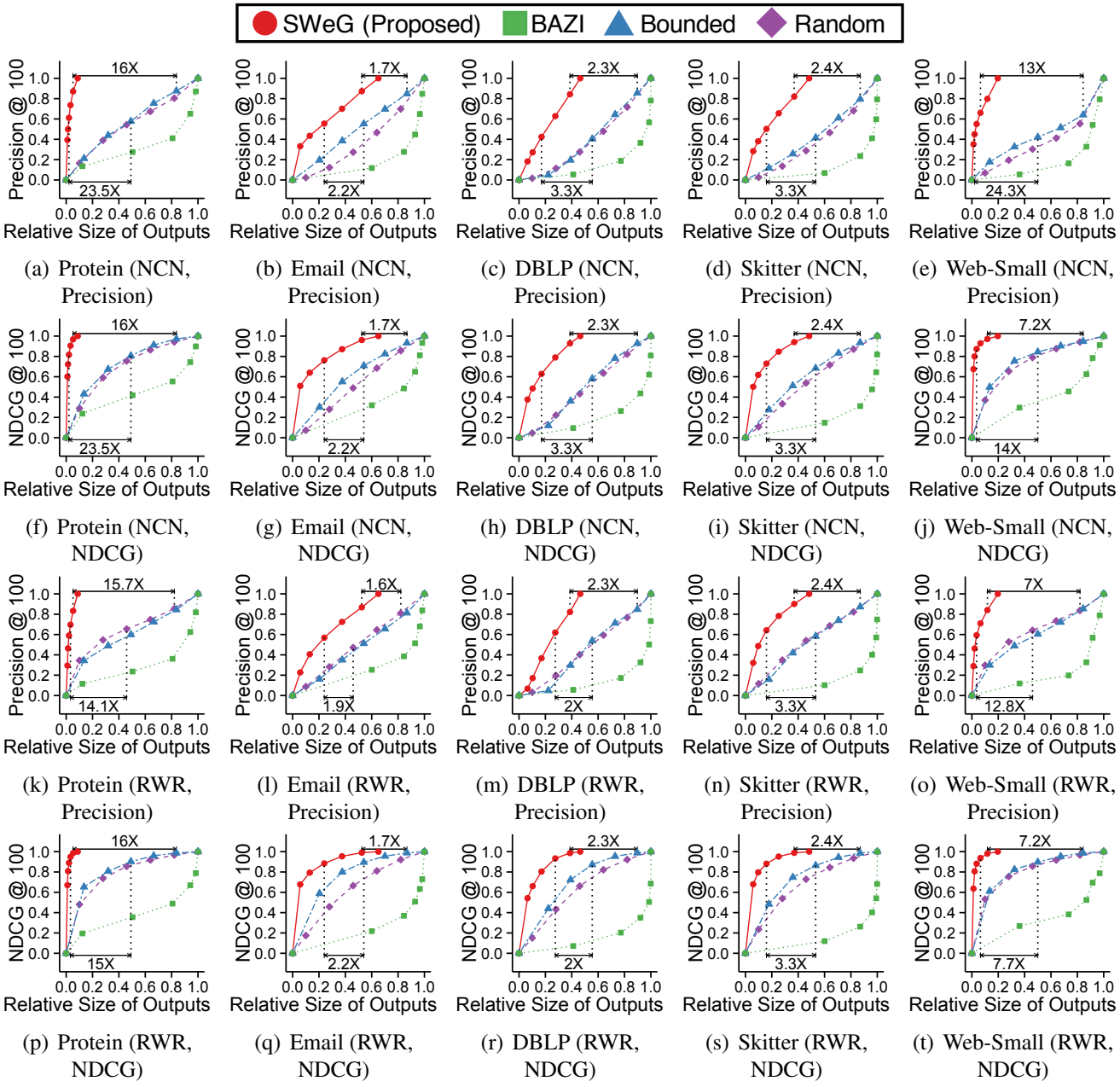
SWEG yielded the most compact and accurate representations. As seen in Figure 7.6, SWEG significantly outperformed the other methods on all the considered datasets. For example, on the Web-Small dataset, SWEG gave a $\mathbf{24.3\times}$ **more compact** and similarly accurate representation than the other methods. BAZI tended to yield representations with most (super) edges since it aims to reduce the number of supernodes instead of minimizing the number of (super) edges (see Section 7.6).

7.5.4 Q3. Scalability

We evaluated the scalability of the shared-memory and MAPREDUCE implementations of SWEG. Specifically, we measured how rapidly their running times change depending on the size of the input graph, the number of threads, and the number of machines (i.e., the numbers of mappers and reducers in the MAPREDUCE framework). In the shared-memory setting, we used graphs with different sizes obtained by sampling different numbers of nodes from the Web-Large dataset. In the MAPREDUCE setting, we used graphs obtained in the same manner using the LinkedIn dataset. When measuring the data scalability, we fixed the number of threads to 8 and the number of machines to 40. When measuring the machine and multi-core scalability, we fixed the size of the input graph.

SWEG scaled linearly with the size of the input graph, as seen in Figures 7.7(a) and 7.7(b). The lossless summarization by SWEG as well as the additional dropping step for lossy summarization (with $\epsilon = 0.1$) scaled near linearly with the number of edges in the input graph in both settings. Note that the largest graph used had more than **20 billion edges**.

SWEG achieved significant speedup in the shared-memory and MAPREDUCE settings. As seen in Figure 7.7(c), the speedup, defined as T_1/T_N where T_N is the running time of SWEG with N threads,



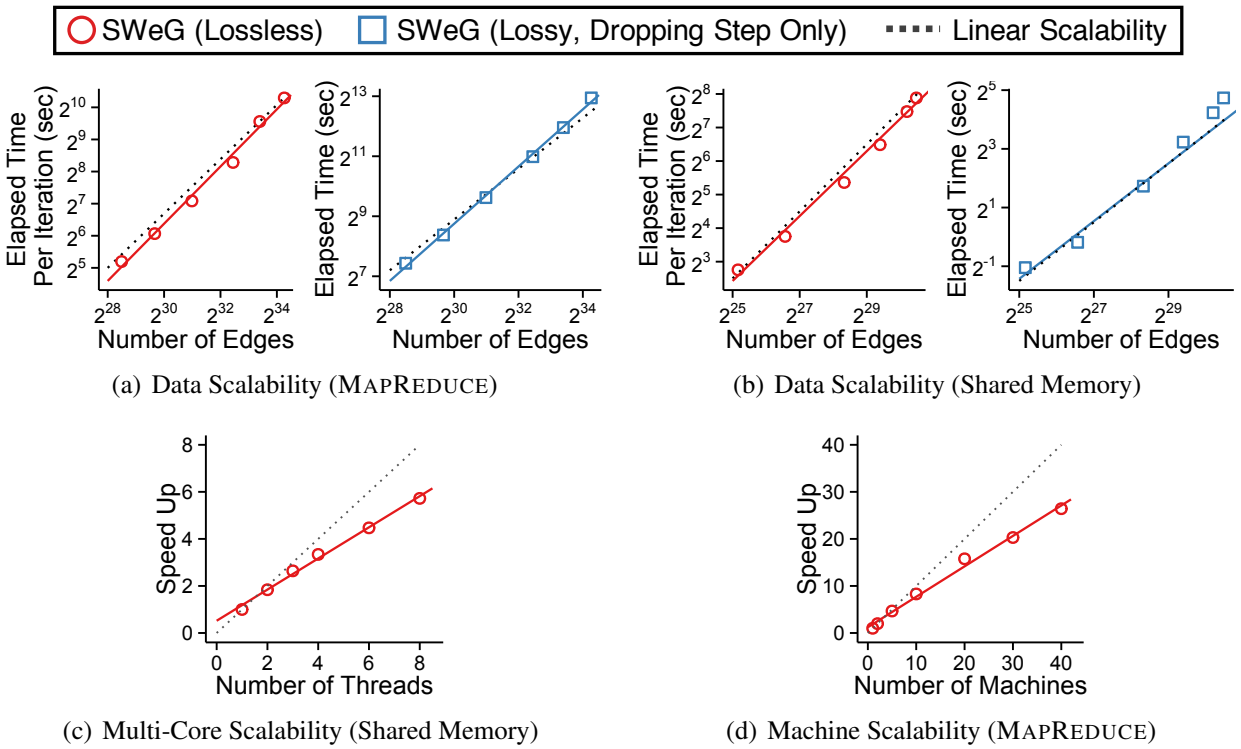
(a) Protein (NCN, Precision) (b) Email (NCN, Precision) (c) DBLP (NCN, Precision) (d) Skitter (NCN, Precision) (e) Web-Small (NCN, Precision)

(f) Protein (NCN, NDCG) (g) Email (NCN, NDCG) (h) DBLP (NCN, NDCG) (i) Skitter (NCN, NDCG) (j) Web-Small (NCN, NDCG)

(k) Protein (RWR, Precision) (l) Email (RWR, Precision) (m) DBLP (RWR, Precision) (n) Skitter (RWR, Precision) (o) Web-Small (RWR, Precision)

(p) Protein (RWR, NDCG) (q) Email (RWR, NDCG) (r) DBLP (RWR, NDCG) (s) Skitter (RWR, NDCG) (t) Web-Small (RWR, NDCG)

Figure 7.6: **SWeG (lossy) significantly outperforms baseline methods for lossy graph summarization.** Specifically, SWeG yielded up to $24.3\times$ *more compact* and similarly accurate representations than the other methods.



(a) Data Scalability (MAPREDUCE)

(b) Data Scalability (Shared Memory)

(c) Multi-Core Scalability (Shared Memory)

(d) Machine Scalability (MAPREDUCE)

Figure 7.7: **SWEG is scalable in both the MAPREDUCE and shared-memory settings.** (a-b) SWEG scaled linearly with the size of the input graph. (c-d) SWEG achieved significant speedups as more machines and cores were used. Note that the largest graph used had more than 20 billion edges.

increased near linearly with the number of threads. Specifically, SWEG provided a speedup of $3.3\times$ with 4 threads and $5.7\times$ with 8 threads in the shared-memory setting. As seen in Figure 7.7(a), the speedup of SWEG, defined as T_1/T_N where T_N is the running time of SWEG with N machines, increased near linearly with the number of machines. Specifically, SWEG provided a speedup of $8.3\times$ with 10 machines and $26.4\times$ with 40 machines in the MAPREDUCE setting.

7.5.5 Q4. Effects of Parameters

We measured how the number of iterations T and the error bound ϵ in SWEG affect the compactness of its output representations using the relative size of outputs (i.e., Eq. (7.15)). For measuring the effect of T, we fixed ϵ to 0 and changed T from 1 to 80. For measuring the effect of ϵ, we fixed T to 80 and changed ϵ from 0 to 0.5.

The larger the number of iterations the more compact the output representations. As seen in Figure 7.8, the size of outputs decreased over iterations and eventually plateaued.

The larger the error bound, the more compact the output representations. As seen in Figure 7.9, the size of outputs decreased near linearly as the error bound increased.

The relative size of outputs was small in web graphs and protein-interaction graphs, where nodes tend to have similar connectivity [CLDG03, KRR+00].

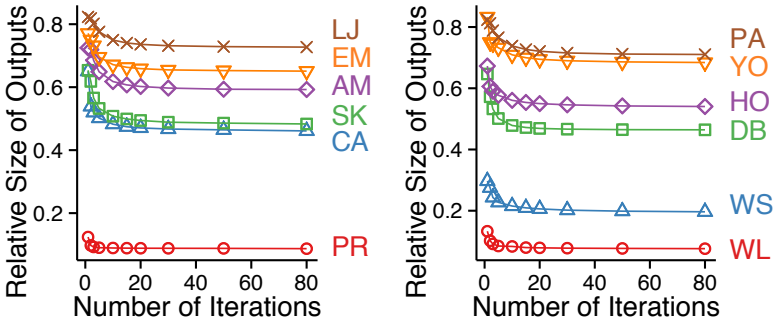


Figure 7.8: **Effects of iterations on the compactness of outputs.** As the number of iterations in SWEG (lossless) increases, the output representations become compact.

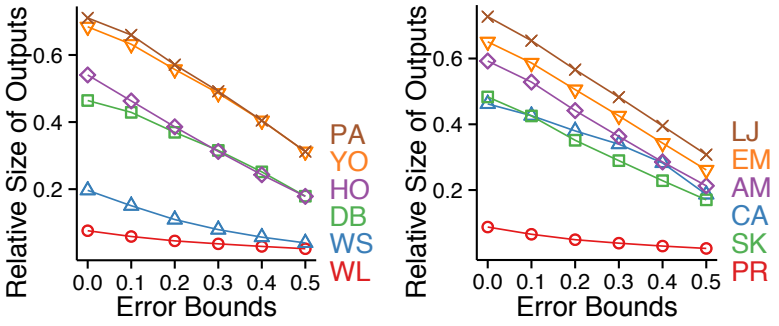


Figure 7.9: **Effects of error bounds on the compactness of outputs.** As the error bound in SWEG (lossy) increases, the output representations become compact.

7.5.6 Q5. Further Compression

We measured how much SWEG+ improves the compression rates of the following advanced graph-compression methods:

(a) BP [DKK+16]: reordering nodes as suggested in [DKK+16] with 20 iterations and using the webgraph framework [BV04].

(b) SHINGLE [CKL+09]: reordering nodes as suggested in [CKL+09] and using the webgraph framework [BV04].

(c) BFS [AD09]: BFS with the default parameter setting in its open-sourced implementation.[2]

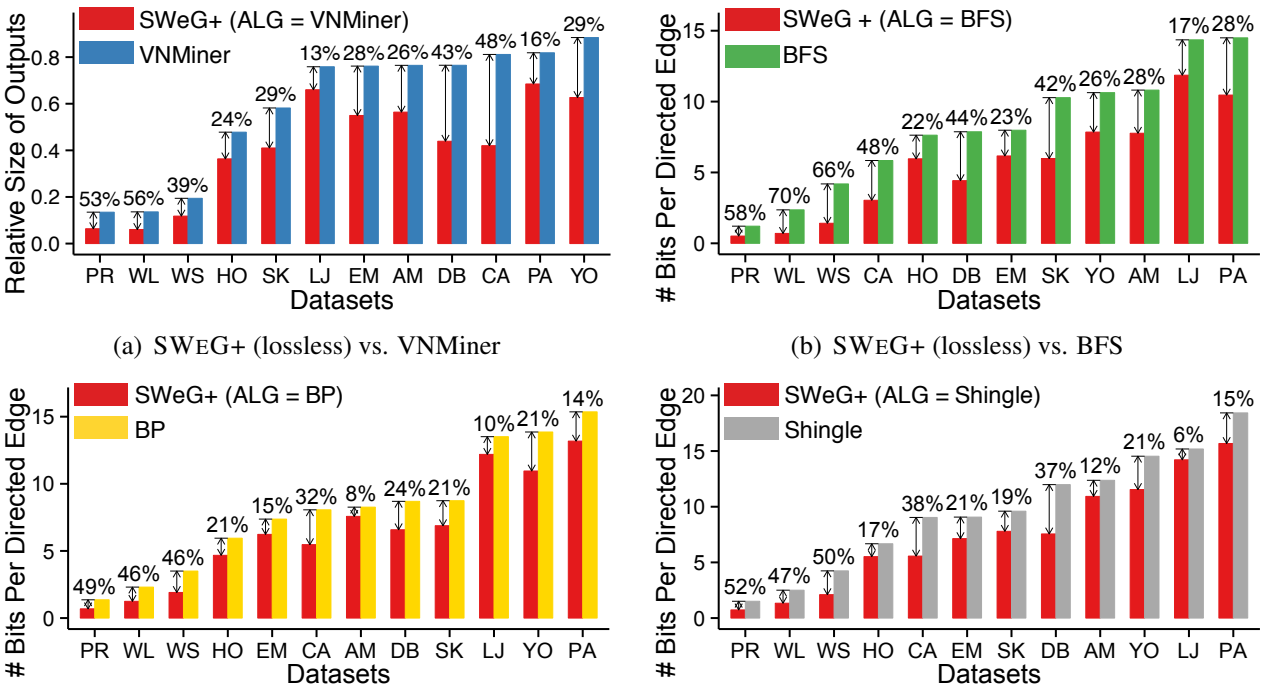
(d) VNMINER [BC08]: VNMINER with 80 iterations.

For the webgraph framework in (a) and (b), we used the default parameter setting (i.e., $r = 3$, $W = 7$, $L_{min} = 7$, and ζ_3). We measured the compression rates using the objective function of each compression algorithm. That is, we used the number of bits per directed edge[3] for (a)-(c) and the relative size of outputs (i.e., Eq. (7.15)) for (d).

SWEG+ achieved further compression. As seen in Figure 7.1(d) in Section 7.1 and Figure 7.10, the lossless version of SWEG+ ($T = 80$ and $\epsilon = 0$) yielded up to **3.4× more compact** representations than all the input compression algorithms on all the datasets. Especially, SWEG+ with BFS as ALG represented the Web-Large dataset using less than **0.7 bits per directed edge**. In terms of the number of bits, SWEG+ yielded the most compact representations when BFS or BP was used as ALG.

[2] https://github.com/drovandi/GraphCompressionByBFS.

[3] We regarded undirected graphs as symmetric directed graphs.



(a) SWEG+ (lossless) vs. VNMiner

(b) SWEG+ (lossless) vs. BFS

(c) SWEG+ (lossless) vs. BP

(d) SWEG+ (lossless) vs. Shingle

Figure 7.10: **SWEG+ (lossless) significantly improves the compression rates of state-of-the-art graph-compression algorithms.**

7.6 Related Work

While this chapter addresses Problem 7.1, the term "graph summarization" has been used for a wider range of problems related to concisely describing static plain graphs [DS13, KKVF14, NRS08, TZHH11, RGSB17], static attributed graphs [CLF+09, FWW13, KNL14, SWL+18, THP08], and dynamic graphs [AZBP17, LSK08, QLJ+14, SKZ+15]. We refer the reader to an excellent survey [LSDK18] for a review on these problems. In this section, we focus on previous work closely related to Problem 7.1.

Lossless graph summarization. GREEDY [NRS08] repeatedly finds and merges a pair of supernodes so that savings in space are maximized given the other supernodes. If there is no pair whose merger leads to non-negative savings, then GREEDY creates a summary graph and corrections so that the number of (super) edges is minimized given the current supernodes $\mathcal{S}$. GREEDY is computationally and memory expensive because it maintains and updates savings for $O(|\mathcal{V}|^2)$ pairs of supernodes. Without maintaining any precomputed savings, RANDOMIZED [NRS08], whose time complexity is $O(|\mathcal{V}| \cdot |\mathcal{E}|)$, randomly chooses a supernode first and then chooses another node to be merged so that savings in space are maximized given the other supernodes. SAGS [KNL15] chooses supernodes to be merged using locality sensitive hashing without computing savings in space, which are expensive to compute. We empirically show in Section 7.5.2 that all these serial algorithms are not satisfactory in terms of speed or compactness of outputs. They either have high computational complexity or significantly sacrifice compactness of outputs for lower complexity. More importantly, they cannot handle large-scale graphs that do not fit in main memory.

Lossy graph summarization. APXMDL [NRS08] reduces the problem of choosing edges to be dropped from outputs of GREEDY to a maximum b-matching problem and uses Gabow's algorithm [Gab83],

whose time complexity is $O(\min(|\mathcal{E}|^2 \log |\mathcal{V}|, |\mathcal{E}| \cdot |\mathcal{V}|^2))$. Due to this high computational complexity, APXMDL does not scale even to moderately-sized graphs. For the same problem, the dropping step of SWEG takes $O(|\mathcal{E}|)$ (see Section 7.4.1). In [NRS08], combining GREEDY and APXMDL into a single step was also discussed. Several algorithms [BAZK18, LT10, RGSB17, LTH+14], including a distributed one [LTH+14], have been developed for a problem that is similar but not identical to Problem 7.1. They aim to find a summary graph with a given number of supernodes so that the difference between the original and restored graphs is minimized without edge corrections. The way of restoring a graph is different from that in Problem 7.1. Since these algorithms aim to reduce the number of supernodes, instead of (super) edges, they are not effective for Problem 7.1 (see Section 7.5.3).

Combination with other compression techniques. In addition to graph summarization, numerous graph-compression techniques have been developed, including relabeling nodes [BV04, AD09, DKK+16, CKL+09], utilizing encoding schemes for integer sequences (e.g., reference, gap, and interval encodings) [BV04], and encoding common structures (e.g., cliques, bipartite-cores, and stars) with fewer bits [KKVF14, BC08, RZ18]. See [BH18] for a comprehensive survey on these techniques. As described in Sections 7.3.5 and 7.5.6, SWEG is readily combinable with any compression technique for static plain graphs. In [SPH+18], tightly combining two specific summarization and compression algorithms [NRS08, WOS06] into a single process was presented.

7.7 Summary

We propose SWEG, a fast parallel algorithm for lossless and lossy summarization of large-scale graphs, which may not fit in main memory. We present efficient implementations of SWEG in shared-memory and MAPREDUCE environments. We also propose SWEG+ where SWEG and other graph-compression methods are combined to achieve better compression than individual methods. We theoretically and empirically show the following strengths of SWEG:

- **Fast**: SWEG provides similarly compact representations up to $5,400\times$ *faster* than existing summarization methods (Figure 7.4).
- **Scalable**: SWEG scales near linearly with the number of edges in the input graph, successfully scaling to graphs with over *20 billion* edges. SWEG also scales up and scales out (Figure 7.7).
- **Compact**: SWEG+ achieves up to $3.4\times$ *better* compression than individual state-of-the-art compression methods that are combined with SWEG (Figures 7.1(d) and 7.10).

7.8 Appendix: Neighbor Queries on Summarized Graphs

Algorithm 7.7 describes how to process neighbor queries (i.e., returning the neighbors $\hat{N}_v$ of a given node $v \in \mathcal{V}$) efficiently on a summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$ and corrections $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$ without restoring entire $\hat{\mathcal{G}}$. Let N_v^- be the neighbors of node $v \in \mathcal{V}$ in $\mathcal{C}^-$. If there is no redundant edge[4] in $\mathcal{C}^+$ and we use a hash table for $\hat{N}_v$ and adjacency lists for $\mathcal{C}^+$, $\mathcal{C}^-$, and $\mathcal{P}$, then the running time of Algorithm 7.7 is proportional to Eq. (7.16)

$$|\hat{N}_v| + 2|N_v^-|. \tag{7.16}$$

[4]As in the outputs of SWEG, if $\{A, B\} \in \mathcal{P}$, $u \in A$, and $v \in B$, then $\{u, v\} \notin \mathcal{C}^+$.

Algorithm 7.7 Neighbor Query Processing on $\overline{\mathcal{G}}$ and $\mathcal{C}$

Input: summary graph $\overline{\mathcal{G}} = (\mathcal{S}, \mathcal{P})$, corrections $\mathcal{C} =< \mathcal{C}^+, \mathcal{C}^- >$, query node $v \in \mathcal{V}$
Output: the set $\hat{N}_v$ of v's neighbors in the restored graph $\hat{\mathcal{G}}$

1: $\hat{N}_v \leftarrow \emptyset$; $S_v \leftarrow$ the supernode in $\mathcal{S}$ where $v \in S_v$
2: **if** S_v has a self-loop in $\overline{\mathcal{G}}$ **then** $\hat{N}_v \leftarrow \hat{N}_v \cup (S_v - \{v\})$
3: **for each** neighbor A $(\neq S_v)$ of S_v in $\overline{\mathcal{G}}$ **do** $\hat{N}_v \leftarrow \hat{N}_v \cup A$
4: $\hat{N}_v \leftarrow (\hat{N}_v \cup N_v^+) - N_v^-$ ▷ N_v^+: v's neighbors in $\mathcal{C}^+$
5: **return** $\hat{N}_v$ ▷ N_v^- : v's neighbors in $\mathcal{C}^-$

In every dataset listed in Table 7.2 in Section 7.5, when SWEG ($T = 80$) was used, $|\mathcal{C}^-|$ was at most 6% of the number of edges in the input graph, regardless of the error bound ϵ. Thus, since $|\hat{N}_v| \leq (1+\epsilon) \cdot |N_v|$ from Eq. (7.2), Eq. (7.16) was at most $(1.12+\epsilon) \cdot |N_v|$ on average.

Chapter based on work that appeared at WSDM 2017 [OSP+17] and its extension [ZOS+19].

Chapter 8

Summarizing Large High-order Tensors

How can we summarize large-scale multi-aspect data using Tucker decomposition on an off-the-shelf workstation with a limited amount of memory?

Tucker decomposition has been used widely for summarizing and analyzing multi-aspect data, which are naturally modeled as tensors. However, existing algorithms for Tucker decomposition have limited scalability, failing to decompose high-order (i.e., 4 or higher order) tensors, since they *explicitly materialize* intermediate data, whose size grows exponentially with the order.

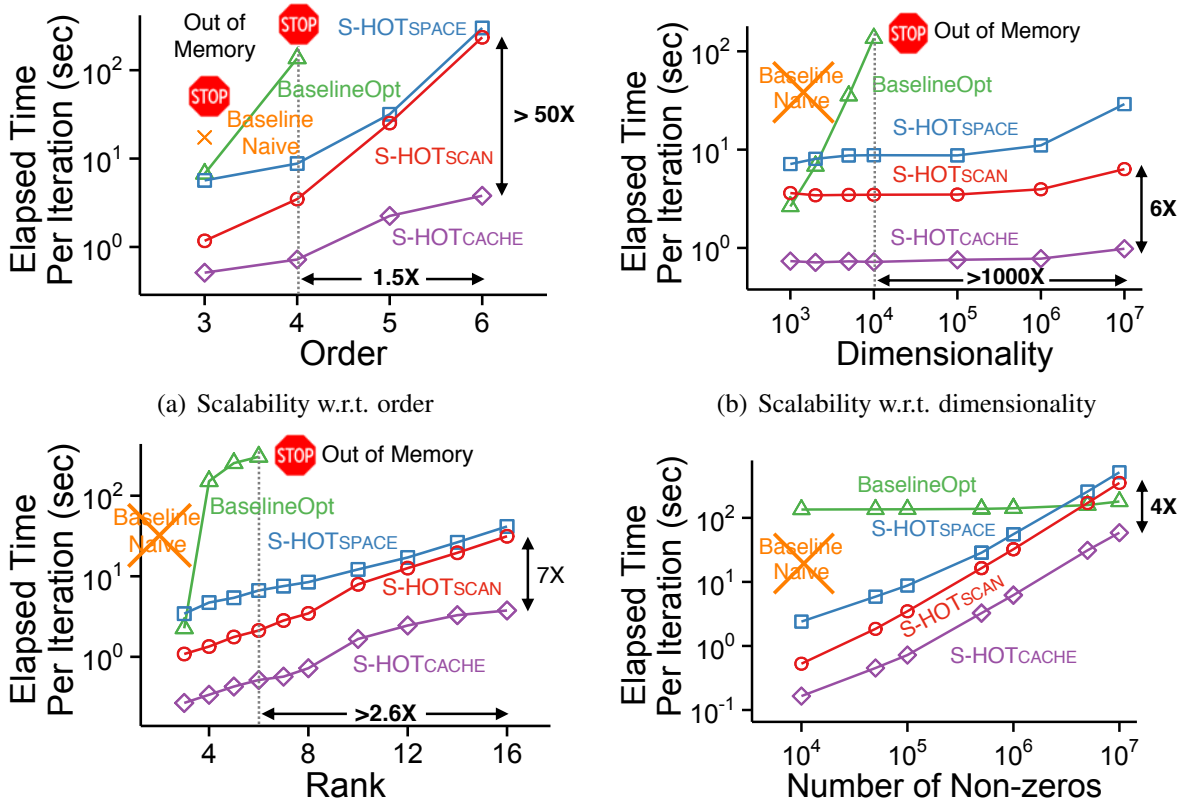
To address this problem, which we call *Materialization Bottleneck*, we propose S-HOT, a scalable algorithm for high-order Tucker decomposition. S-HOT minimizes materialized intermediate data by using an *on-the-fly computation*, and it is optimized for disk-resident tensors not fitting in memory. We theoretically analyze the amount of memory and the number of data scans required by S-HOT. Moreover, we empirically show that S-HOT handles tensors with higher order, dimensionality, and rank than baselines. For example, S-HOT successfully decomposes a tensor from the Microsoft Academic Graph on an off-the-shelf workstation, while all baselines fail. Especially, in terms of dimensionality, S-HOT decomposes $1000\times$ *larger* tensors than baselines.

8.1 Motivation

Tensor decomposition is a widely-used technique for summarization and analysis of multi-aspect data. Multi-aspect data, which are naturally modeled as high-order tensors, frequently appear in many applications [CZL+11, KBK05, MGF11, MJE12, RST10], including the following examples:

- Social media: 4-way tensor (sender, recipient, keyword, timestamp)
- Web search: 4-way tensor (user, keyword, location, timestamp)
- Internet security: 4-way tensor (source IP, destination IP, destination port, timestamp)
- Product reviews: 5-way tensor (user, product, keyword, rating, timestamp)

To summarize and analyze such multi-aspect data, several tensor decomposition methods have been proposed, and we refer interested readers to an excellent survey [KB09]. Tensor decompositions have provided meaningful results in various domains [AÇKY05, KB09, KBK05, LNSS16, CZL+11, FSSS09, KBK05, MJE12, RST10]. Especially, Tucker decomposition [Tuc66] has been successfully applied in many applications, including web search [SZL+05], network forensics [STF06], social network analysis [CTT06], and scientific data compression [ABK16].



(a) Scalability w.r.t. order

(b) Scalability w.r.t. dimensionality

(c) Scalability w.r.t. rank

(d) Scalability w.r.t. non-zeros

Figure 8.1: **S-HOT is scalable**. Every version of S-HOT successfully summarizes tensors with high order, dimensionality, and rank, while the baseline algorithms fail running out of memory, as those three factors increase. Especially, every version of S-HOT handles a tensor with $1000\times$ higher dimensionality. We use two baselines: (1) BaselineNaive: naive algorithm for Tucker decomposition, and (2) BaselineOpt: the state-of-the-art memory-efficient algorithm for Tucker decomposition. Note that all the methods have the same convergence properties (Observation 8.1). See Section 8.5 for details.

Developing a scalable Tucker-decomposition algorithm has been a challenge due to a huge amount of intermediate data generated during the computation. Briefly speaking, Alternating Least Square (ALS), the most widely-used Tucker-decomposition algorithm, repeats two steps: 1) computing an intermediate tensor, denoted by $\mathcal{Y}$, and 2) computing the SVD of the matricized $\mathcal{Y}$ (see Section 8.2 or [KB09] for details). Previous studies [KS08, JPF+16] pointed out that a huge amount of intermediate data are generated during the first step, and they proposed algorithms for reducing the intermediate data by carefully ordering computation.

However, existing algorithms still have limited scalability and easily run out of memory, particularly when dealing with *high-order* (i.e., 4 or higher order) tensors. This is because existing algorithms *explicitly materialize* $\mathcal{Y}$, which is usually thinner but much denser than the input tensor, despite the fact that the amount of space required for storing $\mathcal{Y}$ grows rapidly with respect to the order, dimensionality, and rank of the input tensor. For example, as illustrated in Figure 8.2, the space required for $\mathcal{Y}$, is about *400 Giga Bytes* for a 5-way tensor with 10 million dimensionality when the rank of Tucker decomposition is set to 10. We call this problem Materialization Bottleneck (or *M-Bottleneck* in short). Due to *M-Bottleneck*, existing algorithms are not suitable for decomposing tensors with high order,

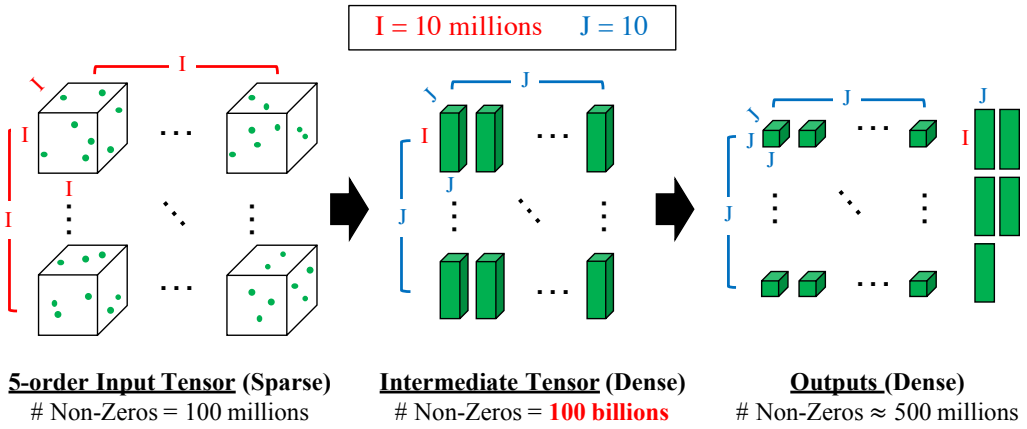


Figure 8.2: **Illustration of the materialization bottleneck** (or *M-Bottleneck* in short). For a high-order sparse input tensor, the amount of space required for the intermediate tensor can be much larger than that for the input tensor and the outputs. As in the figure, the intermediate tensor is usually thinner but much denser than the input tensor. In such a case, materializing intermediate data becomes the scalability bottleneck of existing Tucker-decomposition algorithms.

dimensionality, and/or rank. As seen in Figure 8.1, even state-of-the-art algorithms easily run out of memory as these factors increase.

To avoid *M-Bottleneck*, in this chapter, we propose S-HOT, a scalable Tucker-decomposition algorithm. S-HOT is designed for decomposing high-order tensors on an off-the-shelf workstation. Our key idea is to compute $\mathcal{Y}$ *on the fly*, without materialization, by combining both steps in ALS without changing its results. Specifically, we utilize the *reverse communication interface* of a recent scalable eigensolver called Implicitly Restart Arnoldi Method (IRAM) [LSY98], which enables SVD computation without materializing $\mathcal{Y}$. Moreover, S-HOT performs Tucker decomposition by streaming non-zero tensor entries from the disk, which enables it to handle disk-resident tensors that are too large to fit in memory. We offer the following versions of S-HOT with distinct advantages:

- S-HOT$_{\text{SPACE}}$: the most space-efficient version not requiring additional copies the input tensor.
- S-HOT$_{\text{SCAN}}$: a faster version requiring multiple copies of the input tensor.
- S-HOT$_{\text{CACHE}}$: the fastest version requiring multiple copies of the input tensor and a buffer in main memory.

Our experimental results demonstrate that S-HOT outperforms baseline algorithms by providing significantly better scalability, as shown in Figure 8.1. Specifically, all versions of S-HOT successfully decompose a 6-way tensor, while baselines fail to decompose even a 4-way tensor or a 5-way tensor due to their high memory requirements. The difference is more significant in terms of dimensionality. As seen in Figure 8.1(b), S-HOT decomposes a tensor with *1000× larger dimensionality* than baselines.

Our contributions in this chapter are summarized as follows.

- **Bottleneck Resolution:** We identify *M-Bottleneck* (Figure 8.2), which limits the scalability of existing Tucker-decomposition algorithms, and we avoid it by using an *on-the-fly computation*.i
- **Scalable Algorithm Design:** We propose S-HOT, a scalable Tucker-decomposition algorithm carefully designed for high-order tensors too large to fit in memory. Compared to baseline methods, S-HOT scales up to *1000× larger tensors* (Figure 8.1) with identical convergence properties (Observation 8.1).

Table 8.1: **Table of frequently-used symbols.**

Symbol	Definition
$\mathcal{X}$	N-order input tensor $\in \mathbb{R}^{I_1 \times \cdots \times I_N}$
N	order of $\mathcal{X}$ (i.e., number of modes in $\mathcal{X}$)
I_n	dimensionality of the n-th mode of $\mathcal{X}$
$\mathcal{X}(i_1, ..., i_N)$ $(= x_{i_1 \cdots i_N})$	$(i_1, ..., i_N)$-th entry of $\mathcal{X}$
$nnz(\mathcal{X})$	number of non-zero entries in $\mathcal{X}$
$\Omega(\mathcal{X})$	set of the indices of all non-zero entries in $\mathcal{X}$
$\Omega_i^{(n)}(\mathcal{X})$	subset of $\Omega(\mathcal{X})$ where the n-th mode index is i
$\mathbf{X}_{(n)}$	mode-n unfolding of $\mathcal{X}$
$\mathcal{G}$	N-order core tensor $\in \mathbb{R}^{J_1 \times \cdots \times J_N}$
J_n	number of component (rank) for the n-th mode
$\{\mathbf{A}\}$	set of all the factor matrices of $\mathcal{X}$
$\mathbf{A}^{(n)}$	mode-n factor matrix ($\in \mathbb{R}^{I_n \times J_n}$) of $\mathcal{X}$
$\bar{\mathbf{a}}_i^{(n)}$	i-th row-vector of $\mathbf{A}^{(n)}$
$\mathbf{a}_j^{(n)}$	j-th column-vector of $\mathbf{A}^{(n)}$
$\circ$	outer product
$\bar{\times}_n$	mode-n vector product
$\times_n$	mode-n matrix product

- **Theoretical Analysis:** We provide theoretical analyses on the amount of memory space and the number of data scans that S-HOT requires.

Reproducibility: The source code and datasets used in this chapter are available at *http://dm.postech.ac.kr/shot*.

The rest of the chapter is organized as follows. In Section 8.2, we introduce some preliminary concepts, notations, and a formal problem definition. In Section 8.3, we review related work and discuss *M-Bottleneck*, which limits the scalability of existing methods. In Section 8.4, we propose S-HOT, a scalable algorithm for high-order Tucker decomposition. After sharing some experimental results in Section 8.5, we provide a summary of this chapter in Section 8.6.

8.2 Preliminaries and Problem Definition

In this section, we first introduce some notations and concepts used throughout this chapter. Then, we define the problem of scalable high-order Tucker decomposition.

8.2.1 Notations and Concepts

We give the preliminaries on tensors, basic tensor operations, Tucker decomposition, and Implicitly Restarted ARNOLDI Method. Table 8.1 lists some symbols frequently used in this chapter.

8.2.1.1 Tensors

A tensor is a multi-order array which generalizes a vector (an one-order tensor) and a matrix (a two-order tensor) to higher orders. Let $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ be the input tensor, whose order is denoted by N. Like rows and columns in a matrix, $\mathcal{X}$ has N modes, whose lengths, also called dimensionality, are denoted by $I_1, \ldots, I_N \in \mathbb{N}$, respectively. We use $nnz(\mathcal{X})$ to indicate the number of non-zero entries in $\mathcal{X}$. We denote general N-order tensors by boldface Euler script letters e.g., by $\mathcal{X}$, while matrices and vectors are denoted by boldface capitals, e.g., $\mathbf{A}$, and boldface lowercases, e.g., $\mathbf{a}$, respectively. We use the MATLAB-like notations to indicate the entries of tensors. For example, $\mathcal{X}(i_1, ..., i_N)$ (or $x_{i_1 \cdots i_N}$ in short) indicates the $(i_1, ..., i_N)$-th entry of $\mathcal{X}$. Similar notations are used for matrices and vectors. $\mathbf{A}(i, :)$ and $\mathbf{A}(:, j)$ (or $\bar{\mathbf{a}}_i$ and $\mathbf{a}_j$ in short) indicate the ith row and the jth column of $\mathbf{A}$. The i-th entry of a vector $\mathbf{a}$ is denoted by $\mathbf{a}(i)$ (or a_i in short).

Definition 8.1: Fiber

A mode-n fiber is an 1-order section of a tensor, obtained by fixing all indices except the n-th index.

For example, in a 3-order tensor $\mathcal{X}$, there are 3 types of fibers, $\mathcal{X}(:, j, k)$ (mode-1), $\mathcal{X}(i, :, k)$ (mode-2), and $\mathcal{X}(i, j, :)$ (mode-3) depending on fixed indices.

Definition 8.2: Slice

A slice is a 2-order section of a tensor, obtained by fixing all indices but two.

For example, in a 3-order tensor $\mathcal{X}$, there are 3 types of slices, $\mathcal{X}(i, :, :)$, $\mathcal{X}(:, j, :)$, and $\mathcal{X}(:, :, k)$.

8.2.1.2 Basic Tensor Operations

We review basic tensor operations, which are the building blocks of Tucker decomposition, explained in the following section.

Definition 8.3: Tensor Unfolding/Matricization

Unfolding, also known as matricization, is the process of re-ordering the entries of an N-order tensor into a matrix. The mode-n matricization of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is a matrix $\mathbf{X}_{(n)} \in \mathbb{R}^{I_n \times (\prod_{q \neq n} I_q)}$ whose columns are the mode-n fibers.

For example, the mode-1 unfolding of a 3-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ is denoted by $\mathbf{X}_{(1)} \in \mathbb{R}^{I_1 \times (I_2 I_3)}$. Note that, there are multiple ways to unfold a tensor in terms of the order that the entries of each slice are stacked. For example, the followings are two different ways of mode-1 unfolding

$$\mathbf{X}_{(1)}(i, j + (k-1)I_2) := \mathcal{X}(i, j, k) \text{ and } \mathbf{X}_{(1)}(i, k + (j-1)I_3) := \mathcal{X}(i, j, k).$$

However, specific orders do not have an impact on our algorithm as long as an order is used consistently.

Definition 8.4: N-order Outer Product

The N-order outer product of vectors $\mathbf{v}_1 \in \mathbb{R}^{I_1}$, $\mathbf{v}_2 \in \mathbb{R}^{I_2}$, ..., $\mathbf{v}_N \in \mathbb{R}^{I_N}$ is denoted by $\mathbf{v}_1 \circ \mathbf{v}_2 \circ \cdots \circ \mathbf{v}_N$ and is an N-order tensor in $\mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, where

$$[\mathbf{v}_1 \circ \mathbf{v}_2 \circ \cdots \circ \mathbf{v}_N](i_1, ..., i_N) := \mathbf{v}_1(i_1)\mathbf{v}_2(i_2) \ldots \mathbf{v}_N(i_N).$$

For example, the 3-order outer product of vectors $\mathbf{a} \in \mathbb{R}^I$, $\mathbf{b} \in \mathbb{R}^J$, $\mathbf{c} \in \mathbb{R}^K$ is a 3-order tensor of $\mathbb{R}^{I \times J \times K}$ where each (i, j, k)-th entry is defined as

$$[\mathbf{a} \circ \mathbf{b} \circ \mathbf{c}](i, j, k) = \mathbf{a}(i)\mathbf{b}(j)\mathbf{c}(k).$$

For brevity, we use the following shorthand notations for outer product:

$$\circ_{(i_1,...,i_N)}\{\mathbf{A}\} := \bar{\mathbf{a}}^{(1)}_{i_1} \circ \cdots \circ \bar{\mathbf{a}}^{(N)}_{i_N}, \text{ and}$$
$$\circ^{-n}_{(i_1,...,i_N)}\{\mathbf{A}\} := \bar{\mathbf{a}}^{(1)}_{i_1} \circ \cdots \circ \bar{\mathbf{a}}^{(n-1)}_{i_{n-1}} \circ [1] \circ \bar{\mathbf{a}}^{(n+1)}_{i_{n+1}} \circ \cdots \circ \bar{\mathbf{a}}^{(N)}_{i_N}.$$

Definition 8.5: mode-n Vector Product

The mode-n vector product of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ and a vector $\mathbf{v} \in \mathbb{R}^{I_n}$ is denoted by $\mathcal{X} \bar{\times}_n \mathbf{v}$, and is an ($N$-1)-order tensor in $\mathbb{R}^{I_1 \times \ldots I_{n-1} \times I_{n+1} \times \cdots \times I_N}$, where

$$[\mathcal{X} \bar{\times}_n \mathbf{v}](i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N) := \sum_{i_n=1}^{I_n} \mathcal{X}(i_1, ..., i_N)\mathbf{v}(i_n).$$

Definition 8.6: mode-n Matrix Product

The mode-n matrix product of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ and a matrix $\mathbf{U} \in \mathbb{R}^{J_n \times I_n}$ is denoted by $\mathcal{X} \times_n \mathbf{U}$, and is an N-order tensor in $\mathbb{R}^{I_1 \times \ldots I_{n-1} \times J_n \times I_{n+1} \times \cdots \times I_N}$, where

$$[\mathcal{X} \times_n \mathbf{U}](i_1, \ldots, i_{n-1}, j_n, i_{n+1}, \ldots, i_N) := \sum_{i_n=1}^{I_n} \mathcal{X}(i_1, ..., i_N)\mathbf{U}(j_n, i_n).$$

We adopt the shorthand notations in [KS08] for all-mode matrix product and matrix product in every mode but one:

$$\mathcal{X} \times \{\mathbf{U}\} := \mathcal{X} \times_1 \mathbf{U}^{(1)} \ldots \times_N \mathbf{U}^{(N)}, \text{ and}$$
$$\mathcal{X} \times_{-n} \{\mathbf{U}\} := \mathcal{X} \times_1 \mathbf{U}^{(1)} \ldots \times_{n-1} \mathbf{U}^{(n-1)} \times_{n+1} \mathbf{U}^{(n+1)} \ldots \times_N \mathbf{u}^{(N)}.$$

Algorithm 8.1 Tucker-ALS: a conventional Tucker-decomposition algorithm

Input: (1) $\mathcal{X}$: an N-order tensor of $\mathbb{R}^{I_1 \times \cdots \times I_N}$,
(2) $J_1, \ldots, J_N$: rank for each mode,
(3) T: number of iterations.
Output: (1) $\{\mathbf{A}\}$: a set of factor matrices $\{\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}\}$ where $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times J_n}$,
(2) $\mathcal{G}$: an N-order core tensor of $\mathbb{R}^{J_1 \times \cdots \times J_N}$.

```
initialize all A^(n)
for t ← 1..T do
    for n ← 1..N do
        Y_(n) ← [X ×_{-n} {A^T}]_(n)
        A^(n) ← top-J_n left singular vectors of Y_(n)
G ← Y_(N) ×_N A^(N)^T
return G, {A}
```

8.2.1.3 Tucker decomposition

Tucker decomposition [Tuc66], which is also called N-mode PCA, decomposes a tensor into a core tensor and N factor matrices so that the original tensor is approximated best. Specifically, $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is approximated by

$$\mathcal{X} \approx \mathcal{G} \times \{\mathbf{A}\},$$

where $\mathcal{G} \in \mathbb{R}^{J_1 \times \cdots \times J_N}$, J_n denotes the rank of the mode-n, and $\{\mathbf{A}\}$ is the set of factor matrices $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}$, each of which is in $\mathbb{R}^{I_n \times J_n}$. Using mode-$n$ matrix products and N-order outer products, Tucker is presented as follows:

$$\mathcal{X} \approx \mathcal{G} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \cdots \times_N \mathbf{A}^{(N)} = \sum_{j_1=1}^{J_1} \sum_{j_2=1}^{J_2} \cdots \sum_{j_N=1}^{J_N} \mathcal{G}(j_1, ..., j_N)(\mathbf{a}_{j_1}^{(1)} \circ \mathbf{a}_{j_2}^{(2)} \cdots \circ \mathbf{a}_{j_N}^{(N)}). \quad (8.1)$$

Solving Tucker decomposition is to find the $\mathcal{G}$ and $\{\mathbf{A}\}$ that approximate $\mathcal{X}$ best. It is worth noting that the solution of Tucker is not unique. The most widely used way to solve Tucker is Tucker-ALS (or Higher Order Orthogonal Iteration (HOOI)), which assumes that all column vectors in $\mathbf{A}^{(n)}$ are orthonormal and solves Tucker by Alternating Least Squares (ALS). In addition, [KS08] found that $\mathcal{G}$ can be uniquely computed by $\mathcal{X} \times \{\mathbf{A}^T\}$ once $\{\mathbf{A}\}$ is determined, and simplified the objective function as follows (see [KS08] for details):

$$\max_{\{\mathbf{A}\}} ||\mathcal{X} \times \{\mathbf{A}^T\}||. \quad (8.2)$$

The conventional Tucker-ALS is presented in Algorithm 8.1.

8.2.1.4 Implicitly Restarted Arnoldi Method (IRAM)

Computing the Eigendecomposition of large matrices is important because it is an important foundation of various dimensionality reduction and low-rank approximation techniques. Vector iteration (or power method) is one of the fundamental algorithms for solving large-scale Eigenproblem [Saa11]. Briefly speaking, for a given matrix $\mathbf{U} \in \mathbb{R}^{n \times m}$, Vector iteration finds the leading eigenvector corresponding to the largest eigenvalue by repeating the following updating rule from a randomly initialized $\mathbf{v}^{(0)} \in \mathbb{R}^m$.

$$\mathbf{v}^{(k+1)} = \frac{\mathbf{U}\mathbf{v}^{(k)}}{||\mathbf{U}\mathbf{v}^{(k)}||}.$$

It is known that, as k increases, $\mathbf{v}^{(k+1)}$ converges to the leading eigenvector [Saa11].

ARNOLDI, which is a subspace iteration method, extends Vector iteration to find k leading eigenvectors simultaneously. Specifically, it finds the k eigenvectors from a subspace called Krylov space, which is spanned by $\{\mathbf{v}, \mathbf{U}\mathbf{v}, \ldots, \mathbf{U}^j\mathbf{v}\}$, where $j \geq k-1$. Implicitly Restarted ARNOLDI Method (IRAM) is one of the most advanced techniques for ARNOLDI [Saa11]. Briefly speaking, IRAM only keeps k orthonormal vectors which are a basis of the Krylov space, and updates the basis until it converges, then computes the k leading eigenvectors from the basis. One virtue of IRAM is *reverse communication interface*, which enables users to compute Eigendecomposition by viewing ARNOLDI as a black box. Specifically, the leading k eigenvectors of a square matrix $\mathbf{U}$ are obtained as follows:

S1. User initializes an instance of IRAM.

S2. IRAM returns $\mathbf{v}^{(j)}$ (initially $\mathbf{v}^{(0)}$).

S3. User computes $\mathbf{v}' \leftarrow \mathbf{U}\mathbf{v}^{(j)}$, and gives $\mathbf{v}'$ to IRAM.

S4. After an internal process, IRAM returns new vector $\mathbf{v}^{(j+1)}$.

S5. Repeat steps **S3** and **S4** until the internal variables in IRAM converges.

S6. IRAM computes eigenvalues and eigenvectors from its internal variables, and returns them.

For details of IRAM and *reverse communication interface*, we refer interested readers to [LSY98, Saa11].

8.2.2 Problem Definition

As discussed in Section 8.3.3, the scalability bottleneck of the state-of-the-art algorithms for high-order Tucker decomposition, defined in Section 8.2.1.3, is the explosive increase in memory requirements. In this chapter, we aim to perform high-order Tucker decomposition while keeping memory requirements manageable, as described in Problem 8.1.

Problem 8.1: Scalable High-order Tucker Decomposition

1. **Given:**
   - a large-scale high-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, which may or may not fit in main memory
   - ranks $J_1, \ldots, J_N$
2. **Perform:** Tucker decomposition (see Section 8.2.1.3)
3. **with:** low memory requirements.

8.3 Observation: "Materialization Bottleneck"

We describe the major challenges in scaling Tucker decomposition in Section 8.3.1. Then, in Section 8.3.2, we briefly survey the literature on scalable Tucker decomposition to see how these challenges have been addressed. However, we notice that existing methods still commonly suffer from *M-Bottleneck*, which is described in Section 8.3.3.

8.3.1 Intermediate Data Explosion

The most important challenge in scaling Tucker decomposition is the intermediate data explosion problem which was first identified in [KS08] (Definition 8.7). It states that a naive implementation of Algorithm 8.1, especially the computation of $[\mathcal{X} \times_{-n} \{\mathbf{A}^T\}]_{(n)}$, can produce huge intermediate data that do not fit in memory or even on a disk. We shall refer to this naive method as BaselineNaive.

Definition 8.7: Intermediate Data Explosion in BaselineNaive [KS08]

Let $nnz(\mathcal{X})$ be the number of non-zero entries in $\mathcal{X}$. In Algorithm 8.1, naively computing $\mathcal{X} \times_{-n} \{\mathbf{A}^T\}$ requires $O(nnz(\mathcal{X}) \cdot \prod_{p \neq n} J_p)$ space for intermediate data.

For example, if we assume a 5-order tensor with $nnz(\mathcal{X}) = 100$ millions and $J_n = 10$ for all n, $nnz(\mathcal{X}) \prod_{p \neq n} J_p = 1$ trillions. Thus, if single-precision floating-point numbers are used, computing $\mathcal{X} \times_{-n} \{\mathbf{A}^T\}$ requires about 4TB space, which exceeds the capacity of a typical hard disk as well as RAM.

8.3.2 Scalable Tucker Decomposition

In this section, we introduce recent Tucker decomposition methods that alleviate the intermediate data explosion.

Memory Efficient Tucker (MET) [KS08]: MET carefully orders the computation of $\mathcal{X} \times_{-n} \{\mathbf{A}^T\}$ in Algorithm 8.1 so that space required for intermediate data is reduced. Let $\mathcal{Y} = \mathcal{X} \times_{-n} \{\mathbf{A}^T\}$. Instead of computing entire $\mathcal{Y}$ at a time, MET computes a part of it at a time. Depending on the unit computed at a time, MET has various versions, and MET(2) is the most space-efficient one. In MET(2), each fiber (Definition 8.1) of $\mathcal{X}$ is computed at a time. The specific equation when $\mathcal{X}$ is 3-order is as follows:

$$\mathcal{Y}(:, j_2, j_3) \leftarrow \overbrace{\mathcal{X} \bar{\times}_2 \mathbf{a}^{(2)}_{j_2} \bar{\times}_3 \mathbf{a}^{(3)}_{j_3}}^{I_1}. \tag{8.3}$$

The amount of intermediate data produced during the computation of a fiber in $\mathcal{Y}$ by Eq. (8.3) is only $O(I_1)$. This amount is the same for general N-order tensors. MET(2) is one of the most space-optimized tensor decomposition methods, and we shall refer to MET(2) as BaselineOpt from now on.

Hadoop Tensor Method (HATEN2) [JPF+16]: HATEN2, in the same spirit as MET, carefully orders the computation of $\mathcal{X} \times_{-n} \{\mathbf{A}^T\}$ in Algorithm 8.1 on MAPREDUCE so that the amount of intermediate data and the number of MAPREDUCE jobs are reduced. Specifically, HATEN2 first computes $\mathcal{X} \times_p (\mathbf{A}^{(p)})^T$ for each $p \neq n$, then combines the results to obtain $\mathcal{X} \times_{-n} \{\mathbf{A}^T\}$. However, HATEN2 requires $O(N \cdot nnz(\mathcal{X}) \cdot \sum_{p \neq n} J_p)$ disk space for intermediate data, which is much larger than $O(I_n)$ space that BaselineOpt requires.

Other Work Related to Scalable Tucker Decomposition: Several algorithms were proposed for the case when the input tensor $\mathcal{X}$ is dense so that it cannot fit in memory. Specifically, [Tso10] uses random sampling of non-zero entries to sparsify $\mathcal{X}$, and [ABK16] distributes the entries of $\mathcal{X}$ across multiple machines. However, in this chapter, we assume that $\mathcal{X}$ is a large but sparse tensor, which is

Table 8.2: **The S-HOT family is space efficient**. The S-HOT family requires orders of magnitude less space than state-of-the-art methods. As an example, we assume a tensor where $N = 5$, $I_n = I = 10$ millions for every mode n, and $nnz(\mathcal{X}) = 100$ millions. We also assume that $J_n = J = 10$ for every mode n, and $B = 40$MB, where B is the memory budget for caching in S-HOT$_{\text{CACHE}}$. The space required by IRAM is included in the space for output data. Note that all the methods have the same convergence properties (see Observation 8.1 in Section 8.4).

Methods	Locations	Input Data (in Theory)	Output Data (in Theory)	Intermediate Data (in Theory)
BaselineNaive	Memory	$O(N \cdot nnz(\mathcal{X}))$	$O(NIJ + J^N)$	$O(J^{N-1} \cdot nnz(\mathcal{X}))$
BaselineOpt [KS08]	Memory	$O(N \cdot nnz(\mathcal{X}))$	$O(NIJ + J^N)$	$O(IJ^{N-1})$
HATEN2 [JPF+16]	Memory Disk	- $O(N \cdot nnz(\mathcal{X}))$	- $O(NIJ + J^N)$	$O(J^{N-1}$ + max. degree*) $O(IJ^{N-1} + NJ \cdot nnz(\mathcal{X})^2)$
S-HOT$_{\text{SPACE}}$	Memory Disk	- $O(N \cdot nnz(\mathcal{X}))$	$O(NIJ + J^N)$ -	$O(I + J^{N-1})$ -
S-HOT$_{\text{SCAN}}$	Memory Disk	- $O(N^2 \cdot nnz(\mathcal{X}))$	$O(NIJ + J^N)$ -	$O(J^{N-1})$ -
S-HOT$_{\text{CACHE}}$	Memory Disk	- $O(N^2 \cdot nnz(\mathcal{X}))$	$O(NIJ + J^N)$ -	$O(B + J^{N-1})$ -

* the degree of a mode-n index is the number of non-zero entries with the index (see Definition 8.8 for a formal definition of degree).

Methods	Locations	Input Data (in Example)	Output Data (in Example)	Intermediate Data (in Example)
BaselineNaive	Memory	$\sim$ 2GB	$\sim$ 2GB	$\sim$ 4TB
BaselineOpt [KS08]	Memory	$\sim$ 2GB	$\sim$ 2GB	$\sim$ 400GB
HATEN2 [JPF+16]	Memory Disk	- $\sim$ 2GB	- $\sim$ 2GB	$\gtrsim$ 40KB $\sim$ 500GB
S-HOT$_{\text{SPACE}}$	Memory Disk	- $\sim$ 2GB	$\sim$ 2GB -	$\sim$ 40MB -
S-HOT$_{\text{SCAN}}$	Memory Disk	- $\sim$ 10GB	$\sim$ 2GB -	$\sim$ 40KB -
S-HOT$_{\text{CACHE}}$	Memory Disk	- $\sim$ 10GB	$\sim$ 2GB -	$\sim$ 40MB -

more common in real-world applications. Moreover, our method stores $\mathcal{X}$ in disk, and thus its memory requirement does not depend on the number of non-zero entries of $\mathcal{X}$ (i.e., $nnz(\mathcal{X})$).

Our work is also related to recent variations of Tucker decomposition that are specifically tailored for high-order tensor analysis, such as hierarchical Tucker decomposition [Gra10, PCVS15]. However, such models require additional knowledge that may be application dependent. Note that our method is a scalable algorithm for the original Tucker decomposition with no modification. We leave exploration of such variations for future work.

8.3.3 Materialization Bottleneck

Although BaselineOpt and HATEN2 successfully reduce the space required for intermediate data produced while $\mathbf{Y}_{(n)} \leftarrow [\mathcal{X} \times_{-n} \{\mathbf{A}^T\}]_{(n)}$ is computed, they have an important limitation. Both algorithms materialize $\mathbf{Y}_{(n)}$, but its size $O(I_n \prod_{p \neq n} J_p)$ is usually huge, mainly due to I_n, and more seriously, it grows rapidly as N, I_n or $\{J_n\}_{n=1}^N$ increases. For example, as illustrated in Figure 8.2 in Section 8.1, if we assume a 5-order tensor with $I_n = 10$ millions and $J_p = 10$ for every $p \neq n$, then $I_n \prod_{p \neq n} J_p = 100$ billions. Thus, if single-precision floating-point numbers are used, materializing $\mathbf{Y}_{(n)}$ in a dense matrix format requires about 400GB space, which exceeds the capacity of typical RAM. Note that simply storing $\mathbf{Y}_{(n)}$ in a sparse matrix format does not solve the problem since $\mathbf{Y}_{(n)}$ is usually dense.

Considering this fact and the results in Section 8.3.2, we summarize the amount of intermediate data required during the whole process of tucker decomposition in each algorithm in Table 8.2. Our proposed S-HOT algorithms, which are discussed in detail in the following section, require several orders of magnitude less space for intermediate data.

8.4 Proposed Algorithm: S-HOT

In this section, we propose a set of algorithms, which we call the S-HOT family, for scalable high-order Tucker decomposition. We first provide an overview of the S-HOT family. Then, we discuss the naive version of S-HOT. Next, we present three different versions of S-HOT with distinct advantages.

8.4.1 Overview

The S-HOT family, which we develop in the following subsections, avoids *M-Bottleneck* caused by the materialization of $\mathcal{Y}$. They enable high-order Tucker decomposition to be performed even in an off-the-shelf workstation.

Throughout this section, we focus on memory-efficient computation of the following two steps (lines 4 and 5 of Algorithm 8.1):

$$\mathbf{Y}_{(n)} \leftarrow [\mathcal{X} \times_{-n} \{\mathbf{A}^T\}]_{(n)} \ (\in \mathbb{R}^{I_n \times (\prod_{p \neq n} J_p)})$$
$$\mathbf{A}^{(n)} \leftarrow \text{ top-}J_n \text{ left singular vectors of } \mathbf{Y}_{(n)}.$$

Our key idea is to tightly integrate the above two steps, and compute the singular vectors through IRAM directly from $\mathcal{X}$ without materializing the entire $\mathcal{Y}$ at once. We also use the fact that top-J_n left singular vectors of $\mathbf{Y}_{(n)}$ are equivalent to the top-J_n eigenvectors of $\mathbf{Y}_{(n)}\mathbf{Y}_{(n)}{}^T \in \mathbb{R}^{I_n \times I_n}$. Specifically, if we use *reverse communication interface* of IRAM, the above two steps are computed by simply updating $\mathbf{v}'$ repeatedly as follows:

$$\mathbf{v}' \leftarrow \mathbf{Y}_{(n)}\mathbf{Y}_{(n)}{}^T\mathbf{v}, \tag{8.4}$$

where we do not need to materialize $\mathbf{Y}_{(n)}$ (and thus we can avoid *M-Bottleneck*) if we are able to update $\mathbf{v}'$ directly from the $\mathcal{X}$. Note that, using IRAM does not change the result of the above two steps. Thus, final results of Tucker decomposition are also not changed, while space requirements are reduced drastically.

The remaining problem is how to update $\mathbf{v}'$ directly from $\mathcal{X}$, which is stored in disk, without materializing $\mathbf{Y}_{(n)}$. To address this problem, we first examine a naive method extending BaselineOpt and then eventually propose S-HOT$_{\text{SPACE}}$, S-HOT$_{\text{SCAN}}$, and S-HOT$_{\text{CACHE}}$, which are three versions of S-HOT with distinct advantages.

Note that all our ideas described in this section do not change the outputs of BaselineNaive and BaselineOpt. Thus, all versions of S-HOT have the same convergence properties of BaselineNaive and BaselineOpt, as described in Observation 8.1.

Observation 8.1: Convergence Property of S-HOT

When all initial conditions are identical, S-HOT$_{\text{SPACE}}$, S-HOT$_{\text{SCAN}}$, and S-HOT$_{\text{CACHE}}$ give the same result of BaselineNaive and BaselineOpt after the same number of iterations.

8.4.2 Naive Version: S-HOT$_{\text{NAIVE}}$

How can we avoid *M-Bottleneck*? In other words, how can we compute Eq. (8.4) without materializing the entire $\mathcal{Y}$? We describe S-HOT$_{\text{NAIVE}}$, which computes $\mathcal{Y}$ fiber by fiber, for computing Eq. (8.4). Thus, S-HOT$_{\text{NAIVE}}$ computes $\mathbf{v}'$ progressively on the basis of each column vector of $\mathbf{Y}_{(n)}$, which corresponds to a fiber in $\mathcal{Y}$, as follows:

$$\mathbf{v}' \leftarrow \mathbf{Y}_{(n)}{\mathbf{Y}_{(n)}}^T\mathbf{v} = \sum_j \mathbf{y}_j \left({\mathbf{y}_j}^T\mathbf{v}\right), \tag{8.5}$$

where $\mathbf{y}_j \in \mathbb{R}^{I_n}$ is a column vector of $\mathbf{Y}_{(n)}$.

This equation can be reformulated by $\mathcal{X}$ and $\{\mathbf{A}^T\}$. For ease of explanation, let $\mathcal{X}$ be a 3-order tensor. For each column vector $\mathbf{y}_j$, there exists a fiber $\mathcal{Y}(:, j_2, j_3)$ corresponding to $\mathbf{y}_j$. By plugging Eq. (8.3) into Eq. (8.5), we obtain

$$\begin{aligned}\mathbf{v}' \leftarrow \sum_j \mathbf{y}_j \left({\mathbf{y}_j}^T\mathbf{v}\right) &= \sum_{(j_2,j_3)} \mathcal{Y}(:, j_2, j_3)\left(\mathcal{Y}(:, j_2, j_3)^T\mathbf{v}\right) \\ &= \sum_{(j_2,j_3)} \left(\mathcal{X}\bar{\times}_2\mathbf{a}^{(2)}_{j_2}\bar{\times}_3\mathbf{a}^{(3)}_{j_3}\right)\left(\left(\mathcal{X}\bar{\times}_2\mathbf{a}^{(2)}_{j_2}\bar{\times}_3\mathbf{a}^{(3)}_{j_3}\right)^T\mathbf{v}\right).\end{aligned}$$

As clarified in Eq. (8.3), $\mathcal{X}\bar{\times}_2\mathbf{a}^{(2)}_{j_2}\bar{\times}_3\mathbf{a}^{(3)}_{j_3}$ is computed within $O(I_1)$ space, which is significantly smaller than space required for $\mathbf{Y}_{(n)}$.

However, S-HOT$_{\text{NAIVE}}$ is impractical because the number of scans of $\mathcal{X}$ increases explosively, as Lemma 8.1 and Lemma 8.2 state.

Lemma 8.1: Scan Cost of Computing a Fiber

Computing a fiber *on the fly* requires a complete scan of $\mathcal{X}$.

Proof. Computing a fiber consists of multiple mode-n vector products. Each mode-n vector product is considered as a weighted sum of $(N-1)$-order section of $\mathcal{X}$ as follows:

$$\mathcal{X} \bar{\times}_n \mathbf{v} = \sum_{i_n=1}^{I_n} \mathcal{X}(\underbrace{:, \ldots, :}_{n-1}, i_n, \underbrace{:, \ldots, :}_{N-n}) \mathbf{v}(i_n). \tag{8.6}$$

Thus, a complete scan of $\mathcal{X}$ is required to compute a fiber. ■

Lemma 8.2: Minimum Scan Cost of S-HOT$_{\text{NAIVE}}$

Let B be the memory budget, i.e., the number of floating-point numbers that can be stored in memory at once. Then, S-HOT$_{\text{NAIVE}}$ requires at least $\frac{I_n}{B} \prod_{p \neq n} J_p$ scans of $\mathcal{X}$ for computing Eq. (8.5).

Proof. Since we compute $\mathbf{y}_j \left({\mathbf{y}_j}^T \mathbf{v} \right)$, $\mathbf{y}_j$ should be stored in memory requiring I_n space, until the computation of ${\mathbf{y}_j}^T \mathbf{v}$ finishes. Thus, we can compute at most $\frac{B}{I_n}$ fibers at the same time within one scan of $\mathcal{X}$. Therefore, S-HOT$_{\text{NAIVE}}$ requires at least $\frac{I_n}{B} \prod_{p \neq n} J_p$ scans of $\mathcal{X}$ to compute Eq. (8.5). ■

8.4.3 Space-efficient Version: S-HOT$_{\text{SPACE}}$

How can we avoid the explosion in the number of scans of the input tensor required in S-HOT$_{\text{NAIVE}}$? We propose S-HOT$_{\text{SPACE}}$, which computes Eq. (8.4) within two scans of $\mathcal{X}$. S-HOT$_{\text{SPACE}}$ progressively computes $\mathbf{v}'$ from each row vector of $\mathbf{Y}_{(n)}$. Specifically, $\mathbf{v}'$ is computed by:

$$1 \leq \forall i \leq I_n, \ \mathbf{v}'(i) \leftarrow \bar{\mathbf{y}}_i {\mathbf{Y}_{(n)}}^T \mathbf{v} = \bar{\mathbf{y}}_i \sum_{j=1}^{I_n} \mathbf{v}(j) \bar{\mathbf{y}}_j^T \tag{8.7}$$

where $\bar{\mathbf{y}}_i$ is the ith row vector of $\mathbf{Y}_{(n)}$, which corresponds to an $(N-1)$-order segment of $\mathcal{Y}$ where the mode-n index is fixed to i. When entire $\mathcal{Y}$ does not fit in memory, Eq. (8.7) should be computed in the following two steps:

$$\mathbf{s} \leftarrow \sum_{i=1}^{I_n} \mathbf{v}(i) \bar{\mathbf{y}}_i^T \tag{8.8}$$

$$1 \leq \forall i \leq I_n, \ \mathbf{v}'(i) \leftarrow \bar{\mathbf{y}}_i \mathbf{s}. \tag{8.9}$$

This is since we cannot store all $\bar{\mathbf{y}}_i$ in memory until the computation of $\sum_{i=1}^{I_n} \mathbf{v}(i) \bar{\mathbf{y}}_i^T$ finishes. Algorithm 8.2 gives a formal description of S-HOT$_{\text{SPACE}}$, and Lemma 8.3 states the number of scans required in the algorithm

Lemma 8.3: Scan Cost of S-HOT$_{\text{SPACE}}$

S-HOT$_{\text{SPACE}}$ requires two scans of $\mathcal{X}$ for computing Eq. (8.4).

Proof. Each $\bar{\mathbf{y}}_i$ can be computed as follows.

$$\bar{\mathbf{y}}_i \leftarrow [\mathcal{X} \times_{-n} \{\mathbf{A}^T\}]_{(n)}(i,:) = \sum_{p \in \Omega_i^{(n)}(\mathcal{X})} \mathcal{X}(p) \times_{-n} \{\mathbf{A}^T\} = \sum_{p \in \Omega_i^{(n)}(\mathcal{X})} \mathcal{X}(p) \left[\circ_p^{-n} \{\mathbf{A}\}\right]_{(n)},$$

where p is a tuple $(i_1, ..., i_N)$ whose mode-n index is fixed to i; $\mathcal{X}(p)$ is an entry specified by p. Based on each $\bar{\mathbf{y}}_i$, Eq. (8.8) can be computed progressively as follows:

$$\mathbf{s} \leftarrow \sum_{i=1}^{I_n} \mathbf{v}(i) \bar{\mathbf{y}}_i^T = \sum_{i=1}^{I_n} \mathbf{v}(i) \sum_{p \in \Omega_i^{(n)}(\mathcal{X})} \mathcal{X}(p) \left[\circ_p^{-n} \{\mathbf{A}\}\right]_{(n)} = \sum_{p \in \Omega(\mathcal{X})} \mathbf{v}(i_n) \mathcal{X}(p) \left[\circ_p^{-n} \{\mathbf{A}\}\right]_{(n)}.$$

Thus, computing Eq. (8.8) requires only one scan of $\mathcal{X}$. Similarly, Eq. (8.9) also can be computed within one scan of $\mathcal{X}$. Therefore, Eq. (8.7), which consists of Eq. (8.8) and Eq. (8.9), can be computed within two scans of $\mathcal{X}$. ■

In Lemma 8.4, we prove the amount of space required by S-HOT$_{\text{SCAN}}$ for intermediate data.

Lemma 8.4: Space Complexity in S-HOT$_{\text{SPACE}}$

The update step of S-HOT$_{\text{SPACE}}$ (lines 12-18 of Algorithm 8.2) requires

$$O\left(\max_{1 \le n \le N} (I_n + \prod_{p=1}^{N} J_p / J_n)\right)$$

memory space for intermediate data.

Proof. S-HOT$_{\text{SPACE}}$ maintains $\mathbf{v}$, $\mathbf{v}'$, and $\mathbf{s}$ in its update step. When each factor matrix $\mathbf{A}^{(n)}$ is updated, $\mathbf{v}$ and $\mathbf{v}'$ are I_n by 1 vectors, and $\mathbf{s}$ is a $\prod_{p=1}^{N} J_p/J_n$ by 1 vector. Thus, $O(I_n + \prod_{p=1}^{N} J_p/J_n)$ space is required in the update step for each factor matrix $\mathbf{A}^{(n)}$. Since the factor matrices are update one by one, $O(\max_{1 \le n \le N}(I_n + \prod_{p=1}^{N} J_p/J_n))$ space is required at a time. ■

8.4.4 Faster Version: S-HOT$_{\text{SCAN}}$

How can we further reduce the number of required scans of the input tensor? We propose S-HOT$_{\text{SCAN}}$, which halves the number of scans of $\mathcal{X}$ at the expense of requiring multiple (disk-resident) copies of $\mathcal{X}$ sorted by different mode indices. In effect, S-HOT$_{\text{SCAN}}$ trades off disk space for speed.

Our key idea for the further optimization is to compute J_n right leading singular vectors of $\mathbf{Y}_{(n)}$, which are eigenvectors of $\mathbf{Y}_{(n)}^T \mathbf{Y}_{(n)}$, and use the result to compute the left singular vectors. Let $\mathbf{Y}_{(n)} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ be the SVD of $\mathbf{Y}_{(n)}$. Then,

$$\mathbf{Y}_{(n)} \mathbf{V} \mathbf{\Sigma}^{-1} = \mathbf{U} \mathbf{\Sigma} \mathbf{V}^T \mathbf{V} \mathbf{\Sigma}^{-1} = \mathbf{U}. \tag{8.10}$$

Thus, left singular vectors are obtained from right singular vectors.

S-HOT$_{\text{SCAN}}$ computes top-J_n right singular vectors of $\mathbf{Y}_{(n)}$ by updating the vector $\mathbf{w} \in \mathbb{R}^{\prod_{p \neq n} J_p}$ as follows:

$$\mathbf{w}' \leftarrow \mathbf{Y}_{(n)}^T \mathbf{Y}_{(n)} \mathbf{w} = \sum_{i=1}^{I_n} (\bar{\mathbf{y}}_i^T \mathbf{w}) \bar{\mathbf{y}}_i. \tag{8.11}$$

Algorithm 8.2 S-HOT$_{\text{SPACE}}$ and S-HOT$_{\text{SCAN}}$

Input: (1) $\mathcal{X}$: an N-order tensor of $\mathbb{R}^{I_1 \times \cdots \times I_N}$,
(2) $J_1, \ldots, J_N$: rank for each mode,
(3) T: number of iterations.

Output: (1) $\{\mathbf{A}\}$: a set of factor matrices $\{\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}\}$ where $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times J_n}$,
(2) $\mathcal{G}$: an N-order core tensor of $\mathbb{R}^{J_1 \times \cdots \times J_N}$.

initialize $\{\mathbf{A}\}$
for $t \leftarrow 1..T$ **do**
  for $n \leftarrow 1..N$ **do**
    $\mathbf{v} \leftarrow$ `IRAM_Init`(I_n, J_n)
    repeat
      $\mathbf{v}' \leftarrow$ `SHOT_Space_Update`$(\mathcal{X}, n, \mathbf{v})$ or `SHOT_Scan_Update`$(\mathcal{X}, n, \mathbf{v})$
      $\mathbf{v} \leftarrow$ `IRAM_DoIter`$(\mathbf{v}')$
    until `IRAM_IsConverged()`
    $\mathbf{A}^{(n)} \leftarrow$ `GetSingularVector()`
$\mathcal{G} \leftarrow \mathcal{X} \times \{\mathbf{A}^T\}$
return $\mathcal{G}, \{\mathbf{A}\}$

procedure SHOT_SPACE_UPDATE($\mathcal{X}$, n, $\mathbf{v}$) ▷ Update in S-HOT$_{\text{SPACE}}$
  $\mathbf{s} \leftarrow \mathbf{0}; \mathbf{v}' \leftarrow \mathbf{0}$
  for each $(i_1, ..., i_N) \in \Omega(\mathcal{X})$ **do**
    $\mathbf{s} \leftarrow \mathbf{s} + \mathbf{v}(i_n)\mathcal{X}(i_1, ..., i_N) \left[\circ^{-n}_{(i_1,...,i_N)}\{\mathbf{A}\}\right]_{(n)}$
  for each $(i_1, ..., i_N) \in \Omega(\mathcal{X})$ **do**
    $\mathbf{v}'(i_n) \leftarrow \mathbf{v}'(i_n) + \mathbf{s}^T\mathcal{X}(i_1, ..., i_N) \left[\circ^{-n}_{(i_1,...,i_N)}\{\mathbf{A}\}\right]_{(n)}$
  return $\mathbf{v}'$

procedure SHOT_SCAN_UPDATE($\mathcal{X}$, n, $\mathbf{v}$) ▷ Update in S-HOT$_{\text{SCAN}}$
  $\mathbf{w}' \leftarrow \mathbf{0}$
  for $i \leftarrow 1...I_n$ **do**
    $\mathbf{y}_i \leftarrow \sum_{(i_1,...,i_N) \in \Omega^{(n)}_i(\mathcal{X})} \mathcal{X}(i_1, ..., i_N) \left[\circ^{-n}_{(i_1,...,i_N)}\{\mathbf{A}\}\right]_{(n)}$
    $\mathbf{w}' \leftarrow \mathbf{w}' + (\mathbf{y}_i{}^T\mathbf{w})\mathbf{y}_i$
    deallocate $\mathbf{y}_i$
  return $\mathbf{w}'$

The virtue of S-HOT$_{\text{SCAN}}$ is that it requires only one scan of $\mathcal{X}$ for calculating Eq. (8.11), as Lemma 8.6 states.

Lemma 8.5: Scan Cost for Computing $\bar{\mathbf{y}}_i$

$\bar{\mathbf{y}}_i$ can be computed by scanning only the entries of $\mathcal{X}$ whose mode-n index is i.

Proof. Proven by Eq. (8.10). ■

Lemma 8.6: Scan Cost of S-HOT$_{\text{SCAN}}$

S-HOT$_{\text{SCAN}}$ computes Eq. (8.11) within one scan of $\mathcal{X}$ when $\mathcal{X}$ is sorted by the mode-n index.

Proof. By Lemma 8.5, only a section of tensor whose mode-n index is i is required for computing $\bar{\mathbf{y}}_i$. If $\mathcal{X}$ is sorted by the nth mode index, we can sequentially compute each $\mathbf{y}_i$ *on the fly*. Moreover, once $\bar{\mathbf{y}}_i$ is computed, we can immediately compute $(\bar{\mathbf{y}}_i^T \mathbf{w})\bar{\mathbf{y}}_i$. After that, we do not need $\bar{\mathbf{y}}_i$ anymore, and can discard it. Thus, Eq. (8.11) can be computed *on the fly* within only a single scan of $\mathcal{X}$. ■

We satisfy the sort constraint for all modes by simply keeping N copies of $\mathcal{X}$ sorted by each mode index.

A formal description for S-HOT$_{\text{SCAN}}$ is in Algorithm 8.2. It is assumed that $\mathbf{w}$ is initialized by passing $(\prod_{p \neq n} J_p,\ J_n)$ instead of $(I_n,\ J_n)$ at line 4. Although one additional scan of $\mathcal{X}$ is required for computing left singular vectors from the obtained right singular vectors (Eq. (8.10)), S-HOT$_{\text{SCAN}}$ still requires fewer scans of $\mathcal{X}$ than S-HOT$_{\text{SPACE}}$ since it saves one scan during $\mathbf{w}'$ computation, which is repeated more frequently.

In Lemma 8.7, we prove the amount of space required by S-HOT$_{\text{SCAN}}$ for intermediate data.

Lemma 8.7: Space Complexity of S-HOT$_{\text{SCAN}}$

The update step of S-HOT$_{\text{SCAN}}$ (lines 19-25 of Algorithm 8.2) requires

$$O\left(\max_{1 \leq n \leq N} \left(\prod_{p=1}^{N} J_p / J_n\right)\right)$$

memory space for intermediate data.

Proof. In its update step for each factor matrix $\mathbf{A}^{(n)}$, S-HOT$_{\text{SCAN}}$ maintains $\mathbf{w}$, $\mathbf{w}'$, and $\mathbf{y}_i$ at a time. All of them are $\prod_{p=1}^{N} J_p / J_n$ by 1 vectors. Thus, $O(\prod_{p=1}^{N} J_p / J_n)$ space is required in the update step for each factor matrix $\mathbf{A}^{(n)}$. Since the factor matrices are updated one by one, $O(\max_{1 \leq n \leq N}(\prod_{p=1}^{N} J_p / J_n))$ space is required at a time. ■

Algorithm 8.3 S-HOT$_{\text{CACHE}}$: the fastest version of S-HOT

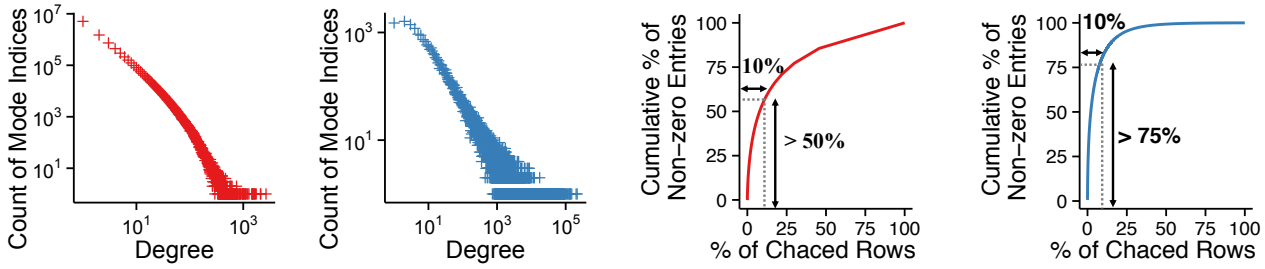
Input: (1) $\mathcal{X}$: an N-order tensor of $\mathbb{R}^{I_1 \times \cdots \times I_N}$, (2) $J_1, \ldots, J_N$: rank for each mode,
(3) T: number of iterations, (4) B (or, equivalently, $k_1, ..., k_N$): memory budget for caching.

Output: (1) $\{\mathbf{A}\}$: a set of factor matrices $\{\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}\}$ where $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times J_n}$,
(2) $\mathcal{G}$: an N-order core tensor of $\mathbb{R}^{J_1 \times \cdots \times J_N}$.

initialize $\{\mathbf{A}\}$
for $t \leftarrow 1..T$ **do**
  for $n \leftarrow 1..N$ **do**
    $\mathbf{v} \leftarrow$ `IRAM_Init`(I_n, J_n)
    `Map`$\leftarrow$ `Cache`$(\mathcal{X}, n, k_n)$
    repeat
      $\mathbf{v}' \leftarrow$ `Update`$(\mathcal{X}, n, \mathbf{v},$ `Map`$)$
      $\mathbf{v} \leftarrow$ `IRAM_DoIter`$(\mathbf{v}')$
    until `IRAM_IsConverged`()
    $\mathbf{A}^{(n)} \leftarrow$ `GetSingularVector`()
$\mathcal{G} \leftarrow \mathcal{X} \times \{\mathbf{A}^T\}$
return $\mathcal{G}, \{\mathbf{A}\}$

procedure CACHE$(\mathcal{X}, n, k_n)$
  `Top`$\leftarrow$ top-k_n highest-degree mode-n indices
  `Map`$\leftarrow$ an empty map
  for each $i \in$`Top` **do**
    `Map.put`$\left(i, \sum_{(i_1,\ldots,i_N)\in\Omega_i^{(n)}(\mathcal{X})} x_{i_1\cdots i_N} \left[\circ_p^{-n}\{\mathbf{A}\}\right]_{(n)}\right)$
  return `Map`

procedure UPDATE$(\mathcal{X}, n, \mathbf{v},$ `Map`$)$
  $\mathbf{w}' \leftarrow \mathbf{0}$
  for $i \leftarrow 1...I_n$ **do**
    if $i \in$ `Map.keys()` **then** $\mathbf{y}_i \leftarrow$ `Map.get`(i)
    else $\mathbf{y}_i \leftarrow \sum_{(i_1,\ldots,i_N)\in\Omega_i^{(n)}(\mathcal{X})} \mathcal{X}(i_1, ..., i_N) \left[\circ_{(i_1,\ldots,i_N)}^{-n}\{\mathbf{A}\}\right]_{(n)}$
    $\mathbf{w}' \leftarrow \mathbf{w}' + (\mathbf{y}_i{}^T\mathbf{w})\mathbf{y}_i$
    deallocate $\mathbf{y}_i$
  return $\mathbf{w}'$



(a) Degree distribution in the author mode

(b) Degree distribution in the keyword mode

(c) % of non-zero entries required for computing cahced rows in the author mode

(d) % of non-zero entries required for computing cahced rows in the keyword mode

Figure 8.3: **Power-law degree distributions in a real-world tensor** (see Section 8.5.1 for a description of the tensor). (a) and (b) show the skewed degree distributions in the author and keyword modes, which are exploited by S-HOT$_{\text{CACHE}}$ for speedup. (c) and (d) show that S-HOT$_{\text{CACHE}}$ can avoid accessing many (e.g., 50-75%) non-zero entries by caching a small percentage (e.g., 10%) of rows.

8.4.5 Fastest Version: S-HOT$_{\text{CACHE}}$

How can we make good use of remaining memory when memory is underutilized by S-HOT$_{\text{SCAN}}$, which requires little space for intermediate data? We propose S-HOT$_{\text{CACHE}}$, which improves the speed of S-HOT$_{\text{SCAN}}$ by caching a part of intermediate data (i.e., some rows of $\mathbf{Y}_{(n)}$) in memory instead of computing all of them on-the-fly. Especially, within a given memory budget, S-HOT$_{\text{CACHE}}$ carefully decides the rows of $\mathbf{Y}_{(n)}$ to be cached so that the speed gain is maximized. A formal description of S-HOT$_{\text{CACHE}}$ is given in Algorithm 8.3.

Given a memory budget B for caching, let k_n be the maximum number of rows of $\mathbf{Y}_{(n)}$ that can be cached within B. When updating each factor matrix $\mathbf{A}^{(n)}$, S-HOT$_{\text{CACHE}}$ caches the k_n rows that are most expensive to compute. Such rows can be found by comparing the *degrees* of the mode-n indices (see Definition 8.8 for the definition of degree), as described in lines 13-18.

Definition 8.8: Degree of Mode Indices

The degree of each mode-n index i is defined as $|\Omega_i^{(n)}(\mathcal{X})|$, i.e., the number of non-zero entries whose mode-n index is i.

This is because computing each row $\mathbf{y}_i$ of $\mathbf{Y}_{(n)}$ takes time proportional the degree of mode-n index i (i.e., $|\Omega_i^{(n)}(\mathcal{X})|$), as shown in Eq. (8.10). The remaining steps for updating $\mathbf{A}^{(n)}$ are the same as those of S-HOT$_{\text{SCAN}}$ except for that the cached rows are used, as described in lines 19-26.

This careful choice of the rows of $\mathbf{Y}_{(n)}$ to be cached is crucial to speed up the algorithm. This is because, in real-world tensors, the degree of mode indices often follows a power-law distribution [CSN09], and thus there exist indices with extremely high degree (see Figures 8.3(a) and 8.3(b) for examples). By caching the rows of $\mathbf{Y}_{(n)}$ corresponding to such high-degree indices, S-HOT$_{\text{CACHE}}$ avoids accessing many non-zero entries (see Figures 8.3(c) and 8.3(d) for examples) and thus saves considerable computation time, as shown empirically in Section 8.5.5.

We prove the scan cost of S-HOT$_{\text{CACHE}}$ in Lemma 8.8 and the space complexity of S-HOT$_{\text{CACHE}}$ In Lemma 8.9.

Lemma 8.8: Scan Cost of S-HOT$_{\text{CACHE}}$

S-HOT$_{\text{CACHE}}$ computes Eq. (8.11) within one scan of $\mathcal{X}$ when $\mathcal{X}$ is sorted by the mode-n index.

Proof. Given cached rows, S-HOT$_{\text{CACHE}}$ computes Eq. (8.11) in the same way as does S-HOT$_{\text{SCAN}}$ only except for that S-HOT$_{\text{CACHE}}$ uses the cached rows. Thus, S-HOT$_{\text{CACHE}}$ and S-HOT$_{\text{SCAN}}$ require the same number of scans of $\mathcal{X}$, which is one, as shown in Lemma 8.6. We do not need an additional scan of $\mathcal{X}$ for the caching step if it is done while Eq. (8.11) is first computed. ■

Lemma 8.9: Space Complexity of S-HOT$_{\text{CACHE}}$

The update step of S-HOT$_{\text{CACHE}}$ (lines 19-26 of Algorithm 8.3) requires

$$O\left(B + \max_{1 \le n \le N} (\prod_{p=1}^{N} J_p / J_n)\right)$$

memory space for intermediate data.

Proof. In addition to those maintained in S-HOT$_{\text{SCAN}}$, which require $O\Big(\max_{1 \le n \le N} (\prod_{p=1}^{N} J_p / J_n)\Big)$ space at a time (see Lemma 8.7), S-HOT$_{\text{CACHE}}$ maintains the cached rows, whose size is within the budget B. Thus, $O(B + \max_{1 \le n \le N} (\prod_{p=1}^{N} J_p / J_n))$ space is required at a time. ■

8.5 Experiments

In this section, we review our experiments for answering the following questions:

- **Q1. Scalability:** How scalable is S-HOT compared to its state-of-the-art competitors with respect to the dimensionality, the rank, the order, and the number of non-zero entries?
- **Q2. S-HOT at Work:** Can S-HOT decompose real-world tensors that are both large-scale and high-order?
- **Q3. Effect of Memory Budget** How does the memory budget affect the speed of S-HOT$_{\text{CACHE}}$?
- **Q4. Effect of Data Skewness** How does the degree distribution of the input tensor affect the speed of S-HOT$_{\text{CACHE}}$?

8.5.1 Experimental Settings

Machine: We conducted all experiments on a machine with Intel Core i7 4700@3.4GHz (4 cores), 32GB RAM, and Ubuntu 14.04 trusty.

Datasets: We used both of synthetic and real-world tensors. We generated synthetic tensors where the degree of their mode indices follows a Zipf distribution, which is common in real-world data [Pow98,

AH02]. Specifically, we created tensors where the mode indices of the non-zero entries follow the following probability density function:

$$p(x) = \frac{x^{-\alpha}}{\sum_{k=1}^{\infty} \frac{1}{k^x}},$$

where α is a parameter determining the skewness of the distribution. We set the value of each entry to 1. However, this does not mean that S-HOT is limited to binary tensors or that our implementation is optimized for binary tensors. We chose binary tensors for simplicity. Generating realistic values, while we control each factor, is not a trivial task. As default parameter values, we used $N = 4$, $M = 10^5$, $I_n = 10^3$ for every n, $J_n = 8$ for every n, and $\alpha = 1.5$. These default values were chosen to effectively compare the scalability of competitors. All experiments using synthetic datasets were repeated nine times (three times for each of three randomly generated tensors), and reported values are the average of the multiple trials.

We also used a real-world tensor obtained from the Microsoft Academic Graph dataset [SSS+15] (a snapshot on Feb 5, 2016). The dataset contains 42 million papers; 1,283 conferences and 23,404 journals; 115 million authors; and 53,834 keywords used to annotate the topics of the papers. We modeled the dataset as a 4-order tensor whose modes are authors, venues, years, and keywords. The papers with missing attributes were ignored, and the final tensor was of size $9,380,418 \times 18,894 \times 2,016 \times 37,000$.

Implementations: Throughout all experiments, we used two baseline methods and three versions of our proposed method:

- BaselineNaive (Algorithm 8.1): a naive method computing $\mathcal{X} \times_{-n} \{\mathbf{A}\}$ in a straight-forward way,
- BaselineOpt [KS08]: the state-of-the-art memory-efficient Tucker-decomposition algorithm which computes $\mathcal{Y}$ fiber by fiber,
- S-HOT$_{\text{SPACE}}$ (Algorithm 8.2): the most space-efficient version of S-HOT,
- S-HOT$_{\text{SCAN}}$ (Algorithm 8.2): a faster version of S-HOT,
- S-HOT$_{\text{CACHE}}$ (Algorithm 8.3): the fastest version of S-HOT with a buffer in main memory. We set the size of the buffer so that it caches up to 30 rows of $\mathbf{Y}_{(n)}$ for each n-th mode unless otherwise stated. The size of the buffer never exceeded 200KB unless otherwise stated.

For BaselineOpt and BaselineNaive, we used the implementation in Tensor Toolbox 2.6 [BK07a], implemented in MATLAB. We excluded HATEN2 because HATEN2 is designed for Hadoop, and thus it takes too much time in a single machine. For example, in order to decompose a synthetic tensor with default parameters, HATEN2 took $10,700$ seconds for an iteration. It was almost *100× slower* than S-HOT$_{\text{SCAN}}$. Every version of S-HOT was implemented in C++ with OpenMP library and AVX instruction set.We used ARPACK [LSY98], which provides IRAM supporting *reverse communication interface*. It is worth noting that ARPACK is an underlying package for a built-in function called eigs(), which is provided in many popular numerical computing environments including SCIPY, GNU OCTAVE, and MATLAB. Therefore, S-HOT is numerically stable and has the similar reconstruction error with eigs() function in the above mentioned numerical computing environments. For fairness, we must note that, a fully optimized C++ implementation could potentially be faster than that of MATLAB although that is unlikely, since MATLAB is extremely well optimized for matrix operations. But in any case, our main contribution still holds: regardless of programming languages, S-HOT scales to much larger settings, thanks to our proposed *on-the-fly* computation (Eqs. (8.7) and (8.11)).

8.5.2 Q1: Scalability

We compared the scalability of the competing methods with respect to various factors: (1) the order, (2) the dimensionality, (3) the number of non-zero entries, and (4) the rank. Specifically, we measured the wall-clock time of a single iteration of each algorithm on synthetic tensors. Note that all the methods have the same convergence properties, as described in Observation 8.1 in Section 8.4.

Scalability w.r.t. the order: We investigated the scalability of the considered methods with respect to the order by controlling the order of the input tensor from 3 to 6 while fixing the other factors to their default values. As shown in Figure 8.1(a), S-HOT outperformed baselines. BaselineNaive failed to decompose the 4-order tensor because it suffers from the *intermediate explosion problem*. BaselineOpt, which avoids the problem, was more memory-efficient than BaselineNaive. However, it failed to decompose a tensor whose order is higher than 4 due to *M-Bottleneck*. On the contrary, every version of S-HOT successfully decomposed even the 6-order tensor. Especially, S-HOT$_{\text{CACHE}}$ is up to $50\times$ faster than S-HOT$_{\text{SPACE}}$ and S-HOT$_{\text{SCAN}}$.

Scalability w.r.t. the dimensionality: We investigated the scalability of the considered methods with respect to the dimensionality. Specifically, we increased the dimensionality I_n of every n from 10^3 to 10^7. That is, since the default order was 4, we increased the tensor from $10^3 \times 10^3 \times 10^3 \times 10^3$ to $10^7 \times 10^7 \times 10^7 \times 10^7$. As shown in Figure 8.1(b), S-HOT was several orders of magnitude scalable than the baselines. Specifically, BaselineNaive failed to decompose any 4-order tensor, and thus it does not appear in the plot. BaselineOpt failed to decompose tensors with dimensionality larger than 10^4 since the space for storing $\mathcal{Y}$ increases rapidly with respect to the size of dimensionality (*M-Bottleneck*). On the contrary, every version of S-HOT successfully decomposed the largest tensor of size $10^7 \times 10^7 \times 10^7 \times 10^7$. Moreover, the running times of S-HOT$_{\text{SCAN}}$ and S-HOT$_{\text{CACHE}}$ were almost constant since they solve the transposed problem, whose size is independent of the dimensionality. Between them, S-HOT$_{\text{CACHE}}$ is up to $6\times$ faster than S-HOT$_{\text{SCAN}}$. On the other hand, the running time of S-HOT$_{\text{SPACE}}$ depends on dimensionality, and it increased as the dimensionality became greater than 10^6. For smaller dimensionalities, however, the effect of dimensionality on its running time was negligible because the outer products (i.e., lines 15 and 17 of Algorithm 8.2) are the major bottleneck.

Scalability w.r.t. the rank: We investigated the scalability of the considered methods with respect to the rank. To show the difference between the competitors clearly, we set the dimensionality of the input tensor to $20,000$ in this experiment. However, the overall trends do not depend on the parameter values. As shown in Figure 8.1(c), the S-HOT had better scalability than baselines. Specifically, BaselineNaive failed to decompose any tensor and does not appear in this plot. BaselineOpt failed to decompose the tensors with rank larger than 6. On the contrary, every version of S-HOT successfully decomposed the tensors with larger ranks. Among them, S-HOT$_{\text{CACHE}}$ was up to $7\times$ faster than S-HOT$_{\text{SCAN}}$ and S-HOT$_{\text{SPACE}}$. S-HOT$_{\text{SCAN}}$ was faster than S-HOT$_{\text{SPACE}}$ but the difference between them decreased as the rank increased. This was because, as the rank increased, the outer products (i.e., lines 15 and 17 of Algorithm 8.2) became the major bottleneck, commonly in S-HOT$_{\text{SPACE}}$ and S-HOT$_{\text{SCAN}}$.

Scalability w.r.t. the number of non-zero entries: We investigated the scalability of the considered methods with respect to the number of non-zero entries. We increased the number of non-zero entries in the input tensor from 10^4 to 10^7. As shown in Figure 8.1(d), every version of S-HOT scaled near linearly with respect to the number of non-zero entries. This was because the S-HOT family scans most non-zero entries (especially, S-HOT$_{\text{SPACE}}$ and S-HOT$_{\text{SCAN}}$ scan all the non-zero entries), and processing each non-zero entry takes almost the same time. Among the different versions of S-HOT, S-HOT$_{\text{CACHE}}$ was $4\times$ faster than S-HOT$_{\text{SCAN}}$ and S-HOT$_{\text{SPACE}}$. With respect to the number of non-

Table 8.3: **S-HOT at work**. We applied S-HOT to the Microsoft Academic Graph dataset and obtained clusters of venues. Sample clusters are given in the table below.

CS-related	International Conference on Networking, Wired/Wireless Internet Communications, Database and Expert Systems Applications, Data Mining and Knowledge Discovery, IEEE Transactions on Robotics, ...
Nanotech.	Nature Nanotechnology, PLOS ONE, Journal of Experimental Nanoscience, Journal of Nanoscience and Nanotechnology, Journal of Semiconductors, Trends in Biotechnology, ...
Clinical	European Journal of Cancer, PLOS Biology, Clinical and Applied Thrombosis-Hemostasis, Journal of Infection Prevention, RBMC Clinical Pharmacology, Regional Anesthesia and Pain ...

zero entries, BaselineOpt showed better scalability than S-HOT since it *explicitly materializes* $\mathcal{Y}$. Once $\mathcal{Y}$ is materialized, since its size does not depend on the number of non-zero entries, the remaining tasks of BaselineOpt are not affected by the number of non-zero entries.

8.5.3 Q2: S-HOT at Work

We showed the scalability of S-HOT again a real-world tensor of size $9,380,418 \times 18,894 \times 2,016 \times 37,000$ from the Microsoft Academic Graph dataset (see Section 8.5.1). We note that, since this tensor is high-order and large-scale, both baseline algorithms failed to handle it running out of memory. However, every version of S-HOT successfully decomposed the tensor.

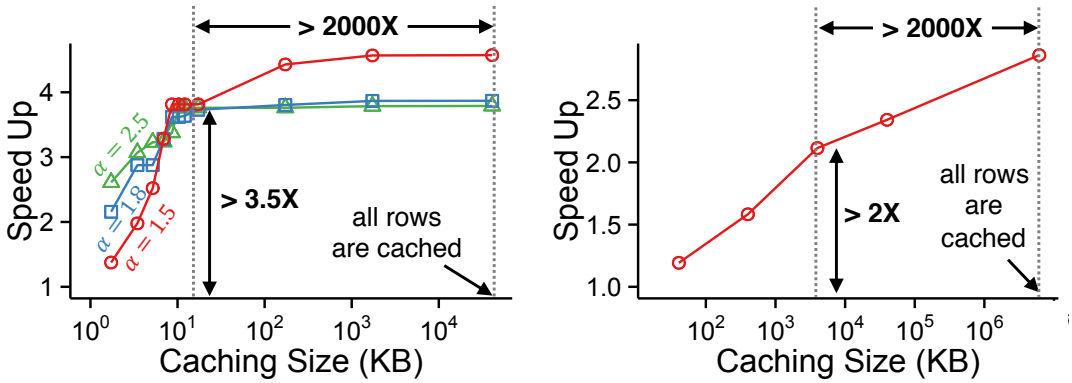
To better interpret the result of Tucker decomposition, we ran k-means clustering [AV07] where we treated each factor matrix as the low-rank embedding of the entities in the corresponding mode, as suggested in [KS08]. Specifically, for Tucker decomposition, we set the rank of each mode to 8 and run 30 iterations. For k-means clustering, we set the number of clusters to 400 and run 100 iterations.

Table 8.3 shows sample clusters in the venue mode. The first cluster contained many venues related to Computer Science. The second one contained many nano-technology-related venues such as Nautre Nanotechnology and Journal of Experimental Nanoscience. The third one had many venues related to Medical Science and Diseases. This result indicates that Tucker decomposition (by the S-HOT family) discovers meaningful concepts and groups entities related to each other. However, there is a vast array of methods for multi-aspect data analysis, and we leave a comparative study as to which one performs the best for future work.

8.5.4 Q3: Effect of the Memory Budget on the Speed of S-HOT$_{\text{CACHE}}$

We measured the effect of memory budget B on the speed of S-HOT$_{\text{CACHE}}$ using synthetic and real-world tensors. We used three 4-order tensors with dimensionality $20,000$ for each mode. All the tensors had 10^6 non-zero entries, while they had different degree distributions characterized by the skewness α of the Zipf distribution. We also used the Microsoft Academic Graph dataset described in Section 8.5.3.

Figure 8.4 shows the result with the synthetic tensors where we set the rank of each mode to 6. S-HOT$_{\text{CACHE}}$ tended to be faster as we use more memory for caching. However, the speed-up slowed down because S-HOT$_{\text{CACHE}}$ prioritizes rows to be cached by the degree of the corresponding mode indices, as described in Section 8.4.5. As the memory budget increased, S-HOT$_{\text{CACHE}}$ cached rows



(a) Speed-up in synthetic tensors

(b) Speed-up in a real-world tensor

Figure 8.4: **S-HOT$_{\text{CACHE}}$ speeds up computation on both synthetic and real-world tensors.** In the synthetic tensors, S-HOT$_{\text{CACHE}}$ achieves over $3.5\times$ *speed-up* by caching less than 0.05% of rows within the 10KB memory budget. In the real-world tensor, S-HOT$_{\text{CACHE}}$ achieves over $2\times$ speed-up by caching less than 0.05% of the rows within the 5MB memory budget.

corresponding to mode indices with smaller degree, which saved less computation. Notice that, with only the 10KB memory budget, S-HOT$_{\text{CACHE}}$ became over $3.5\times$ faster than S-HOT$_{\text{SCAN}}$, which does not use caching.

As shown in Figure 8.4(b), we obtained the same trend with the real-world tensor described in Section 8.5.1. Notice that, with only the 4MB memory budget, S-HOT$_{\text{CACHE}}$ became over $2\times$ faster than S-HOT$_{\text{SCAN}}$, which does not use caching. S-HOT$_{\text{SCAN}}$ saved much computation by caching a small number of rows due to the power-law degree distributions, shown in Figure 8.3.

8.5.5 Q4: Effect of the Skewness of Data on the Speed of S-HOT$_{\text{CACHE}}$

We measured the effect of the skewness of degree distribution on the speed of S-HOT$_{\text{CACHE}}$. To this end, we used three 4-order tensors with different degree distributions characterized by the skewness α of the Zipf distribution. All of them had 10^6 non-zero entries, and their dimensionality for each mode was $20,000$.

Figure 8.4(a) shows how rapidly the speed-up of S-HOT$_{\text{CACHE}}$ increased depending on the skewness α. The speed-up of S-HOT$_{\text{CACHE}}$ tended to increase faster in tensors with larger α. For example, with the 2KB memory budget, S-HOT$_{\text{CACHE}}$ achieved over $2.5\times$ speed-up in the tensor with $\alpha = 2.5$, while it achieved less than $1.5\times$ speed up in the tensor with $\alpha = 1.5$. This was because, with larger α, more non-zero entries were concentrated in few mode indices. For every realistic degree distribution with $\alpha > 1$, S-HOT$_{\text{CACHE}}$ achieved over $3.5\times$ speed-up with the 10KB memory budget.

8.6 Summary

In this chapter, we propose S-HOT, a scalable algorithm for high-order Tucker decomposition. S-HOT solves *M-Bottleneck*, which existing algorithms suffer from, by using an *on-the-fly* computation. We provide three versions of S-HOT: S-HOT$_{\text{SPACE}}$, S-HOT$_{\text{SCAN}}$, and S-HOT$_{\text{CACHE}}$, which provide an interesting trade-off between time and space. We theoretically and empirically show that all versions of S-HOT have better scalability than baselines. In summary, our contributions are as follows:

- **Bottleneck Resolution:** We identify *M-Bottleneck* (Figure 8.2), the scalability bottleneck of existing Tucker-decomposition algorithms and avoid it by a novel approach based on an *on-the-fly computation*.
- **Scalable Algorithm Design:** We propose S-HOT, a Tucker-decomposition algorithm that is carefully optimized for large-scale high-order tensors. S-HOT successfully decomposes *1000× larger* tensors than baselines algorithms (Figure 8.1) with identical convergence properties (Observation 8.1).
- **Theoretical Analysis:** We prove the amount of memory space and the number of data scans that the different versions of S-HOT require (Table 8.2 and Lemmas 8.3-8.9).

Reproducibility: The source code and datasets used in this chapter are available at *http://dm.postech.ac.kr/shot*.

Part II

Anomaly Detection

Chapter based on work that appeared at ICDM 2016 [SERF16] and the KAIS journal [SERF18].

Chapter 9

Finding Patterns and Anomalies in Dense Subgraphs

How do the k-core structures of real-world graphs look like? What are the common patterns and the anomalies? How can we exploit them for applications?

A k-core is the maximal subgraph in which all nodes have degree at least k. This concept has been applied to such diverse areas as hierarchical structure analysis, graph visualization, and graph clustering. Here, we explore pervasive patterns related to k-cores and emerging in graphs from diverse domains.

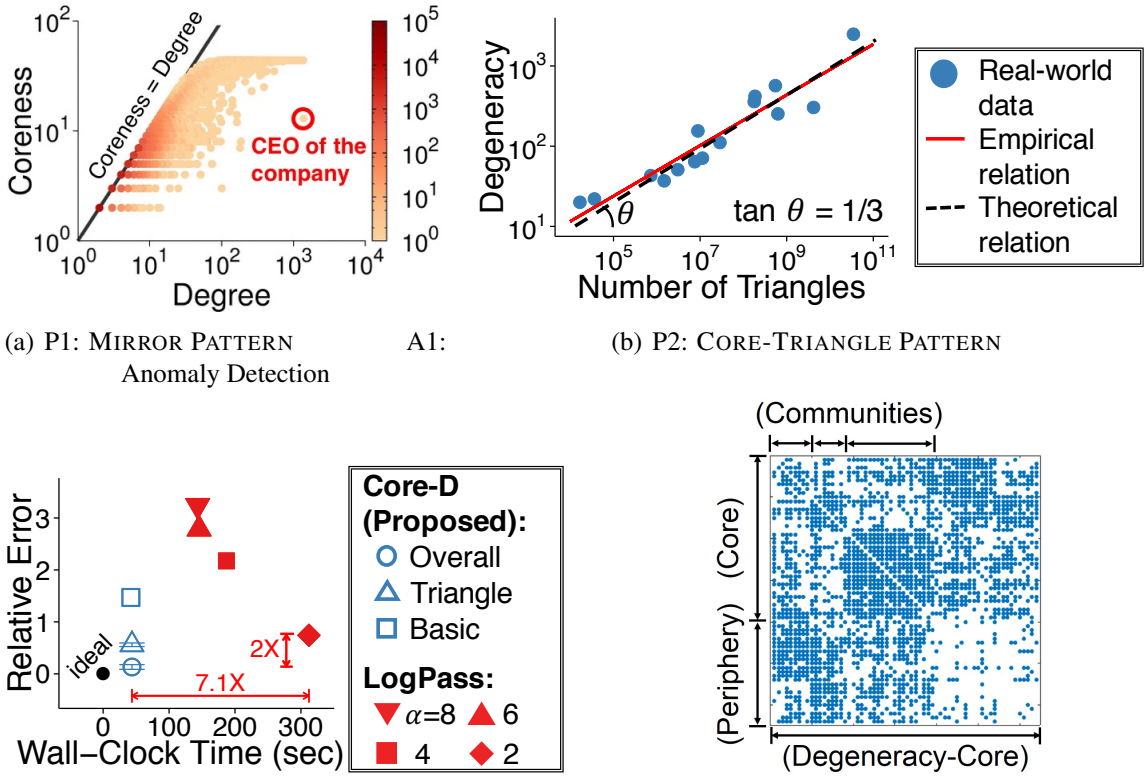
Our discoveries are: (1) MIRROR PATTERN: coreness (i.e., maximum k such that each node belongs to the k-core) is strongly correlated to degree. (2) CORE-TRIANGLE PATTERN: degeneracy (i.e., maximum k such that the k-core exists) obeys a *3-to-1* power law with respect to the count of triangles. (3) STRUCTURED-CORE PATTERN: degeneracy-cores are not cliques but have non-trivial structures such as core-periphery and communities.

Our algorithmic contributions show the usefulness of these patterns. (1) CORE-A, which measures the deviation from MIRROR PATTERN, successfully spots anomalies in real-world graphs, (2) CORE-D, a single-pass streaming algorithm based on CORE-TRIANGLE PATTERN, accurately estimates degeneracy up to *12× faster* than its competitor. (3) CORE-S, inspired by STRUCTURED-CORE PATTERN, identifies influential spreaders up to *17× faster* than its competitors with comparable accuracy.

9.1 Motivation

Given an undirected graph $\mathcal{G}$, the k-core is the maximal subgraph of $\mathcal{G}$ in which every node is adjacent to at least k nodes [BZ03]. This concept has been used extensively in diverse applications, including hierarchical structure analysis [AhBV08], graph visualization [AHDBV06], protein function prediction [WA05], and graph clustering [GMTV14]. An equally useful and closely related concept is the *degeneracy* of $\mathcal{G}$, that is, the maximum k such that the k-core exists in $\mathcal{G}$. For example, a clique of 5 nodes itself is a 4-core and thus has degeneracy 4; a ring of 10 nodes has degeneracy 2; a star of 100 nodes has degeneracy 1. The simplest algorithm to compute k-cores is the so-called *"shaving"* method: repeatedly deleting nodes with degree less than k until no such node is left.

Despite the huge interest in k-cores and their applications, it is not known whether k-cores or degeneracy follow any patterns in real graphs. Our motivating questions are: (1) what are common patterns



(a) P1: MIRROR PATTERN — A1: Anomaly Detection

(b) P2: CORE-TRIANGLE PATTERN

(c) A2: CORE-D Algorithm

(d) P3: STRUCTURED-CORE PATTERN

Figure 9.1: **Three patterns (P1-P3) discovered in real-world graphs, and their applications (A1-A3).** (a) **P1**: Coreness and degree are strongly correlated. **A1**: Anomalies deviate from this pattern. (b) **P2**: Degeneracy and the number of triangles in graphs obey a *3-to-1* power law, which is theoretically supported. (c) **A2**: Our CORE-D algorithm (with OVERALL MODEL) estimates the degeneracy in a graph stream $7\times$ *faster* and $2\times$ *more accurately* than its competitor. (d) **P3**: As seen in the sparsity pattern of the adjacency matrix of a degeneracy-core, degeneracy-cores have structure, such as core-periphery and communities, which can be exploited for identifying influential spreaders (**A3**).

regarding k-cores or degeneracy occurring across graphs in diverse domains? (2) are there anomalies deviating from these patterns? (3) how can these patterns and anomalies be used for algorithm design?

To answer these questions, we present three empirical patterns that govern k-cores or degeneracy across a wide variety of real-world graphs, including social networks, web graphs, internet topologies, and citation networks. We also show the practical uses of these patterns.

Our first MIRROR PATTERN states that the *coreness* of a node (i.e., the maximum k such that the node belongs to the k-core) is strongly correlated to its degree, as seen in Figure 9.1(a). We also observe that anomalies (e.g., the CEO in Figure 9.1(a) and accounts using a 'follower booster' in Twitter) tend to deviate from this pattern. This observation leads to CORE-A, our anomaly detection method based on the degree of deviation from MIRROR PATTERN. We show that CORE-A is complementary to recent densest-subgraph based anomaly detection methods [HSS+17, SHF18], and their combination has the best of the two approaches.

Our second discovery, CORE-TRIANGLE PATTERN, states that, in real-world graphs, the degeneracy and the triangle-count obey a power-law with slope 1/3, as seen in Figure 9.1(b). This relation is theoretically analyzed in very realistic Kronecker graphs [LCKF05], and also utilized in CORE-D, our

single-pass streaming algorithm for estimating degeneracy. CORE-D is up to *12× faster* than a recent multi-pass algorithm [FCT14], while providing comparable accuracy (see Figure 9.1(c)).

Our last discovery, STRUCTURED-CORE PATTERN, states that degeneracy-cores in real-world graphs are not cliques but have non-trivial structures (core-periphery, communities, etc.), as seen in Figure 9.1(d). We also show that nodes central within degeneracy-cores are particularly good spreaders up to *2.6× more influential* than the average nodes in degeneracy-cores, which are already known as good spreaders [KGH+10]. Those spreaders are spotted by CORE-S, our influential spreader identification method, which is up to *17× faster* than its top competitors [KGH+10, RMV15, MSHM12] with similar accuracy.

Our contributions in this chapter are as follows:

- **Patterns**: We discover three empirical patterns that hold across several real-world graphs from diverse domains.
- **Anomalies**: We detect interesting anomalies (e.g., accounts using a 'follower-boosting' service in Twitter) from nodes deviating from the patterns.
- **Algorithms**: The patterns are practically used in our algorithms for detecting anomalies (CORE-A), estimating degeneracy (CORE-D), and identifying influential spreaders (CORE-S). Our experiments show that our algorithms either complement or outperform state-of-the-art algorithms.

Reproducibility: The source code and datasets used in this chapter are available at `http://www.cs.cmu.edu/~kijungs/codes/kcore/`.

The rest of this chapter is organized as follows. In Section 9.2, we give some preliminaries useful in the remaining sections. In Section 9.3, we present MIRROR PATTERN and its application to anomaly detection. In Section 9.4, we present CORE-TRIANGLE PATTERN and CORE-D, a streaming algorithm for estimating degeneracy. In Section 9.5, we present STRUCTURED-CORE PATTERN and its application to influential-spreader detection. After reviewing related work in Section 9.6, we provide a summary of this chapter in Section 9.7.

9.2 Preliminaries

In this section, we first provide the definitions of k-cores and related concepts. Then, we discuss some algorithms for computing k-cores and degeneracy. Lastly, we describe the real-world graphs used in the following sections.

9.2.1 Concepts and Notations

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected unweighted graph. We define $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$. We denote the neighbors of a node $v \in \mathcal{V}$ by $\mathcal{N}_v = \{u \in \mathcal{V} : \{u, v\} \in \mathcal{E}\}$ and its degree by $d_v = |\mathcal{N}_v|$. Likewise, for a subgraph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ of $\mathcal{G}$, we use $\mathcal{N}_v(\mathcal{G}') = \{u \in \mathcal{V}' : \{u, v\} \in \mathcal{E}'\}$ and $d_v(\mathcal{G}') = |\mathcal{N}_v(\mathcal{G}')|$.

The *k-core* or the *core of order* k [BZ03] is the maximal subgraph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ where $\forall v \in \mathcal{V}'$, $d_v(\mathcal{G}') \geq k$. Notice that, for each k, there exists at most one k-core, and it is not necessarily a connected subgraph. Cores are nested, i.e., the k_1-core is a subgraph of the k_2-core if $k_1 \geq k_2$. The *coreness* or *core number* of a node v [BZ03], denoted by c_v, is the order of the highest-order core that v belongs to. By definition, coreness is upper bounded by degree, i.e., $c_v \leq d_v$. The *degeneracy* of a graph $\mathcal{G}$, defined as $k_{max} = \max_{v \in \mathcal{V}} c_v$, is the maximum coreness. The k_{max}-core is also called the *degeneracy-core*. We define the *density* of a subgraph as the ratio between the number of existing edges and the largest

Table 9.1: **Table of frequently-used symbols.**

Symbol	Definition
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	undirected and unweighted graph
A	adjacency matrix of $\mathcal{G}$
n	number of nodes in $\mathcal{G}$
m	number of edges in $\mathcal{G}$
k_{max}	degeneracy of $\mathcal{G}$
n_{max}	number of nodes in the degeneracy-core
m_{max}	number of edges in the degeneracy-core
D_{max}	density of the degeneracy-core
d_{avg}	average degree of $\mathcal{G}$
c_v	coreness of node v
d_v	degree of node v
r	Pearson correlation coefficient
ρ	Spearman's rank correlation coefficient
$dmp(v)$	node v's degree of deviation from MIRROR PATTERN
DSM	densest-subgraph based anomaly detection methods
$a - score(\mathcal{G}')$	anomaly score of subgraph $\mathcal{G}'$
$\#\Delta$	number of triangles in $\mathcal{G}$
λ_i	i-th largest eigenvalue of A
i_v	in-core centrality of node v
β	infection rate in the SIR Model

possible number of edges. If we let n_{max} and m_{max} be the numbers of the nodes and edges in the degeneracy-core, then the density of the degeneracy-core is $D_{max} = m_{max}/\binom{n_{max}}{2}$.

Additionally, we denote the number of triangles in a graph $\mathcal{G}$ by $\#\Delta$. The eigenvalues of the adjacency matrix A of $\mathcal{G}$ are denoted by $(\lambda_1, ..., \lambda_n)$ where $\lambda_i \geq \lambda_j$ if $i < j$. Table 9.1 lists the symbols frequently used in the chapter.

9.2.2 Algorithm for k-Cores and Degeneracy

The k-core remains if we remove nodes with degree less than k and edges incident to them recursively from $\mathcal{G}$ until no node has degree less than k. The $(k+1)$-core can be computed in the same way from the k-core since the $(k+1)$-core is a subgraph of the k-core. Likewise, by computing k-cores sequentially from $k = 1$ to $k = k_{max}$, we divide all nodes according to their coreness. This process, called *core decomposition*, runs in $O(n + m)$ [BZ03] if a graph fits in memory.

However, if a graph does not fit in memory, the computational cost grows. For example, in a graph stream, a recent method LOGPASS [FCT14] requires $O(\log_{\alpha/2}(n))$ passes of the entire graph for α-approximation of the degeneracy, for any real number α larger than 2. It requires $O(n)$ memory space, independent of α. In Section 9.4.3, however, we propose a single-pass algorithm for estimating degeneracy with similar memory requirements. Other algorithms for k-cores in large graphs are discussed in Section 9.6.

Table 9.2: **Summary of the real-world graphs used in the chapter.**

Name	n	m	#Δ	k_{max}	n_{max}	D_{max}	Summary
Hamsterster [Kun13]	1.86K	12.6K	16.8K	20	130	0.24	Social networks
Email [KY04]	36.7K	184K	727K	43	275	0.26	
Catster [Kun13]	150K	5.45M	185M	419	1.28K	0.48	
Youtube [MMG+07]	1.13M	2.99M	3.06M	51	845	0.10	
Flickr [MMG+07]	1.72M	15.6M	548M	568	1.75K	0.49	
Orkut [MMG+07]	3.07M	117M	628M	253	15.7K	0.03	
LiveJournal [MMG+07]	4.00M	34.7M	178M	360	377	0.99	
Twitter [KLPM10]	41.7M	1.20B	34.8B	2.49K	3.19K	0.90	
Friendster [YL15]	65.6M	1.81B	4.17B	304	24.5K	0.02	
Stanford [LLDM09]	282K	1.99M	11.3M	71	387	0.29	Web Graphs
NotreDame [AJB99]	326K	1.09M	8.91M	155	1.37K	0.12	
Caida [LKF07]	26.5K	53.4K	36.3K	22	64	0.53	Internet topologies
Skitter [LKF07]	1.70M	11.1M	28.8M	111	222	0.68	
Arxiv [GGK03]	27.8K	352K	1.48M	37	52	0.86	Citation networks
Patent [HJT01]	3.77M	16.5M	7.52M	64	106	0.73	

9.2.3 Real-world Graph Datasets

We describe the datasets used in the following sections. Since the objective of this chapter is to find pervasive patterns emerging across graphs in diverse domains, our datasets include social networks, web graphs, internet topologies, and citation networks. The direction of edges is ignored in all the datasets because k-cores are defined only in undirected graphs. The datasets are summarized in Table 9.2 with the details below.

Social Networks. The Hamsterster dataset [Kun13] is a friendship network between users of hamsterster.com, an online community for hamster owners. The Catster dataset [Kun13] is a friendship network between users of catster.com, an online community for cat owners. The Youtube dataset [MMG+07] is a friendship network between users of Youtube, a video-sharing web site. The Flickr dataset [MMG+07] is a social network between users of Flickr, a photo sharing site. The Orkut dataset [MMG+07] is a social network between users of Orkut, a social networking site. The LiveJournal dataset [MMG+07] is a friendship network between users of Live Journal, an online blogging community. The Friendster dataset [YL15] is a friendship network between users of Friendster, a former social networking site. The Twitter dataset [KLPM10] is a subscription network between users in Twitter, a microblogging service. The Email dataset [KY04] is an email network between employees of Enron Corporation, an energy, commodities, and services company. This dataset also includes emails between the employees and people outside the company.

Web Graphs. The NotreDame dataset [AJB99] and the Stanford dataset [LLDM09] are hyperlink networks between web pages from University of Notre Dame and Stanford University, respectively.

Internet Topologies. The Caida dataset [LKF07] is an internet topology graph obtained from RouteViews Border Gateway Protocol routing tables. The Skitter dataset [LKF07] is an internet topology graph obtained from traceroute data collected by Skitter, which is Caida's probing tool.

Citation Networks. The Patent dataset [HJT01] is a citation network between patents registered with the United States Patent and Trademark Office. The Arxiv dataset [GGK03] is a citation network between papers submitted to arXiv High Energy Physics Theory Section.

9.3 P1: "Mirror Pattern" and Anomaly Detection

In this section, we discuss MIRROR PATTERN and its use for anomaly detection.

9.3.1 Observation: Pattern in Real-world Graphs

What are the key factors determining the coreness of the nodes in real-world graphs? We find out that a strong positive correlation exists between coreness and degree, which is an upper bound of coreness. Specifically, as seen in Figure 9.2, Spearman's rank correlation coefficient ρ [1] is significantly higher than 0 (no correlation) in all the considered graphs and close to 1 (perfect positive correlation) in many of them. This empirical pattern is described in Observation 9.1.

Observation 9.1: MIRROR PATTERN

In real-world graphs, coreness has a strong positive correlation with degree.

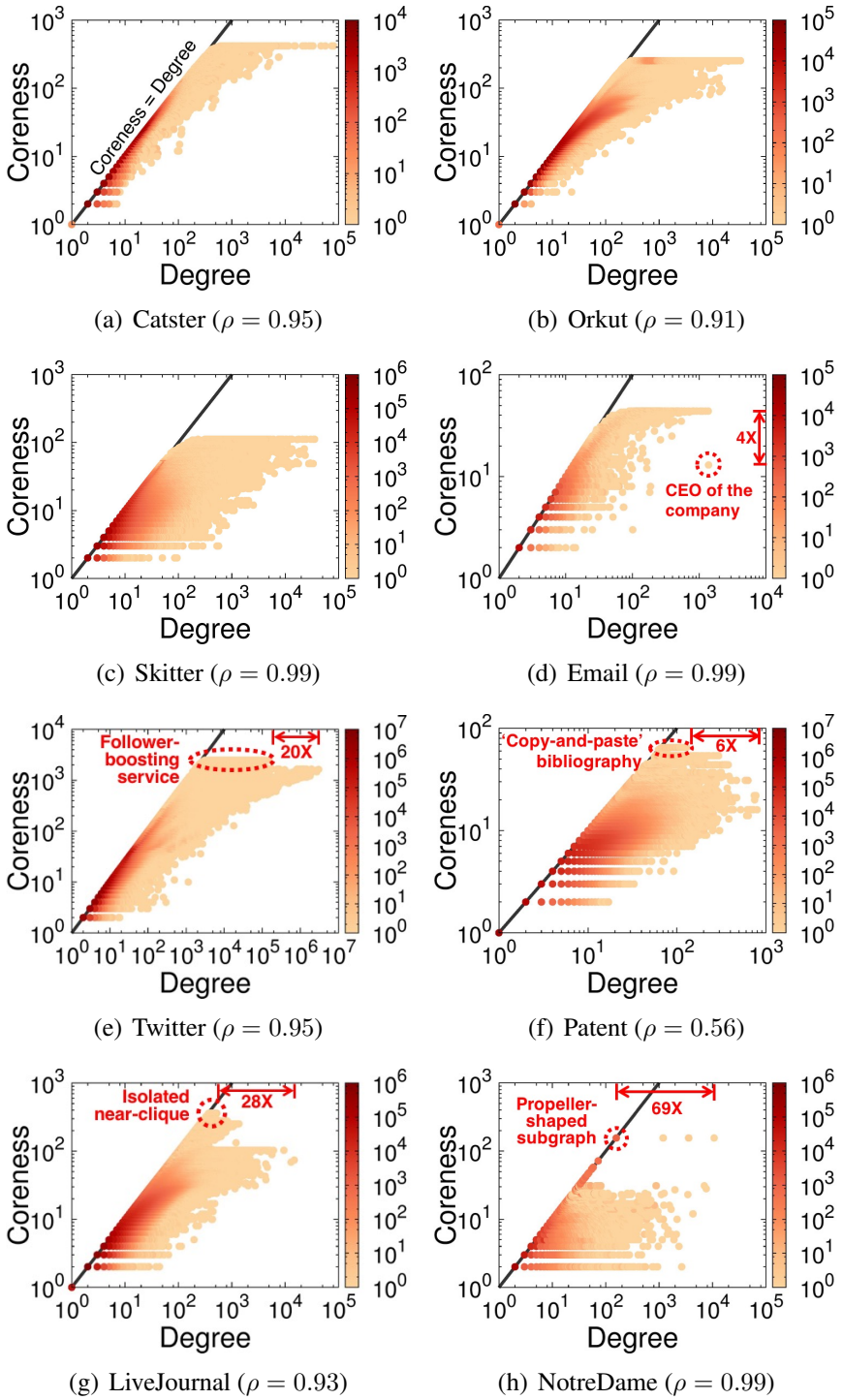
9.3.2 Application: Anomaly Detection

MIRROR PATTERN implies that nodes with high coreness have tendency to have high degree and vice versa. However, the degree-coreness plots in Figure 9.2 highlight some nodes deviating from the pattern, i.e., nodes ranked first in terms of degree but relatively lower in terms of coreness, and vice versa. In this section, we take a close look at these nodes and show that they indicate two different types of anomalies: *'loner-stars'* (i.e., nodes mostly connected to 'loners') or *'lockstep behavior'* (i.e., a group of similarly behaving nodes).

9.3.2.1 Second Email Account of the CEO (Loner-Star)

In the Email dataset, the node marked in Figure 9.2(d) has the highest degree 1,383 but relatively low coreness 12, deviating from MIRROR PATTERN. This node corresponds to the second email account of the former CEO of the company. This account was used only to receive emails, and not a single email was sent from this account. The former CEO used the other email account when sending emails. The 99.6% of the sources of the received emails are outside the company, while only 0.4% are inside. Since email accounts outside the company mostly have small coreness in the dataset (they are 'loners'), this anomalous email account has small coreness despite its high degree.

[1] Spearman's rank correlation coefficient ρ [Spe04] is the standard (Pearson) correlation coefficient r of the ranks. Here, ρ is equivalent to r between the ranks of nodes in terms of degree and their ranks in terms of coreness. Using ρ is known to be robust to outlying values than simply using r. We ignored isolated nodes when computing ρ.



(a) Catster ($\rho = 0.95$) (b) Orkut ($\rho = 0.91$)

(c) Skitter ($\rho = 0.99$) (d) Email ($\rho = 0.99$)

(e) Twitter ($\rho = 0.95$) (f) Patent ($\rho = 0.56$)

(g) LiveJournal ($\rho = 0.93$) (h) NotreDame ($\rho = 0.99$)

Figure 9.2: **Our MIRROR PATTERN is pervasive in real-world graphs; exceptions signal anomalies.** $\rho \in [-1, 1]$ indicates Spearman's rank correlation coefficient; and colors are for heatmap of point density. Degree and coreness have strong positive correlation; exceptions (in red circles) are "strange": the node ranked first in terms of degree but relatively lower in terms of coreness corresponds to an email account of the company's CEO in (d); nodes ranked first in terms of coreness but relatively lower in terms of degree indicate accounts involved in a 'follower-boosting' service in (e), 'copy-and-paste' bibliography in (f), an isolated near-clique in (g), and a propeller-shaped subgraph in (h).

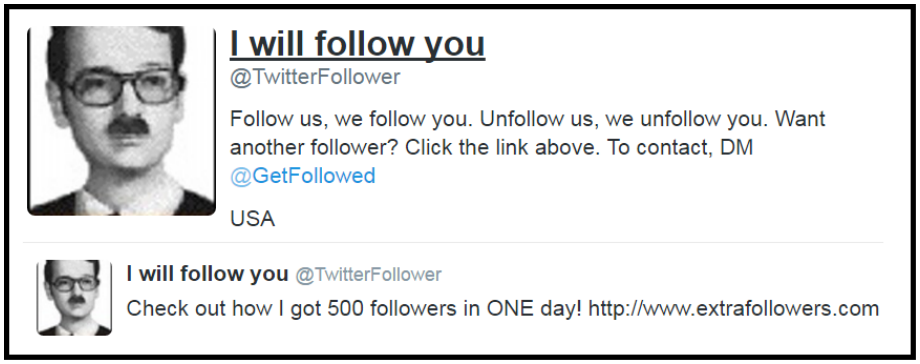


Figure 9.3: **Nodes deviating from MIRROR PATTERN are involved in a 'Follower booster' on Twitter.** 78% of the nodes in the degeneracy-core were following the above Twitter account when the data were crawled. The account still exists without being suspended.

9.3.2.2 'Follower-Boosting' Service on Twitter (Lockstep Behavior)

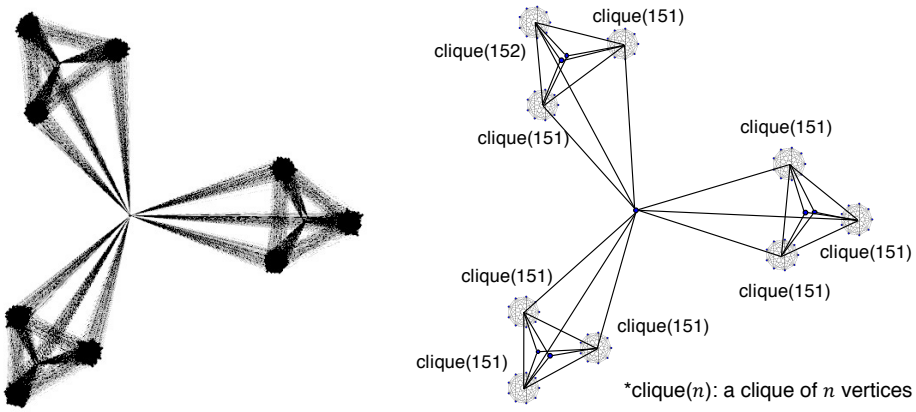
In Twitter, the nodes with the highest coreness, marked in Figure 9.2(e), have relatively low degrees, deviating from MIRROR PATTERN. We find out that at least 78% of the nodes with the highest coreness were directly involved in a 'Follower-Boosting' service (i.e., following '@TwitterFollower' in Figure 9.3) when the Twitter dataset was crawled. Since the accounts involved in the service are densely connected with each other ($D_{max} = 0.90$), to boost the number of followers, they have the highest coreness despite their relatively low degrees. Surprisingly, this misbehavior has been undetected by Twitter, and '@TwitterFollower' account has not been suspended or removed since the data was crawled in 2009.

9.3.2.3 'Copy-and-Paste' Bibliography (Lockstep Behavior)

As on Twitter, the nodes with the highest coreness in the Patent dataset have relatively low degrees, deviating from MIRROR PATTERN (see Figure 9.2(f)). We find out that 88% of these nodes are patents owned by the same pharmaceutical company, and bibliography in previous patents of the company has been reused repeatedly in a 'copy-and-paste' manner in later patents of the company. This results in a dense subgraph in the citation network, and the patents in the subgraph have the highest coreness despite their relatively low degrees.

9.3.2.4 Isolated Near-Clique in Live Journal (Lockstep Behavior)

Nodes with the highest coreness but relatively low degrees are also found in the LiveJournal dataset, as marked in Figure 9.2(g). Although we could not identify actual accounts corresponding to these 377 nodes, their abnormality was supported by the following facts: (1) the nodes form a near-clique with density 99.7%, unlikely to occur naturally, (2) the group formed by the nodes is isolated as judged from the fact that 88% of the neighbors of the nodes are also in the group, while only 12% are outside, and (3) the nodes have suspicious uniformity. Specifically, 127 nodes (one third of the considered nodes) have degrees between 387 and 391.



(a) Degeneracy-core of the NotreDame web graph

(b) The structure of the degeneracy-core of the NotreDame web graph

Figure 9.4: **Nodes deviating from MIRROR PATTERN form a propeller-shaped subgraph in the NotreDame dataset.** The degeneracy-core of the NotreDame dataset is composed by a set of cliques of the same size connected in a symmetric way. This subgraph is unlikely to occur naturally.

9.3.2.5 Propeller-Shaped Subgraph in Web (Lockstep Behavior)

As marked in Figure 9.2(h), in the NotreDame dataset, the nodes with the highest coreness have relatively low degree, deviating from MIRROR PATTERN. The structure of the degeneracy-core, the subgraph formed by these nodes, is shown in Figure 9.4. The degeneracy-core consists of 9 cliques of almost the same size (8 cliques of 151 nodes and a clique of 152 nodes). Moreover, the way these cliques are connected is surprisingly symmetric. That is, the cliques are divided into 3 groups where the cliques in each group are connected to the same two nodes, and every clique is also connected to the center node. Although the actual web pages corresponding to the nodes composing this subgraph are unknown, this subgraph is unlikely to occur naturally. This subgraph seems to be artificially constructed for special purposes.

9.3.3 Proposed Algorithm: CORE-A

Inspired by the observations in the previous section, we propose CORE-A, an anomaly detection method based on the deviation from MIRROR PATTERN. We show that CORE-A is complementary to densest-subgraph based anomaly detection, and their combination has the best of the two methods.

9.3.3.1 Algorithm Description

In the previous section, we show that nodes deviating from MIRROR PATTERN are worth noticing, as they indicate the two types of anomalies: 'loner-stars' (e.g. the CEO in Figure 9.2(d)) and 'lockstep behavior' (e.g., an isolated near-clique in Figure 9.2(g)). What scoring function gives a high score, to both types of anomalies? *Deviation from* MIRROR PATTERN (*dmp*) in Definition 9.1 gives an answer. CORE-A, our proposed anomaly detection method, ranks nodes in decreasing order of *dmp*. The main idea behind our proposed *dmp* measure, is to use the *rank* of each node, and since we expect power-laws, the log of the rank. Specifically, we use $rank_d(v)$, the fractional rank[2] of node v in decreasing degree order, and similarly, $rank_c(v)$, in decreasing coreness order (in case of the same coreness, in decreasing degree order).

Definition 9.1: Deviation from MIRROR PATTERN

A node v's degree of deviation from MIRROR PATTERN in graph $\mathcal{G}$ is defined as

$$\textit{dmp}(v) := |\log(rank_d(v)) - \log(rank_c(v))|.$$

CORE-A has time complexity $O(n+m)$ since the *dmp* scores of all nodes can be computed in $O(n)$ using 'counting sort' once we compute core decomposition in $O(n+m)$ [BZ03].

9.3.3.2 Complementarity of CORE-A

Anomaly detection in graphs (especially in social networks) has been extensively researched (see Section 9.6), and many of them detect dense subgraphs since anomalies tend to form dense subgraphs, as we also show in Section 9.3.2. Especially, recent methods, including M-ZOOM (Chapter 11) and FRAUDAR [HSS+17], are based on densest subgraphs (i.e., subgraphs with maximum average degree). We show that CORE-A and these densest-subgraph based methods (DSM) are complementary as they are good at detecting different-size dense subgraphs.

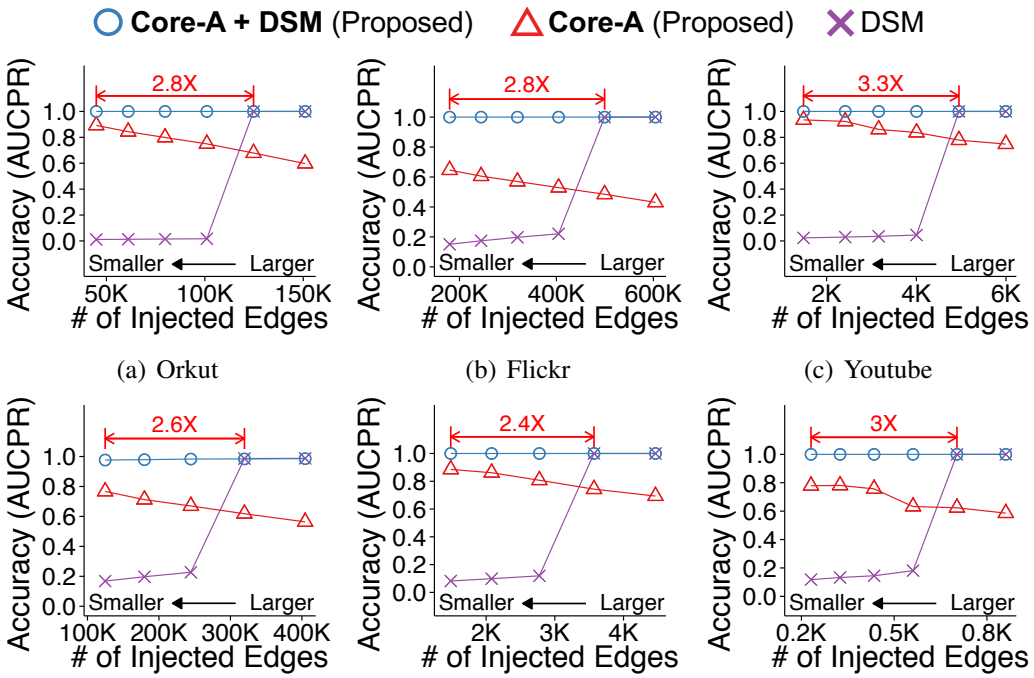
To demonstrate that CORE-A and DSM (specifically, running M-ZOOM on the adjacency matrix) are complementary, we compare their performances when different-size subgraphs are injected into social networks. We randomly choose k nodes and inject $\binom{k}{2}$ edges between them into each network. Then, we compare how precisely and exhaustively each method detects the k chosen nodes using Area Under the Precision-Recall Curve (AUCPR) [DG06].

As seen in Figure 9.5, DSM cannot detect small dense subgraphs accurately, while it detects large ones with near-perfect accuracy. In contrast, CORE-A is more accurate for smaller subgraphs that cannot be detected by DSM. This is explained by the fact that the k chosen nodes have degree and coreness at least $k-1$. If $k \approx c_{max}$ but $k \ll d_{max}$, the nodes tend to have high *dmp* scores since they have small $rank_c$ but are likely to have large $rank_d$. However, if $k \approx d_{max}$, the nodes have low *dmp* scores since they have small $rank_d$ as well as small $rank_c$.

9.3.3.3 Combination with DSM

We can have the best of CORE-A and DSM by combining them. Specifically, we propose to define the *anomaly score* (*a-score*) of a subgraph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ in a graph $\mathcal{G}$ based on *dmp* scores in $\mathcal{G}$ as well as

[2]The fractional rank of an item is one plus the number of items greater than it plus half the number of items equal to it.



(a) Orkut (b) Flickr (c) Youtube

(d) Catster (e) Email (f) Hamsterster

Figure 9.5: **CORE-A is complementary to DSM; their combination has the best of the two.** In social networks, our CORE-A method accurately detects small injected subgraphs that cannot be detected accurately by DSM. The combination of CORE-A and DSM successfully detects both small and large injected subgraphs. The combination detects up to $3.3\times$ *smaller* injected subgraphs than DSM with near-perfect accuracy.

the average degree as follows:

$$a\text{-}score(\mathcal{G}') = \frac{|\mathcal{E}'|}{|\mathcal{V}'|} + w \sum_{v \in \mathcal{V}'} \frac{dmp(v)}{|\mathcal{V}'|} \tag{9.1}$$

where $w > 0$ is a parameter for balancing the two factors: $\frac{|\mathcal{E}'|}{|\mathcal{V}'|}$ and $\sum_{v \in \mathcal{V}'} \frac{dmp(v)}{|\mathcal{V}'|}$. We set w to the ratio of the maximum values of the factors in the given graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The maximum value of $\frac{|\mathcal{E}'|}{|\mathcal{V}'|}$ is close (within a factor of 2) to $\frac{|\mathcal{E}^*|}{|\mathcal{V}^*|}$, where $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$ is the densest subgraph detected by DSM; and the maximum value of $\sum_{v \in \mathcal{V}'} \frac{dmp(v)}{|\mathcal{V}'|}$ is $\max_{v \in \mathcal{V}} dmp(v)$. We set w to their ratio, i.e.,

$$w = \frac{|\mathcal{E}^*|}{|\mathcal{V}^*|} \times \frac{1}{\max_{v \in \mathcal{V}} dmp(v)}. \tag{9.2}$$

Once we set w, we use [SHF18] to identify the subgraph maximizing *a-score* (Eq (9.1)). We classify the nodes in the subgraph into anomalies. This entire process takes $O(m \log n)$, as DSM does [SHF18, HSS+17].

Figure 9.5 illustrates the success of our proposal to combine the scores (Eq. (9.1)): our combination successfully detects both small and large dense subgraphs injected into social networks, outperforming both its component methods (i.e., CORE-A and DSM), Especially, the combination detects up to $3.3\times$ *smaller* dense subgraphs than DSM, with near-perfect accuracy.

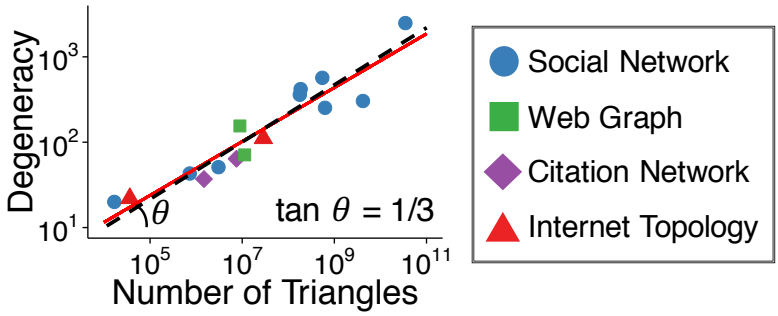


Figure 9.6: **CORE-TRIANGLE PATTERN: triangle count and degeneracy obey a 3-to-1 power law.** Each point corresponds to a graph dataset in Table 9.2. The count of triangles has a strong correlation ($r = 0.94$) with degeneracy in log scale. Moreover, the slope is very close to the theoretical slope $1/3$ (dashed line).

9.4 P2: "Core-Triangle Pattern" and Degeneracy Estimation

In this section, we present CORE-TRIANGLE PATTERN (C-T PATTERN) in real-world graphs and provide mathematical analysis of the pattern. Then, we propose an one-pass streaming algorithm for estimating degeneracy, based on the pattern.

9.4.1 Observation: Pattern in Real-world Graphs

What are the major factors determining degeneracy, the maximum coreness, in real-world graphs? We investigate the relation between degeneracy and various graph measures in real-world graphs. As seen in Figure 9.6, the number of triangles has a particularly strong correlation ($r = 0.94$) with degeneracy in log scale, compared to the node-count ($r = 0.75$) and the edge-count ($r = 0.83$), which have a weaker correlation with degeneracy. Moreover, the slope is 0.32, which is very close to $1/3$. This leads to Observation 9.2.

Observation 9.2: CORE-TRIANGLE PATTERN or C-T PATTERN

In real-world graphs, the triangle count and the degeneracy obey a 3-to-1 power law. That is,

$$k_{max} \propto (\#\Delta)^{\frac{1}{3}}.$$

9.4.2 Theoretical Analysis in the Kronecker and ER Models

Why do real-world graphs obey C-T PATTERN? Here we show that C-T PATTERN holds in the so-called the Kronecker model (Definition 9.2), which is considered as a very realistic graph model obeying common patterns in real-world graphs [LCKF05].

Definition 9.2: Kronecker Graph [LCKF05]

Let $\mathcal{G}_q$ be the q-th power Kronecker graph of a seed graph $\mathcal{G}_1$. If we denote the adjacency matrix of $\mathcal{G}_q$ by A_q, then A_q is defined as:

$$A_q = A_{q-1} \otimes A_1 = \underbrace{A_1 \otimes A_1 \otimes ... \otimes A_1}_{q \text{ times}},$$

where $\otimes$ denotes Kronecker Product.

C-T PATTERN in the model is defined formally in Definition 9.3, where we ignore constant factors for ease of analysis.

Definition 9.3: C-T PATTERN in the Kronecker model

A Kronecker model with seed graph $\mathcal{G}_1$ follows C-T PATTERN if Eq. (9.3) holds in $\{\mathcal{G}_q\}_{q \geq 1}$, graphs generated by the model.

$$k_{max} = \Theta(\#\Delta^{\frac{1}{3}}) \text{ or equivalently } \#\Delta = \Theta(k_{max}^3). \tag{9.3}$$

To prove C-T PATTERN in the Kronecker model, we use Lemmas 9.1 and 9.2, which give upper and lower bounds of degeneracy.

Lemma 9.1: Lower Bound of Degeneracy [Erd63]

The half of the average degree lower bounds the degeneracy. That is, if we let d_{avg} be the average degree, then $k_{max} \geq d_{avg}/2$.

Lemma 9.2: Upper Bound of Degeneracy

The largest eigenvalue upper bounds the degeneracy. That is, if we let λ_1 be the largest eigenvalue of the adjacency matrix, then $k_{max} \leq \lambda_1$.

Proof. Let H be the degeneracy-core (i.e., k_{max}-core) of $\mathcal{G}$ and $d_{min}(H)$ be its minimum degree. By the definition of the k-core and the degeneracy, $d_{min}(H) = k_{max}(\mathcal{G})$. Since the largest eigenvalue is lower bounded by minimum degree [BH11], $k_{max}(\mathcal{G}) = d_{min}(H) \leq \lambda_1(H)$. The largest eigenvalue of a graph is also lower bounded by that of its induced subgraph [BH11]. Since the degeneracy-core is an induced subgraph due to its maximality, $k_{max}(\mathcal{G}) \leq \lambda_1(H) \leq \lambda_1(\mathcal{G}) = \lambda_1$. ■

Lemma 9.3 states that the graph measures used for upper and lower bounding degeneracy in Lemmas 9.1 and 9.2 increase exponentially with q, the power of Kronecker products, in the Kronecker model.

Lemma 9.3: Graph Measures Increasing Exponentially in Kronecker Graphs

The average degree, the degeneracy, and the largest eigenvalue increase exponentially with q in $\{\mathcal{G}_q\}_{q\geq 1}$, graphs generated by the Kronecker model. Specifically,

$$d_{avg}(\mathcal{G}_q) = (d_{avg}(\mathcal{G}_1))^q,\ \forall q \geq 1. \tag{9.4}$$
$$k_{max}(\mathcal{G}_q) \geq (k_{max}(\mathcal{G}_1))^q,\ \forall q \geq 1. \tag{9.5}$$
$$\lambda_1(\mathcal{G}_q) = (\lambda_1(\mathcal{G}_1))^q,\ \forall q \geq 1. \tag{9.6}$$

Proof. Let $n(\mathcal{G})$ be the number of nodes and $nz(\mathcal{G})$ be the number of non-zero entries in the adjacency matrix.

We first show Eq. (9.4). From $d_{avg}(\mathcal{G}) = nz(\mathcal{G})/n(\mathcal{G})$, $n(\mathcal{G}_q) = (n(\mathcal{G}_1))^q$, and $nz(\mathcal{G}_q) = (nz(\mathcal{G}_1))^q$, we have

$$d_{avg}(\mathcal{G}_q) = \frac{nz(\mathcal{G}_q)}{n(\mathcal{G}_q)} = \frac{(nz(\mathcal{G}_1))^q}{(n(\mathcal{G}_1))^q} = \left(\frac{nz(\mathcal{G}_1)}{n(\mathcal{G}_1)}\right)^q = (d_{avg}(\mathcal{G}_1))^q, \forall q \geq 1.$$

We prove Eq. (9.5) by induction. For seed graph $\mathcal{G}_1$, $k_{max}(\mathcal{G}_1) \geq (k_{max}(\mathcal{G}_1))^1$. Assume $k_{max}(\mathcal{G}_i) \geq (k_{max}(\mathcal{G}_1))^i$. Each node in $\mathcal{G}_{i+1}$ can be represented as an ordered pair (v_i, v_1) where v_i is a node of $\mathcal{G}_i$ and v_1 is a node of $\mathcal{G}_1$. Two nodes, (v_i, v_1) and (v_i', v_1'), in $\mathcal{G}_{i+1}$ are adjacent if and only if v_i and v_i' are adjacent in $\mathcal{G}_i$ and v_1 and v_1' are adjacent in $\mathcal{G}_1$ [LCKF05]. Let $\mathcal{G}_i'(\mathcal{V}_i', \mathcal{E}_i')$ be the degeneracy-core of $\mathcal{G}_i(V_i, E_i)$ where $\mathcal{V}_i' = \{v_i \in V_i : c_=(v_i)k_{max}(\mathcal{G}_i)\}$. Then, each node (v_i, v_1) in $S = \{(v_i, v_1) \in V_{i+1} : v_i \in \mathcal{V}_i', v_1 \in \mathcal{V}_1'\}$ are adjacent to $d_{v_i}(\mathcal{G}_i') \times d_{v_1}(\mathcal{G}_1')(\geq k_{max}(\mathcal{G}_i) \times k_{max}(\mathcal{G}_1))$ nodes in S. Therefore, $k_{max}(\mathcal{G}_{i+1}) \geq k_{max}(\mathcal{G}_i) \times k_{max}(\mathcal{G}_1) \geq k_{max}(\mathcal{G}_1)^{(i+1)}$. By induction, $k_{max}(\mathcal{G}_q) \geq (k_{max}(\mathcal{G}_1))^q, \forall q \geq 1$.

Finally, to show Eq. (9.6), let $\lambda(\mathcal{G}) = (\lambda_1, ..., \lambda_n)$ be the eigenvalues of the adjacency matrix of $\mathcal{G}$, and $\lambda_1(\mathcal{G})$ be the largest eigenvalue. Then, $\lambda(\mathcal{G}_q) = sort(\lambda(\mathcal{G}_{q-1}) \otimes \lambda(\mathcal{G}_1))$ [VL00]. As $\lambda_1(\mathcal{G}_q) = \lambda_1(\mathcal{G}_{q-1}) \times \lambda_1(\mathcal{G}_1)$, $\lambda_1(\mathcal{G}_q) = (\lambda_1(\mathcal{G}_1))^q, \forall q \geq 1$ holds. ■

Lemmas 9.4 and 9.5 state how rapidly degeneracy and triangle count increase in the Kronecker model. Both of them increase exponentially with q, the power of Kronecker products, and the base numbers depend on seed graphs. For Lemma 9.5, we have to deal with self-loops in Kronecker graphs which happen naturally. We add one to the degree for each self-loop and define *a triangle in Kronecker graphs* as an unordered node triplet, which can contain multiple instances of the same node, where every instance is connected to the others either by self-loops or other edges. For example, (v_1, v_1, v_2) is a triangle in Kronecker graphs if v_1 has a self-loop and v_1 and v_2 are adjacent. Note that Lemma 9.5 and Theorem 9.1 hold equally, with the original definitions of degree and a triangle, in Kronecker graphs without self-loops.

Lemma 9.4: Degeneracy in the Kronecker model

Degeneracy in $\{\mathcal{G}_q\}_{q\geq 1}$ increases exponentially with q. Let d_{avg} be the average degree and λ_1 be the largest eigenvalue of the adjacency matrix. Then,

$$k_{max}(\mathcal{G}_q) = \Omega(max\{(d_{avg}(\mathcal{G}_1))^q, (k_{max}(\mathcal{G}_1))^q\}). \tag{9.7}$$
$$k_{max}(\mathcal{G}_q) = O((\lambda_1(\mathcal{G}_1))^q). \tag{9.8}$$

Proof. Lemma 9.4 is immediate from Lemmas 9.1, 9.2, and 9.3. ■

Lemma 9.5: Triangles in the Kronecker model

The number of triangles in $\{\mathcal{G}_q\}_{q\geq 1}$ increases exponentially with q. That is, if we let $\lambda(\mathcal{G}_1) = (\lambda_1, ..., \lambda_n)$ be the eigenvalues of the adjacency matrix of the seed graph $\mathcal{G}_1$, then

$$\#\Delta(\mathcal{G}_q) = \Theta\left(\left(\sum_{i=1}^{n} \lambda_i^3\right)^q\right). \tag{9.9}$$

Proof. Let $\lambda(\mathcal{G}_i) = (\lambda_1(\mathcal{G}_i), ..., \lambda_{n^i}(\mathcal{G}_i))$ be the eigenvalues of the adjacency matrix of $\mathcal{G}_i$. The number of walks of length 3 in $\mathcal{G}_i$ that begin and end on the same node is $\sum_{j=1}^{n^i}(\lambda_j(\mathcal{G}_i))^3$ [Tso08] and linearly related to the number of triangles, i.e., $\#\Delta(\mathcal{G}_i) = \Theta(\sum_{j=1}^{n^i}(\lambda_j(\mathcal{G}_i))^3)$. For seed graph $\mathcal{G}_1$, $\sum_{j=1}^{n}(\lambda_j(\mathcal{G}_1))^3 = (\sum_{j=1}^{n}(\lambda_j(\mathcal{G}_1))^3)^1$. Assume $\sum_{j=1}^{n^i}(\lambda_j(\mathcal{G}_i))^3 = (\sum_{j=1}^{n}(\lambda_j(\mathcal{G}_1))^3)^i$. As $\lambda(\mathcal{G}_{i+1}) = sort(\lambda(\mathcal{G}_i) \otimes \lambda(\mathcal{G}_1))$ [VL00],

$$\begin{aligned}\sum_{j=1}^{n^{(i+1)}} (\lambda_j(\mathcal{G}_{i+1}))^3 &= \sum_{r=1}^{n^i}\sum_{s=1}^{n}(\lambda_r(\mathcal{G}_i))^3(\lambda_s(\mathcal{G}_1))^3 \\ &= \left(\sum_{r=1}^{n^i}(\lambda_r(\mathcal{G}_i))^3\right)\left(\sum_{s=1}^{n}(\lambda_s(\mathcal{G}_1))^3\right) = \left(\sum_{s=1}^{n}(\lambda_s(\mathcal{G}_1))^3\right)^{(i+1)}.\end{aligned}$$

By induction, $\sum_{j=1}^{n^q}(\lambda_j(\mathcal{G}_q))^3 = (\sum_{j=1}^{n}(\lambda_j(\mathcal{G}_1))^3)^q$, $\forall q \geq 1$. Hence, $\#\Delta(\mathcal{G}_q) = \Theta(\sum_{j=1}^{n^q}(\lambda_j(\mathcal{G}_q))^3) = \Theta((\sum_{j=1}^{n}(\lambda_j(\mathcal{G}_1))^3)^q)$, $\forall q \geq 1$. ■

Based on the speed of increase of degeneracy and triangle count given in Lemmas 9.4 and 9.5, Theorem 9.1 states a sufficient and a necessary condition for C-T PATTERN to hold in the Kronecker model. Note that $\sum_{i=1}^{n} \lambda_i^3 = \lambda_1^3$ in Eq. (9.10) and $\sum_{i=1}^{n} \lambda_i^3 \leq \lambda_1^3$ in Eq. (9.11) can hold since the eigenvalues can be negative.

Theorem 9.1: C-T PATTERN in the Kronecker model

In a Kronecker model with a seed graph $\mathcal{G}$,

1. A sufficient condition for C-T PATTERN to hold is, in the seed graph $\mathcal{G}$,

$$\max(d_{avg}^3, k_{max}^3) = \sum_{i=1}^{n} \lambda_i^3 = \lambda_1^3. \tag{9.10}$$

2. A seed graph satisfying the sufficient condition exists.
3. A necessary condition for C-T PATTERN to hold is, in the seed graph $\mathcal{G}$,

$$\max(d_{avg}^3, k_{max}^3) \leq \sum_{i=1}^{n} \lambda_i^3 \leq \lambda_1^3. \tag{9.11}$$

Proof. Assume that the sufficient condition holds, and $c = \max(d_{avg}^3, k_{max}^3) = \sum_{i=1}^{n} \lambda_i^3 = \lambda_1^3$. Then, $(k_{max}(\mathcal{G}_q))^3 = \Theta(c^q)$ by Lemma 9.4, and $\#\Delta(\mathcal{G}_q) = \Theta(c^q)$ by Lemma 9.5. Therefore, $\#\Delta(\mathcal{G}_q) = \Theta((k_{max}(\mathcal{G}_q))^3)$, and C-T PATTERN holds. The Mediator seed graph in Table 9.3 satisfies this sufficient condition.

Assume that the necessary condition is not met. By Lemmas 9.4 and 9.5, $(k_{max}(\mathcal{G}_q))^3$ increases faster than $\#\Delta(\mathcal{G}_q)$ if $\sum_{i=1}^{n} \lambda_i^3 < \max(d_{avg}^3, k_{max}^3)$. Instead, $\#\Delta(\mathcal{G}_q)$ increases faster than $(k_{max}(\mathcal{G}_q))^3$ if $\lambda_1^3 < \sum_{i=1}^{n} \lambda_i^3$. Hence, $\#\Delta(\mathcal{G}_q) \neq \Theta((k_{max}(\mathcal{G}_q))^3)$, and C-T PATTERN does not hold. ■

Many realistic seed graphs satisfy the necessary condition for C-T PATTERN, as listed in Table 9.3. Especially, Mediator satisfies also the sufficient condition. Even seed graphs that do not satisfy the sufficient condition empirically follow C-T PATTERN, as seen in Figure 9.7. The slope of the regression line between the number of triangles and degeneracy is close to $1/3$ in log scale with all the seed graphs considered.

In addition to the Kronecker model, C-T PATTERN is proved also in Erdős-Rényi (ER) Model, another mathematically tractable graph generation model where each of possible $\binom{n}{2}$ edges occurs independently with probability p, as formalized in Theorem 9.2. Figure 9.8 shows C-T PATTERN in ER random graphs generated with different p values. The slopes of the regression lines are close to $1/3$ in log scale, regardless of p values.

Theorem 9.2: C-T PATTERN in the ER model

Graphs generated by the ER model with probability p follow C-T PATTERN in terms of expected values if $p = \Omega(\log n/n)$. That is,

$$\mathbb{E}[\#\Delta] = \Theta(\mathbb{E}[k_{max}]^3).$$

Table 9.3: **Sample seed graphs for the Kronecker model.** All graphs satisfy the necessary condition for C-T PATTERN, and Mediator satisfies also the sufficient condition. When computing k_{max} and d_{avg}, we add one to the degree for each self-loop if self-loops exist.

	Core-Periphery	**Mediator**	**Triangle**	**Star**
Shape				
k^3_{max}	1	8	8	1
d^3_{avg}	3.38	8	18.96	5.36
$\sum_{i=1}^{n} \lambda_i^3$	4	8	20	10
λ_1^3	4.24	8	20.39	12.21

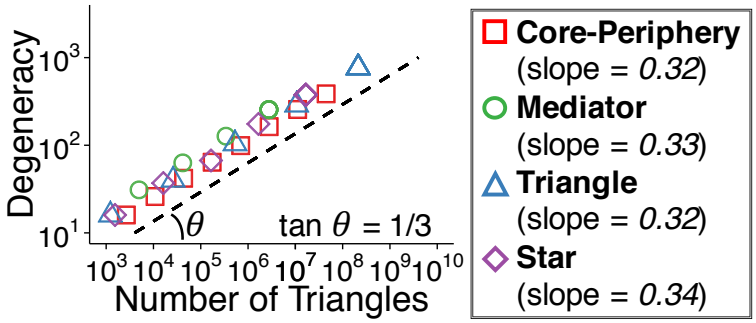


Figure 9.7: **CORE-TRIANGLE PATTERN in the Kronecker model.** Points represent graphs generated by the Kronecker model with different seed graphs. The slopes between the triangle count and the degeneracy are close to $1/3$ (dashed line) in log scale regardless of seed graphs.

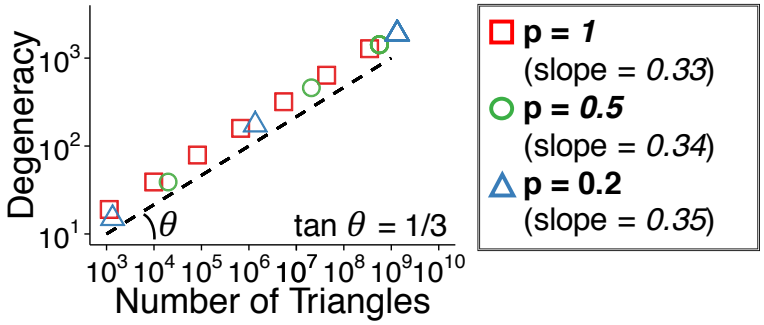


Figure 9.8: **CORE-TRIANGLE PATTERN in the ER model.** Points represent graphs generated by the ER model with different p values. The slopes between the triangle count and the degeneracy are close to $1/3$ (dashed line) in log scale regardless of p.

Proof. From $p = \Omega(\log n/n)$, there exists $c > 0$ such that $p \geq c \log n/n$. Let $\epsilon = \max(2, 12/c)$ (> 1). Then,

$$\begin{aligned}
&\mathbb{P}[\exists v \in V \text{ s.t. } d_>(v)(1+\epsilon)(n-1)p] \\
&\quad \leq n\mathbb{P}[d_>(v)(1+\epsilon)(n-1)p] && \text{(Boole's inequality)} \\
&\quad \leq n\exp\{-(n-1)p\epsilon/3\} && \text{(Chernoff bound)} \\
&\quad \leq n\exp\{-c\log(n)(n-1)\epsilon/3n\} && (p \geq c\log(n)/n) \\
&\quad \leq n\exp\{-4\log(n)(n-1)/n\} && (\epsilon \geq 12/c) \\
&\quad \leq n\exp\{-2\log n\} = n^{-1}.
\end{aligned}$$

Let $q = \mathbb{P}[\exists v \in V \text{ s.t. } d_>(v)(1+\epsilon)(n-1)p]$. Then,

$$\begin{aligned}
\mathbb{E}[k_{max}] \leq \mathbb{E}[d_{max}] &\leq (1-q)(1+\epsilon)(n-1)p + q(n-1) \\
&\leq (1+\epsilon)(n-1)p + (n-1)/n = O(np)
\end{aligned}$$

Hence, $\mathbb{E}[k_{max}] = O(np)$. As $\mathbb{E}[k_{max}] \geq \mathbb{E}[d_{avg}/2] = \Omega(np)$ by Lemma 9.1, $\mathbb{E}[k_{max}] = \Theta(np)$ holds.

On the other hand, the expected number of triangles is the sum of probabilities that each three nodes form a traingle:

$$\mathbb{E}[\#\Delta] = \frac{n(n-1)(n-2)}{6}p^3.$$

Therefore, $\mathbb{E}[\#\Delta] = \Theta(n^3p^3) = \Theta(\mathbb{E}[k_{max}]^3)$ holds. ■

9.4.3 Proposed Algorithm: CORE-D

Based on C-T PATTERN, we propose CORE-D, a single-pass streaming algorithm for estimating degeneracy. We empirically show that CORE-D gives a better trade-off between speed and accuracy than a state-of-the-art method.

9.4.3.1 Algorithm Description

Computing degeneracy in a graph stream not fitting in memory remains as a challenge. As explained in Section 9.2.2, a recent approximate method, namely LOGPASS, needs $O(\log_{\alpha/2}(n))$ passes of a graph stream for α-approximation of its degeneracy, for any real number α greater than 2, with $O(n)$ memory requirements (regardless of α). However, multiple passes of graph streams are time-consuming and not even possible in many real-world settings.

In contrast, the number of triangles can be estimated accurately even in a single pass, as we show in Chapters 4-6. Simply sampling each edge with probability p in a graph stream and estimating the number of triangles in the whole graph from that in the sampled graph [TKMF09] also can be thought as a single-pass streaming algorithm if the sampled graph fits in memory and needs not be streamed again. This sampling method, which our CORE-D method uses, estimates triangle-count accurately even with less than n sampled edges.

CORE-TRIANGLE PATTERN (Observation 9.2), a high correlation between degeneracy and the number of triangles, enables using the accurately estimated triangle-count for estimating degeneracy. Specifically, we consider the following models, whose coefficients are denoted by w, relating the number of triangles and degeneracy:

- **BASIC MODEL** (Baseline): $\log(\hat{k}_{max}) = w_{0,0} + w_{0,1}\log(n) + w_{0,2}\log(m)$

Table 9.4: **Models of CORE-D.** *: p value ≤ 0.05, ****: p value ≤ 0.0001. OVERALL MODEL fits the data best (i.e., has the highest adjusted R^2), and *only the log of triangle-count is statistically significant* with p value < 0.001.

Model	Variable	Coefficient Estimate	Std.Err.	p-value
Basic ($R^2_{adj} = 0.72$)	1	-0.03	0.43	0.94
	$\log(n)$	-0.35	0.28	0.24
	$\log(m)$	0.62	0.24	0.02 *
Triangle ($R^2_{adj} = 0.89$)	1	-0.20	0.23	0.40
	$\log(\#\Delta)$	0.32	0.03	1.3e-07 ****
Overall ($R^2_{adj} = 0.95$)	1	0.03	0.20	0.88
	$\log(n)$	0.18	0.15	0.26
	$\log(m)$	-0.50	0.20	0.03 *
	$\log(\#\Delta)$	0.59	0.09	3.3e-05 ****

Algorithm 9.1 CORE-D with TRIANGLE MODEL

Input: (1) graph stream: $\mathcal{G}$, (2) sampling probability: p, (3) coefficients in triangle model: $(w_{1,0}, w_{1,1})$
Output: Estimated degeneracy: $\hat{k}_{max}$

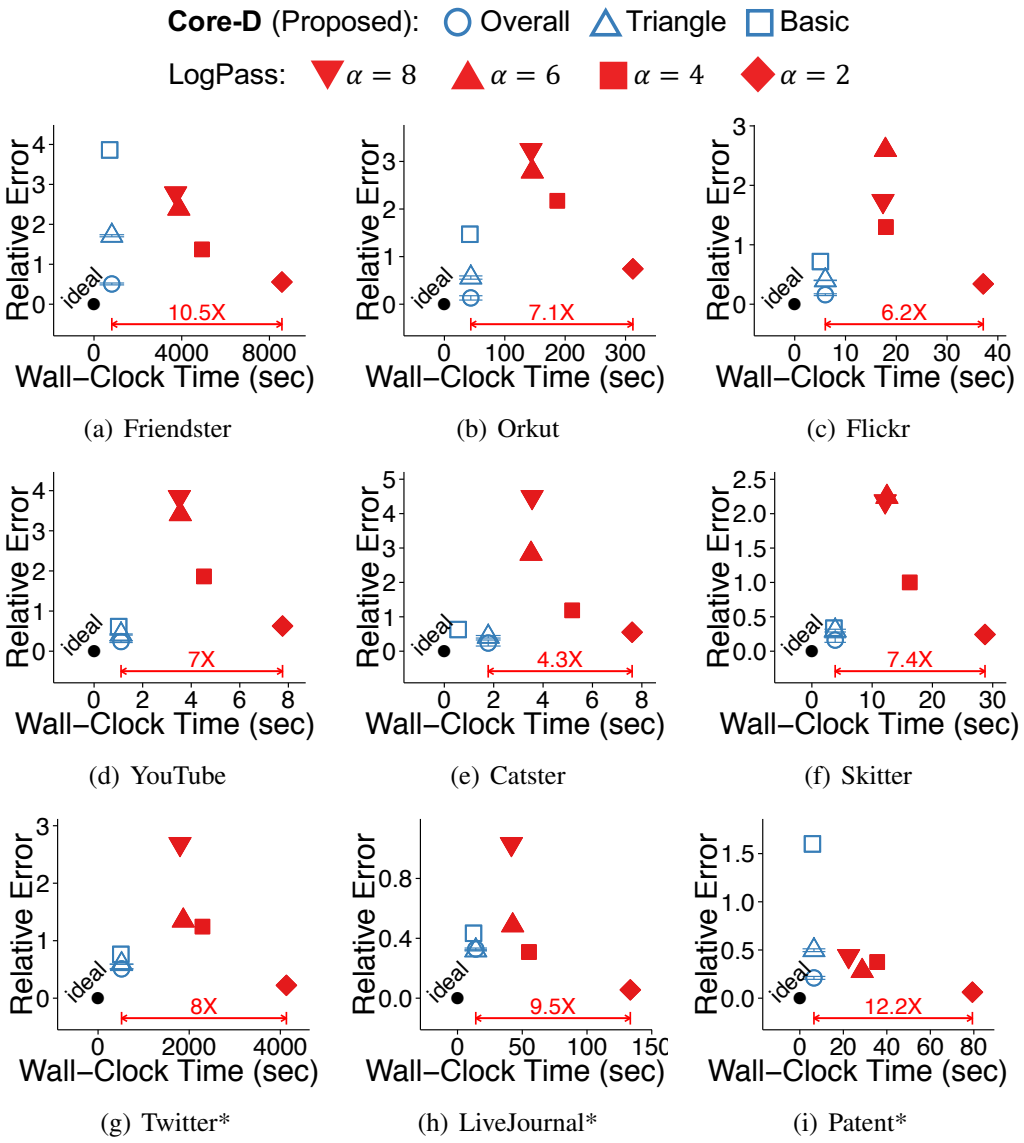
1: $\mathcal{G}_{Sample} = \emptyset$
2: **for each** edge e in $\mathcal{G}$ **do**
3: add e to $\mathcal{G}_{Sample}$ with probability p
4: $\#\Delta_{Sample} \leftarrow$ `InMemoryTriangleCounting`$(\mathcal{G}_{Sample})$ [Sch07]
5: $\hat{\#\Delta} \leftarrow \#\Delta_{Sample} * (1/p)^3$
6: $\hat{k}_{max} \leftarrow exp(w_{1,0} + w_{1,1} \log(\hat{\#\Delta}))$
7: **return** $\hat{k}_{max}$

- **TRIANGLE MODEL**: $\log(\hat{k}_{max}) = w_{1,0} + w_{1,1} \log(\#\Delta)$
- **OVERALL MODEL**: $\log(\hat{k}_{max}) = w_{2,0} + w_{2,1} \log(n) + w_{2,2} \log(m) + w_{2,3} \log(\#\Delta)$

Table 9.4 summarizes the estimates of the coefficients obtained by linear regression on the real-world graphs listed in Table 9.2. The OVERALL MODEL has the highest adjusted R-squared (0.95) among all possible linear models, and the log triangle-count is statistically significant with p-value $<$ 0.001, proving the effectiveness of using triangle-count for estimating degeneracy.

Given a new graph stream, we estimate the node-count, the edge-count, and the triangle-count in the graph in a single pass. Then, by plugging these statistics into one of the models, whose coefficients are given as input parameters, we obtain an estimate of degeneracy. Algorithm 9.1 describes the details of CORE-D with TRIANGLE MODEL. For estimating the triangle-count, CORE-D requires $O(mp)$ memory space on average to store sampled edges. The memory requirement becomes $O(n)$ if we set sampling probability p to $O(n/m)$.

We also need n and m for BASIC MODEL and OVERALL MODEL. We obtain m by simply counting edges in the graph stream. In many real-world settings, n is available or is easily computed from the difference between maximum and minimum node ids. Otherwise, we obtain n by counting distinct node ids with $O(n)$ space. Even when n and m are needed, CORE-D still requires only one pass



(a) Friendster (b) Orkut (c) Flickr

(d) YouTube (e) Catster (f) Skitter

(g) Twitter* (h) LiveJournal* (i) Patent*

* Graphs whose degeneracies are known to be affected by anomalies (see Section 9.3.2)

Figure 9.9: **CORE-D achieves both speed and accuracy.** Points in each plot represent the performances of different methods with different parameters. For randomized algorithms, error bars show $\pm$ one standard deviation over ten runs. Lower-left region indicates better performance. Our proposed CORE-D algorithm provided a better trade-off between speed and accuracy than LOGPASS. Specifically, CORE-D (with OVERALL MODEL) was up to *12*$\times$ *faster* than LOGPASS ($\alpha = 2$), while still providing comparable accuracy. Among the models of CORE-D, OVERALL MODEL yielded the best performance in most datasets.

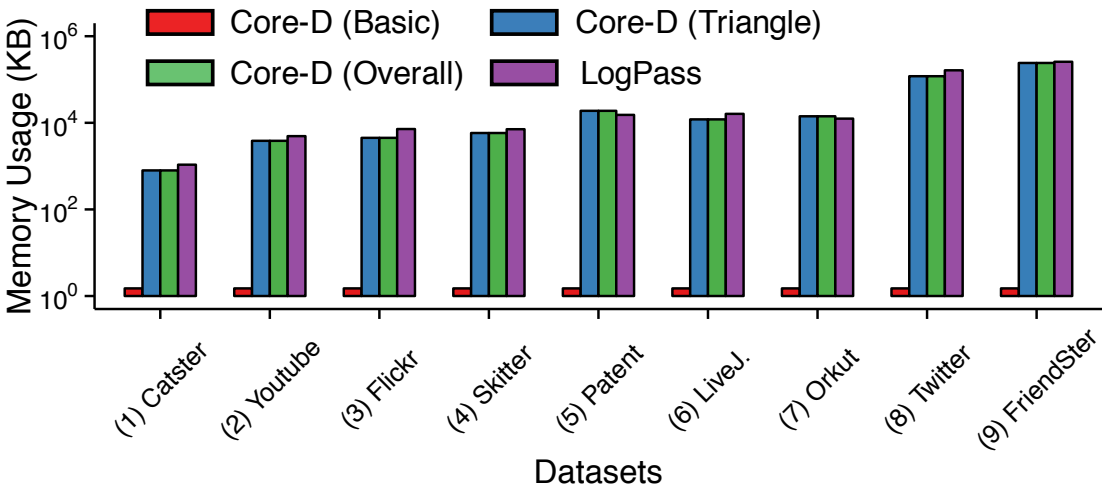


Figure 9.10: **CORE-D is comparable with LOGPASS in terms of memory requirements.** CORE-D with OVERALL MODEL or TRIANGLE MODEL has similar memory requirements to LOGPASS, and CORE-D with BASIC MODEL has the smallest memory requirements.

because both edge sampling (lines 2-3 of Algorithm 9.1) and computing n and m can be conducted at the same time within one pass.

9.4.3.2 Experiments

Experimental Settings. We compare the speed, accuracy, and memory efficiency of CORE-D and LOGPASS. We used a desktop with a 3.6GHz cpu and 16GB memory space, and graphs (see Table 9.2) were streamed from disk whose speed is 192MB/sec for sequential read. We assumed that n is known or is computed easily from node ids in all methods. We set sampling probability p to $n/(5m)$. With this value of p, CORE-D estimated degeneracy reliably and accurately, while using similar amount of memory space to LOGPASS. For the coefficients of the models (e.g., $w_{1,0}$ and $w_{1,1}$ in Algorithm 9.1), we used the values estimated from the real-world graphs listed in Table 9.2. Specifically, we used $\log(n)$, $\log(m)$ and $\log(\#\Delta)$ in all the datasets except the one being tested as training data, and learned the coefficients using linear regression. A graph being tested was excluded from training data for fairly evaluating accuracy in a new (unseen) graph. To measure the accuracy of the considered algorithms, we used *relative error* defined as:

$$relative\_error(k_{max}, \hat{k}_{max}) := |k_{max} - \hat{k}_{max}|/k_{max}.$$

For randomized algorithms, we reported the average over ten runs.

Speed and Accuracy. As seen in Figure 9.9, CORE-D gave a significantly better trade-off between accuracy and speed than LOGPASS. Specifically, CORE-D (with OVERALL MODEL) was up to $12\times$ *faster* than LOGPASS ($\alpha = 2$) with similar accuracy. Mote that CORE-D with OVERALL MODEL was more accurate than LOGPASS in all the datasets except the ones whose degeneracies are known to be affected by anomalies (see Section 9.3.2). Among the models of CORE-D, OVERALL MODEL consistently yielded the best performance in all the datasets. BASIC MODEL, solely based on the numbers of nodes and edges, showed the lowest accuracy especially in the Friendster and Patent datasets. This supports the effectiveness of using the number of triangles for estimating degeneracy.

Memory Efficiency. We experimentally compare the memory requirements of CORE-D and LOGPASS, whose memory requirement does not depend on α. The memory requirement of CORE-D with OVERALL MODEL or TRIANGLE MODEL was similar to that of LOGPASS, as seen in Figure 9.10. Specifically, CORE-D with OVERALL MODEL or TRIANGLE MODEL required 63-124% of the memory space required by LOGPASS. CORE-D with BASIC MODEL, which does not have to sample edges for estimating the triangle count, required the least memory space.

9.5 P3: "Structured Core Pattern" and Influential Spreader Identification

In this section, we describe STRUCTURED-CORE PATTERN and discuss its application to influential spreader identification.

9.5.1 Observation: Pattern in Real-world Graphs

How do the degeneracy-cores in real-world graphs look like? Are they cliques? Our observation shows that degeneracy-cores in real-world graphs are not cliques but have structural patterns such as core-periphery [BE00] (i.e., have a cohesive core and a loosely connected periphery) and communities [New06] (i.e., consist of groups of nodes with dense connections internally and sparser connections between groups). This leads to Observation 9.3, which is supported by the following facts:

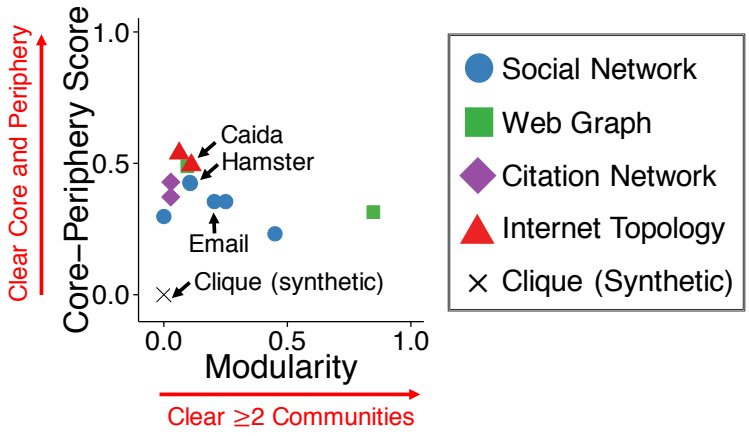
- As shown in Table 9.2, degeneracy-cores have density much less than one in all the datasets (e.g., 0.02 in the Friendster dataset and 0.03 in the Orkut dataset) except the LiveJournal and Twitter datasets, whose degeneracy-cores include anomalies (see Section 9.3.2).
- In all the datasets, degeneracy-cores have significantly higher core-periphery score [3] (e.g., 0.54 in the Skitter dataset and 0.49 in the Stanford dataset) than cliques, as shown in Figure 9.11(a).
- Figure 9.11(a) also indicates that many datasets have significantly higher modularity [4] than cliques (e.g., 0.85 in the NotreDame dataset and 0.47 in the Orkut dataset).
- The sparsity patterns of the adjacency matrices of degeneracy-cores reveal structural patterns such as core-periphery and communities. Figures 9.11(b)-9.11(d) show the sparsity patterns of some real-world degeneracy-cores, where nodes are reordered as proposed in [HSP+16]. In Figure 9.11(b), nodes in the degeneracy-core are clearly divided into the core and the periphery. In Figure 9.11(c), nodes are divided into five communities. In Figure 9.11(d), nodes are clearly divided into the core and the periphery, and the nodes in the core are again divided into three communities.

Observation 9.3: STRUCTURED-CORE PATTERN

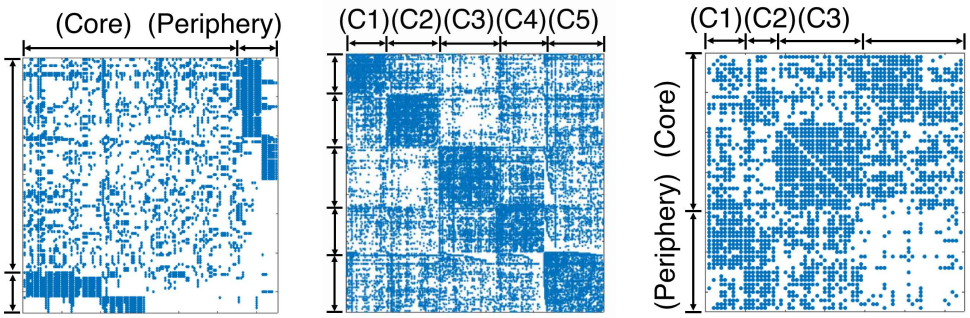
In real-world graphs, degeneracy-cores have structural patterns such as core-periphery and communities.

[3] Strength of core-periphery structure. The correlation between the adjacency matrix of the measured graph and that of a graph with perfect core-periphery structure. See [BE00] for details.

[4] Strength of community structure. The fraction of the edges within communities minus such fraction expected in a randomly connected graph. See [New06] for details.



(a) Structural Property of Real-world Graphs



(b) Core-Periphery (Hamsterster)

(c) Communities (Email)

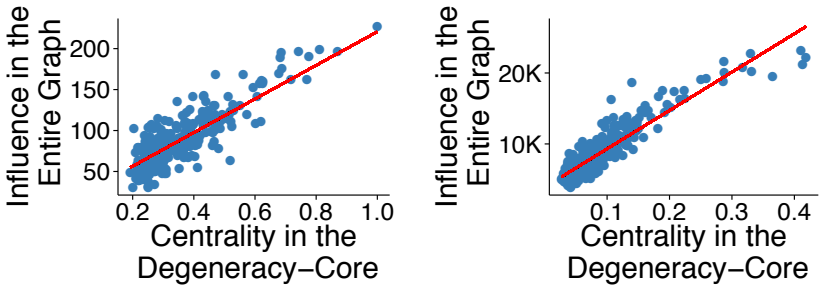
(d) Core-Periphery with Communities (Caida)

Figure 9.11: **Degeneracy-cores of real-world graphs are not cliques but have structural patterns,** such as core-periphery and communities. (a) Core-periphery score ($\in [0, 1]$) and modularity ($\in [-0.5, 1]$) measure the strength of core-periphery and community structure, resp., in graphs. (b), (c), and (d) show the sparsity patterns of the adjacency matrices of degeneracy-cores. Ci denotes the i-th community.

9.5.2 Application: Finding Influential Spreaders

The problem of identifying influential spreaders in social networks has gained considerable attention due to its wide applications, including information spreading, viral marketing, and epidemic disease control (see Section 9.6 for related work). For the problem of finding individual spreaders (instead of a set of spreaders, which is another well-studied problem, called the influence maximization problem [KKT03]), it is shown in [KGH+10] that the ability of nodes to spread information to the large portion of a network is more closely related to their coreness rather than other centrality measures such as degree and betweenness centrality. This implies that the nodes in the degeneracy-core tend to be good spreaders,

Our STRUCTURED-CORE PATTERN reveals that even nodes belonging to the degeneracy-core can be further divided into those in core and those in periphery; or those connecting communities and those inside a community. We observe that this position of a node within the degeneracy-core is highly related to its ability to spread information not just in the degeneracy-core but in the entire graph. Specifically, we find out a strong correlation between influence (see Section 9.8 for the measurement method) and in-core centrality, which we define in Definition 9.4, as shown in Figure 9.12.



(a) Email ($r = 0.84$) (b) Orkut ($r = 0.92$)

Figure 9.12: **Nodes central in degeneracy-cores are influential in entire graphs.** 300 nodes randomly picked in the degeneracy-core of each graph are plotted. r denotes the Pearson correlation coefficient. Influence is measured using the SIR model simulation (see Section 9.8), and in-core centrality (Definition 9.4) is used for measuring centrality.

Definition 9.4: In-Core Centrality

Let $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ be the degeneracy-core of graph $\mathcal{G}$, Then, for each node v in $\mathcal{V}'$, v's in-core centrality in $\mathcal{G}$ is defined as

$$i_v := v\text{'s eigenvector centrality in } \mathcal{G}'.$$

Among many centrality measures, eigenvector centrality (i.e., entries of the eigenvector corresponding to the largest real eigenvalue) is used since it is computationally efficient and is known to be effective in identifying influential spreaders [MSHM12].

This observation is used to further refine influential spreaders in the degeneracy-core in the following section.

9.5.3 Proposed Algorithm: CORE-S

Inspired by STRUCTURED-CORE PATTERN, we propose CORE-S, a top-k influential-spreader identification algorithm based on in-core centrality. We show that CORE-S gives a better trade-off between speed and accuracy than its top competitors.

9.5.3.1 Algorithm Description

As outlined in Algorithm 9.2, CORE-S first runs core decomposition and extracts the degeneracy-core $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$. Then, the in-core centralities of the nodes in $\mathcal{V}'$ are computed using power iteration. As the last step, CORE-S returns the top-k nodes with the highest in-core centralities. The time complexity of CORE-S is $O(n + m + Tm_{max} + n_{max} \log k)$, where $(n + m)$ is for core decomposition, Tm_{max} is for power iteration, and $n_{max} \log k$ is for top-k selection. T denotes the number of iterations in the power iteration.

Algorithm 9.2 CORE-S for top-k spreaders

Input: (1) graph: $\mathcal{G}$, (2) number of spreaders: k ($\leq n_{max}$)
Output: k influential spreaders

1: run the core decomposition of $\mathcal{G}$
2: extract the degeneracy-core $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ from $\mathcal{G}$
3: compute the in-core centrality of the nodes in $\mathcal{V}'$ by power iteration in $\mathcal{G}'$
4: **return** top-k nodes with the highest in-core centralities

9.5.3.2 Experiments

Experimental Settings. The experimental settings were the same with those in Section 9.4.3.2. We compared the average influence of ten nodes chosen by CORE-S with that of the nodes chosen by the following methods:

- K-CORE [KGH+10]: all nodes with the highest coreness.
- K-TRUSS [RMV15]: all nodes with the highest trussness (defined in Section 9.2.1).
- Eigenvector Centrality (EC) [MSHM12]: top-ten nodes with the highest eigenvector centralities in the entire graph.

The influence of each node was measured using SIR simulation (see Section 9.8 for details). We also compared the time taken for choosing influential nodes in each method.

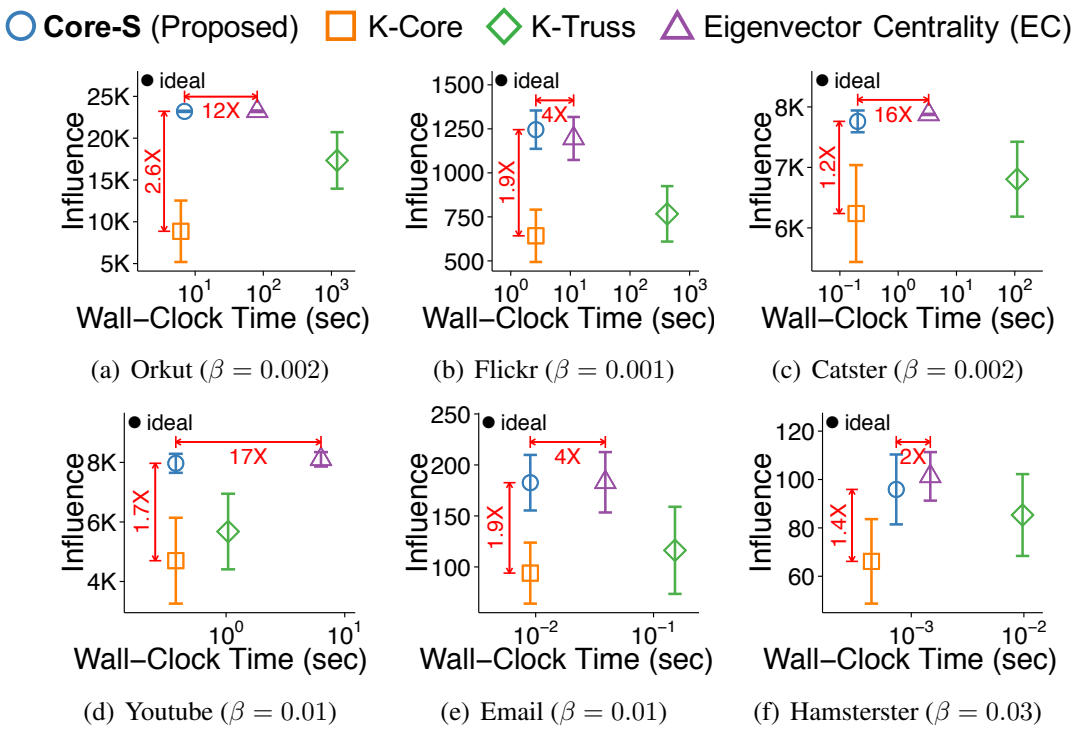
Speed and Accuracy. As seen in Figure 9.13, CORE-S provided the best trade-off between speed and accuracy in social networks. Specifically, the average influence of the nodes chosen by CORE-S was up to $2.6\times$ *higher* than that of all the nodes in the degeneracy-core (K-CORE). However, additional time taken in CORE-S for further refining nodes in degeneracy-cores was at most 12% of the time taken for the core decomposition of entire graphs in all the considered social networks except the smallest Hamsterster dataset. CORE-S was up to $17\times$ *faster* than EC, which computes the eigenvector centrality in entire graphs (instead of only in degeneracy-cores). However, the average influence of the nodes chosen by CORE-S was comparable (95-104%) with that of the nodes chosen by EC.

9.6 Related Work

Related work forms the following groups: applications of k-core analysis, algorithms for k-core analysis, dense subgraphs, graph-based anomaly detection, and influential spreader identification.

Applications of k-core Analysis. The concept of a k-core [Sei83] has been applied to hierarchical structure analysis [AHDBV06], graph visualization [AHDBV06], densest subgraph detection [Cha00] (a special case of DSM in Section 9.3.3.2), important protein identification [WA05], influential spreader detection [KGH+10], and graph clustering [GMTV14]. Degeneracy also has been used as a graph-sparsity measure in many domains such as AI [Fre82] and Bioinformatics [BH03].

Algorithms for k-core Analysis. Core decomposition can be computed in $O(n + m)$ by repeatedly removing nodes with the smallest degree [BZ03]. [SGJS+13] proposed an incremental algorithm, while [CKCÖ11] proposed an external memory algorithm, which requires $O(k_{max})$ scans of graphs. For degeneracy, [FCT14] proposed a streaming algorithm requiring $O(\log_{\alpha/2}(n))$ passes of a graph



(a) Orkut ($\beta = 0.002$) (b) Flickr ($\beta = 0.001$) (c) Catster ($\beta = 0.002$)

(d) Youtube ($\beta = 0.01$) (e) Email ($\beta = 0.01$) (f) Hamsterster ($\beta = 0.03$)

Figure 9.13: **CORE-S achieves both speed and accuracy.** β denotes the infection rate in the SIR model. Points in each plot represent the performances of different methods. Upper-left region indicates better performance. CORE-S provided the best trade-off between speed and accuracy. Specifically, it found up to $2.6\times$ *more influential* nodes than K-CORE with similar speed. Compared with EC, CORE-S was up to $17\times$ *faster*, while still finding nodes with comparable (95-104%) influence.

and n memory space for $\alpha(> 2)$-approximation. Our CORE-D, however, requires only one pass of a graph and n memory space for accurately estimating degeneracy.

Dense Subgraphs. In addition to k-cores, many notions of dense subgraphs have been proposed. The most strict one is a maximal clique [BK73] (i.e, a complete subgraph not included in any other complete subgraphs). Since the definition of a clique is too rigid for many purposes, many relaxed forms have been proposed including n-cliques [Luc50], k-plexes [SF78], n-clans [Mok79], n-clubs [Mok79], and quasi-cliques [ARS02]. However, the computation of these dense subgraphs is NP-hard, while finding k-cores (i.e., core decomposition) runs in $O(n + m)$ [BZ03]. The notion of a k-core also has been generalized [Coh08, SSPC15].

Graph-based Anomaly Detection. There have been diverse approaches (belief propagation [PCWF07], egonet features [AMF10], spectral methods [PSS+10], etc.) for anomaly detection in graphs (see [ATK15] for a survey). Recent methods largely focus on dense subgraphs [HSS+17, SBGF14, BXG+13, ZZY+17] or more generally dense subtensors [SHF18, SHKF18, SHKF17b, JBC+16], which anomalies tend to form (see Chapters 10-13). Especially, many of them are based on densest subgraphs (i.e., subgraphs with maximum average degree). We show that our CORE-A, which detects smaller dense subgraphs consisting of low-degree nodes, are complementary to these densest-subgraph based methods, and the combination of both approaches has the best of both approaches.

Influential Spreader Identification. The problem of identifying influential spreader is sub-categorized into (1) finding a group of spreaders, which is called the influence maximization problem [KKT03],

and (2) finding individual influential spreaders. For the second problem, on which we focus, nodes with high coreness [KGH+10], truss number [RMV15], and eigenvector centrality [MSHM12] are known as good spreaders. Our CORE-S combines these measures so that only the advantages of each measure (i.e., low computational cost of coreness and high accuracy of eigenvector centrality) are taken.

9.7 Summary

In this chapter, we discover three empirical patterns in real-world graphs related to k-cores, and utilize them for several applications. Specifically, our contributions are summarized as follows:

- **MIRROR PATTERN and CORE-A**: We observe a strong correlation between the coreness and the degree of nodes. CORE-A, which measures the deviation from this trend, successfully detects anomalies in real-world graphs and complements a state-of-the-art anomaly detection method.
- **CORE-TRIANGLE PATTERN and CORE-D**: We discover a 3-to-1 power law between degeneracy and triangle count. Our CORE-D method uses this pattern for accurately estimating degeneracy in only one pass of a graph stream and up to *12× faster* than a recent multi-pass method.
- **STRUCTURED-CORE PATTERN and CORE-S**: We observe that degeneracy-cores have non-trivial structures (core-periphery, communities, etc). CORE-S, which finds nodes central within degeneracy-cores, identifies influential spreaders up to *17× faster* than methods with similar accuracy.

Reproducibility: The source code and datasets used in this chapter are available at http://www.cs.cmu.edu/~kijungs/codes/kcore/.

9.8 Appendix: Measuring Influence of Nodes by Simulating the SIR model

To evaluate influence as a spreader, we simulate spreading processes using the SIR model [KGH+10], a widely-used epidemic model. Initially, a vertex chosen as the seed is in the *infectious state* (I-state), while the others are in the *susceptible state* (S-state). Each vertex in the I-state infects each of its neighbors in the S-state with probability β (infection rate) and then enters the *recovered state* (R-state). This is repeated until no vertex is in the I-state. The influence of a seed, the initially infected vertex, can be quantified by the number of vertices infected at any time during the process. To reduce random effects, we repeat the whole process 100 times, and use the average number of infected vertices as the measure of influence. β is set close to the epidemic threshold λ_1^{-1}, as in previous work [RMV15].

Chapter 10

Detecting Dense Subtensors in Large Tensors (0): Preliminaries

In this preliminary chapter on dense-subtensor detection, we (a) provide motivation for dense-subtensor detection, (b) review related work, and (c) introduce some concepts and datasets that are frequently used in the following chapters on dense-subtensor detection.

10.1 Motivation

Imagine that you manage a social review site (e.g., Yelp) and have the records of which accounts wrote reviews for which restaurants. How do you detect suspicious lockstep behavior: for example, a set of accounts which give fake reviews to the same set of restaurants? What about the case where additional information is present, such as the timestamp of each review, and the keywords in each review?

Such problems of detecting suspicious lockstep behavior have been extensively studied from the perspective of dense-subgraph detection, as in Chapter 9. Intuitively, in the above example, highly synchronized behavior induces dense subgraphs in the bipartite review graph of accounts and restaurants. Indeed, methods which detect dense subgraphs have been successfully used to spot fraud in settings ranging from social networks [BXG+13, JCB+14a, JCB+14b, HSS+17, SERF18], auctions [PCWF07], search engines [GKT05], and the web [SERF18].

Additional information helps identify suspicious lockstep behavior more accurately. In the above example, the fact that reviews forming a dense subgraph were also written at about the same time, with the same keywords and rating scores, makes the reviews even more suspicious. A natural and effective way to incorporate such extra information is to model data as a tensor (e.g., a 5-order tensor whose modes are users, restaurants, timestamps, rating scores, and keywords) and find dense subtensors in it. In the following chapters, we show that unusually dense subtensors in real-world tensors indicate many interesting anomalies, including various types of network attacks, spam reviews, and bot activities.

10.2 Related Work

We review previous work on dense-subgraph detection, dense-subtensor detection, and tensor decomposition. See Table 10.1 for a comparison of some algorithms for detecting dense subgraphs or subtensors.

Table 10.1: **Comparison of methods for detecting dense subgraphs or subtensors.** Our proposed algorithms (i.e., M-ZOOM, D-CUBE, DENSESTREAM, and DENSEALERT) provide distinct advantages.

	FRAUDAR [HSS+17]	Densest Subgraph [KS09]	GreedyOQC [TBG+13]	LocalOQC [TBG+13]	CROSSSPOT [JBC+16]	MAF [MGF11]	CPD [KB09]	**M-ZOOM** (Chapter 11)	**D-CUBE** (Chapter 12)	**DENSESTREAM** (Chapter 13)	**DENSEALERT** (Chapter 13)
Graph Data	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Tensor Data					✓	✓	✓	✓	✓	✓	✓
Out-of-core Computation									✓		
Distributed Computation									✓		
Incremental Computation										✓	✓
Sublinear Space Requirements											✓
Approximation Guarantee	✓	✓	✓					✓	✓	✓	✓
Flexibility in Density Measures	✓		✓	✓	✓			✓	✓		
Multiple Subtensor Detection					✓	✓	✓	✓	✓		✓

Dense Subgraph Detection: (1) Theory. The *densest subgraph problem*, the problem of finding the subgraph that maximizes average degree, has been extensively studied in theory (see [LRJA10] for a survey). The two major directions are max-flow based exact algorithms [Gol84, KS09] and greedy algorithms [Cha00, KS09] giving a $1/2$-approximation guarantee. The latter direction has been extended to streaming settings [ELS15, MTVV15, BHNT15] as well as distributed settings [BKV12, BGM14]. Variants allow for size restrictions [AC09], providing a $1/3$-approximation to the densest subgraph for the lower bound case. Variants also allow for multiple dense-subgraph detection [BBC+15, GGT16] or more general density measures, such as edge surplus [TBG+13]. A related line of research deals with dense submatrices in binary matrices where the definition of density is designed for the purpose of frequent itemset mining [SM04] or formal concept mining [CBRB08, IKPZ13].

Dense Subgraph Detection: (2) Application to Anomaly Detection. Dense-subgraph detection has received much attention also in the data mining community, with a focus on its application to anomaly detection. Spectral approaches [PSS+10, JCB+14b, SBGF14] make use of eigendecomposition or SVD of adjacency matrices for dense-subgraph detection. Such approaches have been used to spot anomalous pattens in patent graphs [PSS+10], lockstep followers in social networks [JCB+14b], and stealthy or small-scale attacks in social networks [SBGF14]. Other approaches include NETPROB [PCWF07], which uses belief propagation to detect fraud-accomplice bipartite cores in auction networks, and COPYCATCH [BXG+13], which uses one-class clustering and sub-space clustering to identify 'Like' boosting on Facebook. In addition, ODDBALL [AMF10] spots near-cliques in a graph of posts in blogs based on egonet features. FRAUDAR [HSS+17] and CORE-A ([SERF18]), which generalize a greedy algorithm for the densest subgraph problem [KS09] so that the suspiciousness of each node and edge can be incorporated, spot follower-boosting services on Twitter (see Chapter 9).

Dense Subtensor Detection. Extending dense-subgraph detection to multi-aspect data (i.e., tensors) [JBC+16, SHF18] incorporates additional aspects (i.e., dimensions), such as time, to identify dense regions of interest with greater accuracy and specificity. CROSSSPOT [JBC+16], which starts from a seed subtensor and adjusts it in a greedy way until it reaches a local optimum, shows high accuracy in practice but does not provide any theoretical guarantees on its running time and accuracy. M-ZOOM [SHF18], presented in Chapter 11, starts from the entire tensor and only shrinks it by removing slices one by one in a greedy way. M-ZOOM improves upon CROSSSPOT in terms of speed and approximation guarantees. D-CUBE [SHKF18], presented in Chapter 12, extends M-ZOOM to disk-resident or distributed tensors while providing the same accuracy guarantee of M-ZOOM. DENSESTREAM and DENSEALERT [SHKF17b], presented in Chapter 13, extend M-ZOOM to dynamic tensors, which evolve over time, while providing the same accuracy guarantee of M-ZOOM. Once dense subtensors are detected, for further analyses of detected dense subtensors, ZOORANK [LHS+17] prioritizes the entities forming the dense subtensors based on their contributions to the subtensors. Dense-subtensor detection has been found useful for detecting 'retweet boosting' on social media [JBC+16]; network attacks [MGF11, SHKF17b, SHF18, SHKF18]; spam reviews and rating attacks on review sites [SHKF17b, SHKF18]; and 'edit wars' and bot activities on Wikipedia [SHKF17b, SHF18, SHKF18].

Tensor Decomposition. Spectral methods for dense subgraphs are naturally extended to tensors, and tensor decomposition methods, such as HOSVD and CP Decomposition [KB09], can be used for dense-subtensor detection [MGF11]. Scalable algorithms for tensor decomposition also have been developed, including external-memory algorithms [OSP+17, SSK17], distributed algorithms [KPHF12, SSK17, JPF+16], incremental algorithms [STF06, ZVB+16, GPP18], and approximate algorithms based on sampling [PFS12] and count-min sketch [WTSA15]. However, dense-subtensor detection based on tensor decomposition has serious limitations: it usually detects subtensors with significantly lower density than search-based methods (as shown experimentally in the following chapters), provides no flexibility with regard to the choice of density metrics, and does not provide any approximation guarantee.

10.3 Concepts

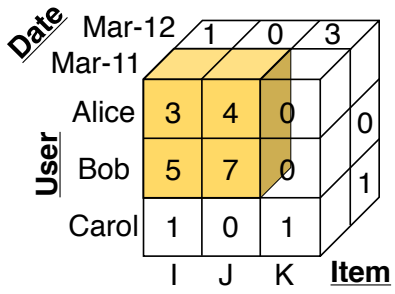
We introduce some concepts frequently used in the following chapters.

10.3.1 Tensors Represented as Relations

We use $[x] = \{1, 2, \cdots, x\}$ for brevity. Let $\mathcal{R}(A_1, \cdots, A_N, X)$ be a relation with N dimension attributes, denoted by $A_1, \cdots, A_N$, and a nonnegative measure attribute, denoted by X (see Example 10.1 for a running example). For each tuple $t \in \mathcal{R}$ and for each $n \in [N]$, $t[A_n]$ and $t[X]$ indicate the values of A_n and X, resp., in t. For each $n \in [N]$, we use $\mathcal{R}_n := \{t[A_n] : t \in \mathcal{R}\}$ to denote the set of distinct values of A_n in $\mathcal{R}$. The relation $\mathcal{R}$ is naturally represented as an N-way tensor of size $|\mathcal{R}_1| \times \cdots \times |\mathcal{R}_N|$. The value of each entry in the tensor is $t[X]$, if the corresponding tuple t exists, and 0 otherwise. Let $\mathcal{B}_n$ be a subset of $\mathcal{R}_n$. Then, a *subtensor* $\mathcal{B}$ in $\mathcal{R}$ is defined as $\mathcal{B}(A_1, ..., A_N, X) := \{t \in \mathcal{R} : \forall n \in [N], t[A_n] \in \mathcal{B}_n\}$, the set of tuples where each attribute A_n has a value in $\mathcal{B}_n$. The relation $\mathcal{B}$ is a 'subtensor' because it forms a subtensor of size $|\mathcal{B}_1| \times \cdots \times |\mathcal{B}_N|$ in the tensor representation of $\mathcal{R}$, as in Figure 10.1(b). We also define the *mass* of $\mathcal{R}$ as $M_\mathcal{R} = mass(\mathcal{R}) := \sum_{t \in \mathcal{R}} t[X]$ (i.e., the sum of the values of attribute X in $\mathcal{R}$), the *size* of $\mathcal{R}$ as $S_\mathcal{R} = size(\mathcal{R}) := \sum_{n=1}^{N} |\mathcal{R}_n|$, and the *volume* of $\mathcal{R}$ as $V_\mathcal{R} = volume(\mathcal{R}) := \prod_{n=1}^{N} |\mathcal{R}_n|$. We denote

User	Item	Date	Count
Alice	I	Mar-11	3
Alice	J	Mar-11	4
Bob	I	Mar-11	5
Bob	J	Mar-11	7
Carol	K	Mar-12	1
⋮	⋮	⋮	⋮

(a) Relation $\mathcal{R}$



(b) Tensor Representation of $\mathcal{R}$

Figure 10.1: **Pictorial description of Example 10.1.** (a) Relation $\mathcal{R}$ (*Purchase*), where the colored tuples compose a subtensor $\mathcal{B}$. (b) Tensor representation of $\mathcal{R}$, where the colored entries compose $\mathcal{B}$.

the set of tuples of $\mathcal{B}$ whose attribute $A_n = a$ by $\mathcal{B}(a, n) := \{t \in \mathcal{B} : t[A_n] = a\}$ and its mass, called the *attribute-value mass of* a *in* A_n, by $M_{\mathcal{B}(a,n)} := \sum_{t \in \mathcal{B}(a,n)} t[X]$.

Example 10.1: Purchase History

Let $\mathcal{R} = Purchase(user, item, date, count)$ depicted in Figure 10.1(a). Each tuple (u, i, d, c) in $\mathcal{R}$ indicates that user u purchased c units of item i on date d. The first three attributes $A_1 = user$, $A_2 = item$, and $A_3 = date$ are dimension attributes, and the other one $X = count$ is the measure attribute. Let $\mathcal{B}_1 = \{$'Alice', 'Bob'$\}$, $\mathcal{B}_2 = \{$'I','J'$\}$, and $\mathcal{B}_3 = \{$Mar-11$\}$. Then, $\mathcal{B}$ is the set of tuples regarding the purchases by 'Alice' or 'Bob' on 'I' or 'J' on Mar-11, and its mass $M_{\mathcal{B}} = 19$, which is the total units sold by such purchases. Likewise, $M_{\mathcal{B}(\text{user, 'Alice'})} = mass(\mathcal{B}(\text{user, 'Alice'})) = 7$, which is the total units of 'I' or 'J' purchased exactly by 'Alice' on Mar-11. In the tensor representation, $\mathcal{B}$ composes a subtensor in $\mathcal{R}$, as depicted in Figure 10.1(b).

10.3.2 Density Measures

Definition of Density Measures. In the following two chapters, we consider four specific density measures although our algorithms are not restricted to them. Below, we slightly abuse notations to emphasize that the density measures are the functions of $M_{\mathcal{B}}$, $\{|\mathcal{B}_n|\}_{n=1}^N$, $M_{\mathcal{R}}$, and $\{|\mathcal{R}_n|\}_{n=1}^N$, where $\mathcal{B}$ is a subtensor of a relation $\mathcal{R}$.

Arithmetic average mass (Definition 10.1) and *geometric average mass* (Definition 10.2) are multi-dimensional extensions of average degree measures, which have been widely used for dense-subgraph detection. The merits of each average degree measure are discussed in [Cha00, KV99], and extensive research based on them is discussed in Section 10.2.

Definition 10.1: Arithmetic Average Mass [Cha00]

The *arithmetic average mass* of a subtensor $\mathcal{B}$ of a relation $\mathcal{R}$ is defined as

$$\rho_{ari}(\mathcal{B}, \mathcal{R}) := \frac{M_{\mathcal{B}}}{S_{\mathcal{B}}/N},$$

or equivalently,

$$\rho_{ari}(M_{\mathcal{B}}, \{|\mathcal{B}_n|\}_{n=1}^N, M_{\mathcal{R}}, \{|\mathcal{R}_n|\}_{n=1}^N) := \frac{M_{\mathcal{B}}}{\frac{1}{N}\sum_{n=1}^N |\mathcal{B}_n|}.$$

Definition 10.2: Geometric Average Mass [Cha00]

The *geometric average mass* of a subtensor $\mathcal{B}$ of a relation $\mathcal{R}$ is defined as

$$\rho_{geo}(\mathcal{B}, \mathcal{R}) := \frac{M_{\mathcal{B}}}{(V_{\mathcal{B}})^{\frac{1}{N}}},$$

or equivalently,

$$\rho_{geo}(M_{\mathcal{B}}, \{|\mathcal{B}_n|\}_{n=1}^N, M_{\mathcal{R}}, \{|\mathcal{R}_n|\}_{n=1}^N) := \frac{M_{\mathcal{B}}}{(\prod_{n=1}^N |\mathcal{B}_n|)^{\frac{1}{N}}}.$$

Another density measure is *entry surplus* (Definition 10.3), defined as the observed mass of $\mathcal{B}$ subtracted by α times the mass expected under the assumption that the value of each entry (in the tensor representation) in $\mathcal{R}$ is i.i.d. Entry surplus is a multi-dimensional extension of *edge surplus*, defined in graphs [TBG+13]. Subtensors with high entry surplus are configurable by adjusting α. With high α values, relatively small compact subtensors have higher entry surplus than large sparse subtensors, while the opposite happens with small α values [SHF18, SHKF18].

Definition 10.3: Entry Surplus [TBG+13]

The *entry surplus* of a subtensor $\mathcal{B}$ of a relation $\mathcal{R}$ is defined as

$$\rho_{es(\alpha)}(\mathcal{B}, \mathcal{R}) := M_{\mathcal{B}} - \alpha \cdot M_{\mathcal{R}} \cdot (V_{\mathcal{B}}/V_{\mathcal{R}}),$$

or equivalently,

$$\rho_{es(\alpha)}(M_{\mathcal{B}}, \{|\mathcal{B}_n|\}_{n=1}^N, M_{\mathcal{R}}, \{|\mathcal{R}_n|\}_{n=1}^N) := M_{\mathcal{B}} - \alpha \cdot M_{\mathcal{R}} \cdot \prod_{n=1}^N \frac{|\mathcal{B}_n|}{|\mathcal{R}_n|},$$

where α is a constant.

The other density measure is *suspiciousness* (Definition 10.4), defined as the negative log likelihood that a subtensor with the same volume of $\mathcal{B}$ has mass $M_{\mathcal{B}}$ under the assumption that the value of each entry (in the tensor representation) of $\mathcal{R}$ is i.i.d. from a Poisson Distribution.

Definition 10.4: Suspiciousness [JBC+16]

The *suspiciousness* of a subtensor $\mathcal{B}$ of a relation $\mathcal{R}$ is defined as

$$\begin{aligned}\rho_{susp}(\mathcal{B},\mathcal{R}) &:= V_{\mathcal{R}} \cdot D_{KL}\left(\frac{M_{\mathcal{B}}}{V_{\mathcal{B}}}\middle\|\frac{M_{\mathcal{R}}}{V_{\mathcal{R}}}\right)\\ &= M_{\mathcal{B}}\cdot\left(\log\frac{M_{\mathcal{B}}}{M_{\mathcal{R}}}-1\right)+M_{\mathcal{R}}\cdot\frac{V_{\mathcal{B}}}{V_{\mathcal{R}}}-M_{\mathcal{B}}\cdot\log\frac{V_{\mathcal{B}}}{V_{\mathcal{R}}},\end{aligned}$$

where D_{KL} is the Kullback-Leibler divergence, or equivalently,

$$\begin{aligned}&\rho_{susp}(M_{\mathcal{B}},\{|\mathcal{B}_n|\}_{n=1}^N, M_{\mathcal{R}}, \{|\mathcal{R}_n|\}_{n=1}^N)\\ &\quad:= M_{\mathcal{B}}\cdot\left(\log\frac{M_{\mathcal{B}}}{M_{\mathcal{R}}}-1\right)+M_{\mathcal{R}}\cdot\prod_{n=1}^N\frac{|\mathcal{B}_n|}{|\mathcal{R}_n|}-M_{\mathcal{B}}\cdot\log\left(\prod_{n=1}^N\frac{|\mathcal{B}_n|}{|\mathcal{R}_n|}\right).\end{aligned}$$

Our methods, discussed in the following two chapters, however, are not restricted to the four measures mentioned above. Our methods, which search for dense subtensors, allow for any density measure ρ that satisfies Axiom 10.1, which any reasonable density measure should satisfy.

Axiom 10.1: Density Axiom

If two subtensors of a relation have the same cardinality for every dimension attribute, the subtensor with higher or equal mass is at least as dense as the other one. Formally,

$$|\mathcal{B}_n| = |\mathcal{B}'_n|, \forall n \in [N] \text{ and } M_{\mathcal{B}} \geq M_{\mathcal{B}'} \Rightarrow \rho(\mathcal{B},\mathcal{R}) \geq \rho(\mathcal{B}',\mathcal{R}).$$

Maximization of Density Measures. Finding the densest subtensors that maximize the above measures is computationally expensive. Even when we restrict our attention to the simplest case where we aim to find a single densest submatrix maximizing arithmetic average mass (i.e., ρ_{ari}) from a binary matrix takes $O(S_{\mathcal{R}}^6)$ time [Gol84].

10.4 Datasets

The real-world tensors used in the following chapters are categorized into four groups: (a) Rating data (AppStore, Yelp, Android, Netflix, and YahooM.), (b) Wikipedia revision histories (KoWiki and EnWiki), (c) 'Like' histories (StackO.), (d) Temporal social networks (Youtube and SMS), and (e) TCP dumps (Darpa and AirForce).

Rating data. Rating data are relations with schema (user, item, timestamp, score, #ratings). Each tuple (u,i,t,s,r) indicates that user u gave item i score s, r times, at timestamp t. In the AppStore dataset [ACF13], the timestamps are in dates, and the items are entertaining software from App Store, an online software market. In the Yelp dataset [1], the timestamps are in dates, and the items are businesses listed on Yelp, a review site. In the Android dataset [MPL15], the timestamps are hours, and the items are Android apps on Amazon, an online store. In the Netflix dataset [BL+07], the timestamps are in dates, and the items are movies listed on Netflix, a movie rental and streaming service. In the YahooM. dataset [DKKW12], the timestamps are in hours, and the items are musical items listed on Yahoo! Music, a provider of various music services.

Wikipedia revision histories. Wikipedia revision histories are relations with schema (user, page, timestamp, #revisions). Each tuple (u,p,t,r) indicates that user u revised page p, r times, at timestamp t (in hour) in Wikipedia, a crowd-sourcing online encyclopedia. In the KoWiki dataset, the pages are from Korean Wikipedia. In the EnWiki dataset, the pages are from English Wikipedia.

'Like' histories. 'Like' histories are relations with schema (user, posting, timestamp, #revisions). Each tuple (u,p,t,1) indicates that user u marked posting p as 'favorite', at timestamp t (in hour). In the StackO. dataset, the postings are from Stack Overflow, a question-and-answer site on topics in computer programming.

Temporal social networks. Temporal social networks are relations with schema (source, destination, timestamp, #interactions). Each tuple (s,d,t,i) indicates that user s interacts with user d, i times, at timestamp t. In the Youtube dataset [MMG+07], the timestamps are in hours, and the interactions are becoming friends on Youtube, a video-sharing website. In the SMS dataset, the timestamps are in hours, and the interactions are sending text messages.

TCP Dumps. The Darpa dataset [LFG+00], collected by the Cyber Systems and Technology Group in 1998, is a relation with schema (source IP, destination IP, timestamp, #connections). Each tuple (s,d,t,c) indicates that c connections were made from IP s to IP d at timestamp t (in minutes). The AirForce dataset, used for KDD Cup 1999 [2], is a relation with schema (protocol, service, src bytes, dst bytes, flag, host count, srv count, #connections) with the following attributes:

- protocol: type of protocol (tcp, udp, etc.).
- service: service on destination (http, telnet, etc.).
- src bytes: bytes sent from source to destination.
- dst bytes: bytes sent from destination to source.
- flag: normal or error status.
- host count: number of connections made to the same host in the past two seconds.
- srv count: number of connections made to the same service in the past two seconds.
- #connections: number of connections with the given dimension attribute values.

[1] https://www.yelp.com/dataset_challenge
[2] http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

Chapter based on work that appeared at ECML/PKDD 2016 [SHF16] and the TKDD journal [SHF18].

Chapter 11

Detecting Dense Subtensors in Large Tensors (1): In-memory Algorithm

Given multi-aspect data (e.g., restaurant reviews with side information) modeled as a tensor, how can we detect dense subtensors in it? Can we spot them in near-linear time but with approximation guarantees?

As discussed in the previous chapter, extensive previous work has shown that unusually dense subtensors in a wide range of real-world tensors tend to indicate anomalous or fraudulent behavior, such as lockstep behavior in social networks. However, available algorithms for detecting dense subtensors are not satisfactory in terms of speed, accuracy, and theoretical guarantees.

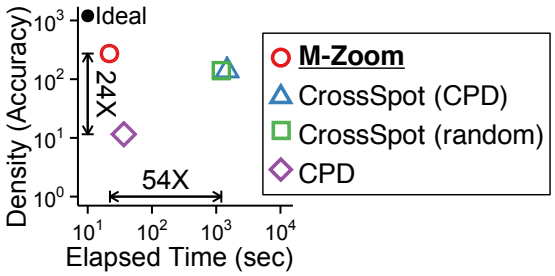
In this chapter, we propose M-ZOOM, a fast and accurate algorithm for dense-subtensor detection. M-ZOOM provides the following advantages: *(1) Fast*: scales near-linearly with all aspects of the input tensor and are up to **114× faster** than state-of-the-art methods with similar accuracy, *(2) Provably accurate*: provides a guarantee on the lowest density of the subtensors it finds, *(3) Effective*: successfully detected edit wars and bot activities on Wikipedia, and spotted network attacks from a TCP dump with near-perfect accuracy (**AUC=0.98**).

11.1 Motivation

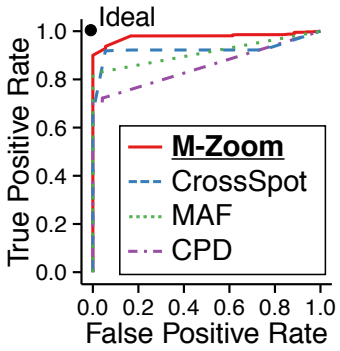
Given reviews on Amazon, how can we detect a group of fraudulent users who review the same set of products with the exact same ratings within a short period of time? Can we spot them in time near linear in the number of reviews, with accuracy guarantees?

As discussed in the previous chapter, modeling data as a tensor and detecting unusually dense subtensors in it has proven an effective way to identify synchronized behavior, which anomalies tend to exhibit. However, neither existing algorithms for dense-subtensor detection nor simple extensions of algorithms for dense-subgraph detection are satisfactory in terms of speed, accuracy, and theoretical guarantees.

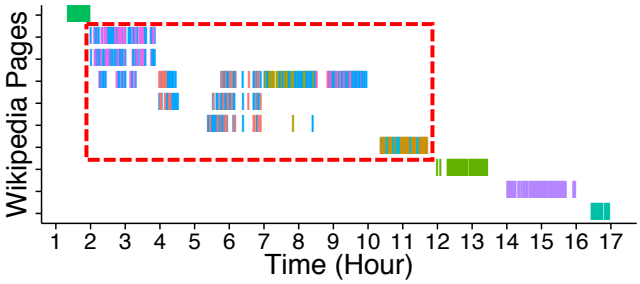
In this second chapter on dense-subtensor detection, we propose M-ZOOM (**M**ultidimensional **Zoom**), which is a fast, accurate, and theoretically sound algorithm for detecting dense subtensors. M-ZOOM starts from the input tensor and greedily removes the slices. Then, it returns the densest subtensor among those encountered, as an output, and removes the subtensor from the input tensor. M-ZOOM repeats these two steps for detecting multiple distinct dense tensors. Despite its simplicity, M-ZOOM has the following advantages over the previous best algorithms:



(a) Fast and Accurate



(b) Effective: Network Intrusion Detection (ROC Curve)



(c) Effective: Edit Wars on Korean Wikipedia

Users	Pages
HBC AIV helperbot3	DeltaQuad/UAA/Time
HBC AIV helperbot5	DeltaQuad/UAA/Wait
HBC AIV helperbot7	Open proxy detection
DeltaQuadBot	WikiProject Spam/LinkReports
COIBot	COIBot/LinkReports
West.andrew.g	West.andrew.g/Dead Links
RonaldB	Cyda/List of current proposed deletion
Cydebot	Cyda/List of requests for unblock

(d) Effective: Bot Activities in English Wikipedia

Figure 11.1: **Strengths of M-ZOOM.** (a) M-ZOOM was up to $54\times$ **faster** with denser subtensors than the previous best algorithms with similar accuracy. (b) M-ZOOM identified network attacks with near-perfect accuracy **(AUC= 0.98)**. (c) M-ZOOM spotted edit wars, during which many users (distinguished by colors) edited the same set of pages hundreds of times within several hours. (d) M-ZOOM spotted bots, and pages edited hundreds of thousands of times by the bots.

- **Fast:** M-ZOOM is up to *114× faster* than its best competitors with similar accuracy (Figure 11.1(a)) thanks to its near-linear scalability with all aspects of the input tensor (Figure 11.4).
- **Provably accurate:** M-ZOOM provides a guarantee on the lowest density of subtensors it finds (Theorem 11.1). In addition, it shows high accuracy similar to its best competitors, in real-world datasets (Figure 11.1(a)).
- **Effective:** M-ZOOM successfully spotted edit wars and bot activities on Wikipedia, and detected network attacks with near-perfect accuracy (*AUC=0.98*) from a TCP dump (Figures 11.1(b-d)).

Reproducibility: The source code and datasets used in this chapter are available at `http://www.cs.cmu.edu/~kijungs/codes/mzoom/`.

The rest of this chapter is organized as follows. In Section 11.2, we give a formal problem definition. In Section 11.3, we present M-ZOOM, our proposed algorithm for dense-subtensor detection. In Section 11.4, we theoretically analyze the accuracy and complexity of M-ZOOM. After sharing some experimental results in Section 11.5, we provide a summary of this chapter in Section 11.6.

Table 11.1: **Table of frequently-used symbols.**

Symbol	Definition
Notations for Tensors Represented as Relations (Defined in Section 10.3.1)	
$\mathcal{R}(A_1, A_2, ..., A_N, X)$	a relation with N dimension attributes and a measure attribute
N	number of dimension attributes in a relation
A_n	n-th dimension attribute in $\mathcal{R}$
X	measure attribute in $\mathcal{R}$
$t[A_n]$ (or $t[X]$)	value of attribute A_n (or X) in tuple t
$\mathcal{B}$	a subtensor in $\mathcal{R}$
$\mathcal{R}_n$ (or $\mathcal{B}_n$)	set of distinct values of A_n in $\mathcal{R}$ (or $\mathcal{B}$)
$\mathcal{R}(a, n)$	set of tuples with attribute $A_n = a$ in $\mathcal{R}$
$M_{\mathcal{R}}$ (or $mass(\mathcal{R})$)	mass of $\mathcal{R}$
$S_{\mathcal{R}}$ (or $size(\mathcal{R})$)	size of $\mathcal{R}$
$V_{\mathcal{R}}$ (or $volume(\mathcal{R})$)	volume of $\mathcal{R}$
Notations for Density Measures (Defined in Section 10.3.2)	
$\rho(\mathcal{B}, \mathcal{R})$	density of a subtensor $\mathcal{B}$ in $\mathcal{R}$ in terms of a density measure ρ
$\rho_{ari}(\mathcal{B}, \mathcal{R})$	arithmetic average mass of a subtensor $\mathcal{B}$ in $\mathcal{R}$
Other Notations	
k	number of subtensors we aim to find
$[x]$	$\{1, 2..., x\}$

11.2 Problem Definition

Throughout this chapter, we represent tensors as relations, and we use the notations and the density measures defined in Section 10.3. As a reminder, Table 11.1 lists some frequently-used symbols. Using the notations, we formally define the problem of detecting the k densest subtensors in a tensor in Problem 11.1.

Problem 11.1: Detecting the Top-k Densest Subtensors

1. **Given:** a relation (i.e., tensor) $\mathcal{R}$, the number of subtensors k, and a density measure ρ,
2. **Find:** k distinct subtensors of $\mathcal{R}$,
3. **to Maximize:** the densities in terms of ρ.

As discussed in Section 10.3.2, solving Problem 11.1 exactly is computationally infeasible for large datasets. Thus, in this chapter, we focus on designing an approximate algorithm that (a) has near-linear scalability with all aspects of $\mathcal{R}$, (b) provides accuracy guarantees at least for some density measures, and (c) produces meaningful results in real-world tensors.

Algorithm 11.1 Outline of M-ZOOM

Input: (1) $\mathcal{R}$: input relation (i.e., input tensor),
(2) k: number of subtensors,
(3) ρ: density measure.
Output: k dense subtensors
1: $\mathcal{R}^{ori} \leftarrow copy(\mathcal{R})$
2: $results \leftarrow \emptyset$
3: **for** $i \leftarrow 1..k$ **do**
4: $\mathcal{B} \leftarrow$ `find_single_subtensor`$(\mathcal{R}, \rho)$ ▷ see Algorithm 11.2
5: $\mathcal{R} \leftarrow \mathcal{R} - \mathcal{B}$
6: $\mathcal{B}^{ori} \leftarrow \{t \in \mathcal{R}^{ori} : \forall n \in [N], t[A_n] \in \mathcal{B}_n\}$
7: $results \leftarrow results \cup \{\mathcal{B}^{ori}\}$
8: **return** $results$

11.3 Proposed Algorithm: M-ZOOM

In this section, we propose M-ZOOM (**M**ultidimensional **Zoom**), a fast and accurate algorithm for finding dense subtensors. We first present an overview of M-ZOOM in Section 11.3.1 and then describe its details in Section 11.3.2.

11.3.1 Overview

The outline of M-ZOOM is given in Algorithm 11.1. M-ZOOM first copies the given relation $\mathcal{R}$ and assigns it to $\mathcal{R}^{ori}$ (line 1). Then, M-ZOOM finds k dense subtensors one by one from $\mathcal{R}$ (line 4). After finding each subtensor from $\mathcal{R}$, M-ZOOM removes the tuples in the subtensor from $\mathcal{R}$ so that the same subtensor is not found repeatedly (line 5). Due to these changes in $\mathcal{R}$, a subtensor found in $\mathcal{R}$ is not necessarily a subtensor of the original relation $\mathcal{R}^{ori}$. Thus, instead of returning the subtensors found in $\mathcal{R}$, M-ZOOM returns the subtensors of $\mathcal{R}^{ori}$ consisting of the same attribute values with the found subtensors (lines 6-7). This also enables M-ZOOM to find overlapped subtensors, i.e., a tuple can be included in two or more subtensors that M-ZOOM finds.

11.3.2 Detailed Description

We present details of M-ZOOM with a focus on single dense-subtensor detection and efficient implementation.

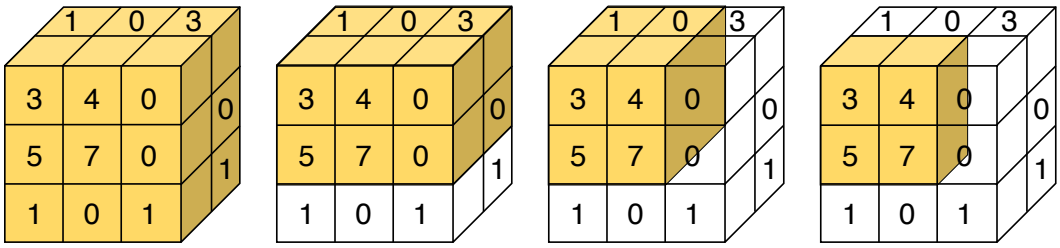
11.3.2.1 Single Dense-subtensor Detection

Algorithm 11.2 and Figure 11.2 describe how M-ZOOM finds a single dense subtensor from the given relation $\mathcal{R}$. The subtensor $\mathcal{B}$ is initialized to $\mathcal{R}$ (lines 1-2). From $\mathcal{B}$, M-ZOOM removes attribute values one by one in a greedy way until no attribute value is left (line 4). Specifically, M-ZOOM finds a dimension $n^- \in [N]$ and a value $a^- \in \mathcal{B}_n$ which are $n \in [N]$ and $a \in \mathcal{B}_n$ maximizing $\rho(\mathcal{B} - \mathcal{B}(a, n), \mathcal{R})$ (i.e., density when all tuples with $A_n = a$ are removed from $\mathcal{B}$) (line 6). Then, the attribute value a^- and the tuples with $A_{n^-} = a^-$ are removed from $\mathcal{B}_{n^-}$ and $\mathcal{B}$, respectively (lines 7-8). Before removing each attribute value, M-ZOOM adds the current $\mathcal{B}$ to the snapshot list (line 5).

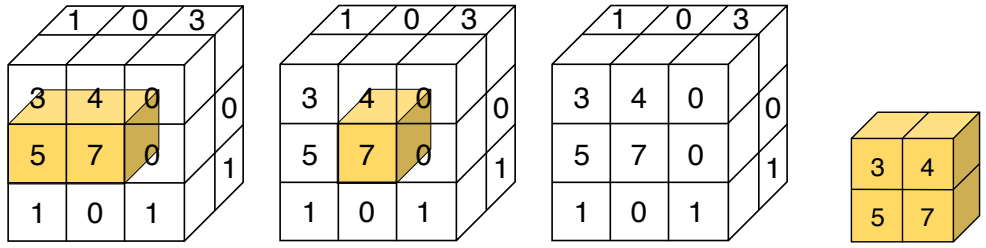
Algorithm 11.2 `find_single_subtensor` in M-ZOOM

Input: (1) $\mathcal{R}$: relation (i.e., tensor) and (2) ρ: density measure:
Output: a dense subtensor

1: $\mathcal{B} \leftarrow copy(\mathcal{R})$
2: $\mathcal{B}_n \leftarrow copy(\mathcal{R}_n), \forall n \in [N]$
3: $snapshots \leftarrow \emptyset$
4: **while** $\exists n \in [N]$ s.t. $\mathcal{B}_n \neq \emptyset$ **do**
5: $\quad snapshots \leftarrow snapshots \cup \{\mathcal{B}\}$
6: $\quad (n^-, a^-) \leftarrow n \in [N]$ and $a \in \mathcal{B}_n$ maximizing $\rho(\mathcal{B} - \mathcal{B}(a, n), \mathcal{R})$ ▷ see Algorithm 11.3
7: $\quad \mathcal{B} \leftarrow \mathcal{B} - \mathcal{B}(a^-, n^-)$
8: $\quad \mathcal{B}_{n^-} \leftarrow \mathcal{B}_{n^-} - \{a^-\}$
9: **return** $\mathcal{B} \in snapshots$ with maximum $\rho(\mathcal{B}, \mathcal{R})$



(a) 1st iteration ($\rho_{ari}(\mathcal{B}, \mathcal{R}) = 3.25$)
(b) 2nd iteration ($\rho_{ari}(\mathcal{B}, \mathcal{R}) = 3.29$)
(c) 3rd iteration ($\rho_{ari}(\mathcal{B}, \mathcal{R}) = 3.33$)
(d) 4th iteration ($\rho_{ari}(\mathcal{B}, \mathcal{R}) = 3.8$)



(e) 5th iteration ($\rho_{ari}(\mathcal{B}, \mathcal{R}) = 3$)
(f) 6th iteration ($\rho_{ari}(\mathcal{B}, \mathcal{R}) = 2.33$)
(g) 7-9th iterations ($\rho_{ari}(\mathcal{B}, \mathcal{R}) = 0$)
(h) Output of Algorithm 11.2

Figure 11.2: **Illustration of Algorithm 11.2** with the tensor $\mathcal{R}$ in Example 10.1. All the invisible entries of $\mathcal{R}$ are zeros. We assume $\rho = \rho_{ari}$. The colored region in each of (a)-(g) shows the subtensor $\mathcal{B}$ added to *snapshots* in each iteration. Note that in each iteration, an attribute value (i.e., a slice in the tensor presentation) is removed from $\mathcal{B}$ so that $\rho_{ari}(\mathcal{B}, \mathcal{R})$ is maximized. (h) shows the output of Algorithm 11.2, the subtensor with maximum $\rho_{ari}(\mathcal{B}, \mathcal{R})$ among those in *snapshots*.

The last step of finding a subtensor is to return the subtensor densest among those in the snapshot list (line 9).

11.3.2.2 Efficient Implementation of M-ZOOM

We discuss efficient implementation of M-ZOOM focusing on greedy attribute-value selection and densest-subtensor selection.

Algorithm 11.3 Greedy Selection Using Min-Heaps in M-ZOOM

Input: (1) $\mathcal{B}$: current subtensor,
(2) ρ: density measure,
(3) $\{H_n^{min}\}_{n=1}^N$: min-heaps
Output: a dimension and an attribute value to be removed

1: **for** each dimension $n \in [N]$ **do**
2: $\quad a_n^- \leftarrow$ attribute value with minimum key in H_n^{min} ▷ key$= M_{\mathcal{B}(a,n)}$
3: $n^- \leftarrow n \in [N]$ maximizing $\rho(\mathcal{B} - \mathcal{B}(a_n^-, n), \mathcal{R})$
4: $a^- \leftarrow a_{n^-}^-$
5: delete a^- from $H_{n^-}^{min}$
6: **for** each tuple $t \in \mathcal{B}(a^-, n^-)$ **do**
7: $\quad$ **for** each dimension $n \in [N] - \{n^-\}$ **do**
8: $\quad\quad$ decrease the key of $t[A_n]$ in H_n^{min} by $t[X]$ ▷ key$= M_{\mathcal{B}(t[A_n],n)}$
9: **return** (n^-, a^-)

Attribute-Value Selection Using Min-Heaps. Finding a dimension $n \in [N]$ and a value $a \in \mathcal{B}_n$ that maximize $\rho(\mathcal{B}-\mathcal{B}(a,n),\mathcal{R})$ (line 6 of Algorithm 11.2) can be computationally expensive if all possible attribute values (i.e., $\{(n,a) : n \in [N], a \in \mathcal{B}_n\}$) should be considered. However, due to Axiom 10.1, which is assumed to be satisfied by considered density measures, the number of candidates is reduced to N if $M_{\mathcal{B}(a,n)}$ is known for each dimension $n \in [N]$ and each attribute value $a \in \mathcal{B}_n$. Lemma 11.1 formalizes this.

Lemma 11.1

If we remove a value of attribute A_n from $\mathcal{B}_n$, removing $a \in \mathcal{B}_n$ with minimum $M_{\mathcal{B}(a,n)}$ results in the highest density. Formally, for each $n \in [N]$,

$$M_{\mathcal{B}(a',n)} \leq M_{\mathcal{B}(a,n)}, \forall a \in \mathcal{B}_n \Rightarrow \rho(\mathcal{B} - \mathcal{B}(a',n), \mathcal{R}) \geq \rho(\mathcal{B} - \mathcal{B}(a,n), \mathcal{R}), \forall a \in \mathcal{B}_n.$$

Proof. Let $\mathcal{B}' = \mathcal{B} - \mathcal{B}(a',n)$ and $\mathcal{B}'' = \mathcal{B} - \mathcal{B}(a,n)$. Then, $|\mathcal{B}'_n| = |\mathcal{B}''_n|, \forall n \in [N]$. In addition, $M_{\mathcal{B}'} \geq M_{\mathcal{B}''}$ since $M_{\mathcal{B}'} = M_{\mathcal{B}} - M_{\mathcal{B}(a',n)} \geq M_{\mathcal{B}} - M_{\mathcal{B}(a,n)} = M_{\mathcal{B}''}$. Hence, by Axiom 10.1, $\rho(\mathcal{B} - \mathcal{B}(a',n), \mathcal{R}) \geq \rho(\mathcal{B} - \mathcal{B}(a,n), \mathcal{R})$. ■

By Lemma 11.1, if we let a_n^- be $a \in \mathcal{B}_n$ with minimum $M_{\mathcal{B}(a,n)}$, we only have to consider the dimension and value pairs in $\{(n, a_n^-) : n \in [N]\}$ instead of $\{(n,a) : n \in [N], a \in \mathcal{B}_n\}$ to find

the attribute value maximizing density when it is removed. To exploit this, our implementation of M-ZOOM maintains a min-heap for each attribute A_n where the key of each value $a \in \mathcal{B}_n$ is $M_{\mathcal{B}(a,n)}$. This key is updated, which takes $O(1)$ time if Fibonacci Heaps are used as min-heaps, whenever the tuples with the corresponding attribute value are removed. Algorithm 11.3 describes in detail how to find the attribute value to be removed based on these min-heaps, and how to update keys in them. Since Algorithm 11.3 considers all promising dimension and value pairs (i.e., $\{(n, a_n^-)\}_{n=1}^N$), it guarantees to find the value that maximizes density when it is removed.

Densest-Subtensor Selection Using Attribute-Value Ordering. As explained in Section 11.3.2.1, M-ZOOM returns the subtensor with maximum density among the snapshots of $\mathcal{B}$ (line 9 of Algorithm 11.2). Explicitly maintaining the list of snapshots, whose length is at most $S_{\mathcal{R}}$, requires $O(N|\mathcal{R}|S_{\mathcal{R}})$ computation and space for copying them. Even maintaining only the current best (i.e., the one with the highest density so far) leads to high computational cost if the current best keeps changing. Instead, our implementation maintains the order by which attribute values are removed as well as the iteration where the density was maximized, which requires only $O(S_{\mathcal{R}})$ space. From these and the original relation $\mathcal{R}$, our implementation restores the snapshot with maximum density in $O(N|\mathcal{R}| + S_{\mathcal{R}})$ time and returns it.

11.4 Theoretical Analysis

In this section, we first prove an approximation guarantee of M-ZOOM. Then, we analyze the time complexity and memory requirements of M-ZOOM.

11.4.1 Accuracy Analysis

We show lower bounds on the densities of the subtensors found by M-ZOOM under the assumption that ρ_{ari} (Definition 10.1) is used as the density measure. Specifically, we show that Algorithm 11.2 is guaranteed to find a subtensor with density at least $1/N$ of the density of the densest subtensor in the given relation (Theorem 11.1). This means that each n-th subtensor returned by Algorithm 11.1 has density at least $1/N$ of the density of the densest subtensor in $\mathcal{R} - \bigcup_{i=1}^{n-1}(i\text{-th subtensor})$.

Let $\mathcal{B}^{(r)}$ be the relation $\mathcal{B}$ at the beginning of the r-th iteration of Algorithm 11.2 with ρ_{ari} as the density measure, and $n^{(r)}$ and $a^{(r)}$ be n^- and a^- in the same iteration. That is, in the r-th iteration, value $a^{(r)} \in \mathcal{B}^{(r)}_{n^{(r)}}$ is removed from attribute $A_{n^{(r)}}$.

Lemma 11.2

$n^{(r)} \in [N]$ and $a^{(r)} \in \mathcal{B}^{(r)}_n$ are $n \in [N]$ and $a \in \mathcal{B}^{(r)}_n$ minimizing $mass(\mathcal{B}^{(r)}(a, n))$.

Proof. By line 6 of Algorithm 11.2, $\rho_{ari}(\mathcal{B}^{(r)} - \mathcal{B}^{(r)}(a^{(r)}, n^{(r)}), \mathcal{R}) \geq \rho_{ari}(\mathcal{B}^{(r)} - \mathcal{B}^{(r)}(a, n), \mathcal{R})$ holds for every $n \in [N]$ and $a \in \mathcal{B}_n^{(r)}$. From this, we have

$$\begin{aligned} mass(\mathcal{B}^{(r)}(a^{(r)}, n^{(r)})) &= mass(\mathcal{B}^{(r)}) - mass(\mathcal{B}^{(r)} - \mathcal{B}^{(r)}(a^{(r)}, n^{(r)})) \\ &= mass(\mathcal{B}^{(r)}) - \rho_{ari}(\mathcal{B}^{(r)} - \mathcal{B}^{(r)}(a^{(r)}, n^{(r)}), \mathcal{R}) \frac{size(\mathcal{B}^{(r)}) - 1}{N} \\ &\leq mass(\mathcal{B}^{(r)}) - \rho_{ari}(\mathcal{B}^{(r)} - \mathcal{B}^{(r)}(a, n), \mathcal{R}) \frac{size(\mathcal{B}^{(r)}) - 1}{N} \\ &= mass(\mathcal{B}^{(r)}) - mass(\mathcal{B}^{(r)} - \mathcal{B}^{(r)}(a, n)) = mass(\mathcal{B}^{(r)}(a, n)). \end{aligned}$$

■

Lemma 11.3

If a subtensor $\mathcal{B}'$ satisfying $M_{\mathcal{B}'(a,n)} \geq c$ for every $n \in [N]$ and every $a \in \mathcal{B}'_n$ exists in $\mathcal{R}$, there exists r satisfying $M_{\mathcal{B}^{(r)}(a,n)} \geq c$ for every $n \in [N]$ and $a \in \mathcal{B}_n^{(r)}$.

Proof. Let r be the first iteration in Algorithm 11.2 where $a^{(r)} \in \mathcal{B}'_{n^{(r)}}$. Then, since $\mathcal{B}^{(r)} \supset \mathcal{B}'$, $mass(\mathcal{B}^{(r)}(a^{(r)}, n^{(r)})) \geq mass(\mathcal{B}'(a^{(r)}, n^{(r)})) \geq c$ holds. By Lemma 11.2, $mass(\mathcal{B}^{(r)}(a, n)) \geq mass(\mathcal{B}^{(r)}(a^{(r)}, n^{(r)})) \geq c$ holds for every $n \in [N]$ and $a \in \mathcal{B}_n^{(r)}$. ■

Theorem 11.1: $1/N$-Approximation Guarantee for Problem 11.1

Given a relation $\mathcal{R}$, let $\mathcal{B}^*$ be the subtensor $\mathcal{B} \subset \mathcal{R}$ with maximum $\rho_{ari}(\mathcal{B}, \mathcal{R})$. Let $\mathcal{B}'$ be the subtensor obtained by Algorithm 11.2. Then,

$$\rho_{ari}(\mathcal{B}', \mathcal{R}) \geq \frac{\rho_{ari}(\mathcal{B}^*, \mathcal{R})}{N}$$

Proof. From the maximality of $\mathcal{B}^*$, $M_{\mathcal{B}^*(a,n)} \geq M_{\mathcal{B}^*}/S_{\mathcal{B}^*}$ holds for every $n \in [N]$ and $a \in \mathcal{B}_n^{(r)}$. Otherwise, a contradiction would result since, for $n \in [N]$ and $a \in \mathcal{B}_n^{(r)}$ where $M_{\mathcal{B}^*(a,n)} < M_{\mathcal{B}^*}/S_{\mathcal{B}^*}$, the following inequality holds:

$$\rho_{ari}(\mathcal{B}^* - \mathcal{B}^*(a, n), \mathcal{R}) = \frac{M_{\mathcal{B}^*} - M_{\mathcal{B}^*(a,n)}}{(S_{\mathcal{B}^*} - 1)/N} > \frac{M_{\mathcal{B}^*} - M_{\mathcal{B}^*}/S_{\mathcal{B}^*}}{(S_{\mathcal{B}^*} - 1)/N} = \rho_{ari}(\mathcal{B}^*, \mathcal{R}).$$

Consider $\mathcal{B}^{(r)}$ where $M_{\mathcal{B}^{(r)}(a,n)} \geq M_{\mathcal{B}^*}/S_{\mathcal{B}^*}$ holds for every $n \in [N]$ and $a \in \mathcal{B}_n^{(r)}$. Such $\mathcal{B}^{(r)}$ exists by Lemma 11.3. Then, $M_{\mathcal{B}^{(r)}} \geq (S_{\mathcal{B}^{(r)}}/N)\ (M_{\mathcal{B}^*}/S_{\mathcal{B}^*}) = (S_{\mathcal{B}^{(r)}}/N)(\rho_{ari}(\mathcal{B}^*, \mathcal{R})/N)$ holds. Hence, $\rho_{ari}(\mathcal{B}', \mathcal{R}) \geq \rho_{ari}(\mathcal{B}^{(r)}, \mathcal{R}) = M_{\mathcal{B}^{(r)}}/(S_{\mathcal{B}^{(r)}}/N) \geq \rho_{ari}(\mathcal{B}^*, \mathcal{R})/N$ holds. ■

11.4.2 Complexity Analysis

The time and space complexities of M-ZOOM depend on the density measure used. We assume that one of the density measures in Section 10.3.2, which satisfy Axiom 10.1, is used.

Theorem 11.2: Time Complexity of M-ZOOM

Let $L = \max_{n\in[N]} |\mathcal{R}_n|$. Then, if $N = O(\log L)$, the time complexity of Algorithm 11.1 is $O(kN|\mathcal{R}| \log L)$.

Proof. In Algorithm 11.3, lines 1-4 take $O(N)$ time for all the considered density measures (i.e., ρ_{ari}, ρ_{geo}, ρ_{susp}, and $\rho_{es(\alpha)}$) if we maintain and update aggregated values (e.g., M_B, S_B, and V_B) instead of computing $\rho(\mathcal{B} - \mathcal{B}(a_n^-, n), \mathcal{R})$ from scratch every time. In addition, line 5 takes $O(\log |\mathcal{R}_{n^-}|)$ time and line 8 takes $O(1)$ time if we use Fibonacci heaps. Algorithm 11.2, whose computational bottleneck is line 6, has time complexity $O(N|\mathcal{R}| + N\sum_{n=1}^{N} |\mathcal{R}_n| + \sum_{n=1}^{N} |\mathcal{R}_n| \log |\mathcal{R}_n|)$ since lines 1-4 of Algorithm 11.3 are executed $S_{\mathcal{R}} = \sum_{n=1}^{N} |\mathcal{R}_n|$ times, line 5 is executed $|\mathcal{R}_{n^-}|$ times for each $n^- \in [N]$, and line 8 is executed $N|\mathcal{R}|$ times. Algorithm 11.1, whose computational bottleneck is line 4, has time complexity $O(kN|\mathcal{R}| + kN\sum_{n=1}^{N} |\mathcal{R}_n| + k\sum_{n=1}^{N} |\mathcal{R}_n| \log |\mathcal{R}_n|)$ since Algorithm 11.2 is executed k times. From $L = \max_{n\in[N]} |\mathcal{R}_n|$, the time complexity of Algorithm 11.1 becomes $O(kN(|\mathcal{R}| + NL + L \log L))$. Since $N = O(\log L)$ (by assumption) and $L \leq |\mathcal{R}|$ (by definition), $|\mathcal{R}| + NL + L \log L = O(|\mathcal{R}| \log L)$. Thus, the time complexity of Algorithm 11.1 is $O(kN|\mathcal{R}| \log L)$. ■

As stated in Theorem 11.2, M-ZOOM scales linearly or sub-linearly with all aspects of relation $\mathcal{R}$ as well as k, the number of subtensors we aim to find. This result is also experimentally supported in Section 11.5.3.

Theorem 11.3: Memory Requirements of M-ZOOM

The amount of memory space required in Algorithm 11.1 is $O(N|\mathcal{R}|)$.

Proof. In Algorithm 11.1, only $\mathcal{B}$, which requires $O(N|\mathcal{R}|)$ space, needs to be loaded into memory at once. The others (i.e., $\mathcal{R}^{ori}$, $\mathcal{R}$, $\mathcal{B}_{ori}$, and $results$) can be read and written sequentially to disk. In Algorithm 11.2, which is called by Algorithm 11.1, $\mathcal{B}$, the min-heaps, and the order by which attribute values are removed need to be loaded into memory at once (see Section 11.3.2.2). They require $O(N|\mathcal{R}|)$, $O(\sum_{n=1}^{N} |\mathcal{R}_n|)$, and $O(\sum_{n=1}^{N} |\mathcal{R}_n|)$ space, respectively. Since $|\mathcal{R}_n| \leq |\mathcal{R}|, \forall n \in [N]$, the sum is $O(N|\mathcal{R}| + \sum_{n=1}^{N} |\mathcal{R}_n|) = O(N|\mathcal{R}|)$. Hence, the memory requirement is $O(N|\mathcal{R}|)$ in total. ■

11.5 Experiments

We review our experiments for answering the following questions:

- **Q1. Speed and Accuracy:** How *fast* and *accurately* does M-ZOOM detect dense subtensors from real-world tensors?
- **Q2. Scalability:** Does M-ZOOM *scale linearly* with all aspects of data?

Table 11.2: **Summary of the real-world tensors used in our experiments.** M: Million, K: Thousand. The underlined attributes are composite primary keys.

Name	Volume	#Tuples
Rating data (user, item, timestamp, rating, #reviews)		
Yelp	552K × 77.1K × 3.80K × 5	2.23M
Android [MPL15]	1.32M × 61.3K × 1.28K × 5	2.64M
Netflix [BL+07]	480K × 17.8K × 2.18K × 5	99.1M
YahooM. [DKKW12]	1.00M × 625K × 84.4K × 101	253M
Wikipedia revision histories (user, page, timestamp, #revisions)		
KoWiki	470K × 1.18M × 101K	11.0M
EnWiki	44.1M × 38.5M × 129K	483M
'Like' histories (user, posting, timestamp, 1)		
StackO. [Kun13]	545K × 96.7K × 1.15K	1.30M
Social networks (user, user, timestamp, #interactions)		
Youtube [MMG+07]	3.22M × 3.22M × 203	18.7M
SMS	1.25M × 7.00M × 4.39K	103M
TCP dumps (protocol, service, src bytes, ⋯, #connections)		
AirForce	3 × 70 × 11 × 7.20K × 21.5K × 512 × 512	648K

- **Q3. Diversity of Subtensors:** Does M-ZOOM detect many *distinct* dense subtensors from real-world tensors?
- **Q4. Effectiveness:** Which *anomalies* or *fraud* does M-ZOOM spot from real-world tensors?

11.5.1 Experimental Settings

Machines: All experiments were conducted on a machine with 2.67 GHz Intel Xeon E7-8837 CPUs and 1TB RAM.

Datasets: We used the real-world tensor datasets listed in Table 11.2. See Section 10.4 for a description of the datasets.

Implementations and Parameter Settings: We compared M-ZOOM with CROSSSPOT [JBC+16], CP Decomposition (CPD) [KB09][1], and MultiAspectForensics (MAF) [MGF11]. Methods only applicable to graphs [HSS+17, PSS+10, SERF18] were excluded from comparison. M-ZOOM and CROSSSPOT[2] were implemented in Java, while Tensor Toolbox [BK07a], which gives the state-of-the-art implementations of tensor decomposition, was used for CPD and MAF. Although CROSSSPOT was originally designed to maximize ρ_{susp}, it can be extended to other density measures. These variants were used

[1] Let $\mathbf{A}^{(1)} \in \mathbb{R}^{|\mathcal{R}_1| \times k}$, $\mathbf{A}^{(2)} \in \mathbb{R}^{|\mathcal{R}_2| \times k}$, ..., $\mathbf{A}^{(N)} \in \mathbb{R}^{|\mathcal{R}_N| \times k}$ be the factor matrices obtained by the rank-k CP Decomposition [KB09] of $\mathcal{R}$. For each $i \in [k]$, we form a subtensor with every attribute value a_n whose corresponding element in the i-th column of $\mathbf{A}^{(n)}$ is at least $1/\sqrt{|\mathcal{R}_n|}$.

[2] We referred the open-sourced implementation at http://github.com/mjiang89/CrossSpot.

depending on the density measure compared in each experiment. In addition, we used CPD as the seed-subtensor selection method of CROSSSPOT, which outperformed HOSVD used in [JBC+16] in terms of both speed and accuracy.

Density Measures: We used the four density measured defined in Section 10.3.2: arithmetic average mass (ρ_{ari}), geometric average mass (ρ_{geo}), suspiciousness (ρ_{susp}), and entry surplus (i.e., $\rho_{es(\alpha)}$). The parameter α in $\rho_{es(\alpha)}$ was set to 1 unless otherwise stated.

11.5.2 Q1. Speed and Accuracy of M-ZOOM

M-ZOOM provides the best trade-off between speed and accuracy. We compared the speed of the considered methods and the densities of the subtensors found by the methods in real-world datasets. Specifically, we measured time taken to find three subtensors and the maximum density among the three subtensors. As seen in 11.3, M-ZOOM clearly provided the best trade-off between speed and accuracy in in most of the other datasets, regardless of density measures. For example, when ρ_{ari} was used as the density measure, M-ZOOM was $114\times$ *faster* than CROSSSPOT, while detecting subtensors with similar densities. Compared with CPD, M-ZOOM detected 2 times denser subtensors $2.8\times$ *faster*. Although the results are not included in Figure 11.3, MAF found several orders of magnitude sparser subtensors than the other methods, with speed similar to that of CPD.

11.5.3 Q2. Scalability of M-ZOOM

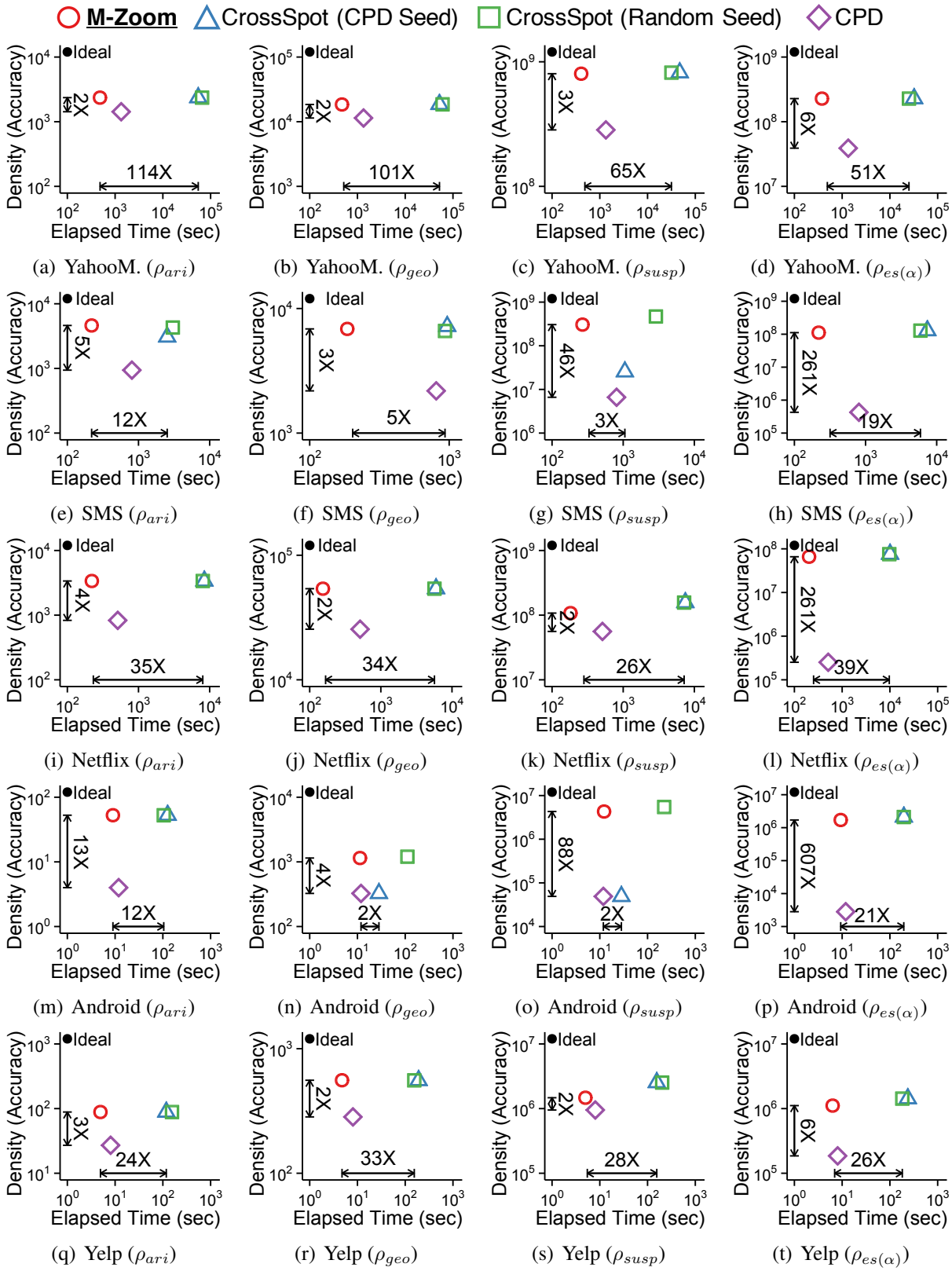
M-ZOOM scales (sub-)linearly with every aspect of the input tensor. We measured the scalability of M-ZOOM with regard to the number of tuples, the number of attributes, the cardinalities of attributes, and the number of subtensors we aim to find. We started with finding one subtensor in a randomly generated 10 millions tuples with three attributes each of whose cardinality is 100 thousands. Then, we measured the running times by changing one factor at a time while fixing the others. As seen in Figure 11.4, M-ZOOM scaled linearly with the number of tuples, the number of attributes, and the number of subtensors we aim to find. Moreover, M-ZOOM scaled sub-linearly with the cardinalities of attributes, as shown mathematically in Theorem 11.2. Although ρ_{susp} was used for the results in Figure 11.4, similar trends were obtained regardless of density measures.

11.5.4 Q3. Diversity of Subtensors Found by M-ZOOM

M-ZOOM successfully detects many distinct dense subtensors. We compared the diversity of dense subtensors found by each method. The ability to detect many distinct dense subtensors is important since distinct subtensors may indicate different anomalies or fraud. We define the diversity as the average dissimilarity between the pairs of subtensors, and the dissimilarity of each pair is defined as

$$dissimilarity(\mathcal{B}, \mathcal{B}') = 1 - \frac{\sum_{n=1}^{N} |\mathcal{B}_n \cap \mathcal{B}'_n|}{\sum_{n=1}^{N} |\mathcal{B}_n \cup \mathcal{B}'_n|}.$$

The average diversity among the three subtensors found by each method is compared in Figure 11.5. In all the datasets, M-ZOOM and CPD successfully detected distinct dense subtensors regardless of the density measures used. CROSSSPOT, however, found the same subtensor repeatedly or subtensors with slight difference, even when it used different seed-subtensor selection methods.



(a) YahooM. (ρ_{ari}) (b) YahooM. (ρ_{geo}) (c) YahooM. (ρ_{susp}) (d) YahooM. ($\rho_{es(\alpha)}$)

(e) SMS (ρ_{ari}) (f) SMS (ρ_{geo}) (g) SMS (ρ_{susp}) (h) SMS ($\rho_{es(\alpha)}$)

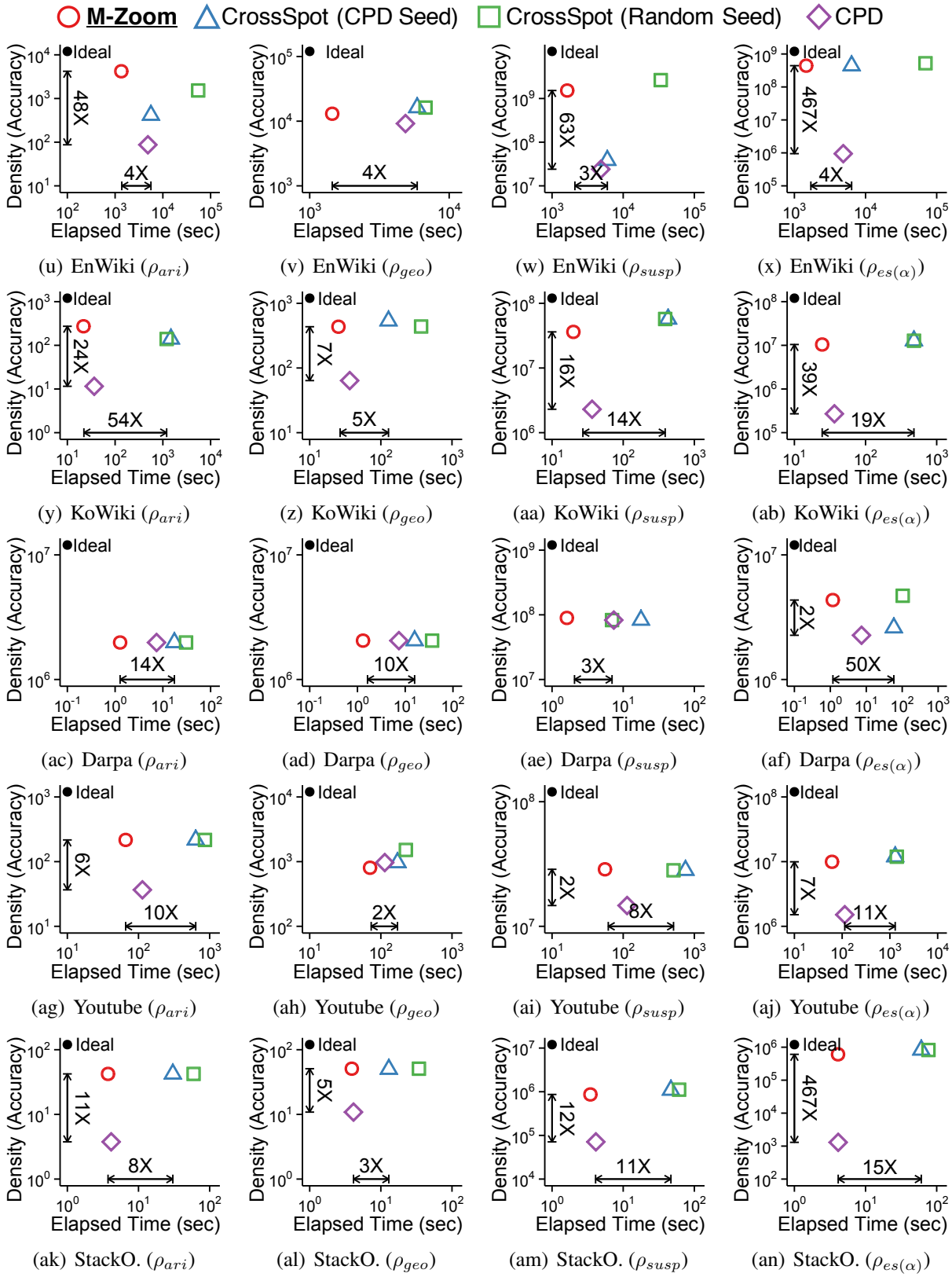
(i) Netflix (ρ_{ari}) (j) Netflix (ρ_{geo}) (k) Netflix (ρ_{susp}) (l) Netflix ($\rho_{es(\alpha)}$)

(m) Android (ρ_{ari}) (n) Android (ρ_{geo}) (o) Android (ρ_{susp}) (p) Android ($\rho_{es(\alpha)}$)

(q) Yelp (ρ_{ari}) (r) Yelp (ρ_{geo}) (s) Yelp (ρ_{susp}) (t) Yelp ($\rho_{es(\alpha)}$)

(Continues on the next page)

(Continues from the previous page)



(u) EnWiki (ρ_{ari}) (v) EnWiki (ρ_{geo}) (w) EnWiki (ρ_{susp}) (x) EnWiki ($\rho_{es(\alpha)}$)

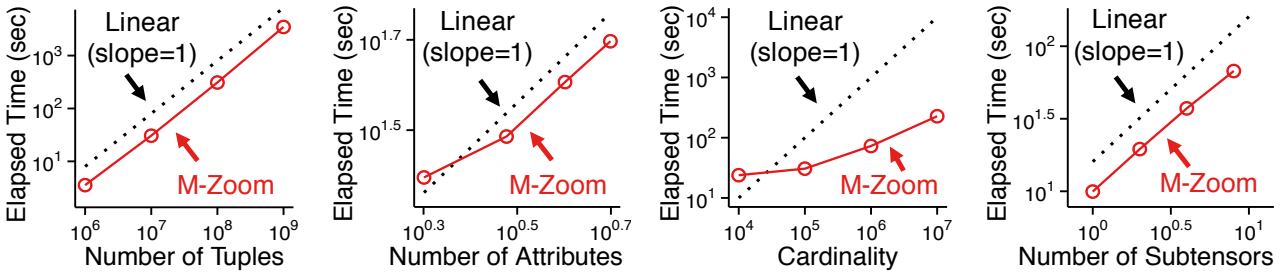
(y) KoWiki (ρ_{ari}) (z) KoWiki (ρ_{geo}) (aa) KoWiki (ρ_{susp}) (ab) KoWiki ($\rho_{es(\alpha)}$)

(ac) Darpa (ρ_{ari}) (ad) Darpa (ρ_{geo}) (ae) Darpa (ρ_{susp}) (af) Darpa ($\rho_{es(\alpha)}$)

(ag) Youtube (ρ_{ari}) (ah) Youtube (ρ_{geo}) (ai) Youtube (ρ_{susp}) (aj) Youtube ($\rho_{es(\alpha)}$)

(ak) StackO. (ρ_{ari}) (al) StackO. (ρ_{geo}) (am) StackO. (ρ_{susp}) (an) StackO. ($\rho_{es(\alpha)}$)

Figure 11.3: **Only M-ZOOM achieves both speed and accuracy.** In each plot, points represent the speed of different methods and the highest density of the three subtensors found by the methods. Upper-left region indicates better performance. M-ZOOM gives the best trade-off between speed and density regardless of used density measures. Specifically, they are up to *114× faster* than CROSSSPOT with similarly dense subtensors.



(a) Scalability w.r.t $|\mathcal{R}|$ (# non-zeros in the input tensor)

(b) Scalability w.r.t N (order of the input tensor)

(c) Scalability w.r.t $|\mathcal{R}_n|$ (dimensionality of the input tensor)

(d) Scalability w.r.t k (# subtensors we aim to find)

Figure 11.4: **M-ZOOM is scalable.** M-ZOOM scales linearly or sub-linearly with the number of tuples, the number of attributes, the cardinalities of attributes, and the number of subtensors we aim to find.

Table 11.3: **M-ZOOM detects bot activities on English Wikipeida.** The table lists the three subtensors detected by M-ZOOM in the EnWiki dataset.

Order	Volume	Mass	Density (ρ_{geo})	Anomaly Type
1	1×1,585×6,733	1.93M	8,772	Bot activities
2	8×12×67.9K	2.43M	13.0K	Bot activities
3	1×1×90	17.6K	3,933	Bot activities

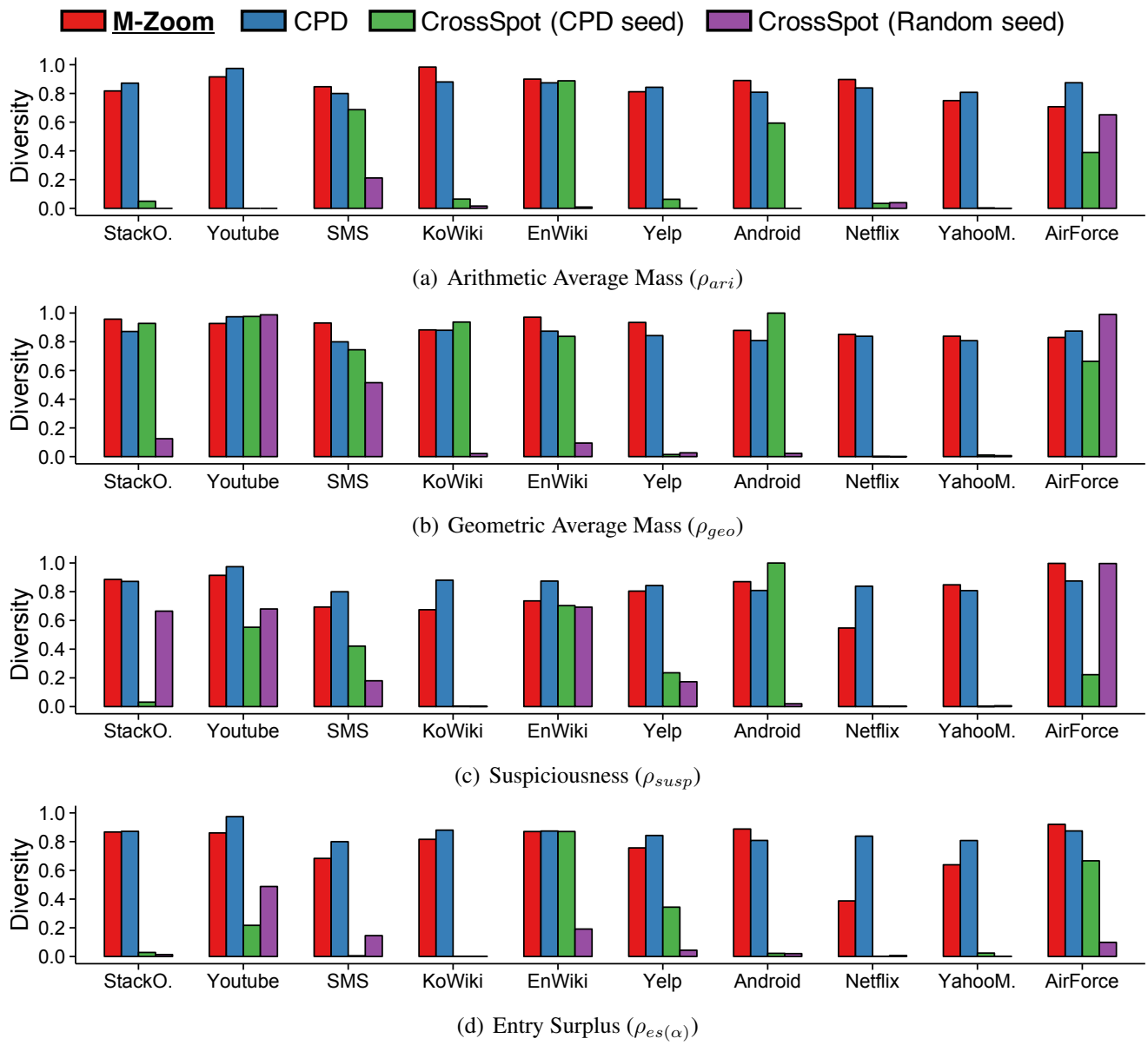
11.5.5 Q4. Effectiveness of M-ZOOM in Real-world Datasets

We demonstrate the effectiveness of M-ZOOM for anomaly and fraud detection by analyzing dense subtensors detected by them in real-world datasets.

M-ZOOM spots bot activities on English Wikipeida. Table 11.4 lists the three dense subtensors found by M-ZOOM in the EnWiki dataset. All the subtensors detected by M-ZOOM indicate the activities of bots, which changed the same pages hundreds of thousands of times. Figure 11.1(d) lists the bots and the pages changed by the bots corresponding to the second subtensor found by M-ZOOM.

M-ZOOM spots edit wars on Korean Wikipeida. Table 11.4 lists the three dense subtensors found by M-ZOOM in the KoWiki dataset. As seen in Figure 11.1(c), which visualizes the third subtensor found by M-ZOOM, all the subtensors detected by M-ZOOM indicate edit wars. That is, users with conflicting opinions revised the same set of pages hundreds of times within several hours.

M-ZOOM spots network intrusions. Table 11.5 lists the five dense subtensors found by M-ZOOM in the AirForce dataset. Based on the provided ground-truth labels, most of the connections composing the subtensors were attacks. This indicates that malicious connections form dense subtensors due to the similarity in their behaviors. Based on this observation, we could accurately separate normal connections and attacks based on the densities of the subtensors they belong (i.e., the denser the densest subtensor including a connection is, the more suspicious the connection is). Especially, we got the highest AUC (Area Under the Curve) 0.98 with M-ZOOM, as shown in Table 11.6 and the ROC curve in Figure 11.1(b). This is since M-ZOOM detects many different dense subtensors accurately, as shown in the previous experiments. For each method, we used the best density measure that led to the highest AUC, which is listed in Table 11.6.



(a) Arithmetic Average Mass (ρ_{ari})

(b) Geometric Average Mass (ρ_{geo})

(c) Suspiciousness (ρ_{susp})

(d) Entry Surplus ($\rho_{es(\alpha)}$)

Figure 11.5: **M-ZOOM detects many distinct dense subtensors.** The dense subtensors found by M-ZOOM and CPD have high diversity, while the dense subtensors found by CROSSSPOT tend to be almost the same, regardless of the density measures used.

Table 11.4: **M-ZOOM detects edit wars on Korean Wikipeida.** The table lists the three subtensors detected by M-ZOOM in the KoWiki dataset.

Order	Volume	Mass	Density (ρ_{ari})	Anomaly Type
1	2×2×2	546	273.0	Edit war
2	2×2×3	574	246.0	Edit war
3	11×10×16	2305	186.9	Edit war

Table 11.5: **M-ZOOM identifies network attacks with near-perfect accuracy.** The five dense subtensors found by M-ZOOM in the AirForce dataset are composed mostly by network attacks.

Order	Volume	Density (ρ_{geo})	# Connections	# Attacks	Attack Type (Ratio)
1	2	2.05M	2.26M	2.26M (100%)	Smurf
2	1	263K	263K	263K (100%)	Smurf
3	8.15K	263K	952K	952K (99.9%)	Neptune
4	1.05M	153K	1.11M	1.06M (95.2%)	Neptune
5	287K	134K	807K	807K (99.9%)	Neptune

Table 11.6: **M-ZOOM identifies network attacks more accurately than its competitors.**

Method	Density Measure	Area Under ROC Curve (AUC)
CPD [KB09]	ρ_{susp}	0.85
MAF [MGF11]	ρ_{susp}	0.91
CrossSpot (CPD Seed) [JBC+16]	ρ_{susp}	0.92
CrossSpot (Random Seed) [JBC+16]	ρ_{geo}	0.93
M-ZOOM [Proposed]	ρ_{geo}	0.98

11.6 Summary

In this chapter, we propose M-ZOOM, which a fast, accurate, and theoretically sound algorithm for dense-subtensor detection. It provides the following advantages over the previous best algorithms:

- **Fast:** M-ZOOM is up to *114× faster* than its competitors with similar accuracy (Figure 11.3) due to its near-linear scalability with all input factors (Figure 11.4).
- **Provably accurate:** M-ZOOM guarantees an approximation bound of the subtensors it finds (Theorems 11.1). In addition, it shows high accuracy in real-world tensors (Figure 11.3).
- **Effective:** M-ZOOM detected network attacks from a TCP dump with near-perfect accuracy (*AUC=0.98*). It also successfully detected edit wars and bot activities on Wikipedia (Tables 11.3-11.6).

Reproducibility: The source code and datasets used in this chapter are available at `http://www.cs.cmu.edu/~kijungs/codes/mzoom/`.

Chapter based on work that appeared at WSDM 2017 [SHKF17a] and its extension [SHKF18].

Chapter 12

Detecting Dense Subtensors in Large Tensors (2): External-memory Algorithm

Given a tensor too large to fit in main memory or even on a disk, how can we detect dense subtensors in it? Can we spot them without sacrificing speed and accuracy of in-memory algorithms?

Continued from the previous chapter, we discuss dense-tensor detection, which has proven useful for identifying various types of anomalies, including 'retweet boosting', network intrusions, and bot accounts. For rapid and accurate detection of dense tensors, various approaches, including tensor decomposition and greedy search, have been developed. However, the existing approaches suffer from low accuracy, or they assume that input tensors are small enough to fit in main memory, which is unrealistic in many real-world applications such as social media and the web.

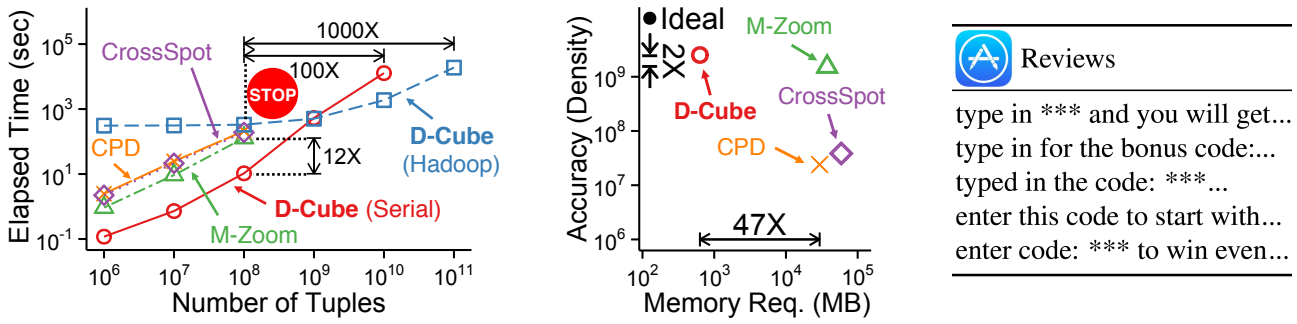
To overcome these limitations, we propose D-CUBE, an external-memory algorithm for dense-subtensor detection. D-CUBE also runs in a distributed manner across multiple machines. Compared to state-of-the-art algorithms, D-CUBE is (1) *Memory Efficient*: requires up to **1,600× less memory** and handles **1,000× larger** data (**2.6TB**), (2) *Fast*: up to **7× faster** due to its near-linear scalability, (3) *Provably Accurate*: gives a guarantee on the densities of the detected subtensors, and (4) *Effective*: spotted network attacks from TCP dumps and synchronized behavior in rating data most accurately.

12.1 Motivation

Given the entire revision histories of all Wikipedia articles, which probably do not fit in main memory or even on a disk, how can we identify suspicious accounts (e.g., bots) and events (e.g., edit wars and vandalism)? How can we utilize the MAPREDUCE framework for rapid identification?

As discussed in the previous chapters, search-based and decomposition-based methods have been developed for dense-subtensor detection, which has proven useful for many anomaly or fraud detection tasks. Search-based methods [JBC+16, SHF18] outperform decomposition-based methods, such as HOSVD and CP Decomposition [MGF11], in terms of accuracy and flexibility with regard to the choice of density metrics. M-ZOOM [SHF18], the latest search method presented in Chapter 11, also provides a guarantee on the densities of the subtensors it finds, while decomposition-based methods do not.

However, the existing search-based methods [JBC+16, SHF18] for dense-subtensor detection assume that the input tensor is small enough to fit in memory. Moreover, they are not directly applicable to tensors stored in disk since using them for such tensors incurs too many disk I/Os due to their



(a) Fast and Scalable (b) Accurate and Memory-Efficient (c) Effective: Spam reviews

Figure 12.1: **Strengths of D-CUBE.** The red stop sign denotes 'out of memory'. (a) *Fast & Scalable*: D-CUBE was **12× faster** and successfully handled **1,000× larger data (2.6TB)** than in-memory algorithms. (b) *Efficient & Accurate*: D-CUBE required **47× less memory** and found denser subtensors than its best competitors from an English Wikipedia revision history. (c) *Effective*: D-CUBE accurately spotted spam reviews on App Store. See Section 12.5 for details.

highly iterative nature. However, real-world applications, such as social media, e-commerce sites, and the World Wide Web, often involve disk-resident tensors with terabytes or even petabytes, which in-memory algorithms cannot handle. This leaves a growing gap that needs to be filled.

In this third chapter on dense-subtensor detection, we propose D-CUBE, a dense-subtensor detection algorithm for disk-resident tensors. D-CUBE works under the W-Stream model [Ruh03], where data are only sequentially read and written during computation. As seen in Table 10.1, only D-CUBE supports out-of-core and distributed computation, which allows it to process data too large to fit in main memory. D-CUBE is optimized for this setting by carefully minimizing the amount of disk I/O and the number of steps requiring disk accesses, without losing accuracy guarantees it provides. Moreover, we present a distributed version of D-CUBE using the MAPREDUCE framework [DG08], specifically its open source implementation HADOOP.[1] The main strengths of D-CUBE are as follows:

- **Memory Efficient:** D-CUBE requires up to *1,600× less memory* and successfully handles *1,000× larger* data (*2.6TB*) than in-memory algorithms (Figure 12.1).
- **Fast:** D-CUBE detects dense subtensors up to *7× faster* in real-world tensors and *12× faster* in synthetic tensors than its best competitors due to its near-linear scalability with all aspects of tensors (Figure 12.1(a)).
- **Provably Accurate:** D-CUBE provides a guarantee on the densities of the subtensors it finds (Theorem 12.1), and it shows similar or higher accuracy than its best competitors on real-world tensors (Figure 12.1(b)).
- **Effective:** D-CUBE successfully spotted many interesting anomalies, including network attacks in TCP dumps and spam reviews in rating data (Figure 12.1(c)).

Reproducibility: The source code and datasets used in this chapter are available at *http://www.cs.cmu.edu/~kijungs/codes/dcube*.

The rest of this chapter is organized as follows. In Section 12.2, we give a formal problem definition. In Section 12.3, we present D-CUBE, an external-memory algorithm for dense-subtensor detection. In Section 12.4, we theoretically analyze the accuracy and complexity of D-CUBE. After sharing some experimental results in Section 12.5, we provide a summary of this chapter in Section 12.6.

[1] http://hadoop.apache.org/

Table 12.1: **Table of frequently-used symbols.**

Symbol	Definition
Notations for Tensors Represented as Relations (Defined in Section 10.3.1)	
$\mathcal{R}(A_1, \cdots, A_N, X)$	relation representing an N-way tensor
N	number of the dimension attributes in $\mathcal{R}$
A_n	n-th dimension attribute in $\mathcal{R}$
X	measure attribute in $\mathcal{R}$
$t[A_n]$ (or $t[X]$)	value of attribute A_n (or X) in tuple t in $\mathcal{R}$
$\mathcal{B}$	a subtensor in $\mathcal{R}$
$\mathcal{R}_n$ (or $\mathcal{B}_n$)	set of distinct values of A_n in $\mathcal{R}$ (or $\mathcal{B}$)
$M_{\mathcal{R}}$ (or $M_{\mathcal{B}}$)	mass of $\mathcal{R}$ (or $\mathcal{B}$)
$\mathcal{B}(a, n)$	set of tuples with attribute $A_n = a$ in $\mathcal{B}$
$M_{\mathcal{B}(a,n)}$	attribute-value mass of a in A_n
Notations for Density Measures (Defined in Section 10.3.2)	
$\rho(\mathcal{B}, \mathcal{R})$	density of a subtensor $\mathcal{B}$ in $\mathcal{R}$ in terms of a density measure ρ
$\rho_{ari}(\mathcal{B}, \mathcal{R})$	arithmetic average mass of a subtensor $\mathcal{B}$ in $\mathcal{R}$
Other Notations	
k	number of subtensors we aim to find
θ	mass-threshold parameter in D-CUBE
$[x]$	$\{1, 2, \cdots, x\}$

12.2 Problem Definition

Throughout this chapter, we represent tensors as relations, and we use the notations and the density measures defined in Section 10.3. As a reminder, Table 12.1 lists some frequently-used symbols. Using the notations, we formally define the problem of detecting the k densest subtensors in a large-scale tensor in Problem 12.1.

Problem 12.1: Detecting the Top-k Densest Subtensors in a Large-scale Tensor

1. **Given:**
   - a large-scale relation (i.e., tensor) $\mathcal{R}$ not fitting in memory,
   - the number of subtensors k,
   - a density measure ρ,
2. **Find:** k distinct subtensors of $\mathcal{R}$,
3. **to Maximize:** the densities in terms of ρ.

As discussed in Section 10.3.2, even when the input tensor fits in main memory, solving Problem 12.1 exactly is computationally infeasible for large datasets Thus, our focus in this chapter is to design an approximate algorithm with (a) near-linear scalability with all aspects of $\mathcal{R}$, which does not fit in

memory, (b) an approximation guarantee at least for some density measures, and (c) meaningful results on real-world data.

12.3 Proposed Algorithm: D-CUBE

In this section, we propose D-CUBE, a disk-based dense-subtensor detection method. We present an overview of D-CUBE in Section 12.3.1 and describe its details in Section 12.3.2. Then, we present our MAPREDUCE implementation of D-CUBE in Section 12.3.3. Throughout these subsections, we assume that the entries of tensors (i.e., the tuples of relations) are stored on disk and read/written only in a sequential way. However, all other data (e.g., distinct attribute-value sets and the mass of each attribute value) are assumed to be stored in memory.

12.3.1 Overview

D-CUBE is a search method that starts with the given relation and removes attribute values (and the tuples with the attribute values) sequentially so that a dense subtensor is left. Contrary to previous approaches, D-CUBE removes multiple attribute values (and the tuples with the attribute values) at a time to reduce the number of iterations and also disk I/Os. In addition to this advantage, D-CUBE carefully chooses attribute values to remove to give the same accuracy guarantee as if attribute values were removed one by one, and shows similar or even higher accuracy empirically.

Algorithm 12.1 D-CUBE: Disk-based Dense-subtensor Detection

Input: (1) $\mathcal{R}$: input relation (i.e., input tensor), (2) k: number of subtensors,
(3) ρ: density measure, (4) $\theta(\geq 1)$: threshold.
Output: k dense subtensors

1: $\mathcal{R}^{ori} \leftarrow copy(\mathcal{R})$
2: compute $\{\mathcal{R}_n\}_{n=1}^{N}$
3: $results \leftarrow \emptyset$ ▷ list of dense subtensors
4: **for** $i \leftarrow 1..k$ **do**
5: $M_{\mathcal{R}} \leftarrow \sum_{t \in \mathcal{R}} t[X]$
6: $\{\mathcal{B}_n\}_{n=1}^{N} \leftarrow$ `find_one`$(\mathcal{R}, \{\mathcal{R}_n\}_{n=1}^{N}, M_{\mathcal{R}}, \rho(,)\theta)$ ▷ see Algorithm 12.2
7: $\mathcal{R} \leftarrow \{t \in \mathcal{R} : \exists n \in [N], t[A_n] \notin \mathcal{B}_n\}$ ▷ $\mathcal{R} \leftarrow \mathcal{R} - \mathcal{B}$
8: $\mathcal{B}^{ori} \leftarrow \{t \in \mathcal{R}^{ori} : \forall n \in [N], t[A_n] \in \mathcal{B}_n\}$
9: $results \leftarrow results \cup \{\mathcal{B}^{ori}\}$
10: **return** $results$

Algorithm 12.1 describes the overall structure of D-CUBE. It first copies and assigns the given relation $\mathcal{R}$ to $\mathcal{R}^{ori}$ (line 1); and computes the sets of distinct attribute values composing $\mathcal{R}$ (line 2). Then, it finds k dense subtensors one by one from $\mathcal{R}$ (line 6) using its mass as a parameter (line 5). The detailed procedure for detecting a single dense subtensor from $\mathcal{R}$ is explained in Section 12.3.2.1. After each subtensor $\mathcal{B}$ is found, the tuples included in $\mathcal{B}$ are removed from $\mathcal{R}$ (line 7) to prevent the same subtensor from being found again. Due to this change in $\mathcal{R}$, subtensors found from $\mathcal{R}$ are not necessarily the subtensors of the original relation $\mathcal{R}^{ori}$. Thus, instead of $\mathcal{B}$, the subtensor in $\mathcal{R}^{ori}$ formed by the same attribute values forming $\mathcal{B}$ is added to the list of k dense subtensors (lines 8-9). Notice that, due to this step, D-CUBE can detect overlapping dense subtensors. That is, a tuple can be included in multiple dense subtensors.

Based on our assumption that the sets of distinct attribute values (i.e., $\{\mathcal{R}_n\}_{n=1}^N$ and $\{\mathcal{B}_n\}_{n=1}^N$) are stored in memory and can be randomly accessed, all the steps in Algorithm 12.1 can be performed by sequentially reading and writing tuples in relations (i.e., tensor entries) in disk without loading all the tuples in memory at once. For example, the filtering steps in lines 7-8 can be performed by sequentially reading each tuple from disk and writing the tuple to disk only if it satisfies the given condition.

Note that this overall structure of D-CUBE is similar to that of M-ZOOM (Chapter 11) except that tuples are stored on disk. However, the methods differ significantly in the way each dense subtensor is found from $\mathcal{R}$, which is explained in the following subsection.

Algorithm 12.2 `find_one` in D-CUBE

Input: (1) $\mathcal{R}$: input relation (i.e., input tensor), (2) $\{\mathcal{R}_n\}_{n=1}^N$: attribute-value sets,
(3) $M_{\mathcal{R}}$: mass, (4) ρ: density measure, (5) $\theta(\geq 1)$: threshold.
Output: attribute values forming a dense subtensor

1: $\mathcal{B} \leftarrow copy(\mathcal{R})$, $M_{\mathcal{B}} \leftarrow M_{\mathcal{R}}$ ▷ initialize the subtensor $\mathcal{B}$
2: $\mathcal{B}_n \leftarrow copy(\mathcal{R}_n), \forall n \in [N]$
3: $\tilde{\rho} \leftarrow \rho(M_{\mathcal{B}}, \{|\mathcal{B}_n|\}_{n=1}^N, M_{\mathcal{R}}, \{|\mathcal{R}_n|\}_{n=1}^N)$ ▷ $\tilde{\rho}$: max ρ so far
4: $r, \tilde{r} \leftarrow 1$ ▷ r: current order of attribute values, $\tilde{r}$: r with $\tilde{\rho}$
5: **while** $\exists n \in [N], \mathcal{B}_n \neq \emptyset$ ▷ until all values are removed **do**
6: compute $\{\{M_{\mathcal{B}(a,n)}\}_{a \in \mathcal{B}_n}\}_{n=1}^N$
7: $i \leftarrow select\_dimension()$ ▷ see Algorithms 12.3 and 12.4
8: $D_i \leftarrow \{a \in \mathcal{B}_i : M_{\mathcal{B}(a,i)} \leq \theta \frac{M_{\mathcal{B}}}{|\mathcal{B}_i|}\}$ ▷ D_i: set to be removed
9: sort D_i in an increasing order of $M_{\mathcal{B}(a,i)}$
10: **for each** value $a \in D_i$ **do**
11: $\mathcal{B}_i \leftarrow \mathcal{B}_i - \{a\}$, $M_{\mathcal{B}} \leftarrow M_{\mathcal{B}} - M_{\mathcal{B}(a,i)}$
12: $\rho' \leftarrow \rho(M_{\mathcal{B}}, \{|\mathcal{B}_n|\}_{n=1}^N, M_{\mathcal{R}}, \{|\mathcal{R}_n|\}_{n=1}^N)$ ▷ ρ' : ρ when a is removed
13: $order(a, i) \leftarrow r$, $r \leftarrow r + 1$
14: **if** $\rho' > \tilde{\rho}$ **then**
15: $\tilde{\rho} \leftarrow \rho'$, $\tilde{r} \leftarrow r$ ▷ update max ρ so far
16: $\mathcal{B} \leftarrow \{t \in \mathcal{B} : t[A_i] \notin D_i\}$ ▷ remove tuples
17: $\tilde{\mathcal{B}}_n \leftarrow \{a \in \mathcal{R}_n : order(a, n) \geq \tilde{r}\}, \forall n \in [N]$ ▷ reconstruct
18: **return** $\{\tilde{\mathcal{B}}_n\}_{n=1}^N$

12.3.2 Detailed Description

We present details of D-CUBE with a focus on single-subtensor detection, dimension selection, and efficient implementation.

12.3.2.1 Single Dense-subtensor Detection (Algorithm 12.2)

Algorithm 12.2 describes how D-CUBE detects each dense subtensor from the given relation $\mathcal{R}$. It first initializes a subtensor $\mathcal{B}$ to $\mathcal{R}$ (lines 1-2) then repeatedly removes attribute values and the tuples of $\mathcal{B}$ with those attribute values until all values are removed (line 5).

Specifically, in each iteration, D-CUBE first chooses a dimension attribute A_i that attribute values are removed from (line 7), Then, it computes D_i, the set of attribute values whose masses are less than $\theta(\geq 1)$ times the average (line 8). We explain how the dimension attribute is chosen, in Section 12.3.2.2

and analyze the effects of θ on the accuracy and the time complexity, in Section 12.4.2. The tuples whose attribute values of A_i are in D_i are removed from $\mathcal{B}$ at once within a single scan of $\mathcal{B}$ (line 16). However, deleting a subset of D_i may achieve higher value of the metric ρ. Hence, D-CUBE computes the changes in the density of $\mathcal{B}$ (line 11) as if the attribute values in D_i were removed one by one, in an increasing order of their masses. This allows D-CUBE to optimize ρ as if we removed attributes one by one, while still benefiting from the computational speedup of removing multiple attributes in each scan. Note that these changes in ρ can be computed exactly without actually removing the tuples from $\mathcal{B}$ or even accessing the tuples in $\mathcal{B}$ since its mass (i.e., $M_{\mathcal{B}}$) and the number of distinct attribute values (i.e., $\{|\mathcal{B}_n|\}_{n=1}^N$) are maintained up-to-date (lines 11-12). This is because removing an attribute value from a dimension attribute does not affect the masses of the other values of the same attribute. The orders that attribute values are removed and when the density of $\mathcal{B}$ is maximized are maintained (lines 13-15) so that the subtensor $\mathcal{B}$ maximizing the density can be restored and returned (lines 17-18), as the result of Algorithm 12.2.

Note that, in each iteration (lines 5-16) of Algorithm 12.2, the tuples of $\mathcal{B}$, which are stored on disk, need to be scanned only twice, once in line 6 and once in line 16. Moreover, both steps can be performed by simply sequentially reading and/or writing tuples in $\mathcal{B}$ without loading all the tuples in memory at once. For example, to compute attribute-value masses in line 6, D-CUBE increases $M_{\mathcal{B}(t[A_n],n)}$ by $t[X]$ for each dimension attribute A_n after reading each tuple t in $\mathcal{B}$ sequentially from disk.

Algorithm 12.3 `select_dimension` by cardinality

Input: $\{\mathcal{B}_n\}_{n=1}^N$: attribute-value sets
Output: a dimension in $[N]$
1: **return** $n \in [N]$ with maximum $|\mathcal{B}_n|$

Algorithm 12.4 `select_dimension` by density

Input: (1) $\{\mathcal{B}_n\}_{n=1}^N$ and $\{\mathcal{R}_n\}_{n=1}^N$ attribute-value sets,
(2) $\{\{M_{\mathcal{B}(a,n)}\}_{a\in\mathcal{B}_n}\}_{n=1}^N$: attribute-value masses,
(3) $M_{\mathcal{B}}$ and $M_{\mathcal{R}}$: masses, (4) ρ: density measure, and (5) $\theta(\geq 1)$: threshold
Output: a dimension in $[N]$
1: $\tilde{\rho} \leftarrow -\infty, \tilde{i} \leftarrow 1$ ▷ $\tilde{\rho}$: max ρ so far, $\tilde{i}$: dimension with $\tilde{\rho}$
2: **for each** dimension $i \in [N]$ **do**
3: **if** $\mathcal{B}_i \neq \emptyset$ **then**
4: $D_i \leftarrow \{a \in \mathcal{B}_i : M_{\mathcal{B}(a,i)} \leq \theta \frac{M_{\mathcal{B}}}{|\mathcal{B}_i|}\}$ ▷ D_i : set to be removed
5: $M'_{\mathcal{B}} \leftarrow M_{\mathcal{B}} - \sum_{a\in D_i} M_{\mathcal{B}(a,i)}$
6: $\mathcal{B}'_i \leftarrow \mathcal{B}_i - D_i$
7: $\rho' \leftarrow \rho(M'_{\mathcal{B}}, \{|\mathcal{B}_n|\}_{n\neq i} \cup \{|\mathcal{B}'_i|\}, M_{\mathcal{R}}, \{|\mathcal{R}_n|\}_{n=1}^N)$ ▷ ρ' : ρ when D_i are removed
8: **if** $\rho' > \tilde{\rho}$ **then**
9: $\tilde{\rho} \leftarrow \rho', \tilde{i} \leftarrow i$ ▷ update max ρ so far
10: **return** $\tilde{i}$

12.3.2.2 Dimension Selection (Algorithms 12.3 and 12.4)

We discuss two policies for choosing a dimension attribute that attribute values are removed from. They are used in line 7 of Algorithm 12.2 offering different advantages.

Maximum cardinality policy (Algorithm 12.3): The dimension attribute with the largest cardinality is chosen, as described in Algorithm 12.3. This simple policy, however, provides an accuracy guarantee (see Theorem 12.1 in Section 12.4.1).

Maximum density policy (Algorithm 12.4): The density of $\mathcal{B}$ when attribute values are removed from each dimension attribute is computed. Then, the dimension attribute leading to the highest density is chosen. Note that the tuples in $\mathcal{B}$, stored on disk, do not need to be accessed for this computation, as described in Algorithm 12.4. Although this policy does not provide the accuracy guarantee given by the maximum cardinality policy, this policy works well with various density measures and tends to spot denser subtensors than the maximum cardinality policy in our experiments with real-world data.

12.3.2.3 Efficient Implementation

We present the optimization techniques used for the efficient implementation of D-CUBE.

Combining disk-accessing steps. The amount of disk I/O can be reduced by combining multiple steps involving disk accesses. In Algorithm 12.1, updating $\mathcal{R}$ (line 7) in an iteration can be combined with computing the mass of $\mathcal{R}$ (line 5) in the next iteration. That is, if we aggregate the values of the tuples of $\mathcal{R}$ while they are written for the update, we do not need to scan $\mathcal{R}$ again for computing its mass in the next iteration. Likewise, in Algorithm 12.2, updating $\mathcal{B}$ (line 16) in an iteration can be combined with computing attribute-value masses (line 6) in the next iteration. This optimization reduces the amount of disk I/O in D-CUBE about 30%.

Caching tensor entries in memory. Although we assume that tuples are stored on disk, storing them in memory up to the memory capacity speeds up D-CUBE up to 3 times in our experiments (see Section 12.5.4). We cache the tuples in $\mathcal{B}$, which are more frequently accessed than those in $\mathcal{R}$ or $\mathcal{R}^{ori}$, in memory with the highest priority.

12.3.3 MapReduce Implementation

We present our MAPREDUCE implementation of D-CUBE, assuming that tuples in relations are stored in a distributed file system. Specifically, we describe four MAPREDUCE algorithms that cover the steps of D-CUBE accessing tuples.

(1) Filtering tuples. In lines 7-8 of Algorithm 12.1 and line 16 of Algorithm 12.2, D-CUBE filters the tuples satisfying the given conditions. These steps are done by the following map-only algorithm, where we broadcast the data used in each condition (e.g., $\{\mathcal{B}_n\}_{n=1}^{N}$ in line 7 of Algorithm 12.1) to mappers using the distributed cache functionality.

- Map-stage: Take a tuple t (i.e., $\langle t[A_1], ..., t[A_N], t[X]\rangle$) and emit t if t satisfies the given condition. Otherwise, the tuple is ignored.

(2) Computing attribute-value masses. Line 6 of Algorithm 12.2 is performed by the following algorithm, where we reduce the amount of shuffled data by combining the intermediate results within each mapper.

- Map-stage: Take a tuple t (i.e., $\langle t[A_1], ..., t[A_N], t[X]\rangle$) and emit N key/value pairs $\{\langle (n, t[A_n]), t[X]\rangle\}_{n=1}^{N}$.
- Combine-stage/Reduce-stage: Take $\langle (n, a)$, values$\rangle$ and emit $\langle (n, a)$, sum(values)$\rangle$.

Each tuple $\langle (n, a)$, value$\rangle$ of the final output indicates that $M_{\mathcal{B}(a,n)}$ =value.

(3) Computing mass. Line 5 of Algorithm 12.1 can be performed by the following algorithm, where we reduce the amount of shuffled data by combining the intermediate results within each mapper.

- Map-stage: Take a tuple t (i.e., $\langle t[A_1], ..., t[A_N], t[X] \rangle$) and emit $\langle 0, t[X] \rangle$.
- Combine-stage/Reduce-stage: Take $\langle 0, \text{values} \rangle$ and emit $\langle 0, \text{sum(values)} \rangle$.

The value of the final tuple corresponds to $M_{\mathcal{R}}$.

(4) Computing attribute-value sets. Line 2 of Algorithm 12.1 can be performed by the following algorithm, where we reduce the amount of shuffled data by combining the intermediate results within each mapper.

- Map-stage: Take a tuple t (i.e., $\langle t[A_1], ..., t[A_N], t[X] \rangle$) and emit N key/value pairs $\{\langle (n, t[A_n]), 0 \rangle\}_{n=1}^{N}$.
- Combine-stage/Reduce-stage: Take $\langle (n, a), \text{values} \rangle$ and emit $\langle (n, a), 0 \rangle$.

Each tuple $\langle (n, a), 0 \rangle$ of the final output indicates that a is a member of $\mathcal{R}_n$.

12.4 Theoretical Analysis

In this section, we first prove an approximation guarantee of D-CUBE. Then, we analyze the time and space complexities of D-CUBE.

12.4.1 Accuracy Analysis

We show that D-CUBE gives the same accuracy guarantee with in-memory algorithms [SHF18], if we set θ to 1, although accesses to tuples (stored on disk) are restricted in D-CUBE to reduce disk I/Os. Specifically, Theorem 12.1 states that the subtensor found by Algorithm 12.2 with the maximum cardinality policy has density at least $\frac{1}{\theta N}$ of the optimum when ρ_{ari} is used as the density measure.

Theorem 12.1: θN-Approximation Guarantee

Let $\mathcal{B}^*$ be the subtensor $\mathcal{B}$ maximizing $\rho_{ari}(\mathcal{B}, \mathcal{R})$ in the given relation $\mathcal{R}$. Let $\tilde{\mathcal{B}}$ be the subtensor returned by Algorithm 12.2 with ρ_{ari} and the maximum cardinality policy. Then,

$$\rho_{ari}(\tilde{\mathcal{B}}, \mathcal{R}) \geq \frac{1}{\theta N} \rho_{ari}(\mathcal{B}^*, \mathcal{R}).$$

Proof. First, the maximal subtensor $\mathcal{B}^*$ satisfies that, for any $i \in [N]$ and for any attribute value $a \in \mathcal{B}_i^*$, its attribute-value mass $M_{\mathcal{B}^*(a,i)}$ is at least $\frac{1}{N}\rho_{ari}(\mathcal{B}^*, \mathcal{R})$. This is since the maximality of $\rho_{ari}(\mathcal{B}^*, \mathcal{R})$ implies $\rho_{ari}(\mathcal{B}^* - \mathcal{B}^*(a,i), \mathcal{R}) \leq \rho_{ari}(\mathcal{B}^*, \mathcal{R})$, and plugging in Definition 10.1 to ρ_{ari} gives $\frac{M_{\mathcal{B}^*} - M_{\mathcal{B}^*(a,i)}}{\frac{1}{N}((\sum_{n=1}^{N} |\mathcal{B}_n^*|) - 1)} = \rho_{ari}(\mathcal{B}^* - \mathcal{B}^*(a,i), \mathcal{R}) \leq \rho_{ari}(\mathcal{B}^*, \mathcal{R}) = \frac{M_{\mathcal{B}^*}}{\frac{1}{N}\sum_{n=1}^{N} |\mathcal{B}_n^*|}$, which reduces to

$$M_{\mathcal{B}^*(a,i)} \geq \frac{1}{N} \rho_{ari}(\mathcal{B}^*, \mathcal{R}). \tag{12.1}$$

Consider the earliest iteration (lines 5-16) in Algorithm 12.2 where an attribute value a of $\mathcal{B}^*$ is included in D_i. Let $\mathcal{B}'$ be $\mathcal{B}$ in the beginning of the iteration. Our goal is to prove $\rho_{ari}(\tilde{\mathcal{B}}, \mathcal{R}) \geq \frac{1}{\theta N}\rho_{ari}(\mathcal{B}^*, \mathcal{R})$, which we will show as $\rho_{ari}(\tilde{\mathcal{B}}, \mathcal{R}) \geq \rho_{ari}(\mathcal{B}', \mathcal{R}) \geq \frac{M_{\mathcal{B}'(a,i)}}{\theta} \geq \frac{M_{\mathcal{B}^*(a,i)}}{\theta} \geq \frac{1}{\theta N}\rho_{ari}(\mathcal{B}^*, \mathcal{R})$.

First, $\rho_{ari}(\tilde{\mathcal{B}}, \mathcal{R}) \geq \rho_{ari}(\mathcal{B}', \mathcal{R})$ is from the maximality of $\rho_{ari}(\tilde{\mathcal{B}}, \mathcal{R})$ among the densities of the subtensors generated in the iterations (lines 13-15 in Algorithm 12.2). Second, applying $|\mathcal{B}'_i| \geq \frac{1}{N}\sum_{n=1}^{N}|\mathcal{B}'_n|$ from the maximum cardinality policy (Algorithm 12.3) to Definition 10.1 of ρ_{ari} gives $\rho_{ari}(\mathcal{B}', \mathcal{R}) = \frac{M_{\mathcal{B}'}}{\frac{1}{N}\sum_{n=1}^{N}|\mathcal{B}'_n|} \geq \frac{M_{\mathcal{B}'}}{|\mathcal{B}'_i|}$. And $a \in D_i$ gives $\theta\frac{M_{\mathcal{B}'}}{|\mathcal{B}'_i|} \geq M_{\mathcal{B}'(a,i)}$. So combining these gives $\rho_{ari}(\mathcal{B}', \mathcal{R}) \geq \frac{M_{\mathcal{B}'(a,i)}}{\theta}$. Third, $\frac{M_{\mathcal{B}'(a,i)}}{\theta} \geq \frac{M_{\mathcal{B}^*(a,i)}}{\theta}$ is from $\mathcal{B}' \supset \mathcal{B}^*$. Fourth, $\frac{M_{\mathcal{B}^*(a,i)}}{\theta} \geq \frac{1}{\theta N}\rho_{ari}(\mathcal{B}^*, \mathcal{R})$ is from Eq. (12.1). Hence, $\rho_{ari}(\tilde{\mathcal{B}}, \mathcal{R}) \geq \frac{1}{\theta N}\rho_{ari}(\mathcal{B}^*, \mathcal{R})$ holds. ■

12.4.2 Complexity Analysis

Theorem 12.2 states the worst-case time complexity, which equals to the worst-case I/O complexity, of D-CUBE.

Lemma 12.1: Maximum Number of Iterations in Algorithm 12.2

Let $L = \max_{n\in[N]}|\mathcal{R}_n|$. Then, the number of iterations (lines 5-16) in Algorithm 12.2 is at most $N\min(\log_\theta L, L)$.

Proof. In each iteration (lines 5-16) of Algorithm 12.2, among the values of the chosen dimension attribute A_i, attribute values whose masses are at most $\theta\frac{M_{\mathcal{B}}}{|\mathcal{B}_i|}$, where $\theta \geq 1$, are removed. The set of such attribute values is denoted by D_i. We will show that, if $|\mathcal{B}_i| > 0$, then

$$|\mathcal{B}_i \backslash D_i| < |\mathcal{B}_i|/\theta. \tag{12.2}$$

Note that, when $|\mathcal{B}_i \backslash D_i| = 0$, Eq. (12.2) trivially holds. When $|\mathcal{B}_i \backslash D_i| > 0$, $M_{\mathcal{B}}$ can be factorized and lower bounded as

$$M_{\mathcal{B}} = \sum\nolimits_{a\in\mathcal{B}_i\backslash D_i} M_{\mathcal{B}(a,i)} + \sum\nolimits_{a\in D_i} M_{\mathcal{B}(a,i)} \geq \sum\nolimits_{a\in\mathcal{B}_i\backslash D_i} M_{\mathcal{B}(a,i)} > |\mathcal{B}_i \backslash D_i| \cdot \theta\frac{M_{\mathcal{B}}}{|\mathcal{B}_i|},$$

where the last strict inequality is from the definition of D_i and that $|\mathcal{B}_i \backslash D_i| > 0$. This strict inequality implies $M_{\mathcal{B}} > 0$, and thus dividing both sides by $\theta\frac{M_{\mathcal{B}}}{|\mathcal{B}_i|}$ gives Eq. (12.2). Now, Eq. (12.2) implies that the number of remaining values of the chosen attribute after each iteration is less than $1/\theta$ of that before the iteration. Hence each attribute can be chosen at most $\log_\theta L$ times before all of its values are removed. Thus, the maximum number of iterations is at most $N\log_\theta L$. Also, by Eq. (12.2), at least one attribute value is removed per iteration. Hence, the maximum number of iterations is at most the number of attribute values, which is upper bounded by NL. Hence the number of iterations is upper bounded by $N\max(\log_\theta L, L)$. ■

Theorem 12.2: Worst-case Time Complexity

Let $L = \max_{n\in[N]}|\mathcal{R}_n|$. If $\theta = O\left(e^{(\frac{N|\mathcal{R}|}{L})}\right)$, which is a weaker condition than $\theta = O(1)$, the worst-case time complexity of Algorithm 12.1 is $O(kN^2|\mathcal{R}|\min(\log_\theta L, L))$.

Proof. From Lemma 12.1, the number of iterations (lines 5-16) in Algorithm 12.2 is $O(N \min(\log_\theta L, L))$. Executing lines 6 and 16 $O(N \min(\log_\theta L, L))$ times takes $O(N^2|\mathcal{R}| \min(\log_\theta L, L))$ time, which dominates the time complexity of the other parts. For example, repeatedly executing line 9 takes $O(NL \log_2 L)$ time, and by our assumption, it is dominated by $O(N^2|\mathcal{R}| \min(\log_\theta L, L))$. Thus, the worst-case time complexity of Algorithm 12.2 is $O(N^2|\mathcal{R}| \min(\log_\theta L, L))$, and that of Algorithm 12.1, which executes Algorithm 12.2, k times, is $O(kN^2|\mathcal{R}| \min(\log_\theta L, L))$. ■

However, this worst-case time complexity, which allows the worst distributions of the measure attribute values of tuples, is too pessimistic. In Section 12.5.4, we experimentally show that D-CUBE scales linearly with k, N, and $\mathcal{R}$; and sub-linearly with L even when θ is its smallest value 1.

Theorem 12.3 states the memory requirement of D-CUBE. Since the tuples do not need to be stored in memory all at once in D-CUBE, its memory requirement does not depend on the number of tuples (i.e., $|\mathcal{R}|$).

Theorem 12.3: Memory Requirements

The amount of memory space required in Algorithm 12.1 is $O(\sum_{n=1}^{N} |\mathcal{R}_n|)$.

Proof. In Algorithm 12.1, $\{\{M_{\mathcal{B}(a,n)}\}_{a \in \mathcal{B}_n}\}_{n=1}^{N}$, $\{\mathcal{R}_n\}_{n=1}^{N}$, and $\{\mathcal{B}_n\}_{n=1}^{N}$ need to be loaded into memory at once. Each has at most $\sum_{n=1}^{N} |\mathcal{R}_n|$ values. Thus, the memory requirement is $O(\sum_{n=1}^{N} |\mathcal{R}_n|)$. ■

12.5 Experiments

We designed and conducted experiments to answer the following questions:

- **Q1. Memory Efficiency**: How much memory space does D-CUBE require for analyzing real-world tensors? How large tensors can D-CUBE handle?
- **Q2. Speed and Accuaracy**: Does D-CUBE spot dense subtensors faster and more accurately than its best competitors?
- **Q3. Scalability**: Does D-CUBE scale linearly with all aspects of data? Does D-CUBE scale out?
- **Q4. Effectiveness**: Which anomalies and fraud does D-CUBE detect in real-world tensors?
- **Q5. Effect of Parameters**: How does the mass threshold θ affect the performance of D-CUBE?

12.5.1 Experimental Settings

Machines: We ran all serial algorithms on a machine with 2.67GHz Intel Xeon E7-8837 CPUs and 1TB memory. We ran MAPREDUCE algorithms on a 40-node Hadoop cluster, where each node has an Intel Xeon E3-1230 3.3GHz CPU and 32GB memory.

Datasets: Table 12.2 lists the real-world tensors used in our experiments. See Section 10.4 for a description of the datasets. We used synthetic tensors for scalability tests. Each tensor was created by generating a random binary tensor and injecting ten random dense subtensors, whose volumes are 10^N and densities (in terms of ρ_{ari}) are between $10\times$ and $100\times$ of that of the entire tensor.

Table 12.2: **Summary of the real-world tensors used in our experiments.** M: Million, K: Thousand. The underlined attributes are composite primary keys.

Name	Volume	#Tuples
Rating data (user, item, timestamp, rating, #reviews)		
AppStore [ACF13]	967K × 15.1K × 1.38K × 5	1.13M
Yelp	552K × 77.1K × 3.80K × 5	2.23M
Android [MPL15]	1.32M × 61.3K × 1.28K × 5	2.64M
Netflix [BL+07]	480K × 17.8K × 2.18K × 5	99.1M
YahooM. [DKKW12]	1.00M × 625K × 84.4K × 101	253M
Wiki revision histories (user, page, timestamp, #revisions)		
KoWiki	470K × 1.18M × 101K	11.0M
EnWiki	44.1M × 38.5M × 129K	483M
Social networks (user, user, timestamp, #interactions)		
Youtube [MMG+07]	3.22M × 3.22M × 203	18.7M
SMS	1.25M × 7.00M × 4.39K	103M
TCP dumps (src IP, dst IP, timestamp, #connections)		
Darpa [LFG+00]	9.48K × 23.4K × 46.6K	522K
TCP dumps (protocol, service, src bytes, $\cdots$, #connections)		
AirForce	3 × 70 × 11 × 7.20K × 21.5K × 512 × 512	648K

Implementations and Parameter Settings: We implemented the following dense-subtensor detection methods for our experiments:

- D-CUBE (Proposed): We implemented D-CUBE in Java with Hadoop 1.2.1. We set the mass-threshold parameter θ to 1 and used the maximum density policy for dimension selection, unless otherwise stated.
- M-ZOOM (Chapter 11): We implemented M-ZOOM in Java.
- CROSSSPOT [JBC+16]: We implemented CROSSSPOT in Java[2]. Although CROSSSPOT was originally designed to maximize ρ_{susp}, we used its variants that directly maximize the density metric compared in each experiment. We used CPD as the seed selection method of CROSSSPOT as in the previous chapter.
- CPD (CP Decomposition): Let $\{\mathbf{A}^{(n)}\}_{n=1}^{N}$ be the factor matrices obtained by CP Decomposition [KB09]. The i-th dense subtensor is composed by every attribute value a_n whose corresponding element in the i-th column of $A^{(n)}$ is greater than or equal to $1/\sqrt{|\mathcal{R}_n|}$. We used Tensor Toolbox [BK07a] for CP Decomposition.
- MAF [MGF11]: We used Tensor Toolbox [BK07a] for CP Decomposition, which MAF is largely based on.

[2]An open-sourced python implementation is available at https://github.com/mjiang89/CrossSpot

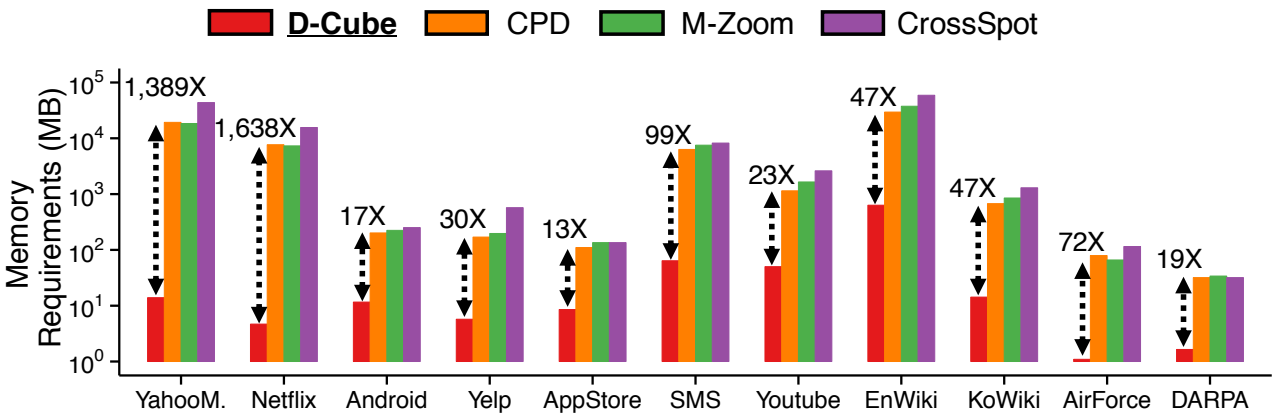


Figure 12.2: **D-CUBE is memory efficient.** D-CUBE requires up to **1,600× less memory** than its competitors.

Density Measures: We used the four density measured defined in Section 10.3.2: arithmetic average mass (ρ_{ari}), geometric average mass (ρ_{geo}), suspiciousness (ρ_{susp}), and entry surplus (i.e., $\rho_{es(\alpha)}$). The parameter α in $\rho_{es(\alpha)}$ was set to 1 unless otherwise stated.

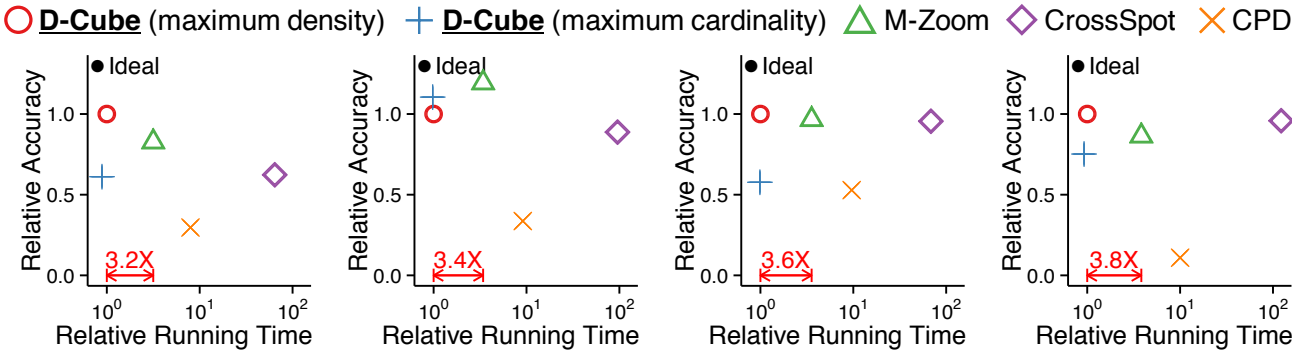
12.5.2 Q1. Memory Efficiency

D-CUBE is memory efficient. We compared the amount of memory required by different methods for handling the real-world datasets. As seen in Figure 12.2, D-CUBE, which does not require tuples to be stored in memory, needed up to *1,600× less memory* than the second best method, which stores tuples in memory. Due to its memory efficiency, D-CUBE successfully handled *1,000× larger data* than its competitors within a memory budget. We ran methods on 3-way synthetic tensors with different numbers of tuples (i.e., $|\mathcal{R}|$), with a memory budget of 16GB per machine. In every tensor, the cardinality of each dimension attribute was $1/1000$ of the number of tuples, i.e., $|\mathcal{R}_n| = |\mathcal{R}|/1000$, $\forall n \in [N]$. Figure 12.1(a) in Section 12.1 shows the result. The HADOOP implementation of D-CUBE successfully spotted dense subtensors in a tensor with 10^{11} tuples (*2.6TB*), and the serial version of D-CUBE successfully spotted dense subtensors in a tensor with 10^{10} tuples (*240GB*), which was the largest tensor that can be stored on a disk. However, all other methods ran out of memory even on a tensor with 10^9 tuples (21GB).

12.5.3 Q2. Speed and Accuracy

D-CUBE provides the best trade-off between speed and accuracy. We compared how rapidly and accurately D-CUBE (the serial version) and its competitors detect dense subtensors in the real-world datasets. We measured the wall-clock time (average over three runs) taken for detecting three subtensors by each method, and we measured the maximum density of the three subtensors found by each method using different density measures in Section 10.3.2. For this experiment, we did not limit the memory budget so that every method can handle every dataset. D-CUBE also utilized extra memory space by caching tuples in memory, as explained in Section 12.3.2.3. Figure 12.3 shows the results averaged over all datasets[3], and Figure 12.4 shows the results in each dataset. D-CUBE provided the best

[3]We computed the relative running time and relative accuracy of each method (compared to the running time and accuracy of D-CUBE with the maximum density policy) in each dataset. Then, we averaged them over all datasets.



(a) ρ_{susp} (b) ρ_{ari} (c) ρ_{geo} (d) $\rho_{es(\alpha)}$

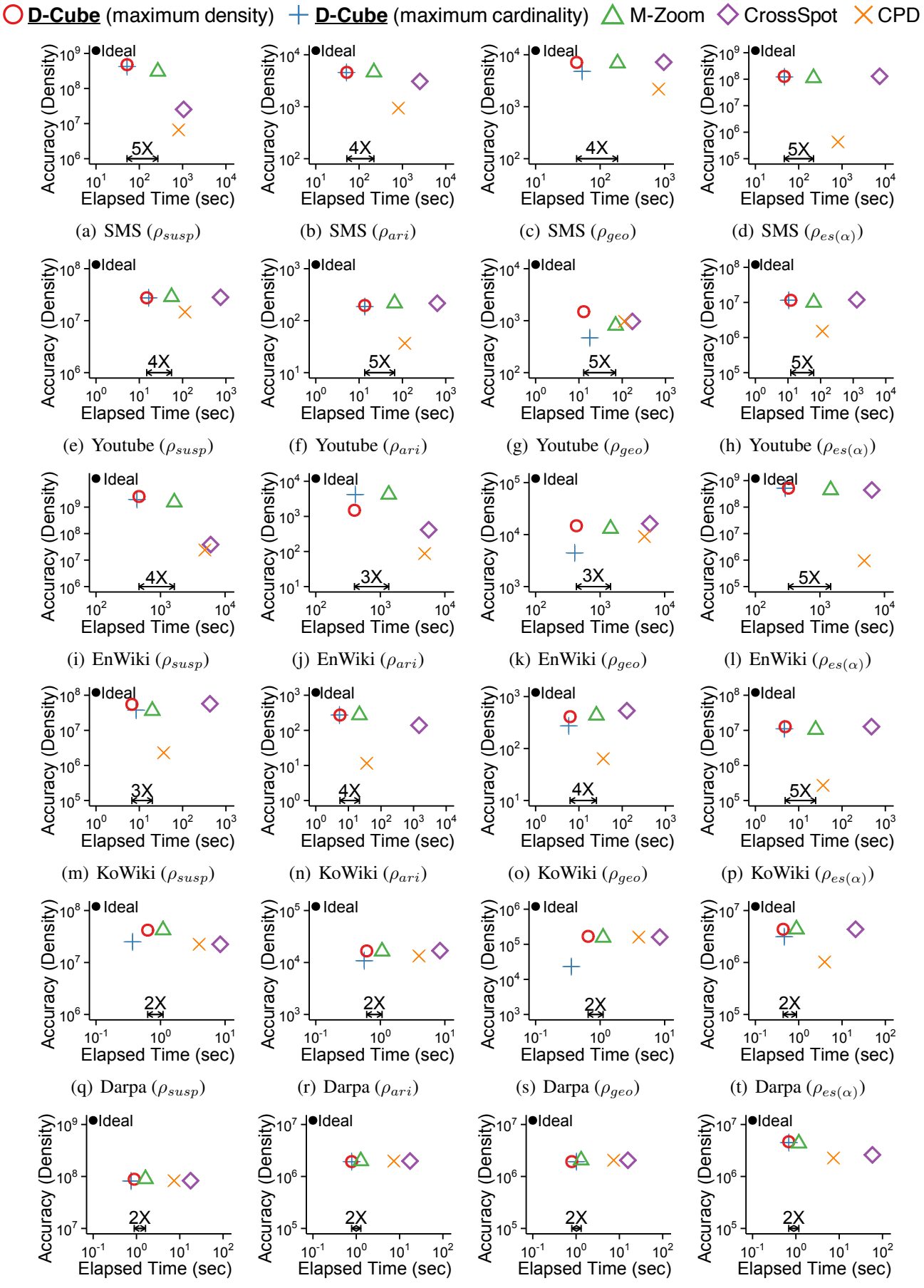
Figure 12.3: **D-CUBE achieves both speed and accuracy (on average).** In each plot, points represent the speed and accuracy of different methods averaged over all real-world tensors. Upper-left region indicates better performance. D-CUBE is about *3.6× faster* than the second best method M-ZOOM. Moreover, D-CUBE with the maximum density consistently shows high accuracy regardless of density measures, while the other methods do not.

trade-off between speed and accuracy. Specifically, D-CUBE was up to *7× faster* (on average 3.6× faster) than the second fastest method M-ZOOM. Moreover, D-CUBE (with the maximum density policy) consistently spotted high-density subtensors, while the accuracies of the other methods varied on density measures. Specifically, on average, D-CUBE (with the maximum density policy) showed the highest accuracy with all the density measures except ρ_{ari}, which M-ZOOM and D-CUBE (with the maximum cardinality policy) were more accurate with. Although MAF does not appear in Figures 12.3 and 12.4, it consistently provided sparser subtensors than CPD with similar speed.

12.5.4 Q3. Scalability

D-CUBE scales (sub-)linearly with every input factor (i.e., the number of tuples, the number of dimension attributes, and the cardinality of dimension attributes, and the number of subtensors that we aim to find). To measure the scalability with each factor, we started with finding a dense subtensor in a synthetic tensor with 10^8 tuples and 3 dimension attributes each of whose cardinality is 10^5. Then, we measured the running time as we changed one factor at a time while fixing the other factors. The threshold parameter θ was fixed to 1. As seen in Figure 12.5, D-CUBE scaled linearly with every factor and sub-linearly with the cardinality of attributes even when θ was set to its minimum value 1. This supports our claim in Section 12.4.2 that the worst-case time complexity of D-CUBE (Theorem 12.2) is too pessimistic. This linear scalability of D-CUBE held both with enough memory budget (blue solid lines in Figure 12.5) to store all tuples and with minimum memory budget (red dashed lines in Figure 12.5) to barely meet the requirements although D-CUBE was up to 3× faster in the former case.

D-CUBE scales out. We also evaluated the machine scalability of the MAPREDUCE implementation of D-CUBE. We measured its running time taken for finding a dense subtensor in a synthetic tensor with 10^{10} tuples and 3 dimension attributes each of whose cardinality is 10^7, as we increased the number of machines running in parallel from 1 to 40. Figure 12.6 shows the changes in the running time and the speed-up, which is defined as T_1/T_M where T_M is the running time with M machines. The speed-up increased near linearly when a small number of machines were used, while it flattened as more machines were added due to the overhead in the distributed system.



(a) SMS (ρ_{susp}) (b) SMS (ρ_{ari}) (c) SMS (ρ_{geo}) (d) SMS ($\rho_{es(\alpha)}$)

(e) Youtube (ρ_{susp}) (f) Youtube (ρ_{ari}) (g) Youtube (ρ_{geo}) (h) Youtube ($\rho_{es(\alpha)}$)

(i) EnWiki (ρ_{susp}) (j) EnWiki (ρ_{ari}) (k) EnWiki (ρ_{geo}) (l) EnWiki ($\rho_{es(\alpha)}$)

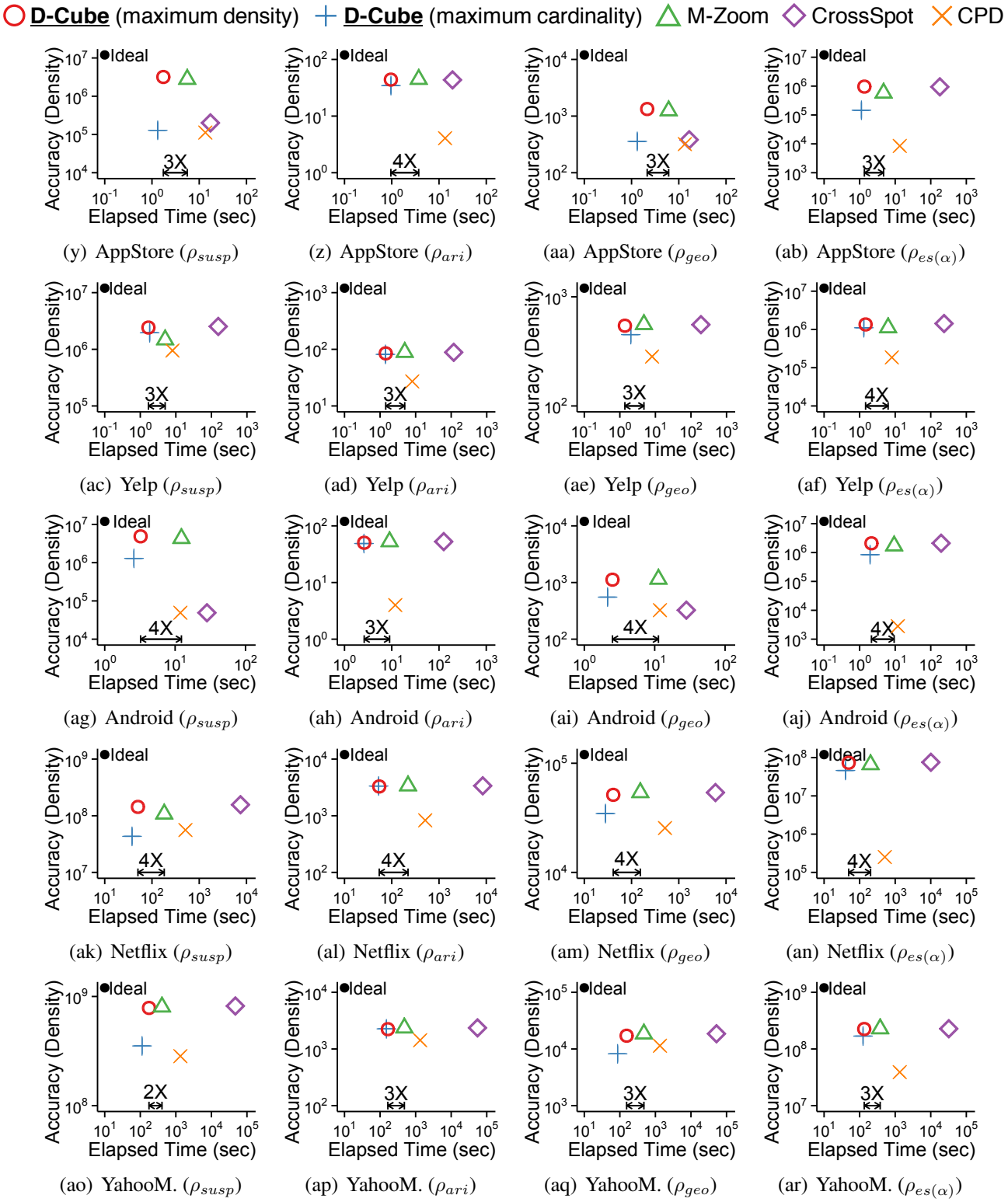
(m) KoWiki (ρ_{susp}) (n) KoWiki (ρ_{ari}) (o) KoWiki (ρ_{geo}) (p) KoWiki ($\rho_{es(\alpha)}$)

(q) Darpa (ρ_{susp}) (r) Darpa (ρ_{ari}) (s) Darpa (ρ_{geo}) (t) Darpa ($\rho_{es(\alpha)}$)

(u) AirForce (ρ_{susp}) (v) AirForce (ρ_{ari}) (w) AirForce (ρ_{geo}) (x) AirForce ($\rho_{es(\alpha)}$)

(continues on the next page)

(continues from the previous page)



(y) AppStore (ρ_{susp}) (z) AppStore (ρ_{ari}) (aa) AppStore (ρ_{geo}) (ab) AppStore ($\rho_{es(\alpha)}$)

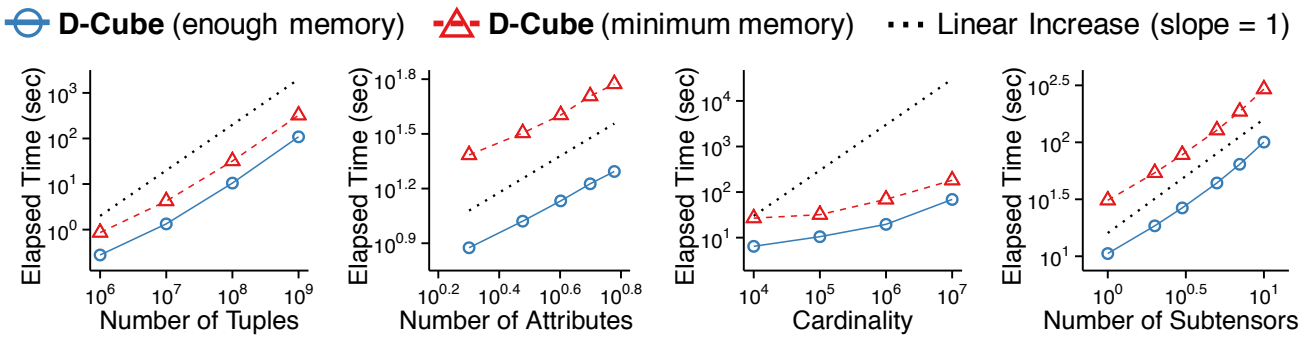
(ac) Yelp (ρ_{susp}) (ad) Yelp (ρ_{ari}) (ae) Yelp (ρ_{geo}) (af) Yelp ($\rho_{es(\alpha)}$)

(ag) Android (ρ_{susp}) (ah) Android (ρ_{ari}) (ai) Android (ρ_{geo}) (aj) Android ($\rho_{es(\alpha)}$)

(ak) Netflix (ρ_{susp}) (al) Netflix (ρ_{ari}) (am) Netflix (ρ_{geo}) (an) Netflix ($\rho_{es(\alpha)}$)

(ao) YahooM. (ρ_{susp}) (ap) YahooM. (ρ_{ari}) (aq) YahooM. (ρ_{geo}) (ar) YahooM. ($\rho_{es(\alpha)}$)

Figure 12.4: **D-CUBE achieves both speed and accuracy (in most datasets).** In each plot, points represent the speed and accuracy of different methods. Upper-left region indicates better performance. D-CUBE is up to 7× *faster* than the second fast method M-ZOOM. Moreover, D-CUBE with the maximum density policy is the only method that is consistently accurate regardless of density measures.



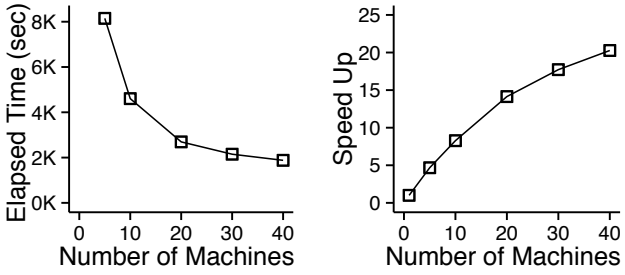
(a) Scalability w.r.t $|\mathcal{R}|$ (# non-zeros in the input tensor)

(b) Scalability w.r.t N (order of the input tensor)

(c) Scalability w.r.t $|\mathcal{R}_n|$ (dimensionality of the input tensor)

(d) Scalability w.r.t k (# subtensors we aim to find)

Figure 12.5: **D-CUBE is scalable.** D-CUBE scales linearly or sub-linearly with the number of tuples, the number of attributes, the cardinalities of attributes, and the number of subtensors we aim to find.



(a) Elapsed Time

(b) Speed Up

Figure 12.6: **D-CUBE scales out.** The MAPREDUCE implementation of D-CUBE is speeded up 8× with 10 machines, and 20× with 40 machines.

12.5.5 Q4. Effectiveness

We demonstrate the effectiveness of D-CUBE in four applications using real-world tensors.

D-CUBE detects network attacks accurately from TCP dumps. We considered two TCP dumps that are modeled differently. The Darpa dataset is a 3-way tensor where the dimension attributes are source IPs, destination IPs, and timestamps in minutes; and the measure attribute is the number of connections. The AirForce dataset, which does not include IP information, is a 7-way tensor where the measure attribute is the same but the dimension attributes are the features of the connections, including protocols and services. Both datasets include labels indicating whether each connection is malicious or not.

Figure 12.1(c) in Section 12.1 lists the five densest subtensors (in terms of ρ_{geo}) found by D-CUBE in each dataset. Notice that the dense subtensors are mostly composed of various types of network attacks. Based on this observation, we classified each connection as malicious or benign based on the density of the densest subtensor containing the connection (i.e., the denser the subtensor including a connection is, the more suspicious the connection is). This led to high accuracy as seen in Table 12.3, which reports the accuracy when each method (with the density measure giving the highest accuracy) was used for dense-subtensor detection. In both datasets, D-CUBE resulted in the highest accuracy.

D-CUBE spots synchronized behavior accurately in rating data. We assumed an attack scenario where fraudsters in a review site, who aim to boost (or lower) the ratings of the set of items, create multiple

Table 12.3: **D-CUBE spots network attacks fastest and most accurately** from TCP dumps, as summarized in the first table. The top-5 subtensors detected by D-CUBE in each of the datasets are described in the second table.

Datasets	AirForce		Darpa	
	Elapsed Time (sec)	Accuracy (AUC)	Elapsed Time (sec)	Accuracy (AUC)
CPD [KB09]	413.2	0.854	105.0	0.926
MAF [MGF11]	486.6	0.912	102.4	0.514
CROSSSPOT [JBC+16]	575.5	0.924	132.2	0.923
M-ZOOM (Chapter 11)	27.7	0.975	22.7	0.923
D-CUBE	**15.6**	**0.987**	**9.1**	**0.930**

Dataset	Order	Volume	Mass	Attack Ratio	Attack Type
AirForce	1	1	1.93M	**100%**	Smurf
	2	8	2.53M	**100%**	Smurf
	3	6,160	897K	**100%**	Neptune
	4	63.5K	1.02M	94.7%	Neptune
	5	930K	1.00M	94.7%	Neptune
Darpa	1	738	1.52M	**100%**	Neptune
	2	522	614K	**100%**	Neptune
	3	402	113K	**100%**	Smurf
	4	1	10.8K	**100%**	Satan
	5	156K	560K	30.4%	SNMP

user accounts and give the same score to the items within a short period of time. This lockstep behavior forms a dense subtensor with volume (# fake accounts $\times$ # target items $\times$ 1 $\times$ 1) in the rating dataset, whose dimension attributes are users, items, timestamps, and rating scores.

We injected 10 such random dense subtensors whose volumes varied from $15\times15\times1\times1$ to $60\times60\times1\times1$ in the Yelp and Android datasets. We compared the number of the injected subtensors detected by each dense-subtensor detection method. We considered each injected subtensor as overlooked by a method, if the subtensor did not belong to any of the top-10 dense subtensors spotted by the method or it was hidden in a natural dense subtensor at least 10 times larger than the injected subtensor. We repeated this experiment 10 times, and the averaged results are summarized in Table 12.4. For each method, we report the results with the density measure giving the highest accuracy. In both datasets, D-CUBE detected a largest number of the injected subtensors. Especially, in the Android dataset, D-CUBE detected 9 out of the 10 injected subtensors, while the second best method detected only 7 injected subtensors on average.

D-CUBE successfully spots spam reviews in the AppStore dataset. The dataset contains reviews from App Store, an online software marketplace. We modeled the dataset as a 4-way tensor whose dimension attributes are users, software, ratings, and timestamps in dates, and we applied D-CUBE (with $\rho = \rho_{ari}$) to the dataset. Table 12.7 shows the statistics of the top-3 dense subtensors. Although ground-truth labels were not available, as the examples in Table 12.5 show, all the reviews composing

Table 12.4: **D-CUBE detects synchronized behavior fastest and most accurately** in rating datasets.

Datasets	Android		Yelp	
	Elapsed Time (sec)	Recall @ Top-10	Elapsed Time (sec)	Recall @ Top-10
CPD [KB09]	59.9	0.54	47.5	0.52
MAF [MGF11]	95.0	0.54	49.4	0.52
CROSSSPOT [JBC+16]	71.3	0.54	56.7	0.52
M-ZOOM (Chapter 11)	28.4	0.70	17.7	0.30
D-CUBE	**7.0**	**0.90**	**4.9**	**0.60**

Table 12.5: **D-CUBE successfully detects spam reviews** in the AppStore dataset.

Subtensor 1 (100% spam)

User	Review	Date
Ti*	type in *** and you will get ...	Mar-4
Fo*	type in for the bonus code: ...	Mar-4
dj*	typed in the code: *** ...	Mar-4
Di*	enter this code to start with ...	Mar-4
Fe*	enter code: *** to win even ...	Mar-4

Subtensor 2 (100% spam)

User	Review	Date
Sk*	invite code***, referral— ...	Apr-18
fu*	use my code for bonus ...	Apr-18
Ta*	enter the code *** for ...	Apr-18
Ap*	bonus code *** for points ...	Apr-18
De*	bonus code: ***, be one ...	Apr-18

Subtensor 3 (at least 48% spam)

User	Review	Date
Mr*	entered this code and got ...	Nov-23
Max*	enter the bonus code: *** ...	Nov-23
Je*	enter *** when it asks...	Nov-23
Man*	just enter *** for a boost ...	Nov-23
Ty*	enter *** ro receive a ...	Nov-23

Table 12.6: **D-CUBE successfully spots bot activities** in the EnWiki dataset.

Subtensor #	Users in each subtensor (100% bots)
1	WP 1.0 bot
2	AAlertBot
3	AlexNewArtBot, VeblenBot, InceptionBot
4	WP 1.0 bot
5	Cydebot, VeblenBot

Table 12.7: **Summary of the three dense subtensors that D-CUBE detects in real-world datasets.**

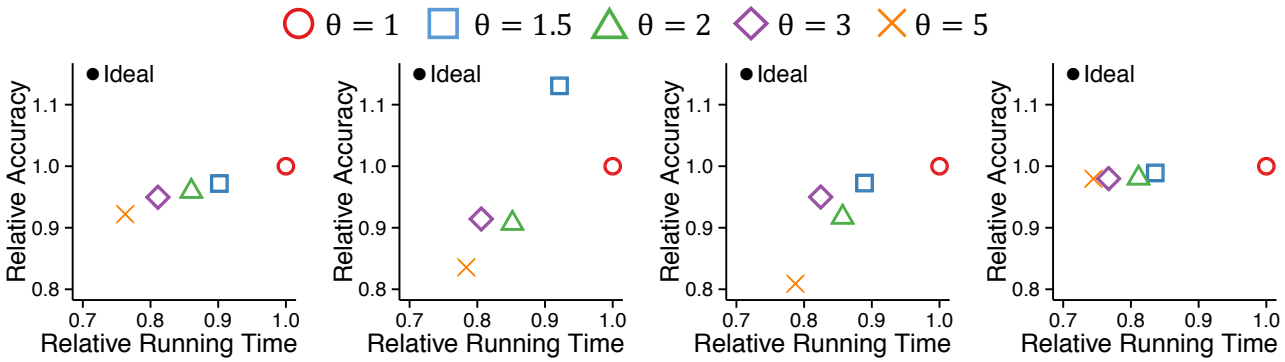
Dataset	Order	Volume	Mass	ρ_{ari}	Type
AppStore	1	120	308	44.0	Spam reviews
	2	612	435	31.6	Spam reviews
	3	231,240	771	20.3	Spam reviews
KoWiki	1	8	546	273.0	Edit war
	2	80	1,011	233.3	Edit war
	3	270	1,126	168.9	Edit war
EnWiki	1	9.98M	1.71M	7,931	Bot activities
	2	541K	343K	4,211	Bot activities
	3	23.5M	973K	3,395	Bot activities

the first and second dense subtensors were obvious spam reviews. In addition, at least 48% of the reviews composing the third dense subtensor were obvious spam reviews.

D-CUBE detects various types of anomalies in Wikipedia revision histories. We modeled the datasets as 3-way tensors whose dimension attributes are users, pages, and timestamps in hours. Table 12.7 gives the statistics of the top-3 dense subtensors detected by D-CUBE (with $\rho = \rho_{ari}$ and the maximum cardinality policy) in the KoWiki dataset and by D-CUBE (with $\rho = \rho_{geo}$ and the maximum density policy) in the EnWiki dataset. All three subtensors detected in the KoWiki dataset indicated edit wars. For example, the second subtensor corresponded to an edit war where 4 users changed 4 pages, 1,011 times, within 5 hours. On the other hand, all three subtensors detected in the Enwiki dataset indicated bot activities. For example, the third subtensor corresponded to 3 bots which edited 1,067 pages 973,747 times. The users composing the top-5 dense subtensors in the EnWiki dataset are listed in Table 12.6. Notice that all of them are bots.

12.5.6 Q5. Effects of Parameter θ on Speed and Accuracy

The parameter θ controls the trade-off between speed and accuracy. We investigated the effects of the mass-threshold parameter θ on the speed and accuracy of D-CUBE in the real-world datasets. We used the serial version of D-CUBE with a memory budget of 16GB, and we measured its relative accuracy and speed as in Section 12.5.3. Figure 12.7 shows the results averaged over all datasets. Different θ values provided a trade-off between speed and accuracy. Specifically, increasing θ tended to make D-CUBE faster but less accurate. This tendency is consistent with our theoretical analyses (Theo-



(a) ρ_{susp} (b) ρ_{ari} (c) ρ_{geo} (d) $\rho_{es(\alpha)}$

Figure 12.7: **The mass-threshold parameter θ controls the trade-off between the speed and accuracy of D-CUBE.** We report the accuracy and speed averaged over all real-world datasets. As θ increases, D-CUBE tends to be faster but less accurate.

rems 12.1-12.2 in Section 12.4.2). The sensitivity of the accuracy to θ depended on the used density measures. Specifically, the sensitivity was lower with $\rho_{es(\alpha)}$ than with the other density measures.

12.6 Summary

We propose D-CUBE, a disk-based dense-subtensor detection method, to deal with disk-resident tensors too large to fit in main memory. D-CUBE is optimized to minimize disk I/Os while providing a guarantee on the quality of the subtensors it finds. Moreover, we propose a distributed version of D-CUBE running on MAPREDUCE for terabyte-scale or larger data distributed across multiple machines. In summary, D-CUBE achieves the following advantages over its state-of-the-art competitors:

- **Memory Efficient:** D-CUBE handles *1,000*× larger data (*2.6TB*) by reducing memory usage up to *1,600*× compared to in-memory algorithms (Figures 12.1(a) and 12.2).
- **Fast:** Even when data fit in memory, D-CUBE is up to *7*× faster than its competitors (Figures 12.3 and 12.4) with near-linear scalability (Figures 12.5 and 12.6).
- **Provably Accurate:** D-CUBE is one of the methods giving the best approximation guarantee (Theorem 12.1) and the densest subtensors in practice (Figures 12.3 and 12.4).
- **Effective:** D-CUBE was most accurate in two applications: detecting network attacks from TCP dumps and lockstep behavior in rating data (Tables 12.3-12.7).

Reproducibility: The source code and datasets used in this chapter are available at `http://www.cs.cmu.edu/~kijungs/codes/dcube`.

Chapter based on work that appeared at KDD 2017 [SHKF17b].

Chapter 13

Detecting Dense Subtensors in Large Tensors (3): Incremental Algorithms

Consider a stream of retweet events on social media - how can we spot fraudulent lock-step behavior in such multi-aspect data evolving over time? Can we detect it in real time, with accuracy guarantees?

In the previous two chapters, we propose rapid and accurate algorithms for detecting dense subtensors, which signal anomalies and fraud in many domains, including social media, e-commerce, and network security. However, the algorithms assume that input tensors are static, while many real-world tensors, including those from the aforementioned domains, evolve over time.

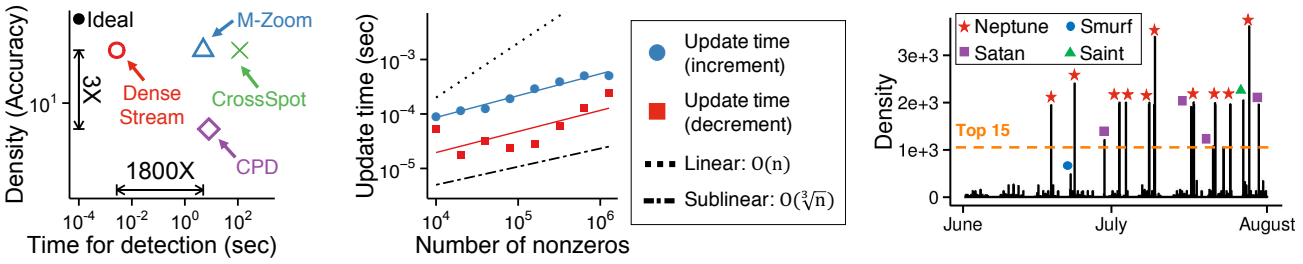
In this chapter, we propose (a) DENSESTREAM, an incremental algorithm that maintains and updates a dense subtensor in a tensor stream (i.e., a sequence of changes in a tensor), and (b) DENSEALERT, an incremental algorithm that spots the sudden appearance of dense subtensors. Our algorithms are: (1) *Fast and 'any time'*: updates by our algorithms are up to **a million times faster** than the fastest batch algorithms, (2) *Provably accurate*: our algorithms guarantee a lower bound on the density of the subtensor they maintain, and (3) *Effective*: DENSEALERT successfully spots anomalies, including those overlooked by existing algorithms, in many real-world tensors.

13.1 Motivation

Given a stream of changes in a tensor that evolves over time, how can we detect the sudden appearances of dense subtensors? Can we detect them incrementally, without having to rescan all tensor entries?

An important application of this problem is intrusion detection systems in networks, where attackers make a large number of connections to target machines to block their availability or to look for vulnerabilities [LFG+00]. Consider a stream of connections where we represent each connection from a source IP address to a destination IP address as an entry in a 3-way tensor (source IP, destination IP, timestamp). Sudden appearances of dense subtensors in the tensor often indicate network attacks. For example, in Figure 13.1(c), all the top 15 densest subtensors concentrated in a short period of time, which are detected by our DENSEALERT algorithm, actually come from network attacks.

Another application is detecting fake rating attacks in review sites, such as Amazon and Yelp. Ratings can be modeled as entries in a 4-way tensor (user, item, timestamp, rating). Injection attacks maliciously manipulate the ratings of a set of items by adding a large number of similar ratings for the



(a) Fast and Accurate (b) Scalable (c) Effective

Figure 13.1: **Strengths of DENSESTREAM and DENSEALERT.** *(a) Fast and accurate:* DENSESTREAM, our incremental algorithm, detects dense subtensors significantly faster than batch algorithms without losing accuracy. *(b) Scalable:* The time taken for each update in DENSESTREAM grows sublinearly with the size of data. *(c) Effective:* DENSEALERT, which detects suddenly emerging dense subtensors, identifies network attacks from a TCP Dump with high accuracy (AUC=0.924). Especially, all the 15 densest subtensors revealed by DENSEALERT indicate actual network attacks of various types. See Section 13.5 for details.

items, creating dense subtensors in the tensor. To guard against such fraud, an alert system detecting suddenly appearing dense subtensors in real time, as they arrive, is desirable.

As discussed in the previous chapters, several algorithms for dense-subtensor detection have been proposed for detecting network attacks [MGF11, SHF18, SHKF18], retweet boosting [JBC+16], rating attacks [SHKF18], and bots [SHF18] as well as for genetics applications [SHK+10]. However, these existing algorithms assume a static tensor rather than a stream of events (i.e., changes in a tensor) over time. In addition, our experiments in Section 13.5 show that they are limited in their ability to detect dense subtensors small but highly concentrated in a short period of time.

In this last chapter on dense-subtensor detection, we propose DENSESTREAM, an incremental algorithm for dense-subtensor detection. It detects dense subtensors in real time as events arrive, and is hence more useful in many practical settings, including those mentioned above. DENSESTREAM is also used as a building block of DENSEALERT, an incremental algorithm for detecting the sudden emergence of dense subtensors. DENSEALERT takes into account the tendency for lock-step behavior, such as network attacks and rating manipulation attacks, to appear within short, continuous intervals of time, which is an important signal for spotting lockstep behavior.

In a nutshell, as the entries of a tensor change, our algorithms work by maintaining a small subset of subtensors that always includes a dense subtensor with a theoretical guarantee on its density. By focusing on this subset, our algorithms detect a dense subtensor in a time-evolving tensor up to a million times faster than the fastest batch algorithms, while providing the same theoretical guarantee on the density of the detected subtensor.

In summary, the main advantages of our algorithms are as follows:

- **Fast and 'any time'**: incremental updates by our algorithms are up to *a million times faster* than the fastest batch algorithms (Figure 13.1(a)).
- **Provably accurate**: our algorithms maintain a subtensor with a theoretical guarantee on its density, and in practice, its density is similar to that of subtensors found by the best batch algorithms (Figure 13.1(a)).

- **Effective**: DENSEALERT successfully detects bot activities and network intrusions (Figure 13.1(c)) in real-world tensors. It also spots small-scale rating manipulation attacks, overlooked by existing algorithms.

Reproducibility: The source code and datasets used in this chapter are available at *http://www.cs.cmu.edu/~kijungs/codes/alert*.

The rest of this chapter is organized as follows. In Section 13.2, we introduce some preliminary concepts, notations, and formal problem definitions. In Section 13.3, we present our proposed algorithms, namely DENSESTREAM and DENSEALERT. In Section 13.4, we theoretically analyze the accuracy and complexity of the algorithms. After sharing some experimental results in Section 13.5, we provide a summary of this chapter in Section 13.6.

13.2 Preliminaries and Problem Definition

In this section, we introduce notations and concepts used in the chapter. Then, we give formal problem definitions. For the ease of presentation, we use a set of notations different from those used in the previous two chapters.

13.2.1 Notations and Concepts

Symbols frequently used in the chapter are listed in Table 13.1, and a toy example is in Example 13.1. We use $[y] = \{1, 2..., y\}$ for brevity.

Notations for Tensors: Tensors are multi-dimensional arrays that generalize vectors (1-way tensors) and matrices (2-way tensors) to higher orders. Consider an N-way tensor $\mathcal{X}$ of size $I_1 \times ... \times I_N$ with non-negative entries. Each $(i_1, ..., i_N)$-th entry of $\mathcal{X}$ is denoted by $x_{i_1 \cdots i_N}$. Equivalently, each *n-mode index* of $x_{i_1 \cdots i_N}$ is i_n. We use $\mathcal{X}_{(n,i_n)}$ to denote the *n-mode slice* (i.e. $(N-1)$-way tensor) obtained by fixing n-mode index to i_n. Then, $Q = \{(n, i_n) : n \in [N], i_n \in [I_n]\}$ indicates all the slice indices. We denote a member of Q by q.

For example, if $N = 2$, $\mathcal{X}$ is a matrix of size $I_1 \times I_2$. Then, $\mathcal{X}_{(1,i_1)}$ is the i_1-th row of $\mathcal{X}$, and $\mathcal{X}_{(2,i_2)}$ is the i_2-th column of $\mathcal{X}$. In this setting, Q is the set of all row and column indices.

Notations for Subtensors: Let S be a subset of Q. $\mathcal{X}(S)$ denotes the subtensor composed of the slices with indices in S, i.e., $\mathcal{X}(S)$ is the subtensor left after removing all the slices with indices not in S.

For example, if $\mathcal{X}$ is a matrix (i.e., $N = 2$) and $S = \{(1, 1), (1, 2), (2, 2), (2, 3)\}$, $\mathcal{X}(S)$ is the submatrix of $\mathcal{X}$ composed of the first and second rows and the second and third columns.

Notations for Orderings: Consider an ordering of the slice indices in Q. A function $\pi : [|Q|] \to Q$ denotes such an ordering where, for each $j \in [|Q|]$, $\pi(j)$ is the slice index in the jth position. That is, each slice index $q \in Q$ is in the $\pi^{-1}(q)$-th position in π. Let $Q_{\pi,q} = \{r \in Q : \pi^{-1}(r) \geq \pi^{-1}(q)\}$ be the slice indices located after or equal to q in π. Then, $\mathcal{X}(Q_{\pi,q})$ is the subtensor of $\mathcal{X}$ composed of the slices with their indices in $Q_{\pi,q}$.

Notations for Slice Sum: We denote the sum of the entries of $\mathcal{X}$ included in subtensor $\mathcal{X}(S)$ by $sum(\mathcal{X}(S))$. Similarly, we define *the slice sum* of $q \in Q$ in subtensor $\mathcal{X}(S)$, denoted by $d(\mathcal{X}(S), q)$, as the sum of the entries of $\mathcal{X}$ that are included in both $\mathcal{X}(S)$ and the slice with index $q \in Q$. For an ordering π and a slice index $q \in Q$, we use $d_\pi(q) = d(\mathcal{X}(Q_{\pi,q}), q)$ for brevity, and define the *cumulative maximum slice sum* of q as $c_\pi(q) = \max\{d_\pi(r) : r \in Q, \pi^{-1}(r) \leq \pi^{-1}(q)\}$, i.e., maximum $d_\pi(\cdot)$ among the slice indices located before or equal to q in π.

Table 13.1: **Table of frequently-used symbols.**

Symbol	Definition
$\mathcal{X}$	an input tensor
N	order of $\mathcal{X}$
$x_{i_1 \cdots i_N}$	entry of $\mathcal{X}$ with index $(i_1, ..., i_N)$
$nnz(\mathcal{X})$	number of non-zero entries in $\mathcal{X}$
Q	set of the slice indices of $\mathcal{X}$
q	a member of Q
$\mathcal{X}(S)$	subtensor composed of the slices in $S \subset Q$
$\pi : [\|Q\|] \to Q$	an ordering of slice indices in Q
$Q_{\pi,q}$	slice indices located after or equal to q in π
$sum(\mathcal{X}(S))$	sum of the entries included in $\mathcal{X}(S)$
$d(\mathcal{X}(S), q)$	slice sum of q in $\mathcal{X}(S)$
$d_\pi(q)$	slice sum of q in $\mathcal{X}(Q_{\pi,q})$
$c_\pi(q)$	cumulative max. slice sum of q in $\mathcal{X}(Q_{\pi,q})$
$((i_1, ..., i_N), \delta, +)$	increment of $x_{i_1 \cdots i_N}$ by δ
$((i_1, ..., i_N), \delta, -)$	decrement of $x_{i_1 \cdots i_N}$ by δ
$\rho(\mathcal{X}(S))$	density of a subtensor $\mathcal{X}(S)$
ρ_{opt}	density of the densest subtensor in $\mathcal{X}$
ΔT	time window in DENSEALERT
$[y]$	$\{1, 2..., y\}$

Notations for Tensor Streams: A tensor stream is a sequence of changes in $\mathcal{X}$. Let $((i_1, ..., i_N), \delta, +)$ be an increment of entry $x_{i_1 \cdots i_N}$ by $\delta > 0$ and $((i_1, ..., i_N), \delta, -)$ be a decrement of entry $x_{i_1 \cdots i_N}$ by $\delta > 0$.

Density Measure: Definition 13.1 gives the density measure used in this chapter. That is, the density of a subtensor is defined as the sum of its entries divided by the number of slices composing it. Note that given an input tensor $\mathcal{X}$, maximizing the density is equivalent to maximizing arithmetic average mass (see Section 10.3.2), which proved effective for anomaly and fraud detection in the previous chapters. We let ρ_{opt} be the density of a densest subtensor in $\mathcal{X}$.

Definition 13.1: Density of a Subtensor

Consider a subtensor $\mathcal{X}(S)$ of a tensor $\mathcal{X}$. The density of $\mathcal{X}(S)$, which is denoted by $\rho(\mathcal{X}(S))$, is defined as

$$\rho(\mathcal{X}(S)) = \frac{sum(\mathcal{X}(S))}{|S|}.$$

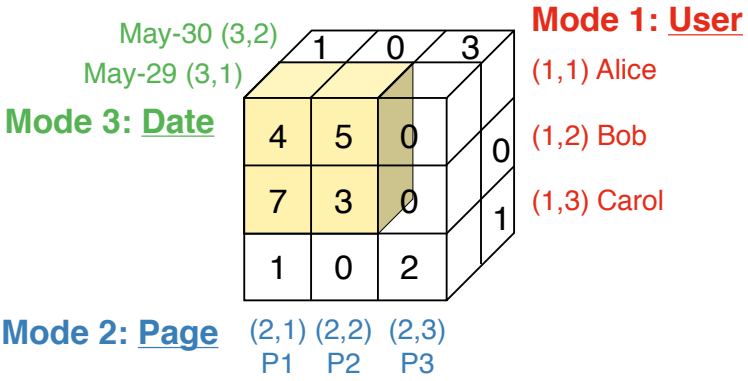


Figure 13.2: **Pictorial depiction of Example 13.1.** The colored region indicates the subtensor $\mathcal{X}(S)$.

Example 13.1: Wikipedia Revision History

Consider the 3-way tensor in Figure 13.2. In the tensor, each entry x_{ijk} indicates that user i revised page j on date k, x_{ijk} times. The set of the slice indices is $Q = \{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2)\}$. Consider its subset $S = \{(1,1),(1,2),(2,1),(2,2),(3,1)\}$. Then, $\mathcal{X}(S)$ is the subtensor composed of the slices with their indices in S, as seen in Figure 13.2. In this setting, $sum(\mathcal{X}(S)) = 4+5+7+3 = 19$, and $d(\mathcal{X}(S),(2,2)) = 5+3 = 8$. Let π be an ordering of Q where $\pi(1) = (1,3)$, $\pi(2) = (2,3)$, $\pi(3) = (3,2)$, $\pi(4) = (2,2)$, $\pi(5) = (1,1)$, $\pi(6) = (1,2)$, $\pi(7) = (2,1)$, and $\pi(8) = (3,1)$. Then, $Q_{\pi,(2,2)} = S$, and $d_\pi((2,2)) = d(\mathcal{X}(Q_{\pi,(2,2)}),(2,2)) = d(\mathcal{X}(S),(2,2)) = 8$.

13.2.2 Problem Definitions

We give the formal definitions of the problems studied in this chapter. The first problem (Problem 13.1) is to maintain the densest subtensor in a tensor that keeps changing.

Problem 13.1: Detecting the Densest Subtensor in a Tensor Stream

1. **Given:** a sequence of changes in a tensor $\mathcal{X}$ with slice indices Q (i.e., a tensor stream)
2. **Maintain:** a subtensor $\mathcal{X}(S)$ where $S \subset Q$,
3. **to Maximize:** its density $\rho(\mathcal{X}(S))$.

As discussed in Section 10.3.2, identifying the exact densest subtensor is computationally expensive even for a static tensor. Thus, we focus on designing an approximation algorithm that maintains a dense subtensor with a provable approximation bound, significantly faster than repeatedly finding a dense subtensor from scratch.

The second problem (Problem 13.2) is to detect suddenly emerging dense subtensors in a tensor stream. For a tensor $\mathcal{X}$ whose values increase over time, let $\mathcal{X}_{\Delta T}$ be the tensor where the value of each entry is the increment in the corresponding entry of $\mathcal{X}$ in the last ΔT time units. Our aim is to spot dense subtensors appearing in $\mathcal{X}_{\Delta T}$, which also keeps changing.

Problem 13.2: Detecting Sudden Dense Subtensors in a Tensor Stream

1. **Given:**
   - a sequence of increments in a tensor $\mathcal{X}$ with slice indices Q (i.e., a tensor stream)
   - a time window ΔT,
2. **Maintain:** a subtensor $\mathcal{X}_{\Delta T}(S)$ where $S \subset Q$,
3. **to Maximize:** its density $\rho(\mathcal{X}_{\Delta T}(S))$.

13.3 Proposed Algorithms: DENSESTREAM and DENSEALERT

In this section, we propose DENSESTREAM, which is an incremental algorithm for dense-subtensor detection in a tensor stream, and DENSEALERT, which detects suddenly emerging dense subtensors. Specifically, we first provide an overview of the algorithms in Section 13.3.1. Then, we present DENSESTATIC, a baseline algorithm for dense-subtensor detection in a static tensor, in Section 13.3.2. After that, we present DENSESTREAM and DENSEALERT in Sections 13.3.3 and 13.3.4, respectively.

13.3.1 Overview

The main idea behind all algorithms discussed in the following subsections is to compute (from scratch or incrementally) a D-ordering to drastically reduce the search space while providing a guarantee on the accuracy. As defined in Definition 13.2, a D-ordering is an ordering of slice indices obtained by choosing a slice index with minimum slice sum repeatedly (as in D-ORDERING() of Algorithm 13.1). With a D-ordering π, the search space of $2^{|Q|}$ possible subtensors can be reduced to $\{\mathcal{X}(Q_{\pi,q}) : q \in Q\}$. In this space of size $|Q|$, however, there always exists a subtensor whose density is at least 1/(order of the input tensor) of maximum density, as formalized in Lemmas 13.1 and 13.2.

Definition 13.2: D-ordering

An ordering π is a **D-ordering** of Q in $\mathcal{X}$ if $\forall q \in Q$, $d(\mathcal{X}(Q_{\pi,q}), q) = \min_{r \in Q_{\pi,q}} d(\mathcal{X}(Q_{\pi,q}), r)$.

Lemma 13.1:

Let $\mathcal{X}(S^*)$ be a subtensor with the maximum density, i.e., $\rho(\mathcal{X}(S^*)) = \rho_{opt}$. Then for any $q \in S^*$,

$$d(\mathcal{X}(S^*), q) \geq \rho(\mathcal{X}(S^*)). \tag{13.1}$$

Proof. The maximality of the density of $\mathcal{X}(S^*)$ implies $\rho(\mathcal{X}(S^*\backslash\{q\})) \leq \rho(\mathcal{X}(S^*))$, and plugging in Definition 2.2 to ρ gives

$$\frac{sum(\mathcal{X}(S^*)) - d(\mathcal{X}(S^*), q)}{|S^*| - 1} = \frac{sum(\mathcal{X}(S^*\backslash\{q\}))}{|S^*| - 1} = \rho(\mathcal{X}(S^*\backslash\{q\})) \leq \rho(\mathcal{X}(S^*)) = \frac{sum(\mathcal{X}(S^*))}{|S^*|},$$

which reduces to Eq. (13.1). ■

Lemma 13.2: Existence of a Dense Subtensor

Given a D-ordering π in an N-way tensor $\mathcal{X}$, there exists $q \in Q$ such that $\rho(\mathcal{X}(Q_{\pi,q})) \geq \rho_{opt}/N$.

Proof. Let $\mathcal{X}(S^*)$ be satisfying $\rho(\mathcal{X}(S^*)) = \rho_{opt}$, and let $q^* \in S^*$ be satisfying that $\forall q \in S^*$, $\pi^{-1}(q^*) \leq \pi^{-1}(q)$. Our goal is to show $\rho(\mathcal{X}(Q_{\pi,q^*})) \geq \frac{1}{N}\rho(\mathcal{X}(S^*))$, which we show as

$$N\rho(\mathcal{X}(Q_{\pi,q^*})) \geq d(\mathcal{X}(Q_{\pi,q^*}), q^*) \geq d(\mathcal{X}(S^*), q^*) \geq \rho(\mathcal{X}(S^*)).$$

To show $N\rho(\mathcal{X}(Q_{\pi,q^*})) \geq d(\mathcal{X}(Q_{\pi,q^*}), q^*)$, note $N\rho(\mathcal{X}(Q_{\pi,q^*})) = \frac{sum(\mathcal{X}(Q_{\pi,q^*}))N}{|Q_{\pi,q^*}|}$, and since $\mathcal{X}$ is an N-way tensor, each entry is included in N slices. Hence

$$\sum\nolimits_{q\in Q_{\pi,q^*}} d(\mathcal{X}(Q_{\pi,q^*}), q) = sum(\mathcal{X}(Q_{\pi,q^*}))N. \tag{13.2}$$

Since π is a D-ordering, $\forall q \in Q_{\pi,q^*}$, $d(\mathcal{X}(Q_{\pi,q^*}), q) \geq d(\mathcal{X}(Q_{\pi,q^*}), q^*)$ holds. Combining this and Eq. (13.2) gives

$$N\rho(\mathcal{X}(Q_{\pi,q^*})) = \frac{sum(\mathcal{X}(Q_{\pi,q^*}))N}{|Q_{\pi,q^*}|} = \frac{\sum_{q\in Q_{\pi,q^*}} d(\mathcal{X}(Q_{\pi,q^*}), q)}{|Q_{\pi,q^*}|} \geq d(\mathcal{X}(Q_{\pi,q^*}), q^*).$$

Second, $d(\mathcal{X}(Q_{\pi,q^*}), q^*) \geq d(\mathcal{X}(S^*), q^*)$ is from that $S^* \subset Q_{\pi,q^*}$. Third, $d(\mathcal{X}(S^*), q^*) \geq \rho(\mathcal{X}(S^*))$ is from Lemma 13.1. From these, $\rho(\mathcal{X}(Q_{\pi,q^*})) \geq \frac{1}{N}\rho(\mathcal{X}(S^*))$ holds. ■

13.3.2 Baseline Algorithm: DENSESTATIC

We present DENSESTATIC, an algorithm for detecting a dense subtensor in a static tensor. Although it eventually finds the same subtensor as M-ZOOM (Chapter 11), DENSESTATIC also computes extra information, including a *D-ordering* (Definition 13.2), required for updating the subtensor in the following sections. As described in Algorithm 13.1, DENSESTATIC has two parts: (a) **D-ordering**: find a D-ordering π and compute $d_\pi(\cdot)$ and $c_\pi(\cdot)$; and (b) **Find-Slices**: find slices forming a dense subtensor from the result of (a). DENSESTATIC detects a subtensor $\mathcal{X}(S_{max})$ of density at least 1/(order of the input tensor) of maximum density, as formalized in Theorem 13.1 in Section 13.4.1.

Algorithm 13.1 DENSESTATIC: Dense-subtensor detection in a static tensor

Input: a tensor $\mathcal{X}$ with slice indices Q
Output: a dense subtensor $\mathcal{X}(S_{max})$

compute $\pi(\cdot)$, $d_\pi(\cdot)$, $c_\pi(\cdot)$ by D-ORDERING()
$S_{max} \leftarrow$ FIND-SLICES()
return $\mathcal{X}(S_{max})$
procedure D-ORDERING(): ▷ to find a D-ordering $\pi(\cdot)$ and compute $d_\pi(\cdot)$ and $c_\pi(\cdot)$
    $S \leftarrow Q$; $c_{max} \leftarrow 0$ ▷ c_{max}: max. $d_\pi(\cdot)$ so far
    for $j \leftarrow 1 ... |Q|$ **do**
        $q \leftarrow \arg\min_{r \in S} d(\mathcal{X}(S), r)$ ▷ q has min. slice sum
        $\pi(j) \leftarrow q$ ▷ $S = Q_{\pi,q}$
        $d_\pi(q) \leftarrow d(\mathcal{X}(S), q)$ ▷ $d_\pi(q) = d(\mathcal{X}(Q_{\pi,q}), q)$
        $c_\pi(q) \leftarrow \max(c_{max}, d_\pi(q))$; $c_{max} \leftarrow c_\pi(q)$
        $S \leftarrow S / \{q\}$
procedure FIND-SLICES(): ▷ to find slices forming a dense subtensor from $\pi(\cdot)$, $d_\pi(\cdot)$, and $c_\pi(\cdot)$
    $S \leftarrow \emptyset$; $m \leftarrow 0$; ▷ m: $sum(\mathcal{X}(S))$
    $\rho_{max} \leftarrow -\infty$; $q_{max} \leftarrow 0$ ▷ ρ_{max}: max. density so far
    for $j \leftarrow |Q| .. 1$ **do**
        $q \leftarrow \pi(j)$; $S \leftarrow S \cup \{q\}$ ▷ $S = Q_{\pi,q}$
        $m \leftarrow m + d_\pi(q)$ ▷ $m = sum(\mathcal{X}(Q_{\pi,q}))$
        if $m/|S| > \rho_{max}$ **then** ▷ $m/|S| = \rho(\mathcal{X}(Q_{\pi,q}))$
            $\rho_{max} \leftarrow m/|S|$; $q_{max} \leftarrow q$
    return $Q_{\pi,q_{max}}$ ▷ $q_{max} = \arg\max_{q \in Q} \rho(\mathcal{X}(Q_{\pi,q}))$

13.3.3 Proposed Algorithm (1): DENSESTREAM

How can we update the subtensor found in Algorithm 13.1 under changes in the input tensor, rapidly, only when necessary, with the same approximation bound? For this purpose, we propose DENSESTREAM, which updates the subtensor while satisfying Property 13.1. We explain the responses of DENSESTREAM to increments of entry values (Section 13.3.3.1), decrements of entry values (Section 13.3.3.2), and changes of the size of the input tensor (Section 13.3.3.3).

Property 13.1: Invariants in DENSESTREAM

For an N-way tensor $\mathcal{X}$ that keeps changing, the ordering π of the slice indices and the dense subtensor $\rho(\mathcal{X}(S_{max}))$ maintained by DENSESTREAM satisfy the following two conditions:

- π is a D-ordering of Q in $\mathcal{X}$
- $\rho(\mathcal{X}(S_{max})) \geq \rho_{opt}/N$.

13.3.3.1 Increment of Entry Values.

Assume that the maintained dense subtensor $\mathcal{X}(S_{max})$ and ordering π (with $d_\pi(\cdot)$ and $c_\pi(\cdot)$) satisfy Property 13.1 in the current tensor $\mathcal{X}$ (such π, $d_\pi(\cdot)$, $c_\pi(\cdot)$, and $\mathcal{X}(S_{max})$ can be initialized by Al-

Algorithm 13.2 DENSESTREAM in the case of increment

Input: (1) current tensor: $\mathcal{X}$ with slice indices Q
(2) current dense subtensor: $\mathcal{X}(S_{max})$ with Property 13.1
(3) current D-ordering: $\pi(\cdot)$ (also $d_\pi(\cdot)$ and $c_\pi(\cdot)$)
(4) a change in $\mathcal{X}$: $((i_1, ..., i_N), \delta, +)$
Output: updated dense subtensor $\mathcal{X}(S_{max})$

1: $x_{i_1 \cdots i_N} \leftarrow x_{i_1 \cdots i_N} + \delta$
2: compute j_L and j_H by Eq. (13.3) and Eq. (13.4) ▷ [FIND-REGION]
3: compute R by Eq. (13.5) ▷ [REORDER]
4: $S \leftarrow \{q \in Q : \pi^{-1}(q) \geq j_L\}$; $RS \leftarrow R \cap S$
5: $c_{max} \leftarrow 0$ ▷ c_{max}: max. $d_\pi(\cdot)$ so far
6: **if** $j_L > 1$ **then** $c_{max} \leftarrow c_\pi(\pi(j_L - 1))$
7: **for** $j \leftarrow j_L ... j_H$ **do**
8: $q \leftarrow \arg\min_{r \in RS} d(\mathcal{X}(S), r)$ ▷ q has min. slice sum
9: $\pi(j) \leftarrow q$ ▷ by Lemma 13.3, $S = Q_{\pi,q}$, $RS = R \cap Q_{\pi,q}$
10: $d_\pi(q) \leftarrow d(\mathcal{X}(S), q)$ ▷ $d_\pi(q) = d(\mathcal{X}(Q_{\pi,q}), q)$
11: $c_\pi(q) \leftarrow \max(c_{max}, d_\pi(q))$; $c_{max} \leftarrow c_\pi(q)$
12: $S \leftarrow S/\{q\}$; $RS \leftarrow RS/\{q\}$
13: **if** $c_{max} \geq \rho(\mathcal{X}(S_{max}))$ **then** ▷ [UPDATE-SUBTENSOR]
14: $S' \leftarrow$FIND-SLICES() in Algorithm 13.1 ▷ time complexity: $O(|Q|)$
15: **if** $S_{max} \neq S'$ **then** $\mathcal{X}(S_{max}) \leftarrow \mathcal{X}(S')$
16: **return** $\mathcal{X}(S_{max})$

gorithm 13.1 if we start from scratch). Algorithm 13.2 describes the response of DENSESTREAM to $((i_1, ..., i_N), \delta, +)$, an increment of entry $x_{i_1 \cdots i_N}$ by $\delta > 0$, for satisfying Property 13.1. Algorithm 13.2 has three steps:(a) **Find-Region**: find a region of the D-ordering π that needs to be reordered, (b) **Reorder**: reorder the region obtained in (a), and (c) **Update-Subtensor**: use π to rapidly update $\mathcal{X}(S_{max})$ only when necessary. Each step is explained below.

(a) Find-Region (Line 2): The goal of this step is to find the region $[j_L, j_H] \subset [1, |Q|]$ of the domain of the D-ordering π that needs to be reordered in order for π to remain as a D-ordering after the change $((i_1, ..., i_N), \delta, +)$. Let $C = \{(n, i_n) : n \in [N]\}$ be the indices of the slices composing the changed entry $x_{i_1 \cdots i_N}$ and let $q_f = \arg\min_{q \in C} \pi^{-1}(q)$ be the one located first in π among C. Then, let $M = \{q \in Q : \pi^{-1}(q) > \pi^{-1}(q_f),\ d_\pi(q) \geq d_\pi(q_f) + \delta\}$ be the slice indices that are located after q_f in π among Q and having $d_\pi(\cdot)$ at least $d_\pi(q_f) + \delta$. Then, j_L and j_H are set as follows:

$$j_L = \pi^{-1}(q_f), \tag{13.3}$$

$$j_H = \begin{cases} \min_{q \in M} \pi^{-1}(q) - 1 & \text{if } M \neq \emptyset, \\ |Q| \text{ (i.e., the last index)} & \text{otherwise.} \end{cases} \tag{13.4}$$

Later in this section, we prove that slice indices whose locations do not belong to $[j_L, j_H]$ need not be reordered by showing that there always exists a D-ordering π' in the updated $\mathcal{X}$ where $\pi'(j) = \pi(j)$ for every $j \notin [j_L, j_H]$.

(b) Reorder (Lines 3-12): The goal of this step is to reorder the slice indices located in the region $[j_L, j_H]$ so that π remains as a D-ordering in $\mathcal{X}$ after the change $((i_1, ..., i_N), \delta, +)$. Let $\mathcal{X}'$ be the

updated $\mathcal{X}$ and π' be the updated π to distinguish them with $\mathcal{X}$ and π before the update. We get π' from π by reordering the slice indices in

$$R = \{q \in Q : \pi^{-1}(q) \in [j_L, j_H]\} \tag{13.5}$$

so that the following condition is met for every $j \in [j_L, j_H]$ and the corresponding $q = \pi'(j)$:

$$d(\mathcal{X}'(Q_{\pi',q}), q) = \min_{r \in R \cap Q_{\pi',q}} d(\mathcal{X}'(Q_{\pi',q}), r). \tag{13.6}$$

This guarantees that π' is a D-ordering in $\mathcal{X}'$, as shown in Lemma 13.3.

Lemma 13.3:

Let π be a D-ordering in $\mathcal{X}$, and let $\mathcal{X}'$ be $\mathcal{X}$ after a change $((i_1, ..., i_N), \delta, +)$. For R (Eq. (13.5)) defined on j_L and j_H (Eq. (13.3) and Eq. (13.4)), let π' be an ordering of slice indices Q where $\forall j \notin [j_L, j_H], \pi'(j) = \pi(j)$ and $\forall j \in [j_L, j_H]$, Eq. (13.6) holds. Then, π' is a D-ordering in $\mathcal{X}'$.

Proof. See Section 13.7.1. ■

(c) Update-Subtensor (Lines 13-15): In this step, we update the maintained dense subtensor $\mathcal{X}(S_{max})$ when two conditions are met. We first check $c_{max} \geq \rho(\mathcal{X}(S_{max}))$, which takes $O(1)$ time if we maintain $\rho(\mathcal{X}(S_{max}))$, since $c_{max} < \rho(\mathcal{X}(S_{max}))$ entails that the updated entry $x_{i_1 \cdots i_N}$ is not in the densest subtensor (see the proof of Theorem 13.2 for details). We then check if there are changes in S_{max}, obtained by FIND-SLICES(). This takes only $O(|Q|)$ time, as shown in Theorem 13.5. Even if both conditions are met, updating $\mathcal{X}(S_{max})$ is simply to construct $\mathcal{X}(S_{max})$ from given S_{max} instead of finding $\mathcal{X}(S_{max})$ from scratch. This conditional update reduces computation but still preserves Property 13.1, as formalized in Theorem 13.2 in Section 13.4.1.

13.3.3.2 Decrement of Entry Values.

As in the previous section, assume that a tensor $\mathcal{X}$, a D-ordering π (also $d_\pi(\cdot)$ and $c_\pi(\cdot)$), and a dense subtensor $\mathcal{X}(S_{max})$ satisfying Property 13.1 are maintained. (such π, $d_\pi(\cdot)$, $c_\pi(\cdot)$, and $\mathcal{X}(S_{max})$ can be initialized by Algorithm 13.1 if we start from scratch). Algorithm 13.3 describes the response of DENSESTREAM to $((i_1, ..., i_N), \delta, -)$, a decrement of entry $x_{i_1 \cdots i_N}$ by $\delta > 0$, for satisfying Property 13.1. Algorithm 13.3 has the same structure of Algorithm 13.2, while they are different in the reordered region of π and the conditions for updating the dense subtensor. The differences are explained below.

For a change $((i_1, ..., i_N), \delta, -)$, we find the region $[j_L, j_H]$ of the domain of π that may need to be reordered. Let $C = \{(n, i_n) : n \in [N]\}$ be the indices of the slices composing the changed entry $x_{i_1 \cdots i_N}$, and let $q_f = \arg\min_{q \in C} \pi^{-1}(q)$ be the one located first in π among C. Then, let $M_{min} = \{q \in Q : d_\pi(q) > c_\pi(q_f) - \delta\}$ and $M_{max} = \{q \in Q : \pi^{-1}(q) > \pi^{-1}(q_f),\ d_\pi(q) \geq c_\pi(q_f)\}$. Note that $M_{min} \neq \emptyset$ since, by the definition of $c_\pi(\cdot)$, there exists $q \in Q$ where $\pi^{-1}(q) \leq \pi^{-1}(q_f)$ and $d_\pi(q) = c_\pi(q_f)$. Then, j_L and j_H are:

$$j_L = \min_{q \in M_{min}} \pi^{-1}(q), \tag{13.7}$$

$$j_H = \begin{cases} \min_{q \in M_{max}} \pi^{-1}(q) - 1 & \text{if } M_{max} \neq \emptyset, \\ |Q| \text{ (i.e., the last index)} & \text{otherwise.} \end{cases} \tag{13.8}$$

Algorithm 13.3 DENSESTREAM in the case of decrement

Input: (1) current tensor: $\mathcal{X}$ with slice indices Q
(2) current dense subtensor: $\mathcal{X}(S_{max})$ with Property 13.1
(3) current D-ordering: $\pi(\cdot)$ (also $c_\pi(\cdot)$ and $d_\pi(\cdot)$)
(4) a change in $\mathcal{X}$: $((i_1, ..., i_N), \delta, -)$
Output: updated dense subtensor$\mathcal{X}(S_{max})$

1: $x_{i_1 \cdots i_N} \leftarrow x_{i_1 \cdots i_N} - \delta$
2: compute j_L and j_H by Eq. (13.7) and (13.8) ▷ [FIND-REGION]
3: Lines 3-12 of Algorithm 13.2 ▷ [REORDER]
4: **if** $x_{i_1 \cdots i_N}$ is in $\mathcal{X}(S_{max})$ **then** ▷ [UPDATE-SUBTENSOR]
5: $S' \leftarrow$FIND-SLICES() in Algorithm 13.1 ▷ time complexity: $O(|Q|)$
6: **if** $S_{max} \neq S'$ **then** $\mathcal{X}(S_{max}) \leftarrow \mathcal{X}(S')$
7: **return** $\mathcal{X}(S_{max})$

As in the increment case, we update π, to remain it as a D-ordering, by reordering the slice indices located in $[j_L, j_H]$ of π. Let $\mathcal{X}'$ be the updated $\mathcal{X}$ and π' be the updated π to distinguish them with $\mathcal{X}$ and π before the update. Only the slice indices in $R = \{q \in Q : \pi^{-1}(q) \in [j_L, j_H]\}$ are reordered in π so that Eq. (13.6) is met for every $j \in [j_L, j_H]$. This guarantees that π' is a D-ordering, as formalized in Lemma 13.4.

Lemma 13.4:

Let π be a D-ordering in $\mathcal{X}$, and let $\mathcal{X}'$ be $\mathcal{X}$ after a change $((i_1, ..., i_N), \delta, -)$. For R (Eq. (13.5)) defined on j_L and j_H (Eq. (13.7) and Eq. (13.8)), let π' be an ordering of slice indices Q where $\forall j \notin [j_L, j_H], \pi'(j) = \pi(j)$ and $\forall j \in [j_L, j_H]$, Eq. (13.6) holds. Then, π' is a D-ordering in $\mathcal{X}'$.

Proof. See Section 13.7.2. ■

The last step of Algorithm 13.3 is to conditionally and rapidly update the maintained dense subtensor $\mathcal{X}(S_{max})$ using π. The subtensor $\mathcal{X}(S_{max})$ is updated if entry $x_{i_1 \cdots i_N}$ belongs to $\mathcal{X}(S_{max})$ (i.e., if $\rho(\mathcal{X}(S_{max}))$ decreases by the change $((i_1, ..., i_N), \delta, -)$) and there are changes in S_{max}, obtained by FIND-SLICES(). Checking these conditions takes only $O(|Q|)$ time, as in the increment case. Even if $\mathcal{X}(S_{max})$ is updated, it is just constructing $\mathcal{X}(S_{max})$ from given S_{max}, instead of finding $\mathcal{X}(S_{max})$ from scratch.

Algorithm 13.3 preserves Property 13.1, as formalized in Theorem 13.3 in Section 13.4.1.

13.3.3.3 Increase or Decrease of Size.

DENSESTREAM also supports the increase and decrease of the size of the input tensor. The increase of the size of $\mathcal{X}$ corresponds to the addition of new slices to $\mathcal{X}$. For example, if the length of the nth mode of $\mathcal{X}$ increases from I_n to $I_n + 1$, the index $q = (n, I_n + 1)$ of the new slice is added to Q and the first position of π. We also set $d_\pi(q)$ and $c_\pi(q)$ to 0. Then, if there exist non-zero entries in the new slice, they are handled one by one by Algorithm 13.2. Likewise, when size decreases, we first handle the

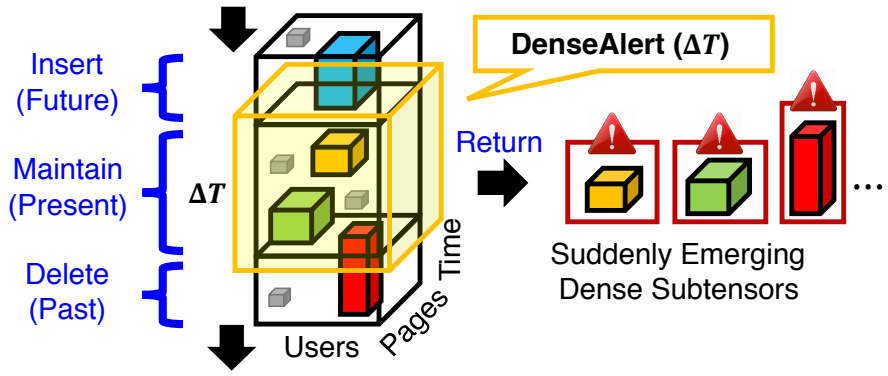


Figure 13.3: **Illustration of DENSEALERT** running on a Wikipedia revision history (Example 13.1). DENSEALERT (yellow box in the figure) spots dense subtensors formed within ΔT time units.

removed non-zero entries one by one by Algorithm 13.3. Then, we remove the indices of the removed slices from Q and π.

13.3.4 Proposed Algorithm (2): DENSEALERT

Based on DENSESTREAM, we propose DENSEALERT, an incremental algorithm for detecting suddenly emerging dense subtensors. For a stream of increments in the input tensor $\mathcal{X}$, DENSEALERT maintains $\mathcal{X}_{\Delta T}$, a tensor where the value of each entry is the increment of the value of the corresponding entry in $\mathcal{X}$ in last ΔT time units (see Problem 13.2 in Section 13.2.2), as described in Figure 13.3 and Algorithm 13.4, To maintain $\mathcal{X}_{\Delta T}$ and a dense subtensor in it, DENSEALERT applies increments by DENSESTREAM (line 4), and undoes the increments after ΔT time units also by DENSESTREAM (lines 5 and 7). The accuracy of DENSEALERT is implied from the accuracy of DENSESTREAM, as formalized in Theorem 13.4 in Section 13.4.1.

Algorithm 13.4 DENSEALERT for sudden dense subtensors

Input: (1) sequence of increments in $\mathcal{X}$
(2) time window: ΔT
Output: suddenly emerging dense subtensors
1: run Algorithm 13.1 with a zero tensor
2: wait until the next change happens at time T
3: **if** the change is $((i_1, ..., i_N), \delta, +)$ **then**
4: run DENSESTREAM (Algorithm 13.2)
5: schedule $((i_1, ..., i_N), \delta, -)$ at time $T + \Delta T$
6: **else if** the change is $((i_1, ..., i_N), \delta, -)$ **then**
7: run DENSESTREAM (Algorithm 13.3)
8: report the current dense subtensor
9: **goto** Line 2

13.4 Theoretical Analysis

We theoretically analyze the accuracies, time complexities, and space complexities of the algorithms discussed in the previous section.

13.4.1 Accuracy Analysis

We prove that DENSESTATIC, DENSESTREAM, and DENSEALERT guarantee an approximation ratio of $1/N$ for the N-way input tensor in Theorems 13.1-13.4.

Theorem 13.1: Accuracy Guarantee of DENSESTATIC

The subtensor returned by Algorithm 13.1 has density at least ρ_{opt}/N.

Proof. By Lemma 13.2, there exists a subtensor with density at least ρ_{opt}/N among $\{\mathcal{X}(Q_{\pi,q}) : q \in Q\}$. The subtensor with the highest density in the set is returned by Algorithm 13.1. ■

Lemma 13.5:

Consider a D-ordering π in $\mathcal{X}$. For every entry $x_{i_1 \cdots i_N}$ with index $(i_1, ..., i_N)$ belonging to the densest subtensor, $\forall n \in [N]$, $c_\pi((n, i_n)) \geq \rho_{opt}$ holds.

Proof. Let $\mathcal{X}(S^*)$ be a subtensor with the maximum density, i.e., $\rho(\mathcal{X}(S^*)) = \rho_{opt}$. Let $q^* \in S^*$ be satisfying that $\forall q \in S^*$, $\pi^{-1}(q^*) \leq \pi^{-1}(q)$. For any entry $x_{i_1 \cdots i_N}$ in $\mathcal{X}(S^*)$ with index $(i_1, ..., i_N)$ and any $q \in \{(n, i_n) : n \in [N]\}$, our goal is to show $c_\pi(q) \geq \rho(\mathcal{X}(S^*))$, which we show as $c_\pi(q) \geq d_\pi(q^*) \geq d(\mathcal{X}(S^*), q^*) \geq \rho(\mathcal{X}(S^*))$.

First, $c_\pi(q) \geq d_\pi(q^*)$ is from the definition of $c_\pi(q)$ and $\pi^{-1}(q^*) \leq \pi^{-1}(q)$. Second, from $S^* \subset Q_{\pi,q^*}$, $d_\pi(q^*) = d(\mathcal{X}(Q_{\pi,q^*}), q^*) \geq d(\mathcal{X}(S^*), q^*)$ holds. Third, $d(\mathcal{X}(S^*), q^*) \geq \rho(\mathcal{X}(S^*))$ is from Lemma 13.1. From these, $c_\pi(q) \geq \rho(\mathcal{X}(S^*))$ holds. ■

Theorem 13.2: Accuracy Guarantee of DENSESTREAM (the Case of Increment)

Algorithm 13.2 preserves Property 13.1, and thus $\rho(\mathcal{X}(S_{max})) \geq \rho_{opt}/N$ holds after Algorithm 13.2 terminates.

Proof. We assume that Property 13.1 holds and prove that it still holds after Algorithm 13.2 is executed. First, the ordering π remains to be a D-ordering in $\mathcal{X}$ by Lemma 13.3. Second, we show $\rho(\mathcal{X}(S_{max})) \geq \rho_{opt}/N$. If the condition in line 13 of Algorithm 13.2 is met, $\mathcal{X}(S_{max})$ is set to the subtensor with the maximum density in $\{\mathcal{X}(Q_{\pi,q}) : q \in Q\}$ by FIND-SLICES(). By Lemma 13.2, $\rho(\mathcal{X}(S_{max})) \geq \rho_{opt}/N$. If the condition in line 13 is not met, for the changed entry $x_{i_1 \cdots i_N}$ with index $(i_1, ..., i_N)$, by

the definition of j_L, there exists $n \in [N]$ such that $\pi(j_L) = (n, i_n)$. Since $j_L \leq j_H$, $c_\pi((n, i_n)) = c_\pi(\pi(j_L)) \leq c_\pi(\pi(j_H)) = c_{max} < \rho(\mathcal{X}(S_{max})) \leq \rho_{opt}$. Then, by Lemma 13.5, $x_{i_1 \cdots i_N}$ does not belong to the densest subtensor, which thus remains the same after the change $((i_1, ..., i_N), \delta, +)$. Since $\rho(\mathcal{X}(S_{max}))$ never decreases, $\rho(\mathcal{X}(S_{max})) \geq \rho_{opt}/N$ still holds by Property 13.1, which we assume. Property 13.1 is preserved because its two conditions are met. ■

Theorem 13.3: Accuracy Guarantee of DENSESTREAM (the Case of Decrement)

Algorithm 13.3 preserves Property 13.1. Thus, $\rho(\mathcal{X}(S_{max})) \geq \rho_{opt}/N$ holds after Algorithm 13.3 terminates.

Proof. We assume that Property 13.1 holds and prove that it still holds after Algorithm 13.3 is executed. First, the ordering π remains to be a D-ordering in $\mathcal{X}$ by Lemma 13.4. Second, we show $\rho(\mathcal{X}(S_{max})) \geq \rho_{opt}/N$. If the condition in line 4 of Algorithm 13.3 is met, $\mathcal{X}(S_{max})$ is set to the subtensor with the maximum density in $\{\mathcal{X}(Q_{\pi,q}) : q \in Q\}$ by FIND-SLICES(). By Lemma 13.2, $\rho(\mathcal{X}(S_{max})) \geq \rho_{opt}/N$. If the condition is not met, $\rho(\mathcal{X}(S_{max}))$ remains the same, while ρ_{opt} never increases. Hence, $\rho(\mathcal{X}(S_{max})) \geq \rho_{opt}/N$ still holds by Property 13.1, which we assume. Since its two conditions are met, Property 13.1 is preserved. ■

Theorem 13.4: Accuracy Guarantee of DENSEALERT

Let $\Delta\rho_{opt}$ be the density of the densest subtensor in the N-way tensor $\mathcal{X}_{\Delta T}$. The subtensor maintained by Algorithm 13.4 has density at least $\Delta\rho_{opt}/N$.

Proof. By Theorems 13.2 and 13.3, DENSEALERT, which uses DENSESTREAM for updates, maintains a subtensor with density at least $1/N$ of the density of the densest subtensor. ■

13.4.2 Complexity Analysis

We prove the time and space complexities of DENSESTATIC, DENSESTREAM, and DENSEALERT.

13.4.2.1 Time Complexity Analysis

The time complexity of DENSESTATIC, described in Algorithm 13.1, is linear with $nnz(\mathcal{X})$, the number of the non-zero entries in $\mathcal{X}$, as formalized in Theorem 13.5. Especially, finding S_{max} takes only $O(|Q|)$ time given $\pi(\cdot)$, $d_\pi(\cdot)$, and $c_\pi(\cdot)$, as shown in Lemma 13.6.

Lemma 13.6:

Let S_{max} be the set of slice indices returned by FIND-SLICES() in Algorithm 13.1. Let $\mathcal{X}(q)$ be the set of the non-zero entries in the slice with index q in $\mathcal{X}$. The time complexity of FIND-SLICES() in Algorithm 13.1 is $O(|Q|)$ and that of constructing $\mathcal{X}(S_{max})$ from S_{max} is $O(N|\bigcup_{q \in S_{max}} \mathcal{X}(q)|)$.

Proof. Assume that, for each slice, the list of the non-zero entries in the slice is stored. In FIND-SLICES(), we iterate over the slices in Q, and each iteration takes $O(1)$ time. Thus, this results in $O(|Q|)$ time. After finding S_{max}, in order to construct $\mathcal{X}(S_{max})$, we have to process each non-zero entry included in any slice in S_{max}. The number of such entries is $|\bigcup_{q \in S_{max}} \mathcal{X}(q)|$. Since processing each entry takes $O(N)$ time, constructing $\mathcal{X}(S_{max})$ takes $O(N|\bigcup_{q \in S_{max}} \mathcal{X}(q)|)$ time. ■

Theorem 13.5: Time Complexity of DENSESTATIC

The time complexity of Algorithm 13.1 is $O(|Q| \log |Q| + N \cdot nnz(\mathcal{X}))$.

Proof. Assume that, for each slice, the list of the non-zero entries in the slice is stored. We first show that the time complexity of D-ORDERING() in Algorithm 13.1 is $O(|Q| \log |Q| + N \cdot nnz(\mathcal{X})|)$. Assume we use a Fibonacci heap to find slices with minimum slice sum (line 7). Computing the slice sum of every slice takes $O(N \cdot nnz(\mathcal{X}))$ time, and constructing a Fibonacci heap where each value is a slice index in Q and the corresponding key is the slice sum of the slice takes $O(|Q|)$ time. Popping the index of a slice with minimum slice sum, which takes $O(\log |Q|)$ time, happens $|Q|$ times, and thus this results in $O(|Q| \log |Q|)$ time. Whenever a slice index is popped, we have to update the slice sums of its dependent slices (two slices are dependent if they have common non-zero entries). Updating the slice sum of each dependent slice, which takes $O(1)$ time in a Fibonacci heap, happens at most $O(N \cdot nnz(\mathcal{X}))$ times, and thus this results in $O(N \cdot nnz(\mathcal{X}))$ time. Thus, D-ORDERING() takes $O(|Q| \log |Q| + N \cdot nnz(\mathcal{X}))$ time in total.

By Lemma 13.6, the time complexity of FIND-SLICES() is $O(|Q|)$, and that of constructing $\mathcal{X}(S_{max})$ from S_{max} is $O(N|\bigcup_{q \in S_{max}} \mathcal{X}(q)|)$.

Since the time complexity of D-ORDERING() dominates that of the remaining parts, the time complexity of Algorithm 13.1 is $O(|Q| \log |Q| + N \cdot nnz(\mathcal{X}))$. ■

Theorem 13.6 gives the time complexity of DENSESTREAM. In the worst case (i.e., $R = Q$), this becomes $O(|Q| \log |Q| + N \cdot nnz(\mathcal{X}))$, which is the time complexity of DENSESTATIC. In practice, however, R is much smaller than Q, and updating $\mathcal{X}(S_{max})$ happens rarely. Thus, in our experiments, scaled sub-linearly with $nnz(\mathcal{X})$ (see Section 13.5.4).

Theorem 13.6: Time Complexity of DENSESTREAM

Let $\mathcal{X}(q)$ be the set of the non-zero entries in the slice with index q in $\mathcal{X}$. The time complexity of Algorithms 13.2 and 13.3 is $O(|R| \log |R| + |Q| + N|\bigcup_{q \in R} \mathcal{X}(q)| + N|\bigcup_{q \in S_{max}} \mathcal{X}(q)|)$.

Proof. Assume that, for each slice, the list of the non-zero entries in the slice is stored, and let $q_f = \pi(j_L)$. Computing j_L, j_H, and R takes $O(|R|)$ time. Assume we use a Fibonacci heap to find slices with minimum slice sum (line 8 of Algorithm 13.2). Computing the slice sum of every slice in R in $\mathcal{X}(Q_{\pi,q_f})$ takes $O(N|\bigcup_{q\in R}\mathcal{X}(q)|)$ time. Then, constructing a Fibonacci heap where each value is a slice index in R and the corresponding key is the slice sum of the slice in $\mathcal{X}(Q_{\pi,q_f})$ takes $O(|R|)$ time. Popping the index of a slice with minimum slice sum, which takes $O(\log|R|)$ time, happens $|R|$ times, and thus this results in $O(|R|\log|R|)$ time. Whenever a slice index is popped we have to update the slice sums of its dependent slices in R (two slices are dependent if they have common non-zero entries). Updating the slice sum of each dependent slice, which takes $O(1)$ time in a Fibonacci heap, happens at most $O(N|\bigcup_{q\in R}\mathcal{X}(q)|)$ times, and thus this results in $O(N|\bigcup_{q\in R}\mathcal{X}(q)|)$ time. On the other hand, by Lemma 13.6, FIND-SLICES() and constructing $\mathcal{X}(S_{max})$ from S_{max} take $O(|Q|+N|\bigcup_{q\in S_{max}}\mathcal{X}(q)|)$ time. Hence, the time complexity of Algorithms 13.2 and 13.3 is the sum of all the times taken, which is $O(|R|\log|R|+|Q|+N|\bigcup_{q\in R}\mathcal{X}(q)|+N|\bigcup_{q\in S_{max}}\mathcal{X}(q)|)$. ■

The time complexity of DENSEALERT is obtained Theorem 13.6 by simply replacing $\mathcal{X}$ with $\mathcal{X}_{\Delta T}$.

13.4.2.2 Space Complexity Analysis

DENSESTATIC (Algorithm 13.1) and DENSESTREAM (Algorithms 13.2 and 13.3) requires $O(N \cdot nnz(\mathcal{X}))$ space for the input and output tensors; and $O(|Q|)$ space for π, $d_\pi(\cdot)$, and $c_\pi(\cdot)$. From $|Q| \leq N \cdot nnz(\mathcal{X})$, their space complexity is $O(N \cdot nnz(\mathcal{X}))$. DENSEALERT needs to store only $\mathcal{X}_{\Delta T}$ (i.e., the changes in the last ΔT units) in memory at a time while discarding older changes. Thus, the space complexity of DENSEALERT, which essentially runs DENSESTREAM on $\mathcal{X}_{\Delta T}$, is $O(N \cdot nnz(\mathcal{X}_{\Delta T}))$.

13.5 Experiments

We design experiments to answer the following questions:

- **Q1. Speed**: How fast are updates in DENSESTREAM compared to batch algorithms?
- **Q2. Accuracy**: How accurately does DENSESTREAM maintain a dense subtensor?
- **Q3. Scalability**: How does the running time of DENSESTREAM increase as input tensors grow?
- **Q4. Effectiveness**: Which anomalies or fraudsters does DENSEALERT spot in real-world tensors?

13.5.1 Experimental Settings.

Machine: We ran all experiments on a machine with 2.67GHz Intel Xeon E7-8837 CPUs and 1TB memory (up to 85GB was used by our algorithms).

Datasets: Table 13.2 lists the real-world tensors used in our experiments. See Section 10.4 for a description of the datasets.

Implementations: We implemented dense-subtensor detection algorithms for comparison. Specifically, we implemented our algorithms, M-ZOOM (Chapter 11), and CROSSSPOT [JBC+16] in Java,

Table 13.2: **Summary of the real-world tensors used in our experiments.** M: Million, K: Thousand. The underlined attributes are composite primary keys.

Name	Size	$\lvert\mathbf{Q}\rvert$	$\mathbf{nnz}(\mathcal{X})$
Ratings: users × items × timestamps × ratings → #reviews			
Yelp	552K × 77.1K × 3.80K × 5	633K	2.23M
Android [MPL15]	1.32M × 61.3K × 1.28K × 5	1.39M	2.64M
YahooM. [DKKW12]	1.00M × 625K × 84.4K × 101	1.71M	253M
Wikipedia edit history: users × pages × timestamps → #edits			
KoWiki [SHF18]	470K × 1.18M × 101K	1.80M	11.0M
EnWiki [SHF18]	44.1M × 38.5M × 129K	82.8M	483M
Social networks: users × users × timestamps → #interactions			
Youtube [MMG+07]	3.22M × 3.22M × 203	6.45M	18.7M
SMS	1.25M × 7.00M × 4.39K	8.26M	103M
TCP dumps: IPs × IPs × timestamps → #connections			
Darpa [LFG+00]	9.48K × 23.4K × 46.6K	79.5K	522K

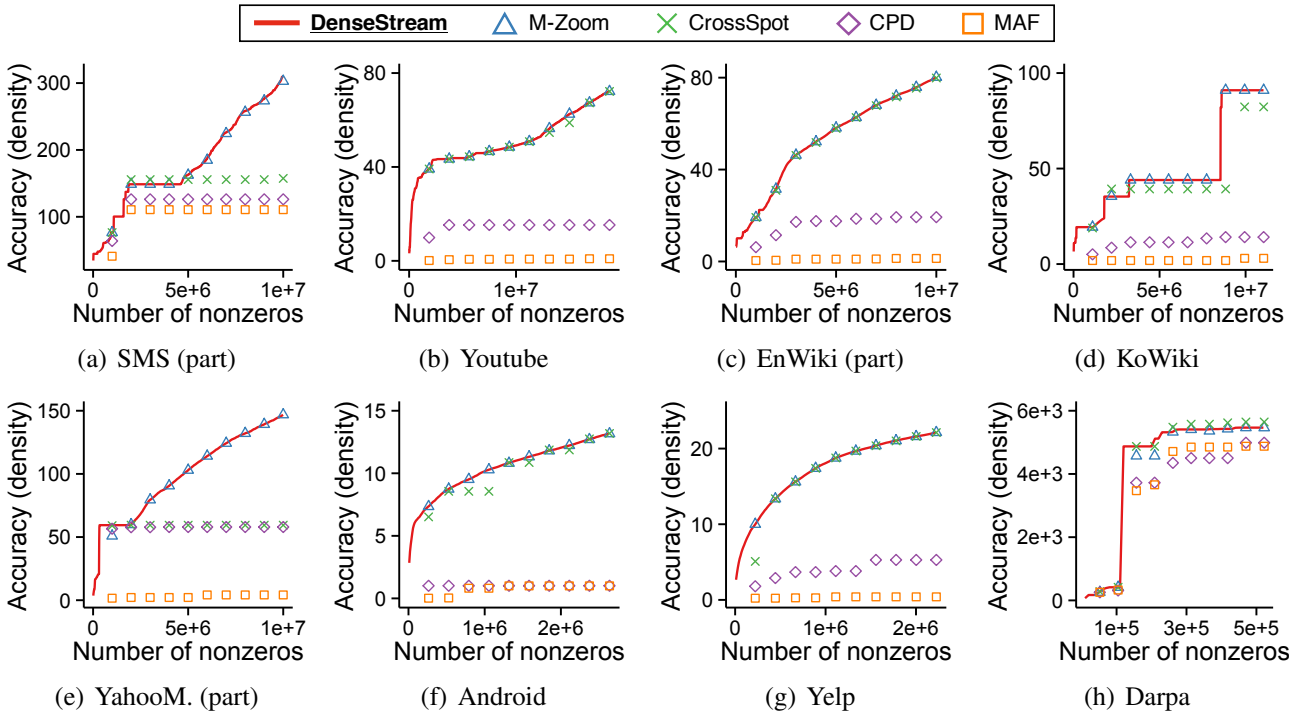
while we used Tensor Toolbox [BK07a] for CP decomposition (CPD)[1] and MAF [MGF11]. In all the implementations, a sparse tensor format was used so that the space usage is proportional to the number of non-zero entries. As in the previous chapters, we used a variant of CROSSSPOT which maximizes the density measure defined in Definition 13.1 and uses CPD for seed selection. For each batch algorithm, we reported the densest one after finding three dense subtensors.

13.5.2 Q1. Speed

Updates by DENSESTREAM are significantly faster than running batch algorithms from scratch. For each tensor stream, we averaged the update times for processing the last 10,000 changes corresponding to increments (blue bars in Figure 13.5). Likewise, we averaged the update times for undoing the first 10,000 increments, i.e., decreasing the values of the oldest entries (red bars in Figure 13.5). Then, we compared them to the time taken for running batch algorithms on the final tensor that each tensor stream results in. As seen in Figure 13.5, updates in DENSESTREAM were up to a million times faster than the fastest batch algorithm. The speed-up was particularly high in sparse tensors having a widespread slice sum distribution (thus having a small reordered region R), as we can expect from Theorem 13.6.

On the other hand, the update time in DENSEALERT, which uses DENSESTREAM as a sub-procedure, was similar to that in DENSESTREAM when the time interval $\Delta T = \infty$, and was less with smaller ΔT. This is since the average number of non-zero entries maintained is proportional to ΔT.

[1] Let $\mathbf{A}^{(1)} \in \mathbb{R}^{I_1 \times k}$, $\mathbf{A}^{(2)} \in \mathbb{R}^{I_2 \times k}$, ..., $\mathbf{A}^{(N)} \in \mathbb{R}^{I_N \times k}$ be the factor matrices obtained by the rank-k CP Decomposition of $\mathcal{X}$. For each $j \in [k]$, we form a subtensor with every slice with index (n, i_n) where the (i_n, j)-th entry of $\mathbf{A}^{(n)}$ is at least $1/\sqrt{I_n}$.



(a) SMS (part) (b) Youtube (c) EnWiki (part) (d) KoWiki

(e) YahooM. (part) (f) Android (g) Yelp (h) Darpa

Figure 13.4: **DENSESTREAM is 'any-time' and accurate.** While tensors grow, DENSESTREAM maintains and instantly updates a dense subtensor, whereas batch algorithms update dense subtensors only once in a time interval. Subtensors maintained by DENSESTREAM have density (red lines) similar to the density (points) of the subtensors found by the best batch algorithms.

13.5.3 Q2. Accuracy

DENSESTREAM is as accurate as the best batch algorithms. We tracked the density of the dense subtensor maintained by DENSESTREAM while each tensor grows,[2] and compared it to the densities of the dense subtensors found by batch algorithms. As seen in Figure 13.4, the subtensors that DENSESTREAM maintained had density (red lines) similar to the density (points) of the subtensors found by the best batch algorithms. Moreover, DENSESTREAM is 'any time'; that is, as seen in Figure 13.4, DENSESTREAM updates the dense subtensor instantly, while the batch algorithms cannot update their dense subtensors until the next batch processes end.

The accuracy and speed (measured in Section 13.5.2) of the algorithms in the Yelp dataset are shown in Figure 13.1(a) in Section 13.1. DENSESTREAM significantly reduces the time gap between the emergence and the detection of a dense subtensor, without losing accuracy.

Note that from the accuracy of DENSESTREAM, the accuracy of DENSEALERT, which uses DENSESTREAM as a sub-procedure, is also obtained.

13.5.4 Q3. Scalability

Each update by DENSESTREAM takes time sub-linear in the number of non-zeros. We measured how rapidly the update time of DENSESTREAM increases as the input tensor grows. For this experiment, we used a $10^5 \times 10^5 \times 10^5$ random tensor stream that has a realistic power-law slice sum

[2]We obtained similar results when the values of entries decrease.

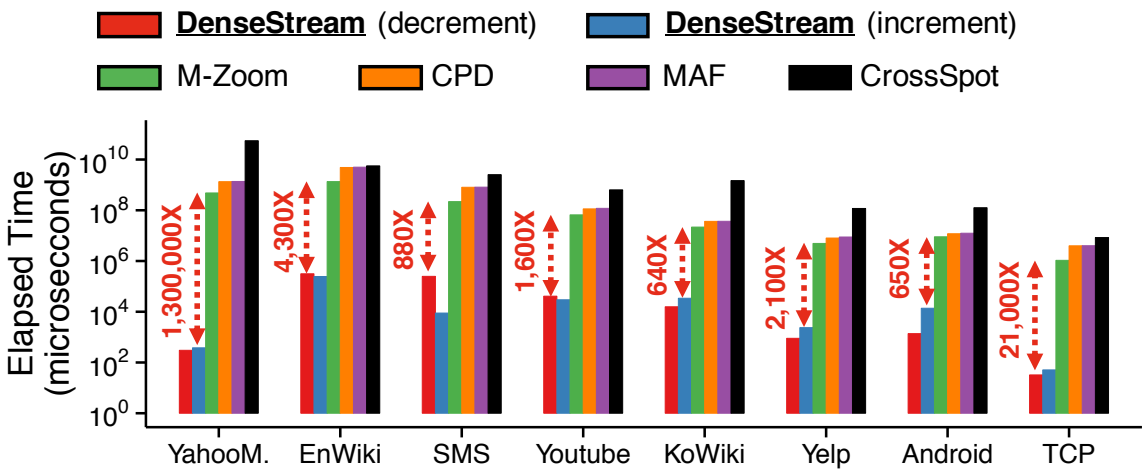


Figure 13.5: **DENSESTREAM rapidly updates dense subtensors.** An update by DENSESTREAM was up to *a million times faster* than the fastest batch algorithm.

distribution in each mode. As seen in Figure 13.1(b) in Section 13.1, update times, for both types of changes, scaled sub-linearly with the number of nonzero entries. Note that DENSEALERT, which uses DENSESTREAM as a sub-procedure, has the same scalability.

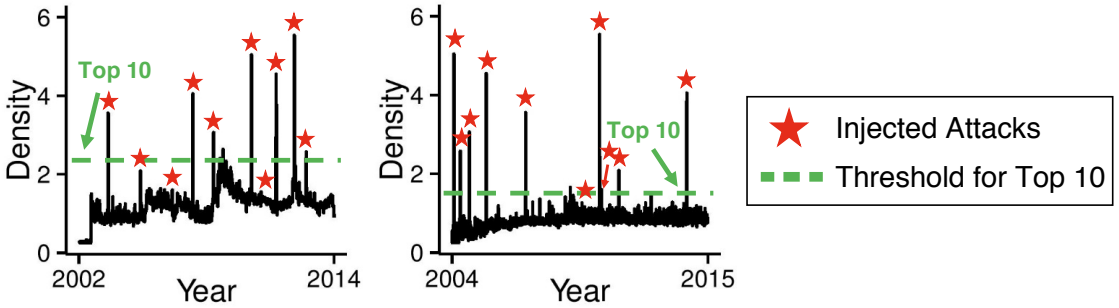
13.5.5 Q4. Effectiveness

In this section, we show the effectiveness of DENSEALERT in practice. We focus on DENSEALERT, which spots suddenly emerging dense subtensors overlooked by existing methods, rather than DENSESTREAM, which is much faster but eventually finds a similar subtensor with previous algorithms, especially M-ZOOM (Chapter 11).

13.5.5.1 Small-scale Attack Detection in Rating Data.

DENSEALERT reveals small-scale attacks in rating data most accurately. For rating datasets, where ground-truth labels are unavailable, we assumed an attack scenario where fraudsters in a rating site, such as Yelp, utilize multiple user accounts and give the same rating to the same set of items (or businesses) in a short period of time. The goal of the fraudsters is to boost (or lower) the ratings of the items rapidly. This lockstep behavior results in a dense subtensor of size $\#fake\_accounts \times \#target\_items \times 1 \times 1$ in rating datasets whose modes are users, items, timestamps, and ratings. Here, we assumed that fraudsters are not blatant but careful enough to adjust their behavior so that only small-scale dense subtensors are formed.

We injected 10 such small random dense subtensors of sizes from $3\times3\times1\times1$ to $12\times12\times1\times1$ in the Yelp and YahooM. datasets, and compared how many of them are detected by each anomaly-detection algorithm. As seen in Figure 13.6(a), DENSEALERT (with ΔT=1 time unit in each dataset) clearly revealed the injected subtensors. Specifically, 9 and 7 among the top 10 densest subtensors spotted by DENSEALERT indeed indicate the injected attacks in the Yelp and YahooM. datasets, respectively. However, the injected subtensors were not revealed when we simply investigated the number of ratings in each time unit. Moreover, as summarized in Figure 13.6(b), none of the injected subtensors was



(a) DENSEALERT in Yelp (left) and YahooM. (right)

	Recall @ Top-10 in Yelp	Recall @ Top-10 in YahooM.
DENSEALERT	**0.9**	**0.7**
Others [HSS+17, JBC+16, SHF18, SHKF18]	0.0	0.0

(b) Comparison with other anomaly detection algorithms

Figure 13.6: **DENSEALERT accurately detects small-scale short-period attacks injected in review datasets.** However, these attacks are overlooked by existing methods.

detected[3] by existing algorithms, including M-ZOOM [SHF18] (Chapter 11) and D-CUBE [SHKF18] (Chapter 12). They failed since they either ignore time information [HSS+17] or only find dense subtensors in the entire tensor [JBC+16, MGF11, SHF18, SHKF18] without using a time window.

13.5.5.2 Network Intrusion Detection.

DENSEALERT spots various types of network attacks from TCP dumps. Figure 13.1(c) shows the changes of the density of the maintained dense subtensor when we applied DENSEALERT to the Darpa dataset with the time window ΔT = 1 minute. We found out that the sudden emergence of dense subtensors (i.e., sudden increase in the density) indicates network attacks of various types. Especially, according to the ground-truth labels, all top 15 densest subtensors correspond to actual network attacks. Classifying each connection as an attack or a normal connection based on the density of the densest subtensor including the connection (i.e., the denser subtensor including a connection is, the more suspicious the connection is) led to high accuracy with AUC (Area Under the Curve) 0.924. This was better than MAF (0.514) and comparable with CPD (0.926), CROSSSPOT (0.923), and M-ZOOM (0.921). The result is still noteworthy since DENSEALERT requires only changes in the input tensor within ΔT time units at a time, while the others require the entire tensor at once.

[3] we consider that an injected subtensor is not detected by an algorithm if the subtensor is not included in the top 10 densest subtensors found by the algorithm or it is hidden in a dense subtensor of size at least 10 times larger than the injected subtensor.

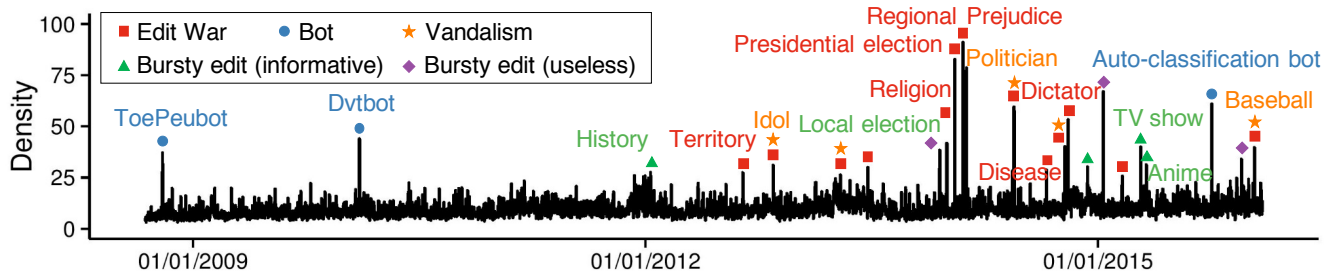


Figure 13.7: **DENSEALERT successfully spots anomalies on Korean Wikipedia.** The sudden appearances of dense subtensors signal actual anomalies of various types including edit wars, bot activities, and vandalism. The densest subtensor indicates an edit war where three users edited two pages about regional prejudice 900 times within a day.

13.5.5.3 Anomaly Detection on Wikipedia.

DENSEALERT detects notable anomalies on Wikipedia. The sudden appearances of dense subtensors also signal anomalies in Wikipedia edit histories. Figure 13.7 depicts the changes of the density of the dense subtensor maintained by DENSEALERT in the KoWiki dataset with the time window $\Delta T = 24$ hours. We investigated the detected dense subtensors and found out that most of them corresponded to actual anomalies including edit wars, bot activities, and vandalism. For example, the densest subtensor, composed by three users and two pages, indicated an edit war where three users edited two pages about regional prejudice 900 times within a day.

13.6 Summary

In this chapter, we propose DENSESTREAM, an incremental algorithm for detecting a dense subtensor in a tensor stream, and DENSEALERT, an incremental algorithm for spotting the sudden appearances of dense subtensors. They have the following advantages:

- **Fast and 'any time'**: our algorithms maintain and update a dense subtensor in a tensor stream, which is up to *a million times faster* than batch algorithms (Figure 13.5).
- **Provably accurate**: The densities of subtensors maintained by our algorithms have provable lower bounds (Theorems 13.2, 13.3, 13.4) and are high in practice (Figure 13.4).
- **Effective**: DENSEALERT successfully detects anomalies, including small-scale attacks, which existing algorithms overlook, in real-world tensors (Figures 13.6 and 13.7).

Reproducibility: The source code and datasets used in the chapter are available at `http://www.cs.cmu.edu/~kijungs/codes/alert`.

13.7 Appendix: Proofs

In this section, we give proofs of Lemmas 13.3 and 13.4. The proofs are based on Lemma 13.7.

Lemma 13.7

Let $\mathcal{X}$ be an N-way tensor, and let $\mathcal{X}'$ be the updated $\mathcal{X}$ after the change of either $((i_1, ..., i_N), \delta, +)$ or $((i_1, ..., i_N), \delta, -)$. Let π be a D-ordering of Q, and let $q_f := \arg\min_{q \in C} \pi^{-1}(q)$ where $C = \{(n, i_n) : 1 \leq \forall n \leq N\}$. For $c \in \mathbb{R}_{>0}$, let $M(c)$ be the set of slice indices that are located after q_f in π and having $d_\pi(\cdot)$ at least c, i.e.

$$M(c) := \{q \in Q : \pi^{-1}(q) > \pi^{-1}(q_f) \wedge d_\pi(q) \geq c\}. \tag{13.9}$$

And let $j_L, j_H \in [|Q|]$ be satisfying

$$j_H = \begin{cases} \min_{q \in M(c)} \pi^{-1}(q) - 1 & \text{if } M(c) \neq \emptyset, \\ |Q| & \text{otherwise}, \end{cases} \tag{13.10}$$

and $j_L \leq j_H$. Let $R = \{q \in Q : \pi^{-1}(q) \in [j_L, j_H]\}$. Let π' be an ordering of slice indices Q in $\mathcal{X}'$ where $\forall j \notin [j_L, j_H]$, $\pi'(j) = \pi(j)$ and $\forall j \in [j_L, j_H]$,

$$d(\mathcal{X}'(Q_{\pi', \pi'(j)}), \pi'(j)) = \min_{r \in R \cap Q_{\pi', \pi'(j)}} d(\mathcal{X}'(Q_{\pi', \pi'(j)}), r). \tag{13.11}$$

Then,

(i) For all $q \in Q$ with $\pi^{-1}(q) \notin [j_L, j_H]$,

$$Q_{\pi, q} = Q_{\pi', q}. \tag{13.12}$$

(ii) For all $q \in Q$ with $\pi^{-1}(q) > j_H$,

$$\mathcal{X}(Q_{\pi, q}) = \mathcal{X}'(Q_{\pi', q}). \tag{13.13}$$

(iii) For all $q \in Q$ with $j_L \leq \pi^{-1}(q) \leq j_H$,

$$d(\mathcal{X}'(Q_{\pi', q}), q) = \min_{r \in R \cap Q_{\pi', q}} d(\mathcal{X}'(Q_{\pi', q}), r). \tag{13.14}$$

(iv) For all $q \in Q$ with $j_L \leq \pi^{-1}(q) \leq j_H$,

$$\min_{r \in Q_{\pi', q} - R} d(\mathcal{X}'(Q_{\pi', q}), r) \geq c. \tag{13.15}$$

(v) For all $q \in Q$ with $j_L \leq \pi^{-1}(q) \leq j_H$, let $r := \arg\min_{q' \in Q_{\pi', q}} \pi^{-1}(q')$ be the slice index located earliest in π among $Q_{\pi', q}$. Then the following inequalities hold:

$$d(\mathcal{X}'(Q_{\pi', q}), q) \leq d(\mathcal{X}'(Q_{\pi', q}), r), \tag{13.16}$$

$$d(\mathcal{X}(Q_{\pi', q}), r) \leq d_\pi(r). \tag{13.17}$$

(vi) For all $q \in Q$ with $\pi^{-1}(q) > j_H$,

$$d(\mathcal{X}'(Q_{\pi', q}), q) = \min_{r \in Q_{\pi', q}} d(\mathcal{X}'(Q_{\pi', q}), r). \tag{13.18}$$

Proof.

Proof of (i): If $\pi^{-1}(q) \notin [j_L, j_H]$, then either $\pi^{-1}(q) < j_L$ or $\pi^{-1}(q) > j_H$. Consider $\pi^{-1}(q) < j_L$ first. Since π and π' coincide on $[1, j_L)$, $\pi'^{-1}(q) = \pi^{-1}(q) < j_L$ holds, so π and π' coincide on $[1, \pi^{-1}(q))$ as well. This implies that

$$Q_{\pi,q} = Q \setminus \{r \in Q : \pi^{-1}(r) < \pi^{-1}(q)\} = Q \setminus \{r \in Q : \pi'^{-1}(r) < \pi'^{-1}(q)\} = Q_{\pi',q}.$$

Now consider $\pi^{-1}(q) > j_H$ case. Since π and π' coincide on $(j_H, |Q|]$, $\pi'^{-1}(q) = \pi^{-1}(q) > j_H$ holds, so π and π' coincide on $[\pi^{-1}(q), |Q|]$ as well. This implies that

$$Q_{\pi,q} = \{r \in Q : \pi^{-1}(r) \geq \pi^{-1}(q)\} = \{r \in Q : \pi'^{-1}(r) \geq \pi'^{-1}(q)\} = Q_{\pi',q}.$$

Hence for either cases, Eq. (13.12) holds.

Proof of (ii): Since $\pi^{-1}(q) > j_H$, Lemma 13.7(i) implies $Q_{\pi,q} = Q_{\pi',q}$; and since $\pi^{-1}(q) > j_H \geq \pi^{-1}(q_f) = \min_{q' \in C} \pi^{-1}(q')$, the changed entry $x_{i_1 \cdots i_N}$ with index $(i_1, ..., i_N)$ is not contained in $\mathfrak{X}(Q_{\pi,q})$. These together imply Eq. (13.13).

Proof of (iii): Note that π and π' coinciding on $[|Q|] \setminus [j_L, j_H]$ implies that $j_L \leq \pi'^{-1}(q) \leq j_H$ as well. Hence this with the condition of π' in Eq. (13.11) implies Eq. (13.14).

Proof of (iv): If $M(c) = \emptyset$, then $Q_{\pi',q} \setminus R = \emptyset$, so there is nothing to show. When $M(c) \neq \emptyset$ so that $Q_{\pi',q} \setminus R \neq \emptyset$, fix any $r \in Q_{\pi',q} \setminus R$, and let $q_h := \pi(j_H + 1)$. We show Eq. (13.15) as

$$d(\mathfrak{X}'(Q_{\pi',q}), r) \geq d(\mathfrak{X}'(Q_{\pi',q_h}), r) \geq d_\pi(q_h) \geq c.$$

First, π and π' coinciding on $[|Q|] \setminus [j_L, j_H]$ implies that $\pi'^{-1}(q) \leq j_H < j_H + 1 = \pi'^{-1}(q_h)$, so $Q_{\pi',q} \supset Q_{\pi',q_h}$. And $\pi'^{-1}(r) = \pi^{-1}(r) \geq j_H + 1 = \pi^{-1}(q_h) = \pi'^{-1}(q_h)$ implies $r \in Q_{\pi',q_h}$, which implies $d(\mathfrak{X}'(Q_{\pi',q}), r) \geq d(\mathfrak{X}'(Q_{\pi',q_h}), r)$. Also, since $\pi^{-1}(q_h) > j_H$, Lemma 13.7(ii) implies $\mathfrak{X}(Q_{\pi,q_h}) = \mathfrak{X}'(Q_{\pi',q_h})$, and hence $d(\mathfrak{X}'(Q_{\pi',q_h}), r) \geq d(\mathfrak{X}(Q_{\pi,q_h}), r)$. Also, π being a D-ordering implies $d(\mathfrak{X}(Q_{\pi,q_h}), r) \geq d(\mathfrak{X}(Q_{\pi,q_h}), q_h) = d_\pi(q_h)$. Also, $M(c) \neq \emptyset$ and $j_H + 1 = \min_{q \in M(c)} \pi^{-1}(q)$ in Eq. (13.10) implies $q_h \in M(c)$, and definition of $M(c)$ in Eq. (13.9) implies that $d_\pi(q_h) \geq c$. These together imply Eq. (13.15).

Proof of (v): Note that π and π' coinciding on $[|Q|] \setminus [j_L, j_H]$ with $\pi^{-1}(q) \in [j_L, j_H]$ implies that $\pi^{-1}(r) \in [j_L, j_H]$, i.e. $r \in R$. Then Eq. (13.16) is from the condition of π' in Eq. (13.11) and that $r \in R \cap Q_{\pi',q}$. Also, $r = \arg\min_{q' \in Q_{\pi',q}} \pi^{-1}(q')$ implies $Q_{\pi',q} \subset Q_{\pi,r}$, which implies Eq. (13.17).

Proof of (vi): We show Eq. (13.18) as

$$d(\mathfrak{X}'(Q_{\pi',q}), q) = d(\mathfrak{X}(Q_{\pi,q}), q) = \min_{r \in Q_{\pi,q}} d(\mathfrak{X}(Q_{\pi,q}), r) = \min_{r \in Q_{\pi,q}} d(\mathfrak{X}'(Q_{\pi',q}), r).$$

First, from Lemma 13.7(ii), $d(\mathfrak{X}'(Q_{\pi',q}), q) = d(\mathfrak{X}(Q_{\pi,q}), q)$ holds. Next, since π is a D-ordering, $d(\mathfrak{X}(Q_{\pi,q}), q) = \min_{r \in Q_{\pi,q}} d(\mathfrak{X}(Q_{\pi,q}), r)$ holds. Lastly, from Lemma 13.7(ii), $\min_{r \in Q_{\pi,q}} d(\mathfrak{X}(Q_{\pi,q}), r) = \min_{r \in Q_{\pi,q}} d(\mathfrak{X}'(Q_{\pi',q}), r)$ holds. These together imply Eq. (13.18). ■

13.7.1 Proof of Lemma 13.3

In the following proof, we use $q_f := \arg\min_{r \in C} \pi^{-1}(r) = \pi(j_L)$ where $C = \{(n, i_n) : 1 \leq \forall n \leq N\}$, $M := \{q \in Q : \pi^{-1}(q) > \pi^{-1}(q_f) \wedge d_\pi(q) \geq d_\pi(q_f) + \delta\}$, and $R := \{q \in Q : \pi^{-1}(q) \in [j_L, j_H]\}$. Note that $\mathfrak{X}$, $\mathfrak{X}'$, π, q_f, j_L, j_H, R, π' satisfies the conditions in Lemma 13.7 with $M = M(d_\pi(q_f) + \delta)$, and hence Lemma 13.7 is applicable.

Proof. From Definition 13.2 of D-ordering, we need to show that for all $q \in Q$,

$$d(\mathcal{X}'(Q_{\pi',q}), q) = \min_{r \in Q_{\pi',q}} d(\mathcal{X}'(Q_{\pi',q}), r). \tag{13.19}$$

We divide into 3 cases depending on the location of q with respect to π:

- **Case (i):** $\pi^{-1}(q) < j_L$,
- **Case (ii):** $j_L \leq \pi^{-1}(q) \leq j_H$,
- **Case (iii):** $\pi^{-1}(q) > j_H$.

Proof of Case (i): For the case of $\pi^{-1}(q) < j_L$, we show Eq. (13.19) as

$$d(\mathcal{X}'(Q_{\pi',q}), q) = d(\mathcal{X}(Q_{\pi,q}), q) = \min_{r \in Q_{\pi,q}} d(\mathcal{X}(Q_{\pi,q}), r) \leq \min_{r \in Q_{\pi',q}} d(\mathcal{X}'(Q_{\pi',q}), r).$$

At first, $\pi^{-1}(q) < j_L$ and Lemma 13.7 (i) imply

$$Q_{\pi',q} = Q_{\pi,q}. \tag{13.20}$$

Also, from $\pi^{-1}(q) < j_L = \min_{q' \in C} \pi^{-1}(q')$ where $C = \{(n, i_n) : 1 \leq \forall n \leq N\}$, the changed entry $x_{i_1 \cdots i_N}$ is not in the slice with index q. From this and Eq. (13.20), $d(\mathcal{X}'(Q_{\pi',q}), q) = d(\mathcal{X}(Q_{\pi,q}), q)$ holds. Next, from π being a D-ordering, we have $d(\mathcal{X}(Q_{\pi,q}), q) = \min_{r \in Q_{\pi,q}} d(\mathcal{X}(Q_{\pi,q}), r)$ holds. Lastly, from Eq. (13.20) and that the slice sums of the slices in Q either increase or remain the same under $((i_1, ..., i_N), \delta, +)$, $\min_{r \in Q_{\pi,q}} d(\mathcal{X}(Q_{\pi,q}), r) \leq \min_{r \in Q_{\pi',q}} d(\mathcal{X}'(Q_{\pi',q}), r)$ holds. From these, $d(\mathcal{X}'(Q_{\pi',q}), q) = \min_{r \in Q_{\pi',q}} d(\mathcal{X}'(Q_{\pi',q}), r)$ holds for $\pi^{-1}(q) < j_L$.

Proof of Case (ii): For the case of $j_L \leq \pi^{-1}(q) \leq j_H$, we show Eq. (13.19) by

$$d(\mathcal{X}'(Q_{\pi',q}), q) = \min_{r \in Q_{\pi',q} \cap R} d(\mathcal{X}'(Q_{\pi',q}), r), \textbf{ and } \min_{r \in Q_{\pi',q} - R} d(\mathcal{X}'(Q_{\pi',q}), r) \geq d_\pi(q_f) + \delta \geq d(\mathcal{X}'(Q_{\pi',q}), q).$$

At first, Lemma 13.7(iii) implies $d(\mathcal{X}'(Q_{\pi',q}), q) = \min_{r \in Q_{\pi',q} \cap R} d(\mathcal{X}'(Q_{\pi',q}), r)$. Also, since $M = M(d_\pi(q_f) + \delta)$, Lemma 13.7(iv) implies $\min_{r \in Q_{\pi',q} - R} d(\mathcal{X}'(Q_{\pi',q}), r) \geq d_\pi(q_f) + \delta$. For $d(\mathcal{X}'(Q_{\pi',q}), q) \leq d_\pi(q_f) + \delta$, let $r := \arg\min_{q' \in Q_{\pi',q}} \pi^{-1}(q')$ be the slice with the index earliest in π among $Q_{\pi',q}$, and we show $d(\mathcal{X}'(Q_{\pi',q}), q) \leq d(\mathcal{X}'(Q_{\pi',q}), r) \leq d_\pi(q_f) + \delta$. $d(\mathcal{X}'(Q_{\pi',q}), q) \leq d(\mathcal{X}'(Q_{\pi',q}), r)$ is from Eq. (13.16) in Lemma 13.7(v). For $d(\mathcal{X}'(Q_{\pi',q}), r) \leq d_\pi(q_f) + \delta$, we divide into cases where $r = q_f$ or $r \neq q_f$. When $r = q_f$, note that slice sums can increase at most δ under $((i_1, ..., i_N), \delta, +)$, so $d(\mathcal{X}'(Q_{\pi',q}), q_f) \leq d(\mathcal{X}(Q_{\pi',q}), q_f) + \delta$ holds; and Eq. (13.17) in Lemma 13.7(v) implies $d(\mathcal{X}(Q_{\pi',q}), q_f) + \delta \leq d_\pi(q_f) + \delta$. Hence these implies $d(\mathcal{X}'(Q_{\pi',q}), r) \leq d_\pi(q_f) + \delta$ for $r = q_f$. When $r \neq q_f = \pi(j_L)$, π and π' coinciding on $[1, j_L)$ implies that $q_f \notin Q_{\pi',q}$, which implies that $x_{i_1 \cdots i_N}$ is not in $\mathcal{X}(Q_{\pi',q})$. Hence $\mathcal{X}(Q_{\pi',q}) = \mathcal{X}'(Q_{\pi',q})$, and this implies $d(\mathcal{X}'(Q_{\pi',q}), r) = d(\mathcal{X}(Q_{\pi',q}), r)$. Then Eq. (13.17) in Lemma 13.7(v) implies $d(\mathcal{X}(Q_{\pi',q}), r) \leq d_\pi(r)$. Then π and π' coinciding on $[|Q|] \backslash [j_L, j_H]$, $\pi^{-1}(q) \leq j_H$, and $r \neq \pi(j_L)$ imply that $j_L < \pi^{-1}(r) \leq \pi^{-1}(q) \leq j_H$, and Eq. (13.2) implies that $r \notin M$. Hence $d_\pi(r) < d_\pi(q_f) + \delta$ holds, and these imply $d(\mathcal{X}'(Q_{\pi',q}), r) \leq d_\pi(q_f) + \delta$ for $r \neq q_f$. Hence $d(\mathcal{X}'(Q_{\pi',q}), r) \leq d_\pi(q_f) + \delta$ holds for any r. This with $d(\mathcal{X}'(Q_{\pi',q}), q) \leq d(\mathcal{X}'(Q_{\pi',q}), r)$ implies that for $j_L \leq \pi^{-1}(q) \leq j_H$ we have

$$d(\mathcal{X}'(Q_{\pi',q}), q) \leq d_\pi(q_f) + \delta. \tag{13.21}$$

Then Lemma 13.7(iii)-(iv), and Eq. (13.21) imply that $d(\mathcal{X}'(Q_{\pi',q}), q) = \min_{r \in Q_{\pi',q}} d(\mathcal{X}'(Q_{\pi',q}), r)$ holds for $j_L \leq \pi^{-1}(q) \leq j_H$.

Proof of Case (iii): For the case of $\pi^{-1}(q) > j_H$, Lemma 13.7(vi) implies Eq. (13.19). ■

13.7.2 Proof of Lemma 13.4

In the following proof, we use $q_f := \arg\min_{r \in C} \pi^{-1}(r)$ where $C = \{(n, i_n) : 1 \leq \forall n \leq N\}$, $M_{\min} := \{q \in Q : d_\pi(q) > c_\pi(q) - \delta\}$, $M_{\max} := \{q \in Q : \pi^{-1}(q) > \pi^{-1}(q_f) \wedge d_\pi(q) \geq c_\pi(q_f)\}$, and $R := \{q \in Q : \pi^{-1}(q) \in [j_L, j_H]\}$. Note that $\mathcal{X}$, $\mathcal{X}'$, π, q_f, j_L, j_H, R, π' satisfies the conditions in Lemma 13.7 with $M_{\max} = M(c_\pi(q_f))$, and hence Lemma 13.7 is applicable.

Proof. From Definition 13.2 of D-ordering, we need to show that for all $q \in Q$,

$$d(\mathcal{X}'(Q_{\pi',q}), q) = \min_{r \in Q_{\pi',q}} d(\mathcal{X}'(Q_{\pi',q}), r). \tag{13.22}$$

We divide into 3 cases depending on the location of q with respect to π:

- **Case (i):** (i) $\pi^{-1}(q) < j_L$,
- **Case (ii):** $j_L \leq \pi^{-1}(q) \leq j_H$,
- **Case (iii):** $\pi^{-1}(q) > j_H$.

Proof of Case (i): For the case of $\pi^{-1}(q) < j_L$, we show as

$$d(\mathcal{X}'(Q_{\pi',q}), r) \geq d_\pi(q) = d(\mathcal{X}'(Q_{\pi',q}), q).$$

At first, $\pi^{-1}(q) < j_L$ and Lemma 13.7 (i) imply

$$Q_{\pi',q} = Q_{\pi,q}. \tag{13.23}$$

Also, $j_L \leq \pi^{-1}(q_f) = \min_{q' \in C} \pi^{-1}(q')$ where $C = \{(n, i_n) : 1 \leq \forall n \leq N\}$, so the changed entry $x_{i_1 \cdots i_N}$ is not in the slice with index q. This and Eq. (13.23) imply that $d_\pi(q) = d(\mathcal{X}(Q_{\pi,q}), q) = d(\mathcal{X}'(Q_{\pi',q}), q)$ holds. For showing $d(\mathcal{X}'(Q_{\pi',q}), r) \geq d_\pi(q)$, we divide into cases when $r \in C$ and $r \in Q_{\pi',q} - C$. For $r \in C$ case, let $x \in Q$ be satisfying $\pi^{-1}(x) \leq \pi^{-1}(q_f)$ and $d_\pi(x) = c_\pi(q_f)$, whose existence is from the definition of $c_\pi(\cdot)$. We show by $d(\mathcal{X}'(Q_{\pi',q}), r) + \delta \geq d(\mathcal{X}(Q_{\pi,q}), r) \geq d(\mathcal{X}(Q_{\pi,x}), r) \geq d_\pi(x) = c_\pi(q_f) \geq d_\pi(q) + \delta$. From Eq. (13.23), since slice sums decrease at most δ under $((i_1, ..., i_N), \delta, -)$, $d(\mathcal{X}'(Q_{\pi',q}), r) + \delta \geq d(\mathcal{X}(Q_{\pi,q}), r)$ holds. Also, $d_\pi(x) = c_\pi(q_f) > c_\pi(q_f) - \delta$ implies that $x \in M_{\min}$ from definition of $M_{\min}$, hence $\pi^{-1}(x) \geq j_L = \min_{q \in M_{\min}} \pi^{-1}(q)$. Then $\pi^{-1}(q) < j_L \leq \pi^{-1}(x)$ implies $Q_{\pi,x} \subset Q_{\pi,q}$ and $\pi^{-1}(x) \leq \pi^{-1}(q_f) \leq \pi^{-1}(r)$ implies $r \in Q_{\pi,x}$. Hence $r \in Q_{\pi,x} \subset Q_{\pi,q}$, which implies $d(\mathcal{X}(Q_{\pi,q}), r) \geq d(\mathcal{X}(Q_{\pi,x}), r)$. Also, $d(\mathcal{X}(Q_{\pi,x}), r) \geq d_\pi(x)$ is from π being a D-ordering. Also, $d_\pi(x) = c_\pi(q_f)$ is from definition of x. Lastly, definition of j_L in Eq. (13.7) and $\pi^{-1}(q) < j_L$ implies $q \notin M_{\min}$, hence $c_\pi(q_f) \geq d_\pi(q) + \delta$ holds. From these, $d(\mathcal{X}'(Q_{\pi',q}), r) \geq d_\pi(q)$ holds for $r \in C$ case. For $r \in Q_{\pi',q} - C$ case, we show as $d(\mathcal{X}'(Q_{\pi',q}), r) = d(\mathcal{X}(Q_{\pi,q}), r) \geq d(\mathcal{X}(Q_{\pi,q}), q)$. Since r is not in C and from Eq. (13.23), $d(\mathcal{X}'(Q_{\pi',q}), r) = d(\mathcal{X}(Q_{\pi,q}), r)$ holds. Then from π being a D-ordering, $d(\mathcal{X}(Q_{\pi,q}), r) \geq d_\pi(q)$ holds for $r \in Q_{\pi',q} - C$ case. Hence for either case, $d(\mathcal{X}'(Q_{\pi',q}), r) \geq d_\pi(q)$ holds. This with $d_\pi(q) = d(\mathcal{X}'(Q_{\pi',q}), q)$ implies $d(\mathcal{X}'(Q_{\pi',q}), q) = \min_{r \in Q_{\pi,q}} d(\mathcal{X}'(Q_{\pi',q}), r)$ for $\pi^{-1}(q) < j_L$.

Proof of Case (ii): For the case of $j_L \leq \pi^{-1}(q) \leq j_H$, we show Eq. (13.22) by

$$d(\mathcal{X}'(Q_{\pi',q}), q) = \min_{r \in Q_{\pi',q} \cap R} d(\mathcal{X}'(Q_{\pi',q}), r), \text{ and } \min_{r \in Q_{\pi',q} - R} d(\mathcal{X}'(Q_{\pi',q}), r) \geq c_\pi(q_f) \geq d(\mathcal{X}'(Q_{\pi',q}), q).$$

At first, Lemma 13.7 (iii) implies $d(\mathcal{X}'(Q_{\pi',q}), q) = \min_{r \in Q_{\pi',q} \cap R} d(\mathcal{X}'(Q_{\pi',q}), r)$. Also, since $M_{\max} = M(c_\pi(q_f))$, Lemma 13.7(iv) implies $\min_{r \in Q_{\pi',q} - R} d(\mathcal{X}'(Q_{\pi',q}), r) \geq c_\pi(q_f)$. For $d(\mathcal{X}'(Q_{\pi',q}), q) \leq c_\pi(q_f)$, let $r := \arg\min_{q' \in Q_{\pi',q}} \pi^{-1}(q')$ be the slice index earliest in π among $Q_{\pi',q}$. We show

$$d(\mathcal{X}'(Q_{\pi',q}), q) \leq d(\mathcal{X}'(Q_{\pi',q}), r) \leq d(\mathcal{X}(Q_{\pi',q}), r) \leq d_\pi(r) \leq c_\pi(q_f).$$

$d(\mathfrak{X}'(Q_{\pi',q}),q) \leq d(\mathfrak{X}'(Q_{\pi',q}),r)$ is from Eq. (13.16) in Lemma 13.7(v). And slice sums either decrease or remain the same under $((i_1,...,i_N),\delta,-)$, so $d(\mathfrak{X}'(Q_{\pi',q}),r) \leq d(\mathfrak{X}(Q_{\pi',q}),r)$ holds. And Eq. (13.17) in Lemma 13.7(v) implies $d(\mathfrak{X}(Q_{\pi',q}),r) \leq d_\pi(r)$. Then π and π' coinciding on $[|Q|]\backslash[j_L,j_H]$, $\pi^{-1}(r) \leq \pi^{-1}(q) \leq j_H$ holds. Then Eq. (13.8) implies that $r \notin M_{\max}$. Hence $\pi^{-1}(r) \leq \pi^{-1}(q_f)$ or $d_\pi(r) < c_\pi(q_f)$ holds. For $\pi^{-1}(r) \leq \pi^{-1}(q_f)$ case, definition of $c_\pi(\cdot)$ implies that $d_\pi(r) \leq c_\pi(r) \leq c_\pi(q_f)$, hence in any case $d_\pi(r) \leq c_\pi(q_f)$ holds. Hence for $j_L \leq \pi^{-1}(q) \leq j_H$ we have

$$d(\mathfrak{X}'(Q_{\pi',q}),q) \leq c_\pi(q_f). \tag{13.24}$$

Then Lemma 13.7(iii)-(iv), and Eq. (13.24) imply that $d(\mathfrak{X}'(Q_{\pi',q}),q) = \min_{r\in Q_{\pi,q}} d(\mathfrak{X}'(Q_{\pi',q}),r)$ holds for $j_L \leq \pi^{-1}(q) \leq j_H$.

Proof of Case (iii): For the case of $\pi^{-1}(q) > j_H$, Lemma 13.7(vi) implies Eq. (13.19). ■

Part III

Behavior Modeling

Chapter based on work that appeared at IJCAI 2017 [SLEP17].

Chapter 14

Modeling Purchases in Social Networks

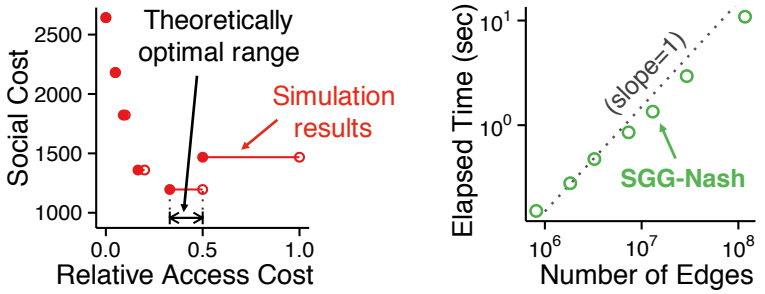
Can it be beneficial to society to charge our friends for borrowing our stuff? If so, how much should we charge to maximize the social benefit?

We consider goods that can be shared with k-hop neighbors (i.e., the set of nodes within k hops from an owner) on a social network (i.e., a graph representing social relationships among individuals). We examine incentives to buy such a good by devising game-theoretic models where each node decides whether to buy the good or free ride and a fast algorithm for finding Nash equilibria of the games. First, we find that social inefficiency, specifically excessive purchase of the good, occurs in Nash equilibria. Second, the social inefficiency decreases as k increases and thus a good can be shared with more nodes. Third, and most importantly, the social inefficiency can also be significantly reduced by charging free riders an access cost and paying it to owners, leading to the conclusion that organizations and system designers should impose such a cost. These findings are supported by our theoretical analysis in terms of the price of anarchy and the price of stability; and by simulations based on synthetic and real social networks with up to hundreds of millions of edges.

14.1 Motivation

Social networks are known to play an important role in the everyday choices people make. In particular, a significant body of work studies the *network effect*, in which there are payoffs from aligning one's decision with those of others [Mar87, Blu93, Ell93, Rog10]. For example, direct payoffs arise when friends or collaborators use compatible technologies instead of incompatible ones, that is, the game rewards *coordination*.

Our work considers the purchase of *shareable goods*, which, in a sense, gives rise to a certain type of *anti-coordination* game. Indeed, buying such a good yields a benefit only when no friend buys the good, since otherwise free riding is possible. An example that would be familiar to most parents is seldom-used baby gear, such as portable cribs, which are frequently borrowed by friends or friends of friends; similar examples include ski gear and hiking equipment. Expensive lab equipment provides a more pertinent example: Confocal laser scanning microscopes, or polymerase chain reaction (PCR) machines, are typically bought by one investigator and used by collaborators. In the realm of AI, one can imagine a multi-agent system populated by heterogeneous software agents that interact and share special computational resources, e.g., a high-end graphics processing unit (GPU) for particularly demanding image processing tasks.



(a) Effect of Access Cost (b) Scalability of SGG-NASH

Figure 14.1: **Effectiveness and scalability of our tools.** *(a) Effectiveness:* using our game-theoretic models (SGG and SGG-AC) and algorithm (SGG-NASH), we analyze the effects of access costs on social cost (formally defined in Section 14.4) and suggest appropriate access costs to minimize the cost. *(b) Scalability:* SGG-NASH, our algorithm for finding Nash equilibria, scales linearly with the size of the input social network. In both figures, we assume goods that can be shared with direct friends. See Section 14.5 for details

To examine incentives to buy shareable goods, we devise game-theoretic models where each node on a network decides whether to buy a good that is shareable with k-*hop neighbors* (i.e., nodes within k hops from an owner), or free ride. Specifically, the good in question is *non-excludable* and *non-rivalrous* in that no k-hop neighbor can be excluded from use, and use by a neighbor does not reduce availability to others. Note that the goods in the examples given above are (essentially) non-rivalrous, as any single person (or agent) requires the good only from time to time.

We find that social inefficiency, specifically excessive purchase of the good, occurs in Nash equilibria (NEs). Moreover, the social inefficiency decreases as k increases and thus a good is shared with more people. Finally, charging free riders an access cost and paying it to owners also significantly reduces the social inefficiency. We support these findings both theoretically and experimentally (see Figure 14.1(a)).

Our contributions in this chapter are threefold:

- *Game-theoretic Models and Equilibrium Analysis:* We develop two game-theoretic models (SGG and SGG-AC), and we provide worst-case analysis of the social inefficiency of NEs of both games, in terms of the price of anarchy and the price of stability.
- *Algorithm and Simulations on Real Social Networks:* We develop SGG-NASH, a fast algorithm for finding NEs of both games (see Figure 14.1(b)). In our simulations using SGG-NASH, we measure the social inefficiency of NEs on real social networks, and the effects of access costs on the inefficiency.
- *Mechanism Design:* We analyze the effects of access costs on the social inefficiency of NEs and suggest an appropriate cost for minimizing the inefficiency.

The rest of the chapter is organized as follows. In Section 14.2, we introduce SGG and SGG-AC, our game-theoretic models. In Section 14.3, we present SGG-NASH, our proposed algorithm for finding Nash equilibria of the games. In Section 14.4, we theoretically analyze the equilibria of our models and the convergence and complexity of SGG-NASH. In Section 14.5, we share some experimental results. After discussing related work in Section 14.6, we provide a summary of this chapter in Section 14.7.

Table 14.1: **Table of frequently-used symbols.**

Symbol	Definition
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	social network
$\mathcal{V} = \{1, 2, ..., n\}$	set of players (i.e., nodes in $\mathcal{G}$)
$n = \lvert\mathcal{V}\rvert$	number of players
$m = \lvert\mathcal{E}\rvert$	number of edges in $\mathcal{G}$
$\mathcal{N}_i^{(k)}$	set of nodes within k-hops from i including i itself
k	maximum possible number of hops between the owner of a good and a renter of it
p	price of a good
b	benefit of a good
a	access cost of a good
$\mathcal{S}_i$	strategy set of each player $i \in \mathcal{V}$
$s_i \in \mathcal{S}_i$	strategy of player $i \in \mathcal{V}$
s_{-i}	strategies taken by all players but $i \in \mathcal{V}$
$s = (s_1, ..., s_n) = (s_i, s_{-i})$	strategy profile
$u_i(s)$	utility of player $i \in \mathcal{V}$ under strategy profile s
$\mathcal{T}$	set of strategy profiles where all players access a good
$\mathcal{F}_i$	followers of player $i \in \mathcal{V}$ (see Section 14.2.1.2 for a definition)
ξ	follower threshold (see Section 14.2.1.2 for a definition)

14.2 Proposed Models: SGG and SGG-AC

In this section, we propose two game-theoretic models of the purchase of shareable goods on a network. Then, we define their equilibria with a proof of their existence.

14.2.1 Notations and Model Description

We formally define our game-theoretic models, namely SGG and SGG-AC. Table 14.1 lists some symbols frequently used in this chapter.

14.2.1.1 Shareable Goods Game (SGG)

Consider an undirected network $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \{1, 2, ..., n\}$. The *players* of the game are nodes in $\mathcal{G}$. Each node decides whether to buy a good or not. The *strategy* of node i is denoted by $s_i \in \mathcal{S}_i$, where $\mathcal{S}_i = \{0, 1\}$ denotes the *strategy set* of node i. If node i buys the good then $s_i = 1$, and otherwise $s_i = 0$ (only pure strategies are considered). Given any *strategy profile* $s = (s_1, s_2, ..., s_n)$, we use s_{-i} to denote the strategies taken by all nodes but i. Then, s is also denoted by (s_i, s_{-i}). The *price* of a good is p (> 0), which is identical for all nodes. A node gets *benefit* b ($> p$) by having access to a good and 0 otherwise. Each node i can access a good if it buys the good itself or has at least one node who buys the good within k (≥ 1) hops. We assume that having access to multiple goods does not increase the benefit of a node and that being accessed by multiple nodes does not decrease the benefit derived from a good (*non-rivalry*).

Table 14.2: **Utility in an SGG.**

state	conditions	utility (i.e., u_i)
buy	$s_i = 1$	$b - p\ (> 0)$
free ride	$s_i = 0, \sum_{j \in \mathcal{N}_i^{(k)}} s_j \geq 1$	$b\ (> 0)$
no access	$s_i = 0, \sum_{j \in \mathcal{N}_i^{(k)}} s_j = 0$	0

Table 14.3: **Utility in an SGG-AC.**

state	conditions	utility (i.e., u_i)
buy	$s_i = i$	$b - p + a\lvert\mathcal{F}_i\rvert (> 0)$
rent	$s_i = j (\neq i), s_j = j$	$b - a (> 0)$
no access	$s_i = j (\neq i), s_j \neq j$	0

In this setting, the *utility* $u_i(s)$ of node i under strategy profile s depends on the strategies of its *k-hop neighbors* $\mathcal{N}_i^{(k)}$ (i.e., the set of nodes within k-hops from i including i itself), as given in Table 14.2. Note that each node gets the highest utility b when it free rides and the second highest utility $b - p$ when it buys the good. Each node gets the lowest utility 0 when neither the node nor its k-hop neighbors buy the good. SGG extends the best-shot game [Hir83], which is equivalent to SGG if $k = 1$, by considering not only direct but k-hop neighbors. SGG-AC, discussed in the following subsection, further extends SGG by considering access costs.

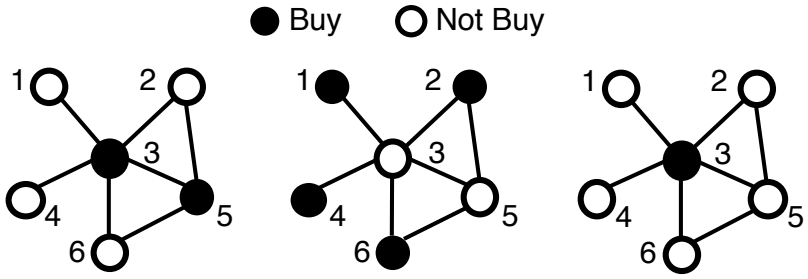
14.2.1.2 Shareable Goods Game with Access Costs (SGG-AC)

In this subsection, we extend the game defined in the previous section to a game we call the shareable goods game with access costs (SGG-AC), where each free rider has to pay an access cost. We focus on the differences from an SGG.

The *strategy set* of each node i is $\mathcal{S}_i = \mathcal{N}_i^{(k)}$, its k-hop neighbors including i itself. If node i buys a good then $s_i = i$, and if node i does not buy a good but wants to access a good bought by node $j \neq i$ then $s_i = j$. If $s_i = j$ for $j \neq i$, and node j actually buys a good (i.e., $s_j = j$) then node i derives benefit from the good at the expense of paying an *access cost* of a ($< p$) to node j. The *followers* of node i, the set of nodes who want to access the good bought by node i, are denoted by $\mathcal{F}_i = \{j \in \mathcal{N}_i^{(k)} \backslash \{i\} : s_j = i\}$. Then, the *utility* $u_i(s)$ of node i under strategy profile s in an SGG-AC is given in Table 14.3. Define the *follower threshold* $\xi = \lceil p/a \rceil - 1$; for ease of exposition we assume that p/a is not an integer. If node i has at least ξ followers (i.e., $|\mathcal{F}_i| \geq \xi$), it has the highest utility when it buys a good (i.e., $s_i = i$). Otherwise, renting a good is preferred. Each node has the lowest utility when it is not accessing any good.

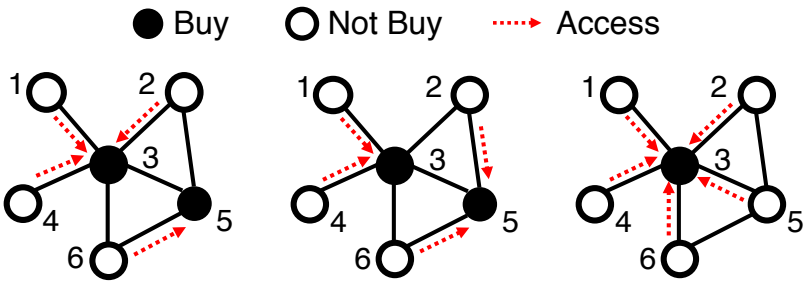
14.2.2 Definition and Existence of Equilibria

We define equilibria in the games described in the previous subsections. To this end, we use the ubiquitous concept of Nash equilibrium (NE) as our solution concept.



(a) Not NE (b) NE ($cost = 4p$) (c) NE ($cost = p$)

Figure 14.2: **Example strategy profiles in an SGG when $k = 1$.** (a) is not an NE since each of node 3 and node 6 would be better off not buying. Between the NEs, (c) leads to a lower social cost than (b).



(a) Not NE (b) NE ($cost = 2p$) (c) NE ($cost = p$)

Figure 14.3: **Example strategy profiles in an SGG-AC when $k = 1$ and $\xi = 2$.** Arrows indicate who accesses whose products. (a) is not an NE since node 6 is better off not buying. Between the NEs, (c) leads to a lower social cost than (b).

Definition 14.1: Nash Equilibrium

A strategy profile $s = (s_i, s_{-i})$ is a *Nash equilibrium (NE)* if no node can increase its utility by changing its strategy given the strategies of the other nodes, i.e.,

$$\forall i \in \mathcal{V},\ \forall s'_i \in \mathcal{S}_i,\ u_i((s_i, s_{-i})) \geq u_i((s'_i, s_{-i})).$$

Figures 14.2 and 14.3 give examples of NEs in an SGG and an SGG-AC, respectively, with explanations. Note that a strategy profile is an NE in an SGG if and only if the set of owners is a *k-independent dominating set* [KN97], i.e., any two owners have distance at least $k + 1$ and every node has distance at most k to some owner. Theorem 14.1 states that an NE always exists in both games.

Theorem 14.1: Existence of Nash Equilibria

An NE exists in any SGG and SGG-AC.

Proof. Given $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, k and ξ, the following procedure gives an NE s for any SGG and s' for any SGG-AC. Choose an arbitrary node i in the graph, let i buy a good ($s_i = 1$, $s'_i = i$), and for each node j within k hops from i, let j follow i ($s_j = 0$, $s'_j = i$). Delete i and all nodes within k hops from i, and

repeat until there is no node left in $\mathcal{G}$. At the end, every node either buys a good or accesses its k-hop neighbor's.

Each node that accesses its k-hop neighbor's good cannot increase its utility by buying a good since the utility of accessing its k-hop neighbor's is greater than that of buying (and no one follows). Each node i that buys a good also cannot increase its utility by following another node because the procedure ensures that there is no node that buys a good and is within distance k from i. Therefore, s and s' are NEs for the given SGG and SGG-AC, respectively. ■

14.3 Proposed Algorithm: SGG-NASH

In this section, we propose SGG-NASH, a fast algorithm for finding NEs in SGGs and SGG-ACs. SGG-NASH, described in Algorithm 14.1, adapts best-response dynamics [Mat92]. A strategy of node i is a *best response* if it maximizes the utility of i given the others' strategies. In *best-response dynamics*, nodes iteratively deviate to a best response, until an NE is reached. SGG-NASH runs best-response dynamics starting from a strategy profile where no node buys a good. The convergence, time complexity, and space complexity of SGG-NASH are analyzed in Section 14.4.

Algorithm 14.1 SGG-NASH: a fast algorithm for searching NEs in SGGs and SGG-ACs

Input: (1) social network: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, (2) benefit: b, (3) price: p, and (4) access cost: a
Output: a strategy profile s corresponding to an NE

for each node $i \in \mathcal{V}$ **do**
  if $a = 0$ **then** $s_i \leftarrow 0$ ▷ in case of an SGG
  else $s_i \leftarrow$ a random strategy in $\mathcal{N}_i^{(k)} \backslash \{i\}$ ▷ in case of an SGG-AC
shuffle nodes in a random order
while strategy profile s is not an NE **do**
  for each node $i \in \mathcal{V}$ in the sorted order **do**
    $u_{max} \leftarrow \max_{x \in \mathcal{S}_i} u_i((x, s_{-i}))$
    if $u_i((s_i, s_{-i})) < u_{max}$ **then**
      $\mathcal{B}_i \leftarrow \{x \in \mathcal{S}_i : u_i((x, s_{-i})) = u_{max}\}$
      $s_i \leftarrow$ a randomly chosen strategy in $\mathcal{B}_i$
return $s = (s_1, ..., s_n)$

14.4 Theoretical Analysis

In this section, we first analyze the efficiency of the equilibria in terms of price of anarchy (PoA) and price of stability (PoS). Then, we analyze the convergence properties, time complexity, and space complexity of SGG-NASH.

14.4.1 Social Inefficiency Analysis

We now turn to the analysis of NEs in our games. It is important to note that a node that does not access any good can increase its utility by buying a good, without decreasing the utilities of the others in both of our games (see Tables 14.2 and 14.3). Thus, if we let $\mathcal{T}$ be the set of strategy profiles where every

Table 14.4: **Summary of our analysis of efficiency of equilibria.**

		in SGGs	in SGG-ACs
$k=1$	PoA	$\Theta(n)$	$\Theta(n)$
	PoS		$\Theta(\xi)$ ($=\mathbf{1}$ if $\xi \leq 2$)
$k>1$	PoA	$\Theta(n/k)$	$\Theta(\max(n/k, n/\xi))$
	PoS		$\Theta(\xi/k)$ ($=\mathbf{1}$ if $\xi \leq 2\lfloor k/2 \rfloor + 1$)

node accesses a good and thus gets benefit b [1], then all NEs belong to $\mathcal{T}$. Due to the same reason, every socially optimal strategy profile (i.e., strategy profile s maximizing social welfare $\sum_{i\in\mathcal{V}} u_i(s)$) belongs to $\mathcal{T}$. Therefore, to define PoA and PoS, we only need to consider the strategy profiles in $\mathcal{T}$. Since all strategy profiles in $\mathcal{T}$ have the same sum of benefits, we can compare them simply by their *social cost*, which is proportional to the number of nodes buying a good (see Definition 14.2). Importantly, access costs in SGG-AC cancel out (they are paid by some players to others) and do not affect the social cost.

Definition 14.2: Social Cost

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the *social cost* of a strategy profile $s = (s_1, ..., s_n) \in \mathcal{T}$ is the sum of prices paid by the nodes, i.e.,

$$cost(s) = \begin{cases} p \cdot |\{i \in \mathcal{V} : s_i = 1\}| & \text{in SGGs} \\ p \cdot |\{i \in \mathcal{V} : s_i = i\}| & \text{in SGG-ACs.} \end{cases}$$

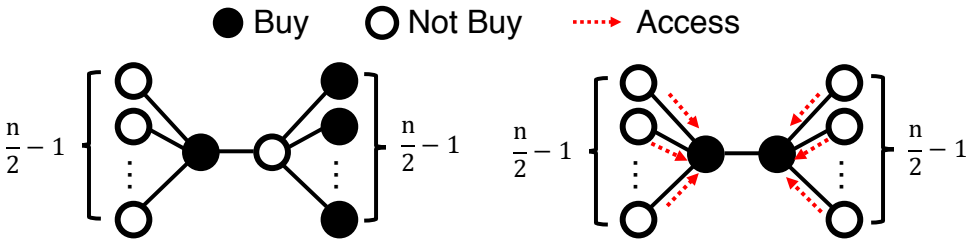
The *price of anarchy (PoA)* is defined as the social cost of the *worst* NE divided by minimum social cost (see Definition 14.3) and the *price of stability (PoS)* is defined as the social cost of the *best* NE divided by minimum social cost (see Definition 14.4). Large PoA and PoS indicate that NEs are socially inefficient.

Definition 14.3: Price of Anarchy

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with n nodes, the *price of anarchy (PoA)* is defined as

$$\text{PoA} = \frac{\max_{s\in\mathcal{T}:\ s \text{ is an NE}} cost(s)}{\min_{s\in\mathcal{T}} cost(s)}.$$

[1] In SGG, $s \in \mathcal{T} \Leftrightarrow \forall i \in \mathcal{V}, \sum_{j\in\mathcal{N}_i^{(k)}} s_j \geq 1$. In SGG-AC, $s \in \mathcal{T} \Leftrightarrow \forall i \in \mathcal{V}, (s_i = i \text{ or } (s_i = j \text{ and } s_j = j))$.



(a) Best NE in SGG (b) Best NE in SGG-AC

Figure 14.4: **An example of social inefficiency in an SGG.** Assume $k = 1$. Arrows indicate who accesses whose products. In this example graph, the best NE in an SGG is (a), whose social cost is $np/2$. In an SGG-AC (with $\xi \leq n/2 - 1$), however, the best NE is (b), whose social cost is $2p$, equal to the minimum social cost.

Definition 14.4: Price of Stability

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with n nodes, the *price of stability (PoS)* is defined as

$$\text{PoS} = \frac{\min_{s \in \mathcal{T}:\ s \text{ is an NE}} cost(s)}{\min_{s \in \mathcal{T}} cost(s)}.$$

In Table 14.4, we summarize the results of our worst-case efficiency analysis. That is, we analyze the two measures in the worst case over all graphs. As stated in Theorem 14.2, both the PoA and PoS in SGGs are $\Theta(n/k)$ in the worst case. That is, not only worst NEs but also best NEs can be severely inefficient, with social cost as high as n/k times the optimum. Figure 14.4(a) shows an example of such inefficiency for $k = 1$, where even the best NEs in the SGG have social cost $np/2$, while the minimum social cost, in Figure 14.4(b), is $2p$.

Theorem 14.2: PoA and PoS in SGG

PoA and PoS in SGGs are both $\Theta(n/k)$ in the worst case.

Proof. For the upper bound on PoA, fix an arbitrary NE $s \in \mathcal{T}$. Let $k' = \lfloor k/2 \rfloor$, and for each node $i \in \mathcal{V}$ that buys a good, consider $\mathcal{N}_i^{(k')}$, the set of nodes within distance k' from i (called a *ball* around i). Since each pair of nodes that buy a good are at distance at least $k + 1$ from each other, these balls are pairwise disjoint.

Call a ball $\mathcal{N}_i^{(k')}$ *big* if it has at least k' nodes, and *small* otherwise. If a ball $\mathcal{N}_i^{(k')}$ is small, i is in a connected component with less than k' nodes, and there is no other node that buys a good in that component. Let c be the number of connected components in $\mathcal{G}$; there are at most n/k' big balls and c small balls. The number of nodes that buy a good is equal to the number of balls, which is at most $n/k' + c$.

Since the optimal social cost is at least cp, the ratio between the social cost of s and the optimum is at most $(n/k' + c)p/(cp) \leq n/k' + 1 = O(n/k)$.

For the lower bound on PoS, given integers $k, m \geq 1$, consider a tree $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where there are two center nodes 1 and 2, and $2m$ simple paths with k nodes. For m of them (called *left arms*), one

of two endpoints is connected to 1. For the other m of them (called *right arms*), one of two endpoints is connected to 2. Finally, 1 and 2 are connected. Figure 14.4 shows such a graph with $k = 1$ and $m = n/2 - 1$. It is easy to see that the optimal social cost is at most $2p$, since if 1 and 2 buy a good, all nodes can access at least one good.

We claim that any NE has social cost at least mp. Fix an NE s. In s, either 1 or 2 does not buy a good, since they can access each other's goods. Without loss of generality, suppose that 1 does not buy a good. Consider a left arm, specifically the endpoint of the arm not connected to 1. The only nodes within distance k from this endpoint are 1 and the nodes in the same arm. Since 1 does not buy a good, there must be a node in the same arm who buys a good. This argument holds for each left arm, so at least m nodes buy a good, establishing the claim.

Since the optimal social cost is at most $2p$ and any NE has social cost at least mp, PoS $\geq mp/(2p) = \Omega(n/k)$. Since PoS $\leq$ PoA, the theorem holds. ■

Intuitively speaking, the main reason for the inefficiency of NEs in SGGs is that high-degree nodes (i.e., nodes with many k-hop neighbors) are less likely to buy goods even when many neighbors can benefit from goods bought by high-degree nodes. Indeed, high-degree nodes are more likely to have a neighbor buying a good, and thus choose to free ride.

To incentivize high-degree nodes to buy goods, we can force their neighbors who access the good to pay an access fee to the node — as we do in SGG-ACs. In Figure 14.4(b), for example, the two high-degree nodes in the center still buy goods, even when they can free ride, since they receive access fees from ξ or more followers, minimizing social cost.

This improvement through access costs is formalized and generalized in Theorem 14.3, where we show that the worst-case PoS in SGG-ACs is $\Theta(\xi/k)$, which is significantly smaller than $\Theta(n/k)$ in SGGs. In particular, if $\xi \leq \max(2\lfloor k/2 \rfloor + 1, 2)$, then the PoS in SGG-ACs is 1, i.e., the social cost in the best equilibria is always optimal even in the worst case. Among ξ values satisfying the condition, the largest one (i.e., $\xi = \max(2\lfloor k/2 \rfloor + 1, 2)$) is preferred to minimize the PoA, which is inversely proportional to ξ, as shown by Theorem 14.4.

Theorem 14.3: PoS in SGG-AC

PoS in SGG-ACs is $\Theta(\xi/k)$ in the worst case. In particular, it is 1 (i.e., there are guaranteed to be socially optimal equilibria) if $\xi \leq \max(2\lfloor k/2 \rfloor + 1, 2)$.

Proof. For the upper bound, given $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, k, and ξ, let $\mathcal{A}^* \subseteq \mathcal{V}$ be a smallest set of the nodes such that for each node $i \in \mathcal{V}$, there is a node $j \in \mathcal{A}^*$ such that i and j are within distance k from each other (i.e., $|\mathcal{A}^*| \cdot p$ is the optimal social cost). Consider a strategy profile s where each node in $\mathcal{A}^*$ buys a good, and all other nodes access a good of a node in $\mathcal{A}^*$ such that for each $i \in \mathcal{A}^*$, the set of nodes that access i's good induces a connected subgraph. Call a node an *owner* if it buys a good. An owner i is called a *rich owner* if $|\mathcal{F}_i| \geq \xi$, and a *poor owner* otherwise. That is, an owner i is a rich owner if and only if $u_i(s)$ is at least the utility of renting a good. Note that $s \in \mathcal{T}$ is not an NE if and only if there is a poor owner who can access a good of another owner.

From s, we show how to construct an NE whose social cost is at most $\Theta(\xi/k) \cdot |\mathcal{A}^*| \cdot p$. If there is a poor owner i who can follow another owner j, let i follow j. For each node who previously followed i, if it can follow another owner, let it follow that owner. Call a node *underprivileged* if it previously followed i but cannot follow any other owner. Scan the list of underprivileged nodes sequentially. When ℓ is considered, let ℓ be an owner (call it a *new owner*) and be followed by all still underprivileged nodes

who can follow ℓ (and remove them from the underprivileged nodes list). At the end of this loop no node is left underprivileged. Repeat until no poor owner i can follow another owner j.

One of the invariants of this procedure is that any new owner ℓ can never access a good of any other owner — ℓ became a new owner since it could not access a good of any other owner, and, subsequently, any node $i \in \mathcal{N}_\ell^{(k)}$ cannot become a new owner, as it can follow ℓ, i.e., $\ell \in \mathcal{N}_i^{(k)}$. Therefore, this procedure always terminates, after having at most $|\mathcal{A}^*|$ owners follow other owners. The final strategy after termination is an NE since there is no underprivileged node and no poor owner who can follow another owner.

To bound the number of new owners, note that when an owner $i \in \mathcal{A}^*$ deviated to follow another owner j, among i's previous followers including i (call them $\mathcal{C}_i$), at most $\max(1, \lfloor \frac{\xi}{\lfloor k/2 \rfloor + 1} \rfloor)$ new owners can be created. This is because $|\mathcal{C}_i| \leq \xi$ (since i was a poor owner) and if we let $k' = \lfloor k/2 \rfloor$, and consider the ball $\mathcal{N}_\ell^{(k')}$ around each new owner ℓ, $|\mathcal{N}_\ell^{(k')} \cap \mathcal{C}_i| \geq \min(k'+1, |\mathcal{C}_i|)$ and all balls are pairwise disjoint (all new owners are at distance at least $k+1$ from each other). The deviation of i creates at most $\max(1, \lfloor \frac{\xi}{k'+1} \rfloor)$ new owners. Therefore, the number of owners in the final NE is at most $|\mathcal{A}^*| \cdot \max(1, \lfloor \frac{\xi}{k'+1} \rfloor) = O(\frac{\xi}{k} \cdot |\mathcal{A}^*|)$. If $\xi \leq 2k'+1$, $\max(1, \lfloor \frac{\xi}{k'+1} \rfloor) = 1$ so the resulting NE is a social optimum. The same conclusion holds when $\xi \leq 2$ since one deviation creates at most one new owner.

For the lower bound of $\Omega(\xi/k)$, for any integers k, ξ such that $m = \frac{\xi-1}{k}$ is an integer, build the same tree as in the proof of Theorem 14.2. As before, the optimal social cost is $2p$, and the social cost at NEs is at least mp. To see why the latter claim holds, note that (as before) 1 and 2 cannot simultaneously be owners because the total number of nodes other than 1 and 2 is $2(\xi - 1)$, so that at least one of 1 and 2 must be a poor owner. Using the same argument as before, the number of owners is at least $m = \frac{\xi-1}{k}$, and PoS $\geq \frac{\xi-1}{2k} = \Omega(\frac{\xi}{k})$. ■

Theorem 14.4: PoA in SGG-AC

PoA in SGG-ACs is $\Theta(\max(n/k, n/\xi))$ in the worst case.

Proof. For the upper bound, fix an arbitrary NE $s \in \mathcal{T}$. Let O be the set of nodes who buy a good in s, and call them *owners*. For each owner $i \in O$, we call i a *rich owner* if $|\mathcal{F}_i| \geq \xi$, i.e., $u_i(s)$ is at least the utility of renting a good. Otherwise, call i a *poor owner*. The number of rich owners is at most $n/(\xi+1)$.

Note that any two poor owners are at distiance at least $k+1$ from each other, since otherwise they prefer to access the other's good. Let $k' = \lfloor k/2 \rfloor$, and for each poor owner $i \in \mathcal{V}$, consider $\mathcal{N}_i^{(k')}$, the set of nodes within distance k' from i (called a *ball* around i). Since each pair of poor owners are at distance at least $k+1$ from each other, these balls are pairwise disjoint.

Call a ball $\mathcal{N}_i^{(k')}$ *big* if it has at least k' nodes, and *small* otherwise. If a ball $\mathcal{N}_i^{(k')}$ is small, i is in a connected component with less than k' nodes, and there is no other owner in that component. Let c be the number of connected components in G. Since there are at most n/k' big balls and c small balls, the number of poor owners is equal to the number of balls, which is at most $n/k' + c$. The total number of owners is at most $n/k' + n/(\xi+1) + c$.

Since the optimal social cost is at least cp, the gap between the optimal social cost and the social cost of s is $\frac{(n/k' + n/(\xi+1) + c)p}{cp} = O(n/k + n/\xi) = O(\max(n/k, n/\xi))$.

For the lower bound of $\Omega(n/k)$, for any integers $k, m, \xi \geq 1$, consider a tree $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where there is a center node 1, and m simple paths with k nodes. For m of them (called *arms*), one of two endpoints is connected to 1. It is easy to see that the optimal social cost is at most p, since if 1 buys a good, all nodes can access it. But if we consider a strategy profile where the endpoint not connected to 1 buys a good for each arm and each node follows the endpoint of its own arm (1 follows an arbitrary one), it is an NE of social cost m. Therefore, the gap is $m = \Omega(n/k)$.

For the lower bound of $\Omega(n/\xi)$, for any integers $k, m, \xi \geq 1$, let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the complete graph on $m(\xi + 1)$ vertices. Obviously the optimal social cost is p, but if m nodes buy a good and have ξ followers each, it becomes an NE with social cost mp. The gap is $\Omega(n/\xi)$. Combining the two lower bounds, the theorem is proved. ■

Note that PoA in SGG-AC has the same order as PoA in SGG if $\xi = \Omega(k)$.

14.4.2 Convergence Analysis

For any SGG and SGG-AC, SGG-NASH, described in Section 14.3, always converges to an NE within a constant number of iterations, as formalized in Theorem 14.5.

Theorem 14.5: Convergence of SGG-NASH

For any SGG and SGG-AC, Algorithm 14.1 converges to an NE after at most three **while** loops in lines 5–10.

Proof. Assume first that the game is an SGG-AC. Given a strategy profile s, call a node *underprivileged* if it is not accessing a good (it may have neighbors buying a good), and call a node an *owner* if it buys a good. There are four types of possible deviations a node can use to improve its utility.

- **Case 1.** An underprivileged node buys a good since it has no k-hop neighbor buying a good.
- **Case 2.** An underprivileged node rents a good (free rides in SGG) since it has a k-hop neighbor buying a good but has at most $\xi - 1$ followers.
- **Case 3.** A non-owner buys a good, even though it has a k-hop neighbor buying a good, because it has at least ξ followers (who were underprivileged).
- **Case 4.** An owner changes its action to renting (free riding in SGG) since it has a k-hop neighbor buying a good but has at most $\xi - 1$ followers.

In SGGs, only Cases 1 and 4 are possible, ignoring the conditions on the number of followers for Case 4. We now prove that after at most 3 iterations of the **while** loop, s is an NE.

Claim 14.1

After the first iteration, Case 3 cannot happen.

Proof. Assume for contradiction that Case 3 happened for a node i after the first iteration. Right before this, i did not own a good, so either i never owned a good, or it owned a good (via Case 1 or 3) and rented a good (free rode in SGG) later (via Case 4). Note that a node cannot get any more followers while it does not own a good. If i never owned a good, it means all its followers followed it from the

beginning, so i should have bought a good in the first iteration, leading to contradiction. If i once owned a good but decided to rent a good (free ride in SGG) later via Case 4, the number of its followers at the moment of the decision was at most $\xi - 1$, and it could not have gained additional followers since then — again leading to a contradiction. ■

Claim 14.2

After the second iteration, Case 4 cannot happen.

Proof. Assume for contradiction that Case 4 happened for a node i after the second iteration, which means that i gave up its good and decided to rent a good (free ride in SGG). Note Case 1 does not apply to i's k-hop neighbors when i owned the item, and by Claim 14.1, even Case 3 does not apply after the first iteration. This implies after the first iteration, no new owner appeared among i's k-hop neighbors, and the number of i's followers only increased. Thus, either Case 4 should have applied to i earlier, or i should not have bought the good (Case 3 did not apply), leading to a contradiction. ■

Claim 14.3

After the third iteration, Case 1 and Case 2 cannot happen.

Proof. If a node is underprivileged after the third iteration, Case 4 must have happened in the third iteration or later, contradicting Claim 14.2. ■

Combining all the claims, s converges to an NE after at most three **while** loops. ■

14.4.3 Complexity Analysis

We analyze the time and space complexities of SGG-NASH, described in Algorithm 14.1.

Time Complexity Analysis. The time complexity of SGG-NASH is formalized in Theorem 14.6. If we consider a good sharable only with direct neighbors (i.e., $k = 1$), the time complexity becomes linear in the number of nodes and edges, as formalized in Corollary 14.1.

Theorem 14.6: Time Complexity of SGG-NASH

For each node $i \in \mathcal{V}$, let $d_i^{(k)} := \sum_{j \in \mathcal{N}_i^{(k-1)}} |\mathcal{N}_j^{(1)}|$. Then, the time complexity of Algorithm 14.1 is $O(\sum_{i=1}^{n} d_i^{(k)})$.

Sketch of Proof. Initializing the strategies (lines 1–3 of Algorithm 14.1) takes $O(\sum_{i=1}^{n} d_i^{(k)})$ time. Shuffling the order of nodes 4 takes $O(n)$ time. Each iteration of the **for** loop (lines 6–10) takes $O(d_i^{(k)})$ time. Thus, each iteration of the **while** loop (lines 5–10) takes $O(\sum_{i=1}^{n} d_i^{(k)})$ time. By Theorem 14.5,

Table 14.5: **Summary of the networks used in our experiments.** M: million, K: thousand.

Name	#Nodes	#Edges	Type
Star	100	99	Star graph
Chain	100	99	Chain graph
Random	50	127	Erdős-Rényi random graph with prob. 0.1
KarateClub [Zac77]	34	78	Friendship network
Hamsterster	1,858	12.5K	Friendship network
Advogato [MST09]	5,155	51.1K	Trust network
Orkut [MMG+07]	3.07M	117M	Friendship network

the **while** loop converges within a constant number of iterations. Thus, the time complexity of Algorithm 14.1 is $O(n + \sum_{i=1}^{n} d_i^{(k)}) = O(\sum_{i=1}^{n} d_i^{(k)})$. ■

Corollary 14.1: Time Complexity of SGG-NASH for Goods Sharable with Direct Neighbors

Let n and m be the numbers of nodes and edges, respectively, in the input social network. If $k = 1$, the time complexity of Algorithm 14.1 is $O(\sum_{i=1}^{n} d_i^{(1)}) = O(\sum_{i=1}^{n} |\mathcal{N}_i^{(1)}|) = O(n + m)$.

Space Complexity Analysis. The space complexity of SGG-NASH is linear in the number of nodes and edges. That is, if we let n and m be the numbers of nodes and edges, respectively, in the input social network, SGG-NASH requires $O(n + m)$ space. This is because SGG-NASH requires $O(m)$ space for maintaining the input graph, and $O(n)$ space for maintaining the current strategy of the nodes.

14.5 Experiments

We review our experiments for answering the following questions:

- **Q1. Inefficiency of NEs in SGGs**: How socially inefficient are NEs on synthetic and real social networks without access costs?
- **Q2. Effect of the Access Cost**: How do access costs affect the social inefficiency of NEs?
- **Q3. Socially Optimal Access Costs**: What is the optimal range of access costs for minimizing the social inefficiency of NEs?
- **Q4. Effect of the Degree of Sharing (i.e., k)**: How do k values in our models affect the social inefficiency of NEs?
- **Q5. Scalability of SGG-NASH**: How does the running time of SGG-NASH scale to the size of the input social network?

14.5.1 Experimental Settings

Machine: We ran all experiments on a PC with a 3.60GHz Intel i7-4790 CPU and 32GB memory.

Table 14.6: **Social costs when $k = 1$.** Social costs of social optima, NEs in SGGs, and NEs in SGG-ACs with different access costs are compared. For NEs, we report the average social cost of $1,000$ NEs returned by SGG-NASH. We set the price p, which is simply a scale factor, to 1. The numbers in the parentheses indicate the standard deviations.

Dataset	Social Cost						
	Optimum	NEs	NEs in SGG-ACs				
	Cost	in SGGs	$\xi = 1$	$\xi = 2$	$\xi = 5$	$\xi = 10$	$\xi = 20$
Star	1	97.6 (9.50)	**1.69 (10.9)**	3.06 (12.6)	5.92 (21.4)	10.2 (28.6)	20.5 (39.1)
Chain	34	43.5 (1.37)	45.5 (1.65)	**43.5 (1.36)**	43.5 (1.38)	43.5 (1.35)	43.5 (1.38)
Random	11	17.0 (1.92)	18.9 (1.46)	**15.9 (1.70)**	17.1 (1.91)	17.1 (1.90)	17.1 (1.90)
KarateClub	4	17.0 (3.31)	10.2 (1.73)	**8.46 (3.14)**	14.3 (4.80)	17.0 (3.36)	17.1 (3.26)
Hamsterster	241	970 (23.6)	526 (12.4)	**426 (16.3)**	581 (36.8)	827 (49.8)	956 (32.1)
Advogato	806	2643 (27.1)	1468 (22.5)	**1196 (30.7)**	1475 (56.2)	1887 (82.6)	2199 (101)

Table 14.7: **Social costs when $k > 1$.** Social costs of social optima, NEs in SGGs, and NEs in SGG-ACs are compared. We set ξ to 6 on KarateClub and 4 on the others. We set the price p, which is simply a scale factor, to 1. For NEs, we report the average social cost of $1,000$ NEs returned by SGG-NASH.

Dataset	Outcomes	Social Cost		
		$k = 2$	$k = 3$	$k = 4$
KarateClub	Optimum Cost	2	1	1
	NEs in SGGs	3.23	1.76	1.28
	NEs in SGG-ACs	3.23	1.75	1.26
Hamsterster	Optimum Cost	65	32	28
	NEs in SGGs	188	90.7	46.3
	NEs in SGG-ACs	128	65.0	41.7
Advogato	Optimum Cost	156	69	58
	NEs in SGGs	670	283	102
	NEs in SGG-ACs	377	129	86.5

Datasets: The synthetic and real social networks used in our experiments are listed in Table 14.5. The Orkut dataset was used only to test the scalability of SGG-NASH.

Calculation of Social Costs: We estimated the social cost of the NEs on each network by (a) finding $1,000$ NEs using SGG-NASH, which we implemented in Java 1.7, and then (b) averaging their social costs. We exactly calculated the optimal social cost on each network by (a) formulating the problem of calculating the optimal social cost as an integer program and then (b) solving it using intlinprog in MATLAB R2015a.[2]

[2]The problem of calculating the optimal social cost is NP-hard, since it is equivalent to the *minimum k-dominating set* problem, which is known to be NP-hard [HL91]. Fortunately, we could easily formulate the problem as an integer program, and solve it using intlinprog in MATLAB.

14.5.2 Q1. Inefficiency of NEs in SGGs

As seen in Table 14.6, the largest inefficiency (i.e., social cost in NEs / the optimal cost) 97.6 was obtained on the star graph. This is because the efficient NE, where only the center node buys a good, is unlikely to be realized, as the center node loses the incentive to buy a good as soon as any of the others does. Thus, the inefficient NE, where all nodes except the center node buy a good, is realized with high probability. On the chain graph and the random graph, however, the inefficiency was only 1.28 and 1.55, respectively, since high-degree nodes, which are main source of inefficiency, do not exist. On real networks, where high-degree nodes exist (albeit not as extreme as the star graph), the inefficiency was between 3.28 and 4.25.

14.5.3 Q2. Effect of the Access Cost on the Inefficiency of NEs

As seen in Table 14.6, the inefficiency significantly decreased in SGG-ACs with an appropriate access cost, compared to SGGs. In particular, on the star graph, the inefficiency decreased by 93%. This is because the inefficient NE, where all nodes except the center node buy the good, is unlikely to be realized, as the center node can still buy a good even when it has neighbors buying goods. For similar reasons, the inefficiency on real networks also decreased by $50\% - 56\%$. Since the inefficiency on the chain and random graphs was already low in SGGs, the improvement was smaller.

14.5.4 Q3. Socially Optimal Access Costs

As seen in Table 14.6, inefficiency was lowest at $\xi = 2$ ($p/3 < a < p/2$), consistent with our suggestion of $\xi = \max(2\lfloor k/2 \rfloor + 1, 2)$ in Section 14.4. The inefficiency tended to increase as ξ increased.

14.5.5 Q4. Effect of the Degree of Sharing (i.e., k) on the Inefficiency of NEs

Social costs on real networks when $k > 1$ are shown in Table 14.7, where we set ξ to 6 on KarateClub and 4 on the others. In both SGGs and SGG-ACs, inefficiency in NEs decreased as k increases, which is consistent with Theorems 14.2, 14.3, and 14.4. Although the inefficiency was still smaller in SGG-ACs than in SGGs, the gap decreased as k increases.

14.5.6 Q5. Scalability of SGG-NASH

We measured the running times of SGG-NASH on social networks of different sizes that were obtained by sampling different numbers of edges from the Orkut dataset. We assumed that goods can be shared only with direct neighbors, and thus we fixed k to 1. As seen in Figure 14.1(b) in Section 14.1, SGG-NASH scaled linearly with the number of edges in the input social network. This result is consistent with Corollary 14.1 in Section 14.4.3.

14.6 Related Work

We first review previous work directly related to network games with shareable goods. See the work of Galeotti et al. [GGJ+10], and the chapter by Jackson and Zenou [JZ15], for an introduction to network games in general.

Bramoullé and Kranton [BK07b] and Bramoullé et al. [BKD14] study a network game where the strategy of each node is its contribution to a public good, and its utility is a function of its own contribution and that of its direct neighbors.

Ballester et al. [BCAZ06] consider a similar game where, however, the utility of each node is a concave function of its effort and a linear function of its neighbors'. With these conditions, the game has a unique NE where the effort of each node is proportional to its Bonacich centrality score. Elliott and Golub [EG13] study an extended game in directed, weighted graphs where an edge (i, j) indicates the marginal benefit that node i can provide to node j. Allouch [All15] develops a model where consumptions of both private goods and public goods are taken into account.

In our models (of Section 14.2), each node simply decides whether to buy a good or access its neighbors', rather than the amount of effort. Instead, we extend the previous models, especially the best-shot game [Hir83], in that (a) access costs can be imposed on free riders and (b) nodes can benefit not only from direct neighbors but also k-hop neighbors. In contrast to previous work, we analyze PoA and PoS, and provide empirical results on real social networks.

Our work is also related to the huge body of literature on the price of anarchy. The concept itself is due to Koutsoupias and Papadimitriou [KP99], and the price of stability was introduced a few years later by Anshelevich et al. [ADK+08]. These concepts underlie much work at the intersection of game theory and AI, e.g., in computational social choice [BCMP13], security games [LV15], and routing [VFH15].

To the best of our knowledge, the price-of-anarchy paper that is most closely related to ours is the one by Kun et al. [KPR13]. They give bounds on the price of anarchy of an anti-coordination game played on a graph, albeit a fundamentally different one: each player chooses a color, and the utility of a player is the number of neighbors with different colors. In their work, k-hop neighbors are not considered, and access costs are incompatible with the model. It is also worth mentioning that our use of access costs to reduce the inefficiency of equilibria is conceptually related to work on taxation in congestion games [CKK06].

14.7 Summary

In this chapter, we propose game-theoretic models, namly SGG and SGG-AC, for capturing incentives to buy a good sharable with k-hop neighbors on a social network. With our models, we find that social inefficiency, specifically overproduction of goods, can occur in Nash Equilibria (NEs). However, we also show that this inefficiency can be reduced significantly by sharing goods with more players (i.e., increasing k) or imposing access costs to free riders. We provide efficiency analysis of NEs (in terms of PoA and PoS) and simulations on real-world social networks to support our findings. For the simulations, we develop SGG-NASH, a fast algorithm for finding NEs of both models.

In our view, the most actionable conclusion from our work is that in the type of scenarios under consideration (shareable goods on a network), access costs should be imposed when possible. This would be hard to do at a societal level for things like ski equipment and portable cribs. However, it certainly seems feasible at the level of an organization. For example, a university could mandate access costs for expensive lab equipment bought by individual researchers, as this would actually decrease the amount of grant money that is invested in buying equipment. For the designer of a multi-agent system, imposing access costs is trivial, and, similarly, might lead (*would* lead, if one trusts our analysis) to significant benefits.

Chapter based on work that appeared at WWW 2018 [SSK+18].

Chapter 15

Modeling Progression of Users on Social Media

How do the behaviors of users in a web service, such as social media, evolve over time? To reach a certain engagement level, what are the common stages that many users go through? How can we represent the stage that each individual user lies in?

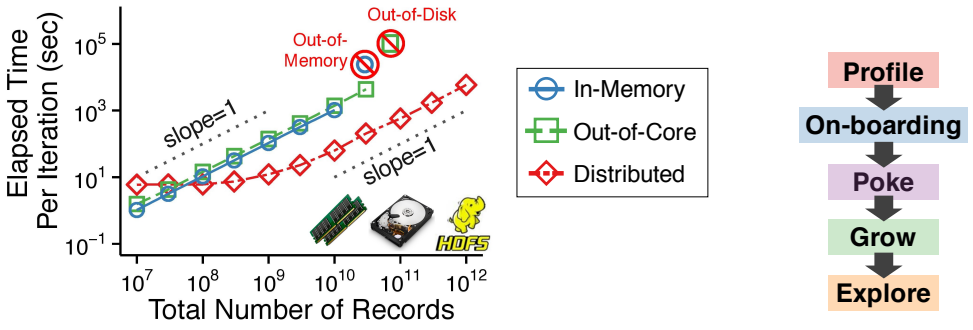
To answer these questions, we propose SWATT, a behavior model for the progressions of users' behaviors from a given starting point – such as a new subscription or first experience of certain features – to a particular target stage such as a predefined engagement level of interest. Under our model, transitions over stages represent progression of users where each stage in our model is characterized by probability distributions over types of actions, frequencies of actions, and next stages to move. Each user performs actions and moves to a next stage following the probability distributions characterizing the current stage.

We also develop SWATTFIT, a fast and memory-efficient algorithm for fitting our model to large-scale behavioral logs, which are modeled as a *tensor* with three modes: users, types of actions, and timestamps. Our algorithm scales linearly with the size of data, and especially, its distributed version implemented in the MAPREDUCE framework successfully handles petabyte-scale data with one trillion actions.

Lastly, we show the effectiveness of SWATT and SWATTFIT by applying them to real-world data from LinkedIn. We discover meaningful stages that LinkedIn users go through leading to predefined target goals. In addition, our trained models are shown to be useful for downstream tasks such as prediction of future actions.

15.1 Motivation

The behaviors of users of web-sites and apps change over time for various reasons including temporal trends [HM16] and shift of personal interests [Liu15]. When these behavioral changes are aligned with a certain direction, we observe the progression of user behavior. For example, in RateBeer, a beer review site, new users have similar tastes, but they start reviewing different types of beers as users gain experience and develop their own preferences [ML13, YML+14]. Another example is Wikipedia, where users utilize different navigation strategies throughout the information seeking process in order to reach the target information [WL12].



(a) Scalability of SWATTFIT (b) Progression on LinkedIn

Figure 15.1: **Scalability and effectiveness of our tools.** *(a) Scalability:* every version of SWATTFIT, our proposed optimization algorithm, scales linearly with the size of the input log data. Especially, the distributed version implemented in the MAPREDUCE framework scales to log data with **one trillion records**. *(b) Effectiveness:* Using SWATT and SWATTFIT, we discover meaningful stages that LinkedIn users go through. See Section 15.5.2 for a detailed description of the stages.

Understanding such progressions is of paramount importance to providing more personalized experiences, which can lead to greater engagement and potentially increasing revenue. To revisit the RateBeer example, the site may want to advertise beers to users based on their current tastes or recommend new flavors of beers considering the next stages in the progression of the users' tastes.

While general understanding of those progression patterns is important, businesses are often interested in behavior progressions only from one state to another state due to strategic or organizational interests. For instance, a company may want to strategically focus on helping onboarding users to reach a certain level of engagement, or it may need to study users who eventually bring revenue. In another example, at LinkedIn, a professional online social network providing services including news feeds and job pages, a product manager for news feeds needs to study the progressions of users who eventually become engaged with the news feed. Another product manager for job pages may be interested in navigational patterns of users who are looking for jobs. Hence, insights about progressions of users with respect to a specific target state are very helpful for establishing practical strategies.

To summarize and better understand such progressions toward a target state, we need to identify common stages that many users go through while interacting with a web service. For thorough summarization, these stages should capture changes in three different aspects that we describe below.

The first aspect is the change in the types of actions performed or features frequently used by users. For example, in an online social network service like LinkedIn, new users focus on making connections, while more established users with enough connections spend more time on consuming content or interacting with their connections. Changes in the types of actions performed by a user will then be considered as transitions between these stages.

Another aspect to consider is how often users visit, perform an action, or use a feature in a web service. Users who visit a web service every day and those who visit it once a year may not be in the same stage even though they perform similar actions or use similar features. This distinction is particularly important because service providers typically distinguish users by their level of activity and often aim to promote user activity and engagement.

The last aspect is the direction of changes toward a given target state. New users who are getting familiar with the web service and more established users who are becoming less active may be in

different stages independent of how similar the other aspects of their behavior are. That is, stages should describe not only current behavior but also transitions to future stages.

In this chapter, we propose SWATT (**S**tages **w**ith **a**ctions, **t**ime gaps, and **t**ransitions), a behavior model with stages characterizing all the aforementioned aspects of changes, while most existing models target only one of these aspects (e.g., only types of actions [ML13, YML+14]). Specifically, each stage in our model is characterized by a probability distribution over types of actions, a probability distribution over frequencies of actions, and transition probabilities of moving to another stage as the next stage. Progressions of users' behaviors can then be defined as transitions between these stages. That is, each user performs actions and moves to the next stage following the probability distributions describing the current stage.

Our algorithmic contribution is SWATTFIT, a fast and memory-efficient algorithm for training the parameters (i.e., probability distributions) of our model. It aims to find the parameters that best describe given behavior logs, modeled as a tensor with three modes: users, types of actions, and timestamps. Our algorithm scales linearly with the number of records (i.e., number of non-zero entries in the input tensor) and handles extremely large datasets that do not fit in main memory. Especially, its distributed version, implemented in the MAPREDUCE framework [DG08], successfully handles a petabyte-scale dataset with one trillion records, as shown in Figure 15.1(a).

We apply our model to real-world data from LinkedIn, discovering meaningful stages that LinkedIn users go through for specific target states (See Figure 15.1(b) for an example). For example, our model accurately captures the on-boarding stages that LinkedIn provides to new users. We also show that stage information inferred by our model is useful for downstream tasks including prediction of future actions. While the empirical evaluations of our model focus on the progressions on an online social network (i.e., LinkedIn), our model can be applied to any dataset with a series of actions by different users over time.

In summary, our main contributions are as follows:

- **Comprehensive behavior model**: we propose SWATT, a probabilistic behavior model that describes progressions of users' behaviors in three different aspects (Figure 15.2).
- **Scalable optimization algorithm**: we propose SWATTFIT, a fast and memory-efficient algorithm that fits the parameters of our model to trillion-scale behavior logs (Figure 15.1).
- **Experiments with real-world data**: we show the effectiveness of SWATT and SWATTFIT by applying them to real-world data from LinkedIn (Tables 15.2 and 15.4)

The rest of this chapter is organized as follows. In Section 15.2, we introduce SWATT, our proposed behavior model. In Section 15.3, we present SWATTFIT, our proposed algorithm for learning the parameters of SWATT. In Section 15.4, we theoretically analyze the time and space complexity of SWATTFIT. In Section 15.5, we share some experimental results. After reviewing related work in Section 15.6, we provide a summary of this chapter in Section 15.7.

15.2 Proposed Model: SWATT

In this section, we describe SWATT (**S**tages **w**ith **a**ctions, **t**ime gaps, and **t**ransitions), our behavior model for capturing progressive changes in users' behaviors. The symbols used to describe our model are listed in Table 15.1.

Table 15.1: **Table of frequently-used symbols.**

Symbol	Definition
$\mathcal{U}$	set of all users
$\mathcal{A}$	set of all types of actions
$\mathcal{D}$	set of all binned time gaps
$\mathcal{S} = \{s_1, ..., s_k\}$	set of all stages
n_u	number of actions done by user $u \in \mathcal{U}$
$g_u \in \{0, 1\}$	whether user $u \in \mathcal{U}$ reaches the target stage after doing n_u actions
$a_{u,j} \in \mathcal{A}$	type of the j-th action done by user $u \in \mathcal{U}$
$t_{u,j}$	timestamp of action $a_{u,j}$
$\delta_{u,j} \in \mathcal{D}$	binned time gap between $t_{u,j}$ and $t_{u,j-1}$
$\bar{a}_u$	sequence of actions performed by user $u \in \mathcal{U}$
$\bar{\delta}_u$	sequence of time gaps in $\bar{a}_u$
$\mathcal{S}_{-k}$	set of all stages excluding the target stage s_k
$z_{u,j} \in \mathcal{S}_{-k}$	stage of user u at $t_{u,j}$
$\bar{z}_u$	sequence of stages assigned to the actions in $\bar{a}_u$
$\theta_{s_i}, \phi_{s_i}, \psi_{s_i}$	probability distributions of actions, time-gaps, and transitions in stage $s_i \in \mathcal{S}_{-k}$
$\lambda_\theta, \lambda_\phi, \lambda_\psi$	hyperparameters regarding the prior distributions of $\theta_{s_i}, \phi_{s_i}, \psi_{s_i}$
ξ	hyperparameter for the initialization step

15.2.1 Notations and Model Description

Consider a set of users doing a sequence of actions. Let $\mathcal{U}$ be the set of users and $\mathcal{A}$ be the set of types of actions that can be done by the users. We bin the time gap between each two consecutive actions and use $\mathcal{D}$ to indicate the set of potential gap sizes.

Let $\mathcal{S} = \{s_1, s_2, ..., s_k\}$ be the set of k stages that the users may go through between the **starting stage** s_1 and the **target stage** s_k. Two stages s_1 and s_k are treated as special cases. That is, we have predefined conditions that determine whether each user has reached them (e.g., users reach the starting stage s_1 if they join LinkedIn and reach the target stage s_k as soon as they reach a certain number of connections).

We assume the monotonicity of the stages in $\mathcal{S}$ to model progression 'towards' the goal stage s_k. That is, for any i and j satisfying $1 \leq i < j \leq k$, users can transit from stage s_i to stage s_j but not in the opposite direction. This constraint, however, does not enforce that all users follow the same path towards the goal stage. Under our model, users are allowed to skip any intermediate stages.

Let $\mathcal{S}_{-k} = \{s_1, ..., s_{k-1}\}$ be the set of non-target stages. Each non-target stage $s_i \in \mathcal{S}_{-k}$ is characterized by probability distributions over types of actions, time gaps, and transitions from the stage, which are defined as follows:

- $\theta_{s_i} \in \mathbb{R}^{|\mathcal{A}|}$: probability distribution over types of actions performed by users in stage s_i.
- $\phi_{s_i} \in \mathbb{R}^{|\mathcal{D}|}$: probability distribution over time gaps between two consecutive actions performed by users in stage s_i.
- $\psi_{s_i} \in \mathbb{R}^{k-i+1}$: transition probability distribution over next stages moving from stage s_i before performing each action.

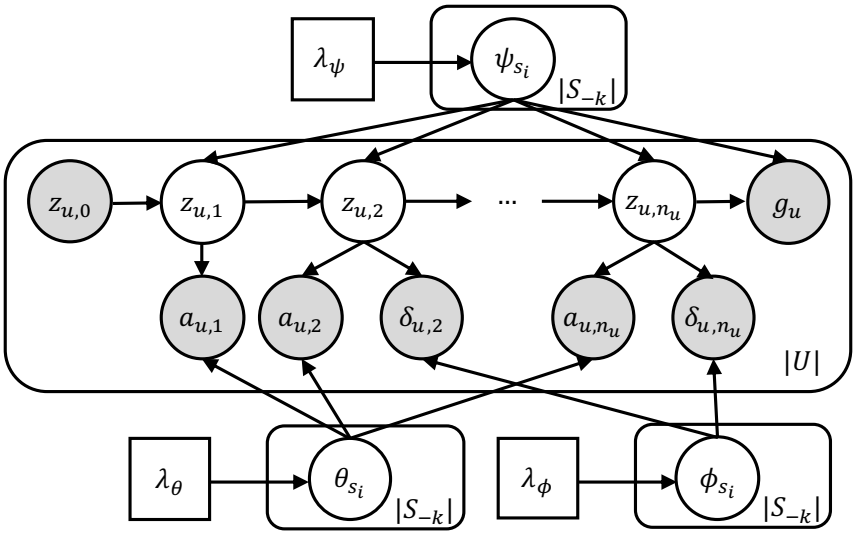


Figure 15.2: **Plate notation [Bun94] for our behavior model SWATT.** Observed variables are colored grey, and unobserved variables are colored white. The type of each action (i.e., each $a_{u,j}$) and the time-gap between each two consecutive actions (i.e., each $\delta_{u,j}$) depend on the current stage (i.e., each $z_{u,j}$) and the probability distributions characterizing each stage (i.e., each θ_{s_i} and ϕ_{s_i}). Transitions between stages depend on the transition probability distribution in each stage (i.e., each ψ_{s_i}). The starting stage (each $z_{u,0}$) and whether the target stage is reached by each user (i.e., each g_u) are observable.

We assume a symmetric Dirichlet prior [Bal06] over θ_{s_i}, ϕ_{s_i}, and ψ_{s_i}:

$$\theta_{s_i} \sim \mathsf{Dirichlet}(1+\lambda_\theta), \quad \phi_{s_i} \sim \mathsf{Dirichlet}(1+\lambda_\phi), \quad \text{and} \quad \psi_{s_i} \sim \mathsf{Dirichlet}(1+\lambda_\psi),$$

where λ_θ, λ_ϕ, and λ_ψ are hyperparameters.

We only consider actions performed after reaching the starting stage and before reaching the target stage (in cases where a user reaches it). For each user $u \in \mathcal{U}$, let n_u be the number of such actions by u, and let $g_u \in \{0,1\}$ indicate whether u reaches the target stage after performing the n_u actions. Then, we use $a_{u,j} \in \mathcal{A}$ to denote the type of the j-th action of u and use $t_{u,j}$ to denote the time when that action is performed. In addition, $\delta_{u,j} \in \mathcal{D}$ indicates the binned time gap between $t_{u,j}$ and $t_{u,j-1}$; and $z_{u,j}$ indicates the stage of u at $t_{u,j}$. For simplicity, we use $\bar{a}_u = (a_{u,1}, ..., a_{u,n_u})$, $\bar{\delta}_u = (\delta_{u,2}, ..., \delta_{u,n_u})$, and $\bar{z}_u = (z_{u,1}, ..., z_{u,n_u})$ to denote the sequences of actions, time-gaps, and assigned stages for user u.

15.2.2 Generative Process

The generative process of our SWATT model is described in Figure 15.2. For user $u \in \mathcal{U}$ who is in stage $z_{u,j-1}$ after doing her j-th action, she

1. moves to stage $z_{u,j}$ (which can be the same as $z_{u,j-1}$), where

$$z_{u,j} \sim \mathsf{Multinomial}(\psi_{z_{u,j-1}})$$

2. performs an action $a_{u,j}$ after a time gap $\delta_{u,j}$, where

$$a_{u,j} \sim \mathsf{Multinomial}(\theta_{z_{u,j}}), \quad \text{and} \quad \delta_{u,j} \sim \mathsf{Multinomial}(\phi_{z_{u,j}}).$$

In the beginning, each user $u \in \mathcal{U}$ moves from the starting stage s_1 (i.e., $z_{u,0} = s_1$), and the time-gap for her first action is ignored since there can be no previous action. Each user repeats this process until she reaches the target stage.

As seen in Figure 15.2, the following are observable from log data: (a) whether each user had reached the target stage before the log data were collected (i.e., $\{g_u\}_{u \in \mathcal{U}}$), (b) the number of actions performed by each user between the starting stage (inclusive) and the target stage (exclusive) before the log data were collected (i.e., $\{n_u\}_{u \in \mathcal{U}}$), (c) the type of each action (i.e., $\{\bar{a}_u\}_{u \in \mathcal{U}}$), and (d) the time gap between each two consecutive actions (i.e., $\{\bar{\delta}_u\}_{u \in \mathcal{U}}$).

15.3 Proposed Algorithm: SWATTFIT

In this section, we propose SWATTFIT, a scalable algorithm for training the parameters of SWATT. We first provide an overview of SWATTFIT with its objective function in Section 15.3.1. Then, we describe the details of SWATTFIT in Section 15.3.2. Lastly, we extend SWATTFIT to external-memory, multi-core, and distributed settings in Section 15.3.3.

15.3.1 Overview

We first introduce the objective function that SWATTFIT aims to optimize, and then we present an outline of SWATTFIT.

15.3.1.1 Objective function

Given behavior logs, which are a tensor with three modes (users, types of actions, and timestamps), we consider the sequences of actions $\{\bar{a}_u\}_{u \in \mathcal{U}}$, the sequences of time-gaps $\{\bar{\delta}_u\}_{u \in \mathcal{U}}$ and whether users reach the target stage $\{g_u\}_{u \in \mathcal{U}}$. SWATTFIT, described in detail in the following subsections, aims to find the most probable sequences of stages $\{\bar{z}_u\}_{u \in \mathcal{U}}$ as well as probability distributions of actions $\{\theta_{s_i}\}_{s_i \in \mathcal{S}_{-k}}$, time-gaps $\{\phi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}$, and transitions $\{\psi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}$. Maximizing the posterior probability of the parameters of our model given the observed states, written as

$$p(\{\theta_{s_i}\}_{s_i \in \mathcal{S}_{-k}}, \{\phi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}, \{\psi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}, \{\bar{z}_u\}_{u \in \mathcal{U}} | \{\bar{a}_u\}_{u \in \mathcal{U}}, \{\bar{\delta}_u\}_{u \in \mathcal{U}}, \{g_u\}_{u \in \mathcal{U}}, \lambda_\theta, \lambda_\phi, \lambda_\psi), \tag{15.1}$$

is equivalent to maximizing the following objective function f:

$$\begin{aligned} &f(\{\bar{z}_u\}_{u \in \mathcal{U}}, \{\theta_{s_i}\}_{s_i \in \mathcal{S}_{-k}}, \{\phi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}, \{\psi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}) \\ &:= \prod_{u \in \mathcal{U}} f_u(\bar{z}_u, \{\theta_{s_i}\}_{s_i \in \mathcal{S}_{-k}}, \{\phi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}, \{\psi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}) \times \prod_{s_i \in \mathcal{S}_{-k}} \left(p(\theta_{s_i} | \lambda_\theta) \times p(\phi_{s_i} | \lambda_\phi) \times p(\psi_{s_i} | \lambda_\psi)\right), \end{aligned} \tag{15.2}$$

where

$$\begin{aligned} &f_u(\bar{z}_u, \{\theta_{s_i}\}_{s_i \in \mathcal{S}_{-k}}, \{\phi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}, \{\psi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}) \\ &:= p(\bar{a}_u | \{\theta_{s_i}\}_{s_i \in \mathcal{S}_{-k}}, \bar{z}_u) \times p(\bar{\delta}_u | \{\phi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}, \bar{z}_u) \times p(\bar{z}_u | \{\psi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}) \times p(g_u | \{\psi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}, \bar{z}_u) \\ &= \left(\prod_{j=1}^{n_u} p(a_{u,j} | z_{u,j}, \theta_{z_{u,j}})\right) \times \left(\prod_{j=2}^{n_u} p(\delta_{u,j} | z_{u,j}, \phi_{z_{u,j}})\right) \times \left(\prod_{j=1}^{n_u} p(z_{u,j} | z_{u,j-1}, \psi_{z_{u,j-1}})\right) \times p(g_u | z_{u,N_u}, \psi_{z_{u,N_u}}). \end{aligned} \tag{15.3}$$

Notice that our objective function is non-convex and may have multiple local optima.

15.3.1.2 Outline of SWATTFIT

We present an outline of SWATTFIT, an iterative refinement algorithm for optimizing our objective function f (i.e., Eq. (15.2)). Different from general optimization algorithms for any graphical model (e.g., EM [DLR77]), SWATTFIT fully utilizes the dependency structure of our model for fast, memory-efficient, and parallel computation. SWATTFIT consists of the following three steps:

- **Initialization step** (Section 15.3.2.3): We initialize the probability distributions θ_{s_i}, ϕ_{s_i}, and ψ_{s_i} in every stage $s_i \in \mathcal{S}_{-k}$.
- **Assignment step** (Section 15.3.2.1): Given the current probability distributions θ_{s_i}, ϕ_{s_i}, and ψ_{s_i} in every stage $s_i \in \mathcal{S}_{-k}$, we update the stage assignments $\bar{z}_u$ of every user $u \in \mathcal{U}$ so that our objective function f is maximized.
- **Update step** (Section 15.3.2.2): Given the current stage assignments $\bar{z}_u$ of every user $u \in \mathcal{U}$, we update the probability distributions θ_{s_i}, ϕ_{s_i}, and ψ_{s_i} in every stage $s_i \in \mathcal{S}_{-k}$ so that our objective function f is maximized.

The initialization step is performed once initially. Then, the assignment and update steps are repeated until our optimization function f converges. Each assignment step and each update step are guaranteed to improve the objective function. Therefore, SWATTFIT is guaranteed to find a local optima.

15.3.2 Detailed Description

We describe each step of SWATTFIT in detail. For ease of explanation, we first present the assignment and update steps then present the initialization step.

15.3.2.1 Assignment Step

In this step, we maximize the objective function f by updating the stage assignments $\{\bar{z}_u\}_{u\in\mathcal{U}}$, while fixing the probability distributions $\{\theta_{s_i}\}_{s_i\in\mathcal{S}_{-k}}$, $\{\phi_{s_i}\}_{s_i\in\mathcal{S}_{-k}}$, and $\{\psi_{s_i}\}_{s_i\in\mathcal{S}_{-k}}$ to their current values. Once the probability distributions are fixed, the stage assignment $\bar{z}_u$ of each user $u \in \mathcal{U}$ can be optimized independently by maximizing f_u (i.e., Eq. (15.3)).

For each user $u \in \mathcal{U}$, we use dynamic programming [Sni10] to update $\bar{z}_u$, as described in detail in Algorithm 15.1. In the algorithm, $f_{u,j}(s_i)$ denotes the maximum posterior probability of the j-th or later actions and time gaps for user u given that $z_{u,j} = s_i$. That is, $f_{u,j}(s_i)$ is defined as follows:

$$f_{u,j}(s_i) := \max_{\{z_{u,l}\}_{l>j}} \Big(p(\{a_{u,l}\}_{l\geq j} | \{\theta_{s_i}\}_{s_i\in\mathcal{S}_{-k}}, \{z_{u,l}\}_{l>j}, z_{u,j} = s_i) \times p(\{\delta_{u,l}\}_{l\geq \max(j,2)} | \{\theta_{s_i}\}_{s_i\in\mathcal{S}_{-k}}, \{z_{u,l}\}_{l>j}, z_{u,j} = s_i) \times p(g_u | \{\psi_{s_i}\}_{s_i\in\mathcal{S}_{-k}}, \{z_{u,l}\}_{l>j}, z_{u,j} = s_i) \times p(\{z_{u,l}\}_{l>j} | \{\psi_{s_i}\}_{s_i\in\mathcal{S}_{-k}}, z_{u,j} = s_i) \Big). \quad (15.4)$$

We observe that $f_{u,j}(s_i)$ is computed easily from $\{f_{u,j+1}(s_i)\}_{s_i\in\mathcal{S}_{-k}}$ (lines 9-13). Our algorithm exploits this observation by computing $\{f_{u,j}(s_i)\}_{s_i\in\mathcal{S}_{-k}}$ in the decreasing order of j from n_u to 1 (lines 2-14). Note that the stage assignments maximizing each $f_{u,j}(s_i)$ is stored in $h_{u,j}(s_i)$ (line 14). Specifically, $h_{u,j}(s_i) = s_l$ means that the stage assignments maximizing $f_{u,j}(s_i)$ are $z_{u,j+1} = s_l$ and those maximizing $f_{u,j+1}(s_l)$.

Once we have computed $\{f_{u,1}(s_i)\}_{s_i\in\mathcal{S}_{-k}}$, which indicates the maximum posterior probabilities of everything except the initial transition, we can easily maximize f_u by finding $z_{u,1}$ maximizing $\psi_{s_1}(z_{u,1}) \cdot f_{u,1}(z_{u,1})$ (line 15). The stage assignments $\{\bar{z}_u\}_{u\in\mathcal{U}}$ maximizing f_u can be obtained from $\{\{h_{u,j}(s_i)\}_{s_i\in\mathcal{S}_{-k}}\}_{1\leq j\leq n_u}$ by following the path backward starting from $z_{u,1}$ (lines 16-17).

Algorithm 15.1 Assignment Step of SWATTFIT

Input: (1)log data: $\{\bar{a}_u\}_{u\in\mathcal{U}}$,$\{\bar{\delta}_u\}_{u\in\mathcal{U}}$,$\{g_u\}_{u\in\mathcal{U}}$
(2) probability distributions: $\{\theta_{s_i}\}_{s_i\in\mathcal{S}_{-k}}$, $\{\phi_{s_i}\}_{s_i\in\mathcal{S}_{-k}}$, $\{\psi_{s_i}\}_{s_i\in\mathcal{S}_{-k}}$
Output: stage assignments: $\{\bar{z}_u\}_{u\in\mathcal{U}}$

for each user $u \in \mathcal{U}$ **do**
  for each stage $s_i \in \mathcal{S}_{-k}$ **do**
    if $g_u = 0$ **then**
      $f_{u,n_u}(s_i) \leftarrow \theta_{s_i}(a_{u,n_u}) \cdot \phi_{s_i}(\delta_{u,n_u}) \cdot (1 - \psi_{s_i}(s_k))$
    else
      $f_{u,n_u}(s_i) \leftarrow \theta_{s_i}(a_{u,n_u}) \cdot \phi_{s_i}(\delta_{u,n_u}) \cdot \psi_{s_i}(s_k)$
  for $j = n_u - 1, ..., 1$ **do**
    for each stage $s_i \in \mathcal{S}_{-k}$ **do**
      $s_l \leftarrow \arg\max_{s_m : i \le m < k} (\psi_{s_i}(s_m) \cdot f_{u,j+1}(s_m))$
      if $j = 1$ **then**
        $f_{u,j}(s_i) \leftarrow \theta_{s_i}(a_{u,j}) \cdot \psi_{s_i}(s_l) \cdot f_{u,j+1}(s_l)$
      else
        $f_{u,j}(s_i) \leftarrow \theta_{s_i}(a_{u,j}) \cdot \phi_{s_i}(\delta_{u,j}) \cdot \psi_{s_i}(s_l) \cdot f_{u,j+1}(s_l)$
      $h_{u,j}(s_i) \leftarrow s_l$
  $z_{u,1} \leftarrow \arg\max_{s_i \in \mathcal{S}_{-k}} (\psi_{s_1}(s_i) \cdot f_{u,1}(s_i))$
  for $j = 2, ..., n_u$ **do**
    $z_{u,j} \leftarrow h_{u,j-1}(z_{u,j-1})$
return $\{\bar{z}_u\}_{u\in\mathcal{U}}$

15.3.2.2 Update Step

In this step, we maximize f by updating the probability distributions $\{\theta_{s_i}\}_{s_i\in\mathcal{S}_{-k}}$, $\{\phi_{s_i}\}_{s_i\in\mathcal{S}_{-k}}$, and $\{\psi_{s_i}\}_{s_i\in\mathcal{S}_{-k}}$, while fixing the stage assignments $\{\bar{z}_u\}_{u\in\mathcal{U}}$ to their current values. To this end, we decompose our objective function based on the probability distribution that each term depends on as follows:

$$f = \prod\nolimits_{s_i\in\mathcal{S}_{-k}} \left(f_\theta(\theta_{s_i}) \times f_\phi(\phi_{s_i}) \times f_\psi(\psi_{s_i})\right),$$

where

$$f_\theta(\theta_{s_i}) = p(\theta_{s_i}|\lambda_\theta) \times \prod_{u\in\mathcal{U}} \prod_{j:z_{u,j}=s_i} p(a_{u,j}|z_{u,j}, \theta_{z_{u,j}}), \tag{15.5}$$

$$f_\phi(\phi_{s_i}) = p(\phi_{s_i}|\lambda_\phi) \times \prod_{u\in\mathcal{U}} \prod_{j\ge 2:z_{u,j}=s_i} p(\delta_{u,j}|z_{u,j}, \phi_{z_{u,j}}), \tag{15.6}$$

$$f_\psi(\psi_{s_i}) = p(\psi_{s_i}|\lambda_\psi) \times \prod_{u\in\mathcal{U}} \Big(p(g_u|z_{u,N_u}, \psi_{z_{u,N_u}}) \times \prod_{j:z_{u,j-1}=s_i} p(z_{u,j}|z_{u,j-1}, \psi_{z_{u,j-1}})\Big). \tag{15.7}$$

Then, we update each probability distribution independently so that the terms depending on it are maximized. Notice that this update has an analytical solution.

Specifically, for each stage $s_i \in \mathcal{S}_{-k}$, we update the action type distribution θ_{s_i} so that $f_\theta(\theta_{s_i})$ (i.e., Eq. (15.5)) is maximized. For each type of action a, the probability $\theta_{s_i}(a)$ is updated as follows:

$$\theta_{s_i}(a) \leftarrow \frac{\lambda_\theta + c_{\mathcal{A}}(s_i, a)}{|\mathcal{A}|\lambda_\theta + \sum_{a'\in\mathcal{A}} c_{\mathcal{A}}(s_i, a')}, \tag{15.8}$$

where $c_{\mathcal{A}}(s_i, a) := \sum_{u \in \mathcal{U}} |\{1 \leq j \leq n_u : a_{u,j} = a \wedge z_{u,j} = s_i\}|$.

Likewise, for each stage $s_i \in \mathcal{S}_{-k}$, we update the time gap distribution ϕ_{s_i} so that $f_\phi(\phi_{s_i})$ (i.e., Eq. (15.6)) is maximized. For each time gap $\delta \in \mathcal{D}$, the probability $\phi_{s_i}(\delta)$ is updated as follows:

$$\phi_{s_i}(\delta) \leftarrow \frac{\lambda_\phi + c_{\mathcal{D}}(s_i, \delta)}{|\mathcal{D}|\lambda_\phi + \sum_{\delta' \in \mathcal{D}} c_{\mathcal{D}}(s_i, \delta')}, \tag{15.9}$$

where $c_{\mathcal{D}}(s_i, \delta) := \sum_{u \in \mathcal{U}} |\{2 \leq j \leq n_u : \delta_{u,j} = \delta \wedge z_{u,j} = s_i\}|$.

Lastly, for each stage $s_i \in \mathcal{S}_{-k}$, we update the transition probability ψ_{s_i} so that $f_\psi(\psi_{s_i})$ (i.e., Eq. (15.7)) is maximized. We update the transition probability $\psi_{s_i}(s_k)$ from s_i to the target stage s_k as follows:

$$\psi_{s_i}(s_k) \leftarrow \frac{\lambda_\psi + c_g(s_i)}{(k - i + 1)\lambda_\psi + c_{\mathcal{S}}(s_i)}, \tag{15.10}$$

where $c_g(s_i) := \sum_{u \in \mathcal{U}} \mathbf{1}(z_{u,N_u} = s_i \wedge g_u = 1)$ and $c_{\mathcal{S}}(s_i) := \sum_{u \in \mathcal{U}} |\{0 \leq j \leq n_u : z_{u,j} = s_i\}|$. Then, for each non-target stages $s_i, s_l \in \mathcal{S}_{-k}$ where $i \leq l$, we update the probability $\psi_{s_i}(s_l)$ of the transition from s_i to s_l as follows:

$$\psi_{s_i}(s_l) \leftarrow (1 - \psi_{s_i}(s_k)) \times \frac{\lambda_\psi + c_{\mathcal{S}}(s_i, s_l)}{(k - i)\lambda_\psi + \sum_{m=i}^{k-1} c_{\mathcal{S}}(s_i, s_m)}, \tag{15.11}$$

where $c_{\mathcal{S}}(s_i, s_l) := \sum_{u \in \mathcal{U}} |\{1 \leq j \leq n_u : z_{u,j-1} = s_i \wedge z_{u,j} = s_l\}|$.

15.3.2.3 Initialization Step

Since our objective function f (i.e., Eq. (15.2)) is non-convex, the solution found by SWATTFIT and its speed of convergence (i.e., the number of iterations required for convergence) depend on initial parameter values. In this section, we present an initialization method that works well in practice, as we experimentally show in Section 15.5.5 and Section 15.5.6.

First, we choose a subset of users and decide their stage variables in a simple way. Specifically, let $\mathcal{U}'$ be the set of users that has performed at least ξ ($\geq k - 1$) actions, where ξ is a hyperparameter. For each user $u \in \mathcal{U}'$, we divide her n_u stage variables into $(k - 1)$ continuous segments of equal length. Then, we assign stage s_i to the variables in each i-th segment.[1] In this process, the users with a small number of actions (i.e., $\mathcal{U} - \mathcal{U}'$) are ignored since they are less likely to have gone through all $(k - 1)$ stages.

Once the stage variables of $\mathcal{U}'$ are set, the action-type probability distributions $\{\theta_{s_i}\}_{s_i \in \mathcal{S}_{-k}}$ and the time-gap probability distributions $\{\phi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}$ are initialized by Eq. (15.8) and Eq. (15.9) with $\mathcal{U}'$ instead of $\mathcal{U}$. We initialize transition probability distributions $\{\psi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}$ so that the transition probability decreases exponentially with the distance from the current stage.[2]

15.3.3 Extensions to External-memory, Multi-core, and Distributed Settings

We extend SWATTFIT, described in the previous subsection, to the external-memory, multi-core, and distributed settings without affecting its outputs. We focus on extending the assignment and update steps. The ideas are also applicable to the initialization step due to its similarity to the other steps.

[1] i.e., for each $1 \leq j \leq n_u$, $z_{u,j}$ is set to stage s_i satisfying $\frac{n_u(i-1)}{(k-1)} < j \leq \frac{n_u i}{(k-1)}$.

[2] i.e., for each stage s_i, $\psi_{s_i}(s_l) \leftarrow \frac{2^{-(l-i)}}{\sum_{m=i}^{k} 2^{-(m-i)}}$.

15.3.3.1 External-memory Settings

SWATTFIT does not require loading the entire input data (i.e., $\{\bar{a}_u\}_{u\in\mathcal{U}}$, $\{\bar{\delta}_u\}_{u\in\mathcal{U}}$, $\{g_u\}_{u\in\mathcal{U}}$, and $\{\bar{z}_u\}_{u\in\mathcal{U}}$) in main memory at once. Instead, it can run by loading the data for each user into main memory, while storing the data for the other users in external memory (e.g., disk). This is particularly useful when the entire input data is too large to fit in main memory.

We assume that the input data are stored in external memory sorted by user ids (by any external sorting algorithm). We sequentially read input data until we load all data for a user into main memory. Then, we assign the user's stages (lines 2-17 of Algorithm 15.1). Based on these stages, we add the user's contributions[3] to the counts (e.g., $c_{\mathcal{A}}(s_i, a)$ and $c_{\mathcal{S}}(s_i, s_l)$) in Eq. (15.8)-Eq. (15.11) for every action type, time gap, and stage. Then, we free the memory space allocated for the current user and move to the next user by continuing reading the input data. After processing the last user, we compute the numerators and denominators of Eq. (15.8)-Eq. (15.11) for every action type, time gap, and stage, simply by adding constants to the sums of the users' contributions. We update the probability distributions by the equations and move to the next iteration.

In Section 15.5.4, we experimentally show that this out-of-core processing using external memory significantly reduces the main memory requirements with a slight compromise in speed.

15.3.3.2 Multi-core Settings

SWATTFIT is easily parallelized in multi-core settings. In the assignment step, the stage assignment for one user (lines 2-17 of Algorithm 15.1) does not depend on that for the other users. Thus, stage assignments for different users can be run in parallel using multiple threads. Likewise, in the update step, the contributions[3] of one user to the counts (e.g., $c_{\mathcal{A}}(s_i, a)$ and $c_{\mathcal{S}}(s_i, s_l)$) in Eq. (15.8)-Eq. (15.11) do not depend on those of the other users. Thus, computing the contributions of different users also can be run in parallel using multiple threads.

In Section 15.5.4, we experimentally show that speed-up by this parallelization is near linear to the number of threads. Notice that this parallel processing performs the same computation as the serial processing and thus has no effect on the outputs of SWATTFIT.

15.3.3.3 Distributed Settings

We combine the extensions in the previous sections for distributed settings. We assume that the input data are distributed across machines so that (a) all data for the same user are stored in one machine and (b) the data in each machine are sorted by user id. In MAPREDUCE [DG08], for example, this can be done by simply shuffling the data by user ids. Each machine sums up the contributions[3] of the assigned users to the counts (e.g., $c_{\mathcal{A}}(s_i, a)$ and $c_{\mathcal{S}}(s_i, s_l)$) in Eqs. (15.8)-(15.11) by sequentially reading the assigned input data, as in Section 15.3.3.1. Then, the contributions are gathered and summed up in one machine, which then updates all probability distributions by Eqs. (15.8)-(15.11) and broadcasts them to all other machines so that they can be used in the next iteration.

In Section 15.5.4, we experimentally show that the MAPREDUCE implementation of SWATTFIT successfully handles petabyte-scale data with one trillion actions.

[3]e.g., the contribution of user u to $c_{\mathcal{A}}(s_i, a)$ is $|\{1 \le j \le n_u : a_{u,j} = a \wedge z_{u,j} = s_i\}|$ and that to $c_{\mathcal{D}}(s_i, \delta)$ is $|\{2 \le j \le n_u : \delta_{u,j} = \delta \wedge z_{u,j} = s_i\}|$.

15.4 Theoretical Analysis

In this section, we analyze the time complexity and memory requirement of our optimization algorithm SWATTFIT.

15.4.1 Time Complexity Analysis

As formalized in Theorem 15.1, the time complexity of SWATTFIT is linear in the number of records (i.e., the total number of actions of all users).

Theorem 15.1: Time Complexity

Let $N = \sum_{u \in \mathcal{U}} n_u$ be the total number of actions and T be the number of iterations. If $N = \Omega(|\mathcal{A}|/k + |\mathcal{D}|/k)$, then the time complexity of our optimization algorithm is $O(TNk^2)$.

Sketch of Proof. The time complexity of the assignment step (i.e., Algorithm 15.1) is $O(Nk^2)$ because line 9, which takes $O(k)$ time, is executed $O(\sum_{u \in \mathcal{U}} n_u k) = O(Nk)$ times.

The time complexity of the update step is $O(N + |\mathcal{A}|k + |\mathcal{D}|k + k^2)$. For each of $\{(u, j) : u \in \mathcal{U}, 0 \leq j \leq n_u\}$, whose size is N, we need to increase a constant number of counts (e.g., $c_{\mathcal{A}}(s_i, a)$ and $c_{\mathcal{S}}(s_i, s_l)$) in Eq. (15.8)-Eq. (15.11). Updating the probability distributions from the counts by Eq. (15.8)-Eq. (15.11) takes $O(|\mathcal{A}|k + |\mathcal{D}|k + k^2)$ time.

Thus, one iteration (i.e., running the assignment and update steps once) takes $O(Nk^2 + |\mathcal{A}|k + |\mathcal{D}|k + k^2)$ time, which is $O(Nk^2)$ by our assumption. Hence, the total time complexity is $O(TNk^2)$. ■

15.4.2 Memory Requirement Analysis

As formalized in Theorem 15.2, the memory requirement of SWATTFIT is linear in the maximum number of actions among all users. Note that it is sub-linear in the number of records (i.e., the total number of actions of all users).

Theorem 15.2: Memory Requirement

If $\max_{u \in \mathcal{U}} n_u = \Omega(|\mathcal{A}| + |\mathcal{D}| + k)$, then the memory requirement of our optimization algorithm is $O(k \max_{u \in \mathcal{U}} n_u)$.

Sketch of Proof. As explained in Section 15.3.3.1, we need to load the input data for each user u (i.e., $\bar{a}_u$, $\bar{\delta}_u$, and g_u), whose sizes are $O(n_u)$, into main memory at a time. To assign the n_u stage variables for u (by lines 2-17 of Algorithm 15.1), we need $O(kn_u)$ memory space for maintaining $\{\{f_{u,j}(s_i)\}_{s_i \in \mathcal{S}_{-k}}\}_{1 \leq j \leq n_u}$, $\{\{h_{u,j}(s_i)\}_{s_i \in \mathcal{S}_{-k}}\}_{1 \leq j \leq n_u}$, and the assigned stage variables $\bar{z}_u$. We also need $O(|\mathcal{A}|k + |\mathcal{D}|k + k^2)$ memory space for maintaining the probability distributions (i.e., $\{\theta_{s_i}\}_{s_i \in \mathcal{S}_{-k}}$, $\{\phi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}$, $\{\psi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}$) and the counts (e.g., $c_{\mathcal{A}}(s_i, a)$ and $c_{\mathcal{S}}(s_i, s_l)$) in Eq. (15.8)-Eq. (15.11). Therefore, the total memory requirement is $O(\max_{u \in \mathcal{U}}(kn_u) + |\mathcal{A}|k + |\mathcal{D}|k + k^2)$, which is $O(k \max_{u \in \mathcal{U}} n_u)$ by our assumption. ■

15.5 Experiments

We review our experiments for answering the following questions:

- **Q1. Effectiveness**: Do SWATT and SWATTFIT discover meaningful progression stages in real-world data?
- **Q2. Applicability**: Are our trained models useful for downstream tasks such as prediction of future actions?
- **Q3. Scalability**: Does SWATTFIT scale linearly with the size of the input data? Can SWATTFIT handle a dataset with trillions of records?
- **Q4. Convergence**: How rapidly does SWATTFIT converge?
- **Q5. Identifiability**: How accurately does SWATTFIT estimate ground-truth parameters?

15.5.1 Experimental Settings

Datasets: We used a dataset provided by LinkedIn. The dataset is the log of the actions performed before mid June of 2017 by a subset of LinkedIn members who joined LinkedIn after mid April of 2017. The dataset has 22 types of actions including the followings:

- *visit*: visit LinkedIn.com.
- *profile-edit*: edit one's profile.
- *profile-view*: view another member's profile.
- *invite-abook*: invite a non-member to LinkedIn on a page that lists some emails imported from one's address book.
- *conn-abook*: send a connection request to a member on a page that lists some members imported from one's address book.
- *conn-ins*: send a connection request to a member on a page that lists some members in the same institution.
- *conn-rec*: send a connection request to a member on a page that lists some members recommended by LinkedIn.
- *conn-search*: send a connection request to a member on a page that shows the results of one's search.
- *conn-profile*: send a connection request to a member on the member's profile page.
- *conn-other*: send a connection request to a member by means other than those explained above.
- *conn-accept*: accept a connection request that one received.
- *job-view*: view a job posting.
- *message*: send a message to a member.

The time gaps between each consecutive actions performed by the same user are binned into *second* (within few seconds), *minute*, *hour*, *day*, *week*, *month*, and *over-a-month* (over a month).

Starting and Target Stages: We used the following two settings:

- CONNECTED: The starting stage is defined as joining LinkedIn, and the target stage is defined as reaching 30 connections. The number of actions between the stages is about 500 millions.

- ENGAGED: The starting stage is defined as having 30 or more connections, and the target stage is defined as visiting LinkedIn 4 days in a week for 5 (not necessarily consecutive) weeks. The number of actions between the stages is about 150 millions.

Implementations: We implemented SWATTFIT in Java 1.7 and Hadoop 2.6.1. We used SMILE v1.3 (https://haifengl.github.io/smile/) for logistic regression [McC84] and k-means++ [AV07].

15.5.2 Q1. Effectiveness: Descriptive Results

We present results demonstrating intuitive patterns extracted by SWATT for both settings of 'CONNECTED' and 'ENGAGED'. As shown in Table 15.2, SWATT[4] extracted the following reasonable latent stages for the CONNECTED setting:

- *profile* (stage 1): The first learned stage shows the behaviors of creating new profiles right after new members join the service.
- *on-boarding* (stages 2-4): The next three stages describe the optional on-boarding process provided to new members. During the process, LinkedIn provides new members with a list of other members to connect to and non-members to invite. New members can rapidly (notice that the most probable time gap is *second*) send connection requests and invitations.
- *poke* (stage 5): Members start poking other services and typically enjoying viewing other members' profiles.
- *grow* (stage 6): Members grow their networks by searching for other members and using connection recommendation services.
- *explore* (stage 7): In addition to growing their networks, members start consuming content. They visit LinkedIn for browsing content (notice that *visit* is the most probable action). They also start seeing job postings. Bursts of actions are reduced.

Notice that not every member goes through every stage. For example, many members in the *profile* stage skip the optional on-boarding process and jump directly to the *poke* or *explore* stage.

For the ENGAGED setting, the *grow* and *explore* stages (i.e., the last two stages extracted for the connected setting) were subdivided into multiple stages by SWATT[5] as follows:

- *active-grow* (stages 1-4): In the first four stages, members actively grow their networks by different means including connecting from profiles (*ag-1*), connecting from search (*ag-2*), importing address books (*ag-3*), and others (*ag-4*).
- *passive-grow* and *explore* (stages 5-7): In the stage *pg-1*, members passively grow their networks by relying on connection recommendation services. The next stage, *pg-2*, captures the transition to the *explore* stage.

Once stages and progression of each user is learned, we can answer simple queries like "what is the stage that most members skip?" or perform post-processing to gain more detailed insights. For example, we can analyze the most discriminative sequence of stages for a particular cohort to achieve a given target. Here, we use a cohort as the members who pass through the *explore* stage (s_7). To extract such paths, for a set of stages $S' \subseteq S = \{s_2, \cdots, s_6\}$, we define the *discriminative score* as:

$$\frac{P(\text{reach target} \mid \text{pass through } s_7 \text{ and all of } S')}{P(\text{reach target} \mid \text{pass through } s_7 \text{ but not all of } S')}.$$

[4] with $k = 8$, $\lambda_\theta = \lambda_\phi = \lambda_\psi = 0.1$, and $\xi = 30$.
[5] with $k = 8$, $\lambda_\theta = \lambda_\phi = \lambda_\psi = 0.1$, and $\xi = 50$.

Table 15.2: **Latent stages extracted by SWATT and SWATTFIT from the LinkedIn dataset.** The first table presents the stages after joining the service and before reaching 30 connections. Members (a) create their profiles, (b) go through on-boarding processes, (c) poke the service, (d) grow their networks, and finally (e) explore the service. The second table presents the stages after reaching 30 connections and before being engaged. Members (a) actively grow their networks by various means, (b) passively grow their networks relying on recommendation services, and finally (c) explore the service. For each stage, we list the three most probable types of actions, time gaps, and transitions, with their probabilities (∗∗∗∗: 80%-100%, ∗ ∗ ∗: 60%-80%, ∗∗: 30%-60%, ∗: 10%-30%). See Section 15.5.1 for descriptions of the types of actions, and see Section 15.5.2 for explanation of the stages.

Stages	***profile*** (s_1)	**on-boarding** ***ob-1*** (s_2)	***ob-2*** (s_3)	***ob-3*** (s_4)	***poke*** (s_5)	***grow*** (s_6)	***explore*** (s_7)
Actions (θ_{s_i})	*profile-edit* (∗∗) *visit* (∗∗) *conn-accept*	*conn-ins* (∗ ∗ ∗) *conn-other* (∗) *invite-abook*	*invite-abook* (∗∗∗∗) *conn-abook* (∗) *conn-ins*	*conn-abook* (∗∗∗∗) *conn-ins* (∗) *invite-abook*	*profile-view* (∗∗) *visit* (∗) *profile-edit*	*conn-rec* (∗ ∗ ∗) *conn-search* (∗) *profile-view*	*visit* (∗ ∗ ∗) *job-view* (∗) *profile-view* (∗)
Time-gaps (ϕ_{s_i})	*hour* (∗∗) *minute* (∗∗) *second* (∗)	*second* (∗∗∗∗) *minute* *hour*	*second* (∗∗∗∗) *minute*	*second* (∗∗∗∗) *hour* *minute*	*minute* (∗∗) *hour* (∗) *second*	*minute* (∗ ∗ ∗) *second* (∗) *hour*	*hour* (∗∗) *minute* (∗) *day* (∗)
Transitions (ψ_{s_i})	*profile* (∗∗∗∗) *explore* *poke*	*ob-1* (∗∗∗∗) *explore* *poke*	*ob-2* (∗∗∗∗) *ob-3* explore	*ob-3* (∗∗∗∗) *poke* *explore*	*poke* (∗∗∗∗) *explore* *grow*	*grow* (∗∗∗∗) *explore* *target*	*explore* (∗∗∗∗) *target*

Stages	**active-grow** ***ag-1*** (s_1)	***ag-2*** (s_2)	***ag-3*** (s_3)	***ag-4*** (s_4)	**passive-grow** ***pg-1*** (s_5)	***pg-2*** (s_6)	***explore*** (s_7)
Actions (θ_{s_i})	*profile-view* (∗∗) *visit* (∗) *conn-profile*	*conn-search* (∗∗) *conn-rec* (∗) *message* (∗)	*invite-abook* (∗∗) *conn-abook* (∗∗) *conn-rec*	*conn-other* (∗) *conn-rec* (∗) *conn-abook* (∗)	*conn-rec* (∗∗∗∗) *profile-view* *visit*	*conn-rec* (∗∗) *profile-view* (∗) *visit* (∗)	*visit* (∗∗) *profile-view* (∗) *job-view*
Time-gaps (ϕ_{s_i})	*minute* (∗ ∗ ∗) *hour* (∗) *second* (∗)	*minute* (∗ ∗ ∗) *second* (∗) *hour*	*second* (∗∗∗∗) *minute*	*second* (∗∗) *minute* (∗∗) *hour*	*second* (∗ ∗ ∗) *minute* (∗) *hour*	*minute* (∗ ∗ ∗) *hour* (∗) *second* (∗)	*minute* (∗∗) *hour* (∗∗) *day* (∗)
Transitions (ψ_{s_i})	*ag-1* (∗∗∗∗) *explore* *pg-2*	*ag-2* (∗∗∗∗) *explore* *pg-2*	*ag-3* (∗∗∗∗) *explore* *ag-4*	*ag-4* (∗∗∗∗) *explore* *pg-2*	*pg-1* (∗∗∗∗) *pg-2*	*pg-2* (∗∗∗∗) *explore*	*explore* (∗∗∗∗) *target*

Table 15.3: **Top-3 discriminative paths for those who pass through the *explore* stage.**

Target setting	Top-3 paths (discriminative score)
CONNECTED	*grow* (11.3) *poke* → *grow* (11.2) *poke* (8.7)
ENGAGED	*ag-3* → *ag-4* → *pg-1* → *pg-2* (1.73) *ag-2* → *ag-3* → *ag-4* → *pg-1* → *pg-2* (1.71) *ag-3* → *ag-4* → *pg-1* → *pg-2* (1.61)

Table 15.4: **Usefulness of our trained model for prediction tasks.** Prediction solely based on our model (i.e., SWATT), which is unsupervised, showed similar accuracy to logistic regression (i.e., LR), which is supervised. Combining our model and logistic regression (i.e., SWATT + LR) was most accurate for all the tasks. See Section 15.5.3 for details.

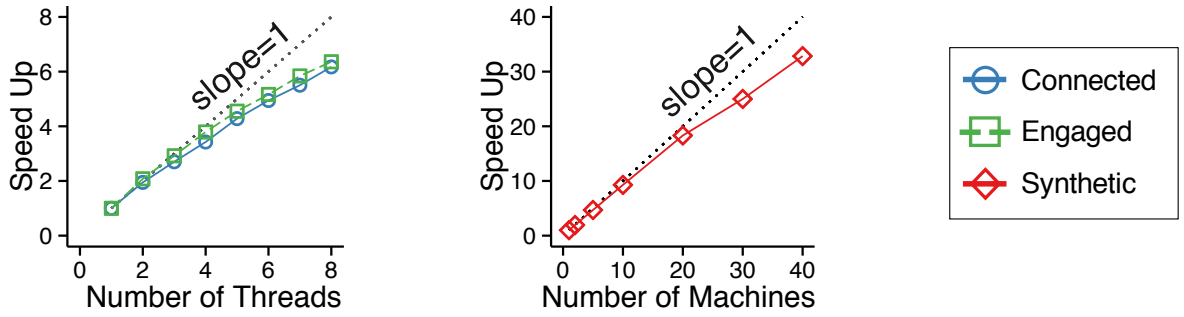
Settings	Prediction Target	Action		Time-gap		Target-stage Reachability	
	Measure	Accuracy	F1	Accuracy	F1	Accuracy	F1
CONNECTED	RANDOM	0.045	0.062	0.142	0.176	0.499	0.506
	FREQUENT	0.273	0.117	0.348	0.180	0.615	0.469
	LR	0.610	0.567	0.551	0.543	0.730	0.710
	SWATT (Proposed)	0.597	0.508	0.577	0.537	N/A	
	K-MEANS + LR	0.610	0.591	0.580	0.543	0.747	0.733
	SWATT + LR (Proposed)	**0.684**	**0.675**	**0.646**	**0.633**	**0.756**	**0.751**
ENGAGED	RANDOM	0.044	0.064	0.144	0.181	0.499	0.504
	FREQUENT	0.273	0.117	0.348	0.180	0.615	0.469
	LR	0.511	0.441	0.554	0.433	0.725	0.718
	SWATT (Proposed)	0.482	0.443	0.564	0.494	N/A	
	K-MEANS + LR	0.511	0.478	0.565	0.489	0.734	0.728
	SWATT + LR (Proposed)	**0.548**	**0.523**	**0.586**	**0.536**	**0.734**	**0.731**

We find the top 3 discriminative sequences of stages for a given target and present the results in Table 15.3. Interestingly, they are very different depending on targets. In the CONNECTED setting, passing through either the *grow* or *poke* stage is important, while progressing step by step is more crucial for the ENGAGED setting. The CONNECTED setting results are intuitive because exploring other content does not necessarily help reaching the target stage without continuous engagement with networking components. On the other hand, the results from the ENGAGED setting imply that exposure to various networking channels is helpful for longer engagement.

15.5.3 Q2. Applicability to Prediction Tasks

To show that SWATT is useful for downstream tasks, we use SWATT to predict (a) the type of each user's next action, (b) the time gap between the current and next actions of each user, and (c) whether each user reaches the target stage within 100 actions. For each task, we compare the following approaches:

- RANDOM: use a randomly chosen label.
- FREQUENT: use the label most frequent in the training set.



(a) Multi-Core Setting (b) Distributed Setting

Figure 15.3: **Near-linear speed-ups of SWATTFIT**. Its speed increased near linearly with the number of threads in the multi-core setting and with the number of machines in the distributed setting.

- LR: use logistic regression [McC84]
- SWATT: use the label most probable (according to SWATT) in the current stage of each user.
- SWATT + LR: divide users depending on their current stages (inferred by SWATT) and use logistic regression separately for each stage based on the actions in the stage.
- K-MEANS + LR: divide users using k-means++ [AV07] and use logistic regression separately for each cluster.

For logistic regression and k-means++, we used $(|\mathcal{A}| + |\mathcal{D}|)$ features corresponding to the frequencies of the action types and time gaps for each user. For action-type and time-gap predictions, we used relative frequencies rather than absolute ones, which led to higher accuracy. We trained our model and logistic regression using randomly chosen half of the users in each dataset and tested using the others. For evaluation, we used the proportion of correct predictions (Accuracy) and a weighted sum [6] of F1 scores (F1).

As shown in Table 15.4, for all the tasks, prediction purely based on our model (SWATT), which is unsupervised, shows similar accuracy to logistic regression (LR), which is supervised. More importantly, their combination (SWATT + LR) was more accurate than combining k-means++ and logistic regression (K-MEANS + LR) as well as the individual methods (SWATT and LR) for all the tasks.

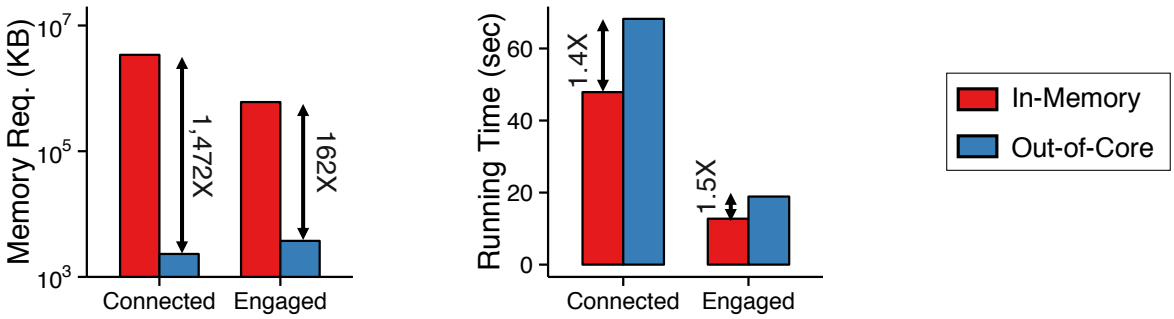
15.5.4 Q3. Scalability

We show the scalability of SWATTFIT, described in Section 15.3. We consider the following implementations:

- IN-MEMORY (MULTI-CORE): in-memory implementation where all data are loaded into memory. It can run in a parallel way in multi-core environments (see Section 15.3.3.2).
- OUT-OF-CORE: out-of-core implementation using disk as the external memory (see Section 15.3.3.1).
- DISTRIBUTED: MAPREDUCE [DG08] implementation on Hadoop 2.6.1 (see Section 15.3.3.3).

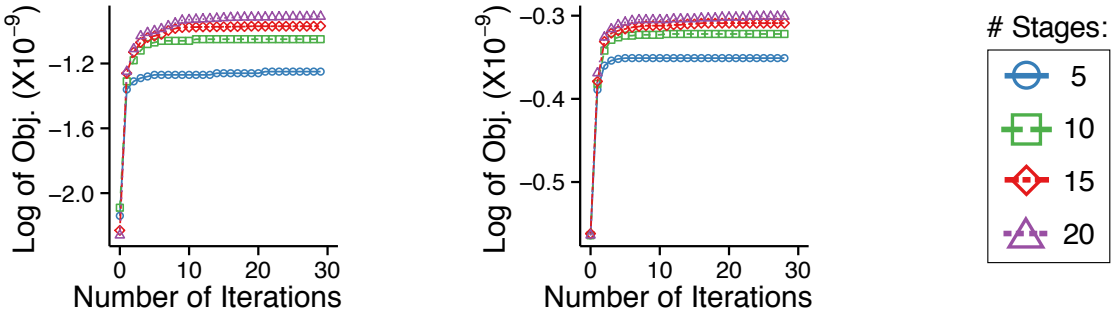
First, we show that all implementations scale linearly with the number of records in the input data. To this end, we used synthetic datasets with different numbers of users while fixing $|\mathcal{A}| = |\mathcal{D}| = k = 10$ and $n_u = 1000$ for every user u. As seen in Figure 15.1 in Section 15.1, the per-iteration running times of all implementations increased linearly with the total number of actions. Especially, 'DISTRIBUTED'

[6] each weight is the proportion to the number of the corresponding label in the test set.



(a) Main memory requirements (b) Running time

Figure 15.4: **Memory efficiency of our out-of-core implementation** of SWATTFIT in the real-world datasets. Out-of-core processing using external memory reduced the amount of required main memory space by up to $1,472\times$ at the expense of a slight decrease in speed.



(a) Convergence (CONNECTED) (b) Convergence (ENGAGED)

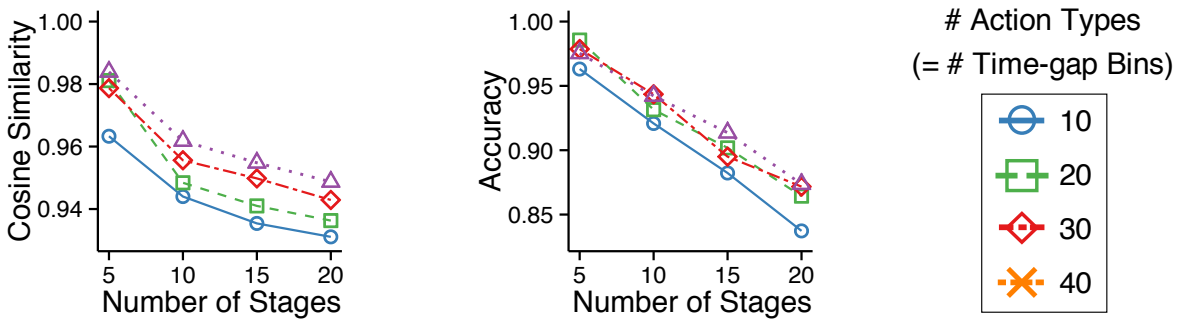
Figure 15.5: **Fast convergence of SWATTFIT.** It converged within 20 iterations in the real-world datasets.

processed a dataset with one trillion actions with per-iteration time less than 2 hours. We obtained similar results when we increased the number of actions per user, while fixing the number of users. These results are consistent with our theoretical analysis in Section 15.4.

Second, we show the near-linear speed-ups of our parallel and distributed implementations. To this end, we measured the speed-up [7] of 'MULTI-CORE' with different numbers of threads and that of 'DISTRIBUTED' with different number of machines. For 'MULTI-CORE', we used the LinkedIn datasets with $k = 10$, and for 'DISTRIBUTED', we used a larger synthetic dataset where $|\mathcal{A}| = |\mathcal{D}| = k = 10$, $|\mathcal{U}| = 10$ millions, and $n_u = 1000$ for every user u. As seen in Figure 15.3, both implementations showed near-linear speed-ups.

Lastly, we show significant reductions in main memory requirements by out-of-core processing using external memory. To this end, we compared the amount of main memory space required by 'IN-MEMORY' and 'OUT-OF-CORE' for processing the LinkedIn datasets with $k = 10$. As seen in Figure 15.4, 'OUT-OF-CORE' required up to $1,472\times$ less main memory space than 'IN-MEMORY'. However, the increase in the running time was at most 50%.

[7] relative speed compared to when a single thread (or a single machine) is used.



(a) Cosine similarity of probability distributions

(b) Accuarcy of stage assignments

Figure 15.6: **Accuracy of SWATTFIT.** The parameters trained by SWATTFIT were reasonably close to ground-truth parameters.

15.5.5 Q4. Convergence

We show that SWATTFIT converges within a small number of iterations. Figure 15.5 shows the value of our objective function (i.e., Eq. (15.2)) in each iteration ofSWATTFIT in the LinkedIn datasets. The number of iterations required for convergence increased with the number of stages (i.e., k). However, even with 20 stages, SWATTFIT converged within 20 iterations.

15.5.6 Q5. Identifiability

We show that SWATTFIT learns parameters reasonably close to ground-truth parameters. We first created synthetic datasets by (a) choosing random probability distributions (i.e., $\{\theta_{s_i}\}_{s_i \in \mathcal{S}_{-k}}$, $\{\phi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}$, $\{\psi_{s_i}\}_{s_i \in \mathcal{S}_{-k}}$) [8] and (b) generating action sequences of $100,000$ users by following the generative process of SWATT (see Section 15.2.2). Then, we compared the probability distributions learned by SWATTFIT with the ground-truth distributions in terms of cosine similarity [Sin01]. We also compared the learned stage assignments with the ground-truth assignments in terms of accuracy (i.e., the proportion of correct assignments). Figure 15.6 shows the results averaged over $1,000$ trials for each setting. As the number of stages (i.e., k) to be learned increased, similarity between trained and ground-truth parameters decreased. However, the increase in the number of action types (i.e., $|\mathcal{A}|$) and time gap bins (i.e., $|\mathcal{D}|$) increased the similarity by making different stages more likely to have distinct probability distributions. In every setting, the learned parameters were reasonably close to the true values. Specifically, the cosine similarity was higher than 0.93 and the accuracy was higher than 0.82 in every setting. These results were not sensitive to the values of the hyperparameters.

15.6 Related Work

Modeling user behavior has been extensively studied to tackle various kinds of tasks. One line of work aims to predict events in the future by learning the hidden patterns from historical behavior. At a microscopic level, for predicting each event, many models and algorithms have been proposed, including

[8] we used $\lambda_\theta = \lambda_\phi = \lambda_\psi = 0.1$. We ignored probability distributions if there exists a short-lived stage (i.e., the probability of self-transition is less than 80%) or an isolated stage (i.e., the probability that a user reaches the stage is less than 10%).

frequency-based pattern mining [LTW08, BFH+12], mechanistic model approaches [BKT16], marked temporal point processes [MA17, DDT+16], generative models using latent cluster variables [MSF+12, FRAF16], and LSTM-based approaches [LJT+17, ZLL+17]. In terms of modeling approaches, our work is closest to the generative latent variable models. However, our model is designed to infer more interpretable, coarse-grained patterns of event sequences. Such macroscopic progressions are not directly extracted through the use of microscopic event models.

On the other hand, another line of literature presents methods for capturing more macroscopic views of user behaviors through clustering event sequences. Those methods can typically visualize the overall view of user behavior from a certain aspect – such as navigation patterns on websites [DFA+15], diagram of user activity transitions [BRCA09], clusters of user types [WZT+16], and topics of event streams [DFA+15, MVGR17]. Our work is related to this line of work in the sense that it provides high-level insights. However, our model presents multi-dimensional insights on each individual user's change, clusters of user actions, and transitions between these clusters.

Some recent work has focused on such multi-dimensional insights by grouping events and representing each entity's progression over coarse event groups. This approach is thus able to account for relationships between groups of events [SYS+13], regime shifts in event streams [MS16], and evolution of users [ML13].

In particular, modeling the progression of users over the latent stages has been proposed to distinguish different patterns of progression including development of various diseases [YML+14]. Our work is closely related to this chapter in the sense that both assume latent stages that given event sequences progress through as well as each observed event in the sequences depends on the current latent stage. However, while the previous work differently models the step-by-step progression through each latent class, our work does not address different classes but model different patterns of progression using transition probabilities between stages. Furthermore, our model defines starting moments and goal stages to clearly illustrate the multiple paths in a particular progression region of interest, whereas the previous work does not contain a component triggering the start or the goal.

15.7 Summary

In this chapter, we propose SWATT, a behavior model where progressions of users' behaviors from a starting state to a goal state are modeled as transitions over latent stages. The latent stages capture the progressions in three aspects: types of actions, frequencies of actions, and directions of changes.

To fit SWATT to web-scale behavior logs, modeled as a tensor, we propose SWATTFIT, a fast and memory-efficient optimization algorithm. We also extend SWATTFIT to multi-core, external-memory, and distributed settings. We theoretically and empirically demonstrate that SWATTFIT scales linearly with the size of the input data. Especially, the distributed version of SWATTFIT, implemented in the MAPREDUCE framework, successfully handles a petabyte-scale dataset with one trillion actions.

We demonstrate the effectiveness of SWATT and SWATTFIT using datasets from LinkedIn. They, however, can be applied to any dataset with a series of actions by different users over time. Using SWATT and SWATTFIT, we discover meaningful stages summarizing the progressions of LinkedIn users towards certain target states. We also show that stage information inferred by SWATT is useful for downstream tasks including prediction of next actions.

Part IV

Conclusions and Future Directions

Chapter 16

Conclusions

This thesis focuses on *mining large dynamic graphs and tensors*, which naturally represent a wealth of information in the real world. Specifically, throughout this thesis, we develop scalable algorithms and tools for three closely related tasks: *structure analysis*, *anomaly detection*, and *behavior modeling*. Our algorithms and tools achieve the highest performance and scalability by (a) employing mathematical techniques, including approximation and sampling, (b) using distributed computing frameworks, and/or (c) exploiting pervasive patterns in real-world data. Thereby, they are successfully applied to a variety of datasets with billions or even trillions of records. Below, we summarize the contributions and impact of our work.

16.1 Contributions

16.1.1 Part I: Structure Analysis

In Part I, we address two tasks related to structure analysis: (a) computing the count of triangles, which many structure-related measures are based on, in large dynamic graphs, and (b) summarizing (i.e., concisely representing) large graphs and tensors.

- **Counting Triangles in Graph Streams:** We develop *four* unbiased, one-pass, sublinear-space algorithms for estimating the count of triangles in large dynamic graphs modeled as graph streams. They provide distinct advantages as described below.
  - **Exploiting Temporal Patterns (Chapter 4):** We discover temporal locality in triangles of real-world graph streams. Then, we develop WRS, which exploits the temporal locality for accurate triangle counting in graph streams. Given the same space budget, WRS is up to $1.9\times$ *more accurate* than its best competitors.
  - **Utilizing Multiple Machines (Chapter 5):** We develop TRI-FLY and COCOS, the *first* distributed algorithms for triangle counting in graph streams. Given the same space budget, COCOS is up to $39\times$ *more accurate* than TRI-FLY, which significantly outperforms the best single-machine algorithms.
  - **Handling Deletions (Chapter 6):** We develop THINKD, an accurate algorithm for triangle counting in *fully-dynamic graph streams*, where edges can be both added and deleted over time. Given the same space budget, it is up to $4.3\times$ *more accurate* than its best competitors.
- **Summarizing Large Graphs and Tensors:** We develop distributed and out-of-core algorithms for summarizing large graphs and tensors, respectively, as described below.

- **Summarizing Large Graphs (Chapter 7):** We develop SWEG, a distributed algorithm for graph summarization. It summarizes a $25\times$ *larger graph* with over *20 billion edges* than its best competitors, without quality loss. By employing SWEG, we losslessly compress a billion-scale web graph with an unprecedented compression rate.
- **Summarizing Large High-order Tensors (Chapter 8):** We develop S-HOT, an external-memory algorithm for high-order Tucker decomposition. It summarizes a tensor with $1000\times$ larger dimensionality without quality loss than its best competitors. Using S-HOT, we analyze a large-scale high-order tensor from Microsoft Academic Graph, which cannot be analyzed by the previously best Tucker-decomposition algorithms.

16.1.2 Part II: Anomaly Detection

In Part II, we develop fast approximate algorithms for detecting unusually dense subgraphs and subtensors, which signal interesting anomalies, in large graphs and tensors.

- **Finding Patterns and Anomalies in Dense Subgraphs (Chapter 9):** We discover *three* empirical patterns in dense subgraphs of real-world graphs. Then, we design *three algorithms* that exploit the patterns for anomaly detection, degeneracy estimation, and influence spreader identification in large graphs. Especially, our anomaly-detection algorithm, namely CORE-A, identifies many interesting anomalies, including a 'follower booster' on Twitter, and 'copy-and-paste' bibliography.
- **Detecting Dense Subtensors in Large Tensors:** We develop *four* algorithms for detecting the densest subtensors, which we show signal many notable anomalies, including 'edit wars' on Wikipedia, spam reviews on App Store, and various types of network attacks. They achieve *an approximation ratio of* $1/n$ for n-order tensors, and they provide distinct advantages, as described below.
  - **In-memory Algorithm (Chapter 11):** We develop M-ZOOM, a near-linear time algorithm for detecting dense subtensors. It is up to $114\times$ *faster* than its best competitors with similar accuracy.
  - **External-memory Algorithm (Chapter 12):** We develop D-CUBE, the first external-memory algorithm for detecting dense subtensors. Our MAPREDUCE implementation of D-CUBE scales to a $1000\times$ *larger* tensor with *100 billion* non-zero entries than in-memory algorithms.
  - **Incremental Algorithms (Chapter 13):** We develop DENSESTREAM and DENSEALERT, the *first* incremental algorithms for detecting dense subtensors. Each update by them is up to $10^6\times$ *faster* than running batch algorithms from scratch.

16.1.3 Part III: Behavior Modeling

In Part III, we focus on developing behavior models of individuals in graph and tensor data.

- **Modeling Purchases in Social Networks (Chapter 14):** We develop SGG and SGG, game-theoretic models for purchases of sharable goods on a social network, modeled as a graph. Then, we develop SGG-NASH, a fast algorithm for finding Nash equilibria of both games. We also suggest a socially optimal range of rental fees for minimizing the social inefficiency of the Nash Equlibria, based on our theoretical analysis and simulation results.
- **Modeling Progression of Users on Social Media (Chapter 15):** We develop SWATT, a comprehensive behavior model for progressions of individuals on social media. Then, we develop SWATTFIT, a linear-time distributed algorithm for fitting SWATT to behavior logs, modeled as a tensor. Our MAPREDUCE implementation of SWATTFIT successfully scales to *one trillion* records. Using SWATT and SWATTFIT, We discover meaningful patterns in the progression of the users in *LinkedIn*.

16.2 Overall Impact

In addition to the contributions above, our work has broad impact on a wide range of domains where large, dynamic, and rich information are modeled as graphs or tensors. Especially, thorough experiments using real-world datasets, we show that our work has direct applications in search engines (Chapters 7 and 9), online social networks (Chapters 9 and 15), computer security (Chapters 11, 12, and 13), e-commerce (Chapter 12), and public policy (Chapter 14). Our work also has been used in the following settings:

Impact in Academia:

- We have *open-sourced* most of the algorithms developed throughout this thesis. They have been *downloaded* over 350 times from 24 countries.
- Our work on patterns and anomalies in dense subgraphs [SERF18] was included in *MIT*'s graduate course on graph analytics (MIT 6.886). It was also featured in an *ICDM 2016* tutorial on core decomposition, and an *ECML/PKDD 2017* tutorial on the same topic.

Impact in Industry:

- Our tools for modeling progression of individuals on social media (SWATT and SWATTFIT [SSK+18]) were used at *LinkedIn Inc.* for user behavior analysis.
- Our graph-summarization algorithm (SWEG [SGKR19]) was filed for a patent by *LinkedIn Inc.* in September 2018.
- Our anomaly-detection algorithm (D-CUBE [SHKF18]) was used in *production* at *NAVER Corp.*, which handled 74.7% of web searches in South Korea in 2017, to identify and filter spam sites.

Awards and Media Coverage:

- Our work on patterns and anomalies in dense subgraphs [SERF16] was selected as one among the *"Bests of ICDM 2016"* and invited to the Knowledge and Information Systems journal [SERF18].
- Our work on modeling purchases in social networks [SLEP17] was featured in *New Scientist* in May 2017 (available at `https://www.newscientist.com/article/2132926`).

Chapter 17

Vision and Future Directions

Throughout this thesis, we develop scalable algorithms and tools for mining large dynamic graphs and tensors, with a focus on structure analysis, anomaly detection, and behavior modeling. Below, we outline several research directions that extend our work towards our ultimate vision, which is *to fully understand and utilize large, dynamic and rich data in the real world*.

Distributed Processing of Graph Streams

While graph-stream algorithms are suitable for real-time processing of dynamic graphs, most of them are assumed to run on a single machine. On the other hand, most distributed graph algorithms are designed for batch processing of static graphs. Our work on triangle counting in Chapter 5 shows that "the best of both worlds" can be achieved by distributed graph-stream algorithms. A natural next step is to extend our idea of distributed graph-stream algorithms to diverse graph mining tasks, such as community detection [GN02], motif analysis [WF94], and graph summarization [NRS08]. Another future step is to develop programming models and systems for efficient distributed graph-stream processing.

Detecting Adversarial Fraudsters

Most graph or tensor based fraud-detection methods, including ours discussed in Chapters 9-13, are based on the behavioral patterns of fraudsters. In practice, however, intelligent fraudsters can adjust their behaviors in response to the advance of fraud-detection algorithms. A first step for deterring such fraudsters is to model their utility functions. Then, natural next steps are (a) to evaluate fraud-detection methods in terms of how costly it is to avoid being detected by the methods, and (b) to develop fraud-detection methods that are costly to avoid.

Modeling Co-evolution of Beliefs, Behaviors, and Social Networks

How do individuals' behaviors (e.g., drinking behaviors) and beliefs (e.g., beliefs on legalizing marijuana) affect the evolution of social networks, and vice versa? To answer this question, we recently developed models for explaining and predicting the co-evolution of beliefs, behaviors, and social networks [NSB+18]. These simple models based on linear regression, however, showed limited ability to explain and predict the co-evolution. A next step is to explore more sophisticated models, such as game-theoretic models. Then, these advanced models can be employed to counteract negative social phenomena such as disagreement and polarization.

Bibliography

[ABK16] Woody Austin, Grey Ballard, and Tamara G Kolda. Parallel tensor compression for large-scale scientific data. In *2016 IEEE International Parallel and Distributed Processing Symposium*, pages 912–922. IEEE, 2016. 1.2.1.2, 8.1, 8.3.2

[AC09] Reid Andersen and Kumar Chellapilla. Finding dense subgraphs with size bounds. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 25–37. Springer, 2009. 10.2

[ACF13] Leman Akoglu, Rishi Chandy, and Christos Faloutsos. Opinion fraud detection in online reviews by network effects. *Seventh International AAAI Conference on Weblogs and Social Media*, 13:2–11, 2013. 10.4, 12.2

[AÇKY05] Evrim Acar, Seyit A Çamtepe, Mukkai S Krishnamoorthy, and Bülent Yener. Modeling and multiway analysis of chatroom tensors. In *International Conference on Intelligence and Security Informatics*, pages 256–268. Springer, 2005. 8.1

[AD09] Alberto Apostolico and Guido Drovandi. Graph compression by bfs. *Algorithms*, 2(3):1031–1044, 2009. 7.1, 7.3.5, (c), 7.6

[ADK+08] Elliot Anshelevich, Anirban Dasgupta, Jon Kleinberg, Eva Tardos, Tom Wexler, and Tim Roughgarden. The price of stability for network design with fair cost allocation. *SIAM Journal on Computing*, 38(4):1602–1623, 2008. 14.6

[ADNK14] Nesreen K Ahmed, Nick Duffield, Jennifer Neville, and Ramana Kompella. Graph sample and hold: A framework for big-graph analytics. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1446–1455. ACM, 2014. 3.1, 3.2, 5.1

[ADWR17] Nesreen K. Ahmed, Nick Duffield, Theodore L. Willke, and Ryan A. Rossi. On sampling from massive graph streams. *Proceedings of the VLDB Endowment*, 10(11):1430–1441, 2017. 3.1, 3.2, 5.1, 6.1

[AH02] Lada A Adamic and Bernardo A Huberman. Zipf's law and the internet. *Glottometrics*, 3(1):143–150, 2002. 8.5.1

[AhBV08] Jose Ignacio Alvarez-hamelin, Alain Barrat, and Alessandro Vespignani. k-core decomposition of internet graphs: hierarchies, self-similarity and measurement biases. In *Networks and Heterogeneous Media*. Citeseer, 2008. 9.1

[AHDBV06] J Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Advances in neural information processing systems*, pages 41–50, 2006. 9.1, 9.6

[AJB99] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Internet: Diameter of the world-wide web. *nature*, 401(6749):130, 1999. 9.2, 9.2.3

[AKM13] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 529–538. ACM, 2013. 3.1, 3.2, 5.1

[All15] Nizar Allouch. On the private provision of public goods on networks. *Journal of Economic Theory*, 157:527–552, 2015. 14.6

[AMF10] Leman Akoglu, Mary McGlohon, and Christos Faloutsos. Oddball: spotting anomalies in weighted graphs. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 410–421. Springer, 2010. 9.6, 10.2

[ARS02] James Abello, Mauricio GC Resende, and Sandra Sudarsky. Massive quasi-clique detection. In *Latin American symposium on theoretical informatics*, pages 598–612. Springer, 2002. 9.6

[ATK15] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, 29(3):626–688, 2015. 9.6

[AV07] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007. 8.5.3, 15.5.1, 15.5.3

[AZBP17] Bijaya Adhikari, Yao Zhang, Aditya Bharadwaj, and B Aditya Prakash. Condensing temporal networks using propagation. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, pages 417–425. SIAM, 2017. 7.6

[Bal06] Narayanaswamy Balakrishnan. *Continuous multivariate distributions*. Wiley Online Library, 2006. 15.2.1

[BAZK18] Maham Anwar Beg, Muhammad Ahmad, Arif Zaman, and Imdadullah Khan. Scalable approximation algorithm for graph summarization. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 502–514. Springer, 2018. (b), 7.6

[BBC+15] Oana Denisa Balalau, Francesco Bonchi, TH Chan, Francesco Gullo, and Mauro Sozio. Finding subgraphs with maximum total density and limited overlap. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, pages 379–388. ACM, 2015. 10.2

[BBCG10] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient algorithms for large-scale local triangle counting. *ACM Transactions on Knowledge Discovery from Data*, 4(3):13, 2010. 1.2.1.1, 3.1, 3.2

[BC08] Gregory Buehrer and Kumar Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, pages 95–106. ACM, 2008. 7.1, 7.3.5, (d), 7.6

[BCAZ06] Coralio Ballester, Antoni Calvó-Armengol, and Yves Zenou. Who's who in networks. wanted: The key player. *Econometrica*, 74(5):1403–1417, 2006. 14.6

[BCFM00] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000. 7.3.2.1

[BCMP13] Simina Brânzei, Ioannis Caragiannis, Jamie Morgenstern, and Ariel D Procaccia. How bad is selfish voting? In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, pages 138–144. AAAI Press, 2013. 14.6

[BE00] Stephen P Borgatti and Martin G Everett. Models of core/periphery structures. *Social networks*, 21(4):375–395, 2000. 9.5.1, 3

[BFH+12] Iyad Batal, Dmitriy Fradkin, James Harrison, Fabian Moerchen, and Milos Hauskrecht. Mining recent temporal patterns for event detection in multivariate time series data. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 280–288. ACM, 2012. 15.6

[BGM14] Bahman Bahmani, Ashish Goel, and Kamesh Munagala. Efficient primal-dual graph algorithms for mapreduce. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 59–78. Springer, 2014. 10.2

[BGMZ97] Andrei Z Broder, Steven C Glassman, Mark S Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8):1157–1166, 1997. 7.3.2.1

[BH03] Gary D Bader and Christopher WV Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC bioinformatics*, 4(1):2, 2003. 9.6

[BH11] Andries E Brouwer and Willem H Haemers. *Spectra of graphs*. Springer Science & Business Media, 2011. 9.4.2

[BH18] Maciej Besta and Torsten Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *arXiv preprint arXiv:1806.01799*, 2018. 7.6

[BHLP11] Jonathan W Berry, Bruce Hendrickson, Randall A LaViolette, and Cynthia A Phillips. Tolerating the community detection resolution limit with edge weighting. *Physical Review E*, 83(5):056119, 2011. 3.1

[BHNT15] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. Space-and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 173–182. ACM, 2015. 10.2

[BK73] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973. 9.6

[BK07a] Brett W Bader and Tamara G Kolda. Efficient matlab computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2007. 8.5.1, 11.5.1, 12.5.1, 13.5.1

[BK07b] Yann Bramoullé and Rachel Kranton. Public goods in networks. *Journal of Economic Theory*, 135(1):478–494, 2007. 14.6

[BKD14] Yann Bramoullé, Rachel Kranton, and Martin D'amours. Strategic interaction and networks. *American Economic Review*, 104(3):898–930, 2014. 14.6

[BKT16] Austin R Benson, Ravi Kumar, and Andrew Tomkins. Modeling user consumption sequences. In *Proceedings of the 25th International Conference on World Wide Web*, pages 519–529. International World Wide Web Conferences Steering Committee, 2016. 15.6

[BKV12] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. Densest subgraph in streaming and mapreduce. *Proceedings of the VLDB Endowment*, 5(5):454–465, 2012. 10.2

[BL+07] James Bennett, Stan Lanning, et al. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York, NY, USA, 2007. 10.4, 11.2, 12.2

[Blu93] Lawrence E Blume. The statistical mechanics of strategic interaction. *Games and economic behavior*, 5(3):387–424, 1993. 1.2.3.1, 14.1

[BRCA09] Fabrício Benevenuto, Tiago Rodrigues, Meeyoung Cha, and Virgílio Almeida. Characterizing user behavior in online social networks. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, pages 49–62. ACM, 2009. 15.6

[Bun94] Wray L Buntine. Operations for learning with graphical models. *Journal of Artificial Intelligence Research*, pages 159–225, 1994. 15.2

[BV04] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602. ACM, 2004. 7.1, 7.3.5, 7.2, (a), (b), 7.6

[BXG+13] 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. ACM, 2013. 1.2.2.2, 9.6, 10.1, 10.2

[BYKS02] Ziv Bar-Yossef, Ravi Kumar, and D Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632. Society for Industrial and Applied Mathematics, 2002. 3.1

[BZ03] Vladimir Batagelj and Matjaz Zaversnik. An o (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003. 9.1, 9.2.1, 9.2.2, 9.3.3.1, 9.6

[BZ07] Vladimir Batagelj and Matjaž Zaveršnik. Short cycle connectivity. *Discrete Mathematics*, 307(3-5):310–318, 2007. 1.2.1.1, 3.1

[CBRB08] Loïc Cerf, Jérémy Besson, Céline Robardet, and Jean-François Boulicaut. Data-peeler: Constraint-based closed pattern mining in n-ary relations. In *proceedings of the 2008 SIAM International conference on Data Mining*, pages 37–48. SIAM, 2008. 10.2

[Cha00] Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 84–95. Springer, 2000. 9.6, 10.2, 10.3.2, 10.1, 10.2

[CKCÖ11] James Cheng, Yiping Ke, Shumo Chu, and M Tamer Özsu. Efficient core decomposition in massive networks. In *IEEE 27th International Conference on Data Engineering*, pages 51–62. IEEE, 2011. 9.6

[CKK06] Ioannis Caragiannis, Christos Kaklamanis, and Panagiotis Kanellopoulos. Taxes for linear atomic congestion games. In *European Symposium on Algorithms*, pages 184–195. Springer, 2006. 14.6

[CKL+09] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228. ACM, 2009. 7.1, 7.3.5, (b), 7.6

[CLDG03] Fan Chung, Linyuan Lu, T Gregory Dewey, and David J Galas. Duplication models for biological networks. *Journal of computational biology*, 10(5):677–687, 2003. 7.5.5

[CLF+09] Chen Chen, Cindy X Lin, Matt Fredrikson, Mihai Christodorescu, Xifeng Yan, and Jiawei Han. Mining graph patterns efficiently via randomized summaries. *Proceedings of the VLDB Endowment*, 2(1):742–753, 2009. 7.6

[Coh08] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, 16, 2008. 3.1, 9.6

[Coh09] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009. 3.1, 3.2, 5.1

[CSN09] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009. 8.4.5

[CTT06] Yun Chi, Belle L Tseng, and Junichi Tatemura. Eigen-trend: trend analysis in the blogosphere based on singular value decompositions. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 68–77. ACM, 2006. 1.2.1.2, 8.1

[CZL+11] Yuanzhe Cai, Miao Zhang, Dijun Luo, Chris Ding, and Sharma Chakravarthy. Low-order tensor decompositions for social tagging recommendation. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 695–704. ACM, 2011. 8.1

[DDT+16] Nan Du, Hanjun Dai, Rakshit Trivedi, Utkarsh Upadhyay, Manuel Gomez-Rodriguez, and Le Song. Recurrent marked temporal point processes: Embedding event history to vector. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1555–1564. ACM, 2016. 15.6

[DFA+15] Nan Du, Mehrdad Farajtabar, Amr Ahmed, Alexander J Smola, and Le Song. Dirichlet-hawkes processes with applications to clustering continuous-time document streams. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 219–228. ACM, 2015. 15.6

[DG06] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006. 9.3.3.2

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 12.1, 15.1, 15.3.3.3, 15.5.4

[DKK+16] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1535–1544. ACM, 2016. 7.1, 7.3.5, (a), 7.6

[DKKW12] Gideon Dror, Noam Koenigstein, Yehuda Koren, and Markus Weimer. The yahoo! music dataset and kdd-cup'11. In *KDD Cup*, 2012. 10.4, 11.2, 12.2, 13.2

[DLR77] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (methodological)*, pages 1–38, 1977. 15.3.1.2

[DS13] Cody Dunne and Ben Shneiderman. Motif simplification: improving network visualization readability with fan, connector, and clique glyphs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3247–3256. ACM, 2013. 7.6

[EG13] Matthew Elliott and Benjamin Golub. A network approach to public goods. In *Proceedings of the fourteenth ACM conference on Electronic commerce*, pages 377–378. ACM, 2013. 14.6

[Ell93] Glenn Ellison. Learning, local interaction, and coordination. *Econometrica: Journal of the Econometric Society*, pages 1047–1071, 1993. 1.2.3.1, 14.1

[ELM+15] Alessandro Epasto, Silvio Lattanzi, Vahab Mirrokni, Ismail Oner Sebe, Ahmed Taei, and Sunita Verma. Ego-net community mining applied to friend suggestion. *Proceedings of the VLDB Endowment*, 9(4):324–335, 2015. 3.1

[ELS15] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. Efficient densest subgraph computation in evolving graphs. In *Proceedings of the 24th International Conference on World Wide Web*, pages 300–310. International World Wide Web Conferences Steering Committee, 2015. 10.2

[EM02] Jean-Pierre Eckmann and Elisha Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings of the national academy of sciences*, 99(9):5825–5829, 2002. 3.1

[Erd63] P Erdös. On the structure of linear graphs. *Israel Journal of Mathematics*, 1(3):156–160, 1963. 9.1

[FCT14] Martin Farach-Colton and Meng-Tsung Tsai. Computing the degeneracy of large graphs. In *Latin American Symposium on Theoretical Informatics*, volume 8392, pages 250–260, 2014. 9.1, 9.2.2, 9.6

[FRAF16] Flavio Figueiredo, Bruno Ribeiro, Jussara M Almeida, and Christos Faloutsos. Tribeflow: Mining & predicting user trajectories. In *Proceedings of the 25th international conference on world wide web*, pages 695–706. International World Wide Web Conferences Steering Committee, 2016. 15.6

[Fre82] Eugene C Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982. 9.6

[FSSS09] Thomas Franz, Antje Schultz, Sergej Sizov, and Steffen Staab. Triplerank: Ranking semantic web data by tensor decomposition. In *International semantic web conference*, pages 213–228. Springer, 2009. 8.1

[FWW13] Wenfei Fan, Xin Wang, and Yinghui Wu. Diversified top-k graph pattern matching. *Proceedings of the VLDB Endowment*, 6(13):1510–1521, 2013. 7.6

[Gab83] Harold N Gabow. An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 448–456. ACM, 1983. 7.6

[GGJ+10] Andrea Galeotti, Sanjeev Goyal, Matthew O Jackson, Fernando Vega-Redondo, and Leeat Yariv. Network games. *The review of economic studies*, 77(1):218–244, 2010. 14.6

[GGK03] Johannes Gehrke, Paul Ginsparg, and Jon Kleinberg. Overview of the 2003 kdd cup. *ACM SIGKDD Explorations Newsletter*, 5(2):149–151, 2003. 4.6.1, 4.2, 5.4, 9.2, 9.2.3

[GGT16] Esther Galbrun, Aristides Gionis, and Nikolaj Tatti. Top-k overlapping densest subgraphs. *Data Mining and Knowledge Discovery*, 30(5):1134–1165, 2016. 10.2

[GKT05] David Gibson, Ravi Kumar, and Andrew Tomkins. Discovering large dense subgraphs in massive graphs. In *Proceedings of the 31st international conference on Very large data bases*, pages 721–732. VLDB Endowment, 2005. 1.2.2.2, 10.1

[GLH08] Rainer Gemulla, Wolfgang Lehner, and Peter J Haas. Maintaining bounded-size sample synopses of evolving datasets. *The VLDB Journal*, 17(2):173–201, 2008. 6.3.3, 6.5

[GMTV14] Christos Giatsidis, Fragkiskos D Malliaros, Dimitrios M Thilikos, and Michalis Vazirgiannis. Corecluster: A degeneracy based graph clustering framework. In *AAAI*, volume 14, pages 44–50, 2014. 9.1, 9.6

[GN02] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002. 17

[Gol84] Andrew V Goldberg. *Finding a maximum density subgraph*. Technical Report, 1984. 10.2, 10.3.2

[GPP18] Ekta Gujral, Ravdeep Pasricha, and Evangelos E Papalexakis. Sambaten: Sampling-based batch incremental tensor decomposition. In *Proceedings of the 2018 SIAM International Conference on Data Mining*, pages 387–395. SIAM, 2018. 10.2

[Gra10] Lars Grasedyck. Hierarchical singular value decomposition of tensors. *SIAM Journal on Matrix Analysis and Applications*, 31(4):2029–2054, 2010. 8.3.2

[Hir83] Jack Hirshleifer. From weakest-link to best-shot: The voluntary provision of public goods. *Public choice*, 41(3):371–386, 1983. 14.2.1.1, 14.6

[HJT01] Bronwyn H Hall, Adam B Jaffe, and Manuel Trajtenberg. The nber patent citation data file: Lessons, insights and methodological tools. Technical report, National Bureau of Economic Research, 2001. 4.6.1, 4.2, 6.2, 7.2, 9.2, 9.2.3

[HL91] Stephen T. Hedetniemi and Renu C. Laskar. *Topics on Domination*. Elsevier, 1991. 2

[HM16] Ruining He and Julian McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *Proceedings of the 25th international conference on World Wide Web*, pages 507–517. ACM, 2016. 1.2.3.2, 15.1

[HS17] Guyue Han and Harish Sethu. Edge sample and discard: A new algorithm for counting triangles in large dynamic graphs. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 44–49. ACM, 2017. 3.1, 3.2, 6.1, 6.5.1

[HSP+16] Bryan Hooi, Hyun Ah Song, Evangelos Papalexakis, Rakesh Agrawal, and Christos Faloutsos. Matrices, compression, learning curves: formulation, and the groupnteach algorithms. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 376–387. Springer, 2016. 9.5.1

[HSS+17] Bryan Hooi, Kijung Shin, Hyun Ah Song, Alex Beutel, Neil Shah, and Christos Faloutsos. Graph-based fraud detection in the face of camouflage. *ACM Transactions on Knowledge Discovery from Data*, 11(4):44, 2017. 9.1, 9.3.3.2, 9.3.3.3, 9.6, 10.1, 10.1, 10.2, 11.5.1, 13.6(b), 13.5.5.1

[IKPZ13] Dmitry I Ignatov, Sergei O Kuznetsov, Jonas Poelmans, and Leonid E Zhukov. Can triconcepts become triclusters? *International Journal of General Systems*, 42(6):572–593, 2013. 10.2

[JBC+16] Meng Jiang, Alex Beutel, Peng Cui, Bryan Hooi, Shiqiang Yang, and Christos Faloutsos. Spotting suspicious behaviors in multimodal data: A general metric and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 28(8):2187–2200, 2016. 1.2.2.2, 9.6, 10.1, 10.2, 10.4, 11.5.1, 11.6, 12.1, 12.5.1, 12.3, 12.4, 13.1, 13.5.1, 13.6(b), 13.5.5.1

[JCB+14a] Meng Jiang, Peng Cui, Alex Beutel, Christos Faloutsos, and Shiqiang Yang. Catchsync: catching synchronized behavior in large directed graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 941–950. ACM, 2014. 10.1

[JCB+14b] Meng Jiang, Peng Cui, Alex Beutel, Christos Faloutsos, and Shiqiang Yang. Inferring strange behavior from connectivity pattern in social networks. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 126–138. Springer, 2014. 10.1, 10.2

[JK02] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems*, 20(4):422–446, 2002. 7.5.3

[JPF+16] Inah Jeon, Evangelos E Papalexakis, Christos Faloutsos, Lee Sael, and U Kang. Mining billion-scale tensors: algorithms and discoveries. *The VLDB Journal*, 25(4):519–544, 2016. 8.1, 8.3.2, 8.2, 10.2

[JSP13] 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. ACM, 2013. 3.1, 3.2, 6.1

[JTGV+05] G Joshi-Tope, Marc Gillespie, Imre Vastrik, Peter D'Eustachio, Esther Schmidt, Bernard de Bono, Bijay Jassal, GR Gopinath, GR Wu, Lisa Matthews, et al. Reactome: a knowledgebase of biological pathways. *Nucleic acids research*, 33(suppl_1):D428–D432, 2005. 7.2

[JZ15] Matthew O Jackson and Yves Zenou. Games on networks. In *Handbook of game theory with economic applications*, volume 4, pages 95–163. Elsevier, 2015. 14.6

[KB09] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009. 8.1, 10.1, 10.2, 11.5.1, 1, 11.6, 12.5.1, 12.3, 12.4

[KBK05] Tamara G Kolda, Brett W Bader, and Joseph P Kenny. Higher-order web link analysis using multilinear algebra. In *Data Mining, Fifth IEEE International Conference on*, pages 8–pp. IEEE, 2005. 8.1

[KGH+10] Maksim Kitsak, Lazaros K Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H Eugene Stanley, and Hernán A Makse. Identification of influential spreaders in complex networks. *Nature Physics*, 6(11):888–893, 2010. 9.1, 9.5.2, 9.5.3.2, 9.6, 9.8

[KH18] Seongyun Ko and Wook-Shin Han. Turbograph++: A scalable and fast graph analytics system. In *Proceedings of the 2018 International Conference on Management of Data*, pages 395–410. ACM, 2018. 3.2

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998. 7.3.2.1

[KKT03] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the ninth ACM SIGKDD international con-*

ference on Knowledge discovery and data mining, pages 137–146. ACM, 2003. 9.5.2, 9.6

[KKVF14] Danai Koutra, U Kang, Jilles Vreeken, and Christos Faloutsos. Vog: Summarizing and understanding large graphs. In *Proceedings of the 2014 SIAM International Conference on Data Mining*, pages 91–99. SIAM, 2014. 7.1, 7.6

[KLPM10] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. AcM, 2010. 9.2, 9.2.3

[KN97] Bernd Kreuter and Till Nierhoff. Greedily approximating the r-independent set and k-center problems on random instances. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 43–53. Springer, 1997. 14.2.2

[KNL14] Kifayat Ullah Khan, Waqas Nawaz, and Young-Koo Lee. Set-based unified approach for attributed graph summarization. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*, pages 378–385. IEEE, 2014. 7.6

[KNL15] Kifayat Ullah Khan, Waqas Nawaz, and Young-Koo Lee. Set-based approximate approach for lossless graph summarization. *Computing*, 97(12):1185–1207, 2015. 7.1, (b), 7.6

[KP99] Elias Koutsoupias and Christos Papadimitriou. Worst-case equilibria. In *Proceedings of the 16th annual conference on Theoretical aspects of computer science*, pages 404–413. Springer, 1999. 14.6

[KP13] Konstantin Kutzkov and Rasmus Pagh. On the streaming complexity of computing local clustering coefficients. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 677–686. ACM, 2013. 3.2, 5.1

[KP14] Konstantin Kutzkov and Rasmus Pagh. Triangle counting in dynamic graph streams. In *Scandinavian Workshop on Algorithm Theory*, pages 306–318. Springer, 2014. 3.2, 6.1

[KP17] John Kallaugher and Eric Price. A hybrid sampling scheme for triangle counting. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1778–1797. Society for Industrial and Applied Mathematics, 2017. 3.2, 3.1, 5.1

[KPHF12] U Kang, Evangelos Papalexakis, Abhay Harpale, and Christos Faloutsos. Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 316–324. ACM, 2012. 10.2

[KPR13] Jeremy Kun, Brian Powers, and Lev Reyzin. Anti-coordination games and stable graph colorings. In *International Symposium on Algorithmic Game Theory*, pages 122–133. Springer, 2013. 14.6

[KRR+00] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D Sivakumar, Andrew Tomkins, and Eli Upfal. Stochastic models for the web graph. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 57–65. IEEE, 2000. 7.5.5

[KS08] Tamara G Kolda and Jimeng Sun. Scalable tensor decompositions for multi-aspect data mining. In *IEEE Eighth International Conference on Data Mining*, pages 363–372. IEEE, 2008. 8.1, 8.2.1.2, 8.2.1.3, 8.3.1, 8.7, 8.3.2, 8.2, 8.5.1, 8.5.3

[KS09] Samir Khuller and Barna Saha. On finding dense subgraphs. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming*, pages 597–608. Springer-Verlag, 2009. 10.1, 10.2

[Kun13] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013. 9.2, 9.2.3, 11.2

[KV99] Ravi Kannan and V Vinay. *Analyzing the structure of large graphs*. Rheinische Friedrich-Wilhelms-Universität Bonn Bonn, 1999. 10.3.2

[KY04] Bryan Klimt and Yiming Yang. Introducing the enron corpus. In *Proceedings of the conference on Collaboration, Electronic messaging, Anti-Abuse and Spam*, 2004. 4.6.1, 4.2, 7.2, 9.2, 9.2.3

[LAH07] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. The dynamics of viral marketing. *ACM Transactions on the Web*, 1(1):5, 2007. 7.2

[LCKF05] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 133–145. Springer, 2005. 9.1, 9.4.2, 9.2, 9.4.2

[Lea00] Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000. 7.3.3

[LFG+00] Richard P Lippmann, David J Fried, Isaac Graf, Joshua W Haines, Kristopher R Kendall, David McClung, Dan Weber, Seth E Webster, Dan Wyschogrod, Robert K Cunningham, et al. Evaluating intrusion detection systems: The 1998 darpa off-line intrusion detection evaluation. In *DARPA Information Survivability Conference and Exposition*, volume 2, pages 12–26. IEEE, 2000. 10.4, 12.2, 13.1, 13.2

[LGF09] Chao Liu, Fan Guo, and Christos Faloutsos. Bbm: bayesian browsing model from petabyte-scale data. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 537–546. ACM, 2009. 7.1

[LHS+17] Hemank Lamba, Bryan Hooi, Kijung Shin, Christos Faloutsos, and Jürgen Pfeffer. zoorank: Ranking suspicious entities in time-evolving tensors. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 68–84. Springer, 2017. 10.2

[Liu15] Xin Liu. Modeling users' dynamic preference for personalized recommendation. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, pages 1785–1791, 2015. 1.2.3.2, 15.1

[LJK18] Yongsub Lim, Minsoo Jung, and U Kang. Memory-efficient and accurate sampling for counting local triangles in graph streams: from simple to multigraphs. *ACM Transactions on Knowledge Discovery from Data*, 12(1):4, 2018. 1.2.1.1, 3.1, 3.1, 3.2, 3.3.2, 4.5.2, 4.6.1, 5.1, 5.5.1, 6.1, 6.5.1

[LJT+17] Liangyue Li, How Jing, Hanghang Tong, Jaewon Yang, Qi He, and Bee-Chung Chen. Nemo: Next career move prediction with contextual embedding. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 505–513. International World Wide Web Conferences Steering Committee, 2017. 15.6

[LKF07] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data*, 1(1):2, 2007. 7.2, 9.2, 9.2.3

[LLDM09] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009. 5.4, 6.2, 9.2, 9.2.3

[LNSS16] Hemank Lamba, Vaishnavh Nagarajan, Kijung Shin, and Naji Shajarisales. Incorporating side information in tensor completion. In *Proceedings of the 25th International Conference Companion on World Wide Web*, pages 65–66. International World Wide Web Conferences Steering Committee, 2016. 8.1

[LRJA10] Victor E Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*, pages 303–336. Springer, 2010. 10.2

[LSDK18] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. Graph summarization methods and applications: A survey. *ACM Computing Surveys*, 51(3):62, 2018. 7.6

[LSK08] Yu-Ru Lin, Hari Sundaram, and Aisling Kelliher. Summarization of social activity over time: people, actions and concepts in dynamic networks. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 1379–1380. ACM, 2008. 7.6

[LSY98] Richard B Lehoucq, Danny C Sorensen, and Chao Yang. *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*. SIAM, 1998. 8.1, 8.2.1.4, 8.5.1

[LT10] Kristen LeFevre and Evimaria Terzi. Grass: Graph structure summarization. In *Proceedings of the 2010 SIAM International Conference on Data Mining*, pages 454–465. SIAM, 2010. 7.6

[LTH+14] Xingjie Liu, Yuanyuan Tian, Qi He, Wang-Chien Lee, and John McPherson. Distributed graph summarization. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 799–808. ACM, 2014. 7.6

[LTW08] Srivatsan Laxman, Vikram Tankasali, and Ryen W White. Stream prediction using a generative model based on frequent episodes in event sequences. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 453–461. ACM, 2008. 15.6

[Luc50] R Duncan Luce. Connectivity and generalized cliques in sociometric group structure. *Psychometrika*, 15(2):169–190, 1950. 9.6

[LV15] Jian Lou and Yevgeniy Vorobeychik. Equilibrium analysis of multi-defender security games. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, pages 596–602, 2015. 14.6

[MA05] P Massa and P Avesani. Controversial users demand local trust metrics: an experimental study on epinions. com community. In *20th Conference of American Association for Artificial Intelligence*. ACM, 2005. 6.2

[MA17] Emaad Manzoor and Leman Akoglu. Rush!: Targeted time-limited coupons via purchase forecasts. In *Proceedings of the 23rd ACM SIGKDD International Conference on*

Knowledge Discovery and Data Mining, pages 1923–1931. ACM, 2017. 15.6

[Mar87] M Lynne Markus. Toward a "critical mass" theory of interactive media: Universal access, interdependence and diffusion. *Communication research*, 14(5):491–511, 1987. 1.2.3.1, 14.1

[Mat92] Akihiko Matsui. Best response dynamics and socially stable strategies. *Journal of Economic Theory*, 57(2):343–362, 1992. 14.3

[McC84] Peter McCullagh. Generalized linear models. *European Journal of Operational Research*, 16(3):285–292, 1984. 15.5.1, 15.5.3

[MGF11] Koji Maruhashi, Fan Guo, and Christos Faloutsos. Multiaspectforensics: Pattern mining on large-scale heterogeneous networks with tensor analysis. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 203–210. IEEE, 2011. 1.2.2.2, 8.1, 10.1, 10.2, 11.5.1, 11.6, 12.1, 12.5.1, 12.3, 12.4, 13.1, 13.5.1, 13.5.5.1

[Mis09] Alan Mislove. *Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems*. PhD thesis, Rice University, 2009. 4.6.1, 4.2

[MJE12] Samaneh Moghaddam, Mohsen Jamali, and Martin Ester. Etf: extended tensor factorization model for personalizing prediction of review helpfulness. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pages 163–172. ACM, 2012. 8.1

[ML13] Julian John McAuley and Jure Leskovec. From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews. In *Proceedings of the 22nd international conference on World Wide Web*, pages 897–908. ACM, 2013. 1.2.3.2, 15.1, 15.1, 15.6

[MMG+07] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 29–42. ACM, 2007. 5.4, 6.2, 7.2, 9.2, 9.2.3, 10.4, 11.2, 12.2, 13.2, 14.5

[Mok79] Robert J Mokken. Cliques, clubs and clans. *Quality and quantity*, 13(2):161–173, 1979. 9.6

[MPL15] Julian McAuley, Rahul Pandey, and Jure Leskovec. Inferring networks of substitutable and complementary products. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM, 2015. 10.4, 11.2, 12.2, 13.2

[MS16] Yasuko Matsubara and Yasushi Sakurai. Regime shifts in streams: Real-time forecasting of co-evolving time sequences. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1045–1054. ACM, 2016. 15.6

[MSF+12] Yasuko Matsubara, Yasushi Sakurai, Christos Faloutsos, Tomoharu Iwata, and Masatoshi Yoshikawa. Fast mining and forecasting of complex time-stamped events. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 271–279. ACM, 2012. 15.6

[MSHM12] Brian Macdonald, Paulo Shakarian, Nicholas Howard, and Geoffrey Moores. Spreaders in the network sir model: An empirical study. *arXiv preprint arXiv:1208.4269*, 2012. 9.1, 9.5.2, 9.5.3.2, 9.6

[MSLC01] Miller McPherson, Lynn Smith-Lovin, and James M Cook. Birds of a feather: Homophily in social networks. *Annual review of sociology*, 27(1):415–444, 2001. 3.1

[MST09] Paolo Massa, Martino Salvetti, and Danilo Tomasoni. Bowling alone and trust decline in social network sites. In *Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing*, pages 658–663. IEEE, 2009. 14.5

[MTVV15] Andrew McGregor, David Tench, Sofya Vorotnikova, and Hoa T Vu. Densest subgraph in dynamic graph streams. In *International Symposium on Mathematical Foundations of Computer Science*, pages 472–482. Springer, 2015. 10.2

[MVGR17] Charalampos Mavroforakis, Isabel Valera, and Manuel Gomez-Rodriguez. Modeling the dynamics of learning activity on the web. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1421–1430. International World Wide Web Conferences Steering Committee, 2017. 15.6

[MVLB14] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. Graph structure in the web—revisited: a trick of the heavy tail. In *Proceedings of the 23rd international conference on World Wide Web*, pages 427–432. ACM, 2014. 7.1

[New03] Mark EJ Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003. 1.2.1.1, 3.1

[New06] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006. 9.5.1, 4

[NRS08] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 419–432. ACM, 2008. 7.1, 7.2.2, (c), (d), 7.6, 17

[NSB+18] Aastha Nigam, Kijung Shin, Ashwin Bahulkar, Bryan Hooi, David Hachen, Boleslaw Szymanski, Christos Faloutsos, and Nitesh Chawla. One-m: Modeling the co-evolution of opinions and network connections. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 122–140. Springer, 2018. 17

[OSP+17] Jinoh Oh, Kijung Shin, Evangelos E Papalexakis, Christos Faloutsos, and Hwanjo Yu. S-hot: Scalable high-order tucker decomposition. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 761–770. ACM, 2017. 1.2.1.2, 8, 10.2

[PC13] Ha-Myung Park and Chin-Wan Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 539–548. ACM, 2013. 3.1, 3.2, 5.1

[PCVS15] Ioakeim Perros, Robert Chen, Richard Vuduc, and Jimeng Sun. Sparse hierarchical tucker factorization and its application to healthcare. In *2015 IEEE International Conference on Data Mining*, pages 943–948. IEEE, 2015. 8.3.2

[PCWF07] Shashank Pandit, Duen Horng Chau, Samuel Wang, and Christos Faloutsos. Netprobe: a fast and scalable system for fraud detection in online auction networks. In *Proceedings of the 16th international conference on World Wide Web*, pages 201–210. ACM, 2007. 9.6, 10.1, 10.2

[PFS12] Evangelos E Papalexakis, Christos Faloutsos, and Nicholas D Sidiropoulos. Parcube: Sparse parallelizable tensor decompositions. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 521–536. Springer, 2012. 10.2

[PMK16] Ha-Myung Park, Sung-Hyon Myaeng, and U Kang. Pte: enumerating trillion triangles on distributed systems. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1115–1124. ACM, 2016. 3.1, 3.2, 5.1

[Pow98] David MW Powers. Applications and explanations of zipf's law. In *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*, pages 151–160. Association for Computational Linguistics, 1998. 8.5.1

[PSKP14] Ha-Myung Park, Francesco Silvestri, U Kang, and Rasmus Pagh. Mapreduce triangle enumeration with guarantees. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 1739–1748. ACM, 2014. 3.1, 3.2, 5.1

[PSP+18] Ha-Myung Park, Francesco Silvestri, Rasmus Pagh, Chin-Wan Chung, Sung-Hyon Myaeng, and U Kang. Enumerating trillion subgraphs on distributed systems. *ACM Transactions on Knowledge Discovery from Data*, 12(6):71, 2018. 3.1, 3.2, 5.1

[PSS+10] B Aditya Prakash, Ashwin Sridharan, Mukund Seshadri, Sridhar Machiraju, and Christos Faloutsos. Eigenspokes: Surprising patterns and scalable community chipping in large graphs. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 435–448. Springer, 2010. 7.3.2.1, 9.6, 10.2, 11.5.1

[PT12] Rasmus Pagh and Charalampos E Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters*, 112(7):277–281, 2012. 3.2, 5.1

[PTT13] Aduri Pavan, Kanat Tangwongan, and Srikanta Tirthapura. Parallel and distributed triangle counting on graph streams. *Technical report, IBM*, 2013. 3.2, 5.1

[PTTW13] Aduri Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Counting and sampling triangles from a graph stream. *Proceedings of the VLDB Endowment*, 6(14):1870–1881, 2013. 3.1, 3.2, 5.1, 6.1

[QLJ+14] Qiang Qu, Siyuan Liu, Christian S Jensen, Feida Zhu, and Christos Faloutsos. Interestingness-driven diffusion process summarization in dynamic networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 597–613. Springer, 2014. 7.6

[RGSB17] Matteo Riondato, David García-Soriano, and Francesco Bonchi. Graph summarization with quality guarantees. *Data mining and knowledge discovery*, 31(2):314–349, 2017. 7.6

[RMV15] Maria-Evgenia G Rossi, Fragkiskos D Malliaros, and Michalis Vazirgiannis. Spread it good, spread it fast: Identification of influential nodes in social networks. In *Proceedings of the 24th International Conference on World Wide Web*, pages 101–102. ACM, 2015. 9.1, 9.5.3.2, 9.6, 9.8

[Rog10] Everett M Rogers. *Diffusion of innovations*. Simon and Schuster, 2010. 1.2.3.1, 14.1

[RST10] Steffen Rendle and Lars Schmidt-Thieme. Pairwise interaction tensor factorization for personalized tag recommendation. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 81–90. ACM, 2010. 8.1

[Ruh03] Jan Matthias Ruhl. *Efficient algorithms for new computational models*. PhD thesis, Massachusetts Institute of Technology, 2003. 12.1

[RZ18] Ryan A Rossi and Rong Zhou. Graphzip: a clique-based sparse graph compression method. *Journal of Big Data*, 5(1):10, 2018. 7.1, 7.3.5, 7.6

[Saa11] Yousef Saad. *Numerical methods for large eigenvalue problems: revised edition*, volume 66. Siam, 2011. 8.2.1.4

[SBGF14] Neil Shah, Alex Beutel, Brian Gallagher, and Christos Faloutsos. Spotting suspicious link behavior with fbox: An adversarial perspective. In *IEEE 14th International Conference on Data Mining*, pages 959–964. IEEE, 2014. 9.6, 10.2

[Sch07] Thomas Schank. *Algorithmic aspects of triangle-based network analysis*. PhD thesis, PhD Thesis, Universität Karlsruhe (TH), 2007. 4

[SDB15] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *Data Compression Conference (DCC), 2015*, pages 403–412. IEEE, 2015. 7.1

[Sei83] Stephen B Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983. 1, 9.6

[SERF16] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. Corescope: Graph mining using k-core analysis—patterns, anomalies and algorithms. In *IEEE 16th International Conference on Data Mining*, pages 469–478. IEEE, 2016. 1.1, 1.2.2.1, 9, 16.2

[SERF18] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. Patterns and anomalies in k-cores of real-world graphs with applications. *Knowledge and Information Systems*, 54(3):677–710, 2018. 1.1, 3.1, 9, 10.1, 10.2, 11.5.1, 16.2

[SERU17] Lorenzo De Stefani, Alessandro Epasto, Matteo Riondato, and Eli Upfal. Trièst: counting local and global triangles in fully dynamic streams with fixed memory size. *ACM Transactions on Knowledge Discovery from Data*, 11(4):43, 2017. 3.1, 3.2, 3.3.1, 4.5.1, 4.2, 4.5.2, 4.6.1, 5.1, 5.3.2, 5.4.1.1, 5.2, 5.4.1.2, 5.4.1.2, 5.5.1, 6.1, 6.5.1

[SF78] Stephen B Seidman and Brian L Foster. A graph-theoretic generalization of the clique concept. *Journal of Mathematical sociology*, 6(1):139–154, 1978. 9.6

[SGJS+13] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment*, 6(6):433–444, 2013. 9.6

[SGKR19] Kijung Shin, Amol Ghoting, Myunghwan Kim, and Hema Raghavan. Sweg: Lossless and lossy summarization of web-scale graphs. In *Proceedings of the 28th International Conference on World Wide Web*. ACM, 2019. 1.1, 1.2.1.2, 7, 16.2

[SHF16] Kijung Shin, Bryan Hooi, and Christos Faloutsos. M-zoom: Fast dense-block detection in tensors with quality guarantees. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 264–280. Springer, 2016. 11

[SHF18] Kijung Shin, Bryan Hooi, and Christos Faloutsos. Fast, accurate and flexible algorithms for dense subtensor mining. *ACM Transactions on Knowledge Discovery from Data*, 12(3):28:1–28:30, 2018. 1.2.2.2, 9.1, 9.3.3.3, 9.6, 10.2, 10.3.2, 11, 12.1, 12.4.1, 13.1, 13.2, 13.6(b), 13.5.5.1

[Shi17] Kijung Shin. Wrs: Waiting room sampling for accurate triangle counting in real graph streams. In *IEEE 17th International Conference on Data Mining*, pages 1087–1092. IEEE, 2017. 1.2.1.1, 3.2, 4, 5.1, 6.1

[SHK+10] Barna Saha, Allison Hoch, Samir Khuller, Louiqa Raschid, and Xiao-Ning Zhang. Dense subgraphs with restrictions and applications to gene annotation graphs. In *Annual International Conference on Research in Computational Molecular Biology*, pages 456–472. Springer, 2010. 13.1

[SHKF17a] Kijung Shin, Bryan Hooi, Jisu Kim, and Christos Faloutsos. D-cube: Dense-block detection in terabyte-scale tensors. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 681–689. ACM, 2017. 12

[SHKF17b] Kijung Shin, Bryan Hooi, Jisu Kim, and Christos Faloutsos. Densealert: Incremental dense-subtensor detection in tensor streams. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1057–1066, 2017. 1.2.2.2, 9.6, 10.2, 13

[SHKF18] Kijung Shin, Bryan Hooi, Jisu Kim, and Christos Faloutsos. Out-of-core and distributed algorithms for dense subtensor mining. *arXiv preprint arXiv:1802.01065*, 2018. 1.1, 1.2.2.2, 9.6, 10.2, 10.3.2, 12, 13.1, 13.6(b), 13.5.5.1, 16.2

[SHL+18] Kijung Shin, Mohammad Hammoud, Euiwoong Lee, Jinoh Oh, and Christos Faloutsos. Tri-fly: Distributed estimation of global and local triangle counts in graph streams. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 651–663. Springer, 2018. 1.2.1.1, 3.2, 5, 5.1, 6.1

[Sin01] Amit Singhal. Modern information retrieval: A brief overview. *IEEE Data Engineering Bulletin*, 24(4):35–43, 2001. 15.5.6

[SKHF18] Kijung Shin, Jisu Kim, Bryan Hooi, and Christos Faloutsos. Think before you discard: Accurate triangle counting in graph streams with deletions. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 141–157. Springer, 2018. 1.2.1.1, 3.2, 6

[SKHF19] Kijung Shin, Jisu Kim, Bryan Hooi, and Christos Faloutsos. Fast, accurate and provable triangle counting in fully dynamic graph streams. *Manuscript submitted for publication*, 2019. 6

[SKZ+15] Neil Shah, Danai Koutra, Tianmin Zou, Brian Gallagher, and Christos Faloutsos. Timecrunch: Interpretable dynamic graph summarization. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1055–1064. ACM, 2015. 7.6

[SLEP17] Kijung Shin, Euiwoong Lee, Dhivya Eswaran, and Ariel D. Procaccia. Why you should charge your friends for borrowing your stuff. In *Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 395–401, 2017. 1.1, 1.2.3.1, 14, 16.2

[SLO+19] Kijung Shin, Euiwoong Lee, Jinoh Oh, Mohammad Hammoud, and Christos Faloutsos. Cocos: Fast and accurate distributed triangle counting in graph streams. In *Manuscript submitted for publication*, 2019. 1.2.1.1, 3.2, 5

[SM04] Jouni K Seppänen and Heikki Mannila. Dense itemsets. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 683–688. ACM, 2004. 10.2

[Sni10] Moshe Sniedovich. *Dynamic programming: foundations and principles*. CRC press, 2010. 15.3.2.1

[Spe04] Charles Spearman. The proof and measurement of association between two things. *The American journal of psychology*, 15(1):72–101, 1904. 3.3.2, 1

[SPH+18] Hojin Seo, Kisung Park, Yongkoo Han, Hyunwook Kim, Muhammad Umair, Kifayat Ullah Khan, and Young-Koo Lee. An effective graph summarization and compression technique for a large-scaled graph. *The Journal of Supercomputing*, pages 1–15, 2018. 7.1, 7.6

[SSK17] Kijung Shin, Lee Sael, and U Kang. Fully scalable methods for distributed tensor factorization. *IEEE Transactions on Knowledge and Data Engineering*, 29(1):100–113, 2017. 10.2

[SSK+18] Kijung Shin, Mahdi Shafiei, Myunghwan Kim, Aastha Jain, and Hema Raghavan. Discovering progression stages in trillion-scale behavior logs. In *Proceedings of the 27th International Conference on World Wide Web*, pages 1765–1774. ACM, 2018. 1.1, 1.2.2.1, 1.2.3.2, 15, 16.2

[SSPC15] Ahmet Erdem Sariyuce, C Seshadhri, Ali Pinar, and Umit V Catalyurek. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *Proceedings of the 24th International Conference on World Wide Web*, pages 927–937. International World Wide Web Conferences Steering Committee, 2015. 9.6

[SSS+15] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-june Paul Hsu, and Kuansan Wang. An overview of microsoft academic service (mas) and applications. In *Proceedings of the 24th international conference on world wide web*, pages 243–246. ACM, 2015. 8.5.1

[STF06] Jimeng Sun, Dacheng Tao, and Christos Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 374–383. ACM, 2006. 1.2.1.2, 8.1, 10.2

[SV11] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World Wide Web*, pages 607–614. ACM, 2011. 3.1, 3.2, 5.1

[SWL+18] Qi Song, Yinghui Wu, Peng Lin, Luna Xin Dong, and Hui Sun. Mining summaries for knowledge graph search. *IEEE Transactions on Knowledge & Data Engineering*, 30(10):1887–1900, 2018. 7.6

[SYS+13] Dafna Shahaf, Jaewon Yang, Caroline Suen, Jeff Jacobs, Heidi Wang, and Jure Leskovec. Information cartography: creating zoomable, large-scale maps of information. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1097–1105. ACM, 2013. 15.6

[SZL+05] Jian-Tao Sun, Hua-Jun Zeng, Huan Liu, Yuchang Lu, and Zheng Chen. Cubesvd: a novel approach to personalized web search. In *Proceedings of the 14th international conference on World Wide Web*, pages 382–390. ACM, 2005. 1.2.1.2, 8.1

[TBG+13] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 104–112. ACM, 2013. 10.1, 10.2, 10.3.2, 10.3

[TDM+11] Charalampos E Tsourakakis, Petros Drineas, Eirinaios Michelakis, Ioannis Koutis, and Christos Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining*, 1(2):75–81, 2011. 1.2.1.1, 3.1

[TFP06] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Fast random walk with restart and its applications. In *Sixth IEEE International Conference on Data Mining*, pages 613–622. IEEE, 2006. 7.5.3

[THP08] Yuanyuan Tian, Richard A Hankins, and Jignesh M Patel. Efficient aggregation for graph summarization. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 567–580. ACM, 2008. 7.6

[TKMF09] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 837–846. ACM, 2009. 3.2, 9.4.3.1

[TPT13] Kanat Tangwongsan, Aduri Pavan, and Srikanta Tirthapura. Parallel triangle counting in massive streaming graphs. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 781–786. ACM, 2013. 3.1, 3.2, 6.1

[Tso08] Charalampos E Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *Eighth IEEE International Conference on Data Mining*, pages 608–617. IEEE, 2008. 9.4.2

[Tso10] Charalampos E Tsourakakis. Mach: Fast randomized tensor decompositions. In *Proceedings of the 2010 SIAM International Conference on Data Mining*, pages 689–700. SIAM, 2010. 8.3.2

[Tuc66] Ledyard R Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966. 1.2.1.2, 8.1, 8.2.1.3

[TZHH11] Hannu Toivonen, Fang Zhou, Aleksi Hartikainen, and Atte Hinkka. Compression of weighted graphs. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 965–973. ACM, 2011. 7.6

[VFH15] Shoshana Vasserman, Michal Feldman, and Avinatan Hassidim. Implementing the wisdom of waze. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, pages 660–666, 2015. 14.6

[Vit85] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985. 3.3.1, 6.3.3

[VL00] Charles F Van Loan. The ubiquitous kronecker product. *Journal of computational and applied mathematics*, 123(1):85–100, 2000. 9.4.2, 9.4.2

[VMCG09] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM workshop on Online social networks*, pages 37–42. ACM, 2009. 4.6.1, 4.2, 5.4, 6.2

[WA05] Stefan Wuchty and Eivind Almaas. Peeling the yeast protein network. *Proteomics*, 5(2):444–449, 2005. 9.1, 9.6

[WCW+17] Xiao Wang, Peng Cui, Jing Wang, Jian Pei, Wenwu Zhu, and Shiqiang Yang. Community preserving network embedding. In *Proceedings of the Thirty-First AAAI Conference on*

Artificial Intelligence, pages 203–209, 2017. 7.3.2.1

[WF94] Stanley Wasserman and Katherine Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994. 1.2.1.1, 3.1, 4.3, 17

[WL12] Robert West and Jure Leskovec. Human wayfinding in information networks. In *Proceedings of the 21st international conference on World Wide Web*, pages 619–628. ACM, 2012. 15.1

[WOS06] Kesheng Wu, Ekow J Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006. 7.6

[WS98] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998. 3.1

[WTSA15] Yining Wang, Hsiao-Yu Tung, Alexander J Smola, and Anima Anandkumar. Fast and guaranteed tensor decomposition via sketching. In *Advances in Neural Information Processing Systems*, pages 991–999, 2015. 10.2

[WZT+16] Gang Wang, Xinyi Zhang, Shiliang Tang, Haitao Zheng, and Ben Y Zhao. Unsupervised clickstream clustering for user behavior analysis. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 225–236. ACM, 2016. 15.6

[WZTT10] Nan Wang, Jingbo Zhang, Kian-Lee Tan, and Anthony KH Tung. On triangulation-based dense neighborhood graph discovery. *Proceedings of the VLDB Endowment*, 4(2):58–68, 2010. 3.1

[YL15] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015. 5.4, 6.2, 7.2, 9.2, 9.2.3

[YML+14] Jaewon Yang, Julian McAuley, Jure Leskovec, Paea LePendu, and Nigam Shah. Finding progression stages in time-evolving event sequences. In *Proceedings of the 23rd international conference on World wide web*, pages 783–794. ACM, 2014. 15.1, 15.1, 15.6

[Zac77] Wayne W Zachary. An information flow model for conflict and fission in small groups. *Journal of anthropological research*, 33(4):452–473, 1977. 14.5

[ZLL+17] Yu Zhu, Hao Li, Yikang Liao, Beidou Wang, Ziyu Guan, Haifeng Liu, and Deng Cai. What to do next: Modeling user behaviors by time-lstm. In *Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 3602–3608, 2017. 15.6

[ZOS+19] Jiyuan Zhang, Jinoh Oh, Kijung Shin, Evangelos E Papalexakis, Christos Faloutsos, and Hwanjo Yu. Fast and memory-efficient algorithms for high-order tucker decomposition. *Manuscript submitted for publication*, 2019. 8

[ZVB+16] Shuo Zhou, Nguyen Xuan Vinh, James Bailey, Yunzhe Jia, and Ian Davidson. Accelerating online cp decompositions for higher order tensors. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1375–1384. ACM, 2016. 10.2

[ZZY+17] Si Zhang, Dawei Zhou, Mehmet Yigit Yildirim, Scott Alcorn, Jingrui He, Hasan Davulcu, and Hanghang Tong. Hidden: hierarchical dense subgraph detection with application to financial fraud detection. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, pages 570–578. SIAM, 2017. 9.6