UNIVERSITY of CALIFORNIA
Santa Barbara

Mining and Managing Large-Scale Temporal Graphs

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

Doctor of Philosophy

in

Computer Science

by

Bo Zong

Committee in Charge:

Professor Ambuj K. Singh, Co-Chair

Professor Xifeng Yan, Co-Chair

Professor Subhash Suri

December 2015

The Dissertation of Bo Zong is approved.

Professor Xifeng Yan, Co-Chair

Professor Subhash Suri

Professor Ambuj K. Singh, Committee Chair

December 2015

Mining and Managing Large-Scale Temporal Graphs

Copyright © 2015

by

Bo Zong

To my parents and my wife, for your endless love and unconditional support that make me happy everyday, lend me courage to explore unknown world, and lead me to where I am.

Acknowledgements

First of all, I would like to express my sincere gratitude to my advisors, Prof. Ambuj K. Singh and Prof. Xifeng Yan. I am fortunate to work with two great advisors who give me invaluable guidance on my research path and also grant me freedom to explore independent work. In my five years' PhD study, I have learned how to perform good research from my advisors' enthusiasm, immense knowledge, and patience. Outside of academia, Ambuj and Xifeng are also the advisors in my life. They share with me the wisdom of how to live a happy, healthy, and meaningful life, and give me great support and encouragement.

Second, I sincerely thank my committee member Prof. Subhash Suri. I am deeply grateful for his valuable feedback and insightful suggestions on my work.

My sincere thanks also go to Prof. Yinghui Wu and Dr. Misael Mongiovì. It is my great pleasure to work with them on multiple interesting projects. I am also grateful to Dr. Nan Li, Dr. Kyle Chipman, Dr. Arijit Khan, Prof. Petko Bogdanov, Dr. Nicholas D. Larusso, and Prof. Sayan Ranu for their various forms of help during my PhD study.

I am indebted to all my labmates: Dr. Shengqi Yang, Dr. Xuan-Hong Dang, Dr. Yang Li, Minh X. Hoang, Huan Sun, Arlei Lopes da Silva, Victor Amelkin, Fangqiu Han, Sourav Medya, Honglei Liu, Anh Nguyen, Yu Su, Haraldur Hallgrimsson, Izzeddin Gür, Theodore Georgiou, Semih Yavuz, and Hongyuan You.

I would like to acknowledge my collaborators and mentors for their valuable advice and numerous discussions: Dr. Ramya Raghavendra, Dr. Mudhakar Srivatsa, Dr. Christos Gkantsidis, Dr. Milan Vojnovic, Dr. Zhichun Li, Dr. Xusheng Xiao, Dr. Zhenyu Wu, and Prof. Zhiyun Qian.

Finally, I would like to thank my loving family. Thanks to my father Yonghua Zong and my mother Qiandi Shao for their endless love, and for always being there for me. Thanks to my wife Dr. Xiaohan Zhao for her unconditional love, support, and encouragement.

Curriculum Vitæ

Bo Zong

Education

2015	Ph.D in Computer Science, University of California, Santa Barbara.
2010	Master of Science in Computer Science, Nanjing University, China.
2007	Bachelor of Science in Computer Science, Nanjing University, China.

Experience

08/2010 – 09/2015	Research Assistant, University of California, Santa Barbara.
03/2015 – 06/2015	Teaching Assistant, University of California, Santa Barbara.
06/2015 – 07/2015	Research Mentor, University of California, Santa Barbara.
06/2014 – 09/2014	Research Intern, NEC Labs America, Princeton, NJ, USA.
05/2013 – 08/2013	Research Intern, Microsoft Research, Cambridge, UK.
06/2012 – 09/2012	Research Intern, IBM Research T. J. Watson, Yorktown Heights, NY, USA.

Publication

(Publications marked with ’*’ order authors alphabetically.)

Bo Zong, Christos Gkantsidis, and Milan Vojnovic. “Herding ‘Small’ Streaming Queries”. *International Conference on Distributed Event-Based Systems (DEBS)*, July 2015.

Bo Zong, Yinghui Wu, Jie Song, Ambuj K. Singh, Hasan Cam, Jiawei Han, and Xifeng Yan. “Towards Scalable Critical Alert Mining”. *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, Aug. 2014.

Bo Zong, Ramya Raghavendra, Mudhakar Srivatsa, Xifeng Yan, Ambuj K. Singh, and Kang-Won Lee. “Cloud Service Placement via Subgraph Matching”. *International Conference on Data Engineering (ICDE)*, Apr. 2014.

Bo Zong, Yinghui Wu, Ambuj K. Singh, and Xifeng Yan. “Inferring the Underlying Structure of Information Cascades”. *International Conference on Data Mining (ICDM)*, Dec. 2012.

Shengqi Yang, Xifeng Yan, **Bo Zong**, and Arijit Khan. “Towards Effective Partition Management for Large Graphs”. *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, May. 2012.

Misael Mongiovi, Konstantinos Psounis, Ambuj K. Singh, Xifeng Yan, and **Bo Zong**. *“Efficient Multicasting for Delay Tolerant Networks using Graph Indexing”. *Annual International Conference on Computer Communications (INFOCOM)*, Mar. 2012.

Bo Zong, Feng Xu, Jun Jiao, and Jian Lu. “A Broker-assisting Trust and Reputation System Based on Artificial Neural Network”. *International Conference on Systems, Man and Cybernetics (SMC)*, Oct. 2009.

Abstract

Mining and Managing Large-Scale Temporal Graphs

by

Bo Zong

Large-scale temporal graphs are everywhere in our daily life. From online social networks, mobile networks, brain networks to computer systems, entities in these large complex systems communicate with each other, and their interactions evolve over time. Unlike traditional graphs, temporal graphs are dynamic: both topologies and attributes on nodes/edges may change over time. On the one hand, the dynamics have inspired new applications that rely on mining and managing temporal graphs. On the other hand, the dynamics also raise new technical challenges. First, it is difficult to discover or retrieve knowledge from complex temporal graph data. Second, because of the extra time dimension, we also face new scalability problems. To address these new challenges, we need to develop new methods that model temporal information in graphs so that we can deliver useful knowledge, new queries with temporal and structural constraints where users can obtain the desired knowledge, and new algorithms that are cost-effective for both mining and management tasks.

In this dissertation, we discuss our recent works on mining and managing large-scale temporal graphs.

First, we investigate two mining problems, including node ranking and link prediction problems. In these works, temporal graphs are applied to model the data generated from computer systems and online social networks. We formulate data mining tasks that extract knowledge from temporal graphs. The discovered knowledge can help domain experts identify critical alerts in system monitoring applications and recover the complete traces for information propagation in online social networks. To address computation efficiency problems, we leverage the unique properties in temporal graphs to simplify mining processes. The resulting mining algorithms scale well with large-scale temporal graphs with millions of nodes and billions of edges. By experimental studies over real-life and synthetic data, we confirm the effectiveness and efficiency of our algorithms.

Second, we focus on temporal graph management problems. In these study, temporal graphs are used to model datacenter networks, mobile networks, and subscription relationships between stream queries and data sources. We formulate graph queries to retrieve knowledge that supports applications in cloud service placement, information routing in mobile networks, and query assignment in stream processing system. We investigate three types of queries, including subgraph matching, temporal reachability, and graph partitioning. By utilizing the relatively stable components in these temporal graphs, we develop flexible data management techniques to enable fast query processing and handle graph dynamics. We evaluate the soundness of the proposed techniques by both real and synthetic data.

Through these study, we have learned valuable lessons. For temporal graph mining, temporal dimension may not necessarily increase computation complexity; instead, it may reduce computation complexity if temporal information can be wisely utilized. For temporal graph management, temporal graphs may include relatively stable components in real applications, which can help us develop flex-

ible data management techniques that enable fast query processing and handle dynamic changes in temporal graphs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Temporal graphs are ubiquitous in our daily life. From online social networks [101, 208], mobile networks [128, 133], road networks [63], brain networks [160], to computer systems [124, 209], entities in these large complex systems are not isolated. Instead, they communicate with each other, and their interactions evolve over time, which makes temporal graph a natural data model for such dynamic relational data. To extract meaningful knowledge from these data, it is critical to provide efficient and effective tools that is able to mine and manage temporal graphs.

Unlike traditional graphs, temporal graphs are dynamic. First, topologies in temporal graphs can evolve over time [118, 149, 152, 204]. For example, in applications of online social networks [208], temporal graphs are applied to model dynamic communications between users, where nodes are users, and edges indicate at which time who talked with whom. In this case, the topologies of these temporal graphs evolve with the communication records between users. Second, attributes on nodes/edges can be changed over time [27, 63, 130]. Take datacenter management as an example [207]. Temporal graphs are used to model datacenter networks, where nodes represent servers, edges are connections between servers, and attributes on nodes/edges denote the amount of available computation resources. Node/edge attributes in these temporal graphs can change over time, due to the dynamic workload in datacenters (*e.g.,* new tasks join a datacenter, or old tasks are finished and then leave a datacenter). The dynamics in temporal graphs has inspired new applications that rely on mining and managing temporal graphs.

A big track of applications rely on the knowledge discovered by mining temporal graphs. Temporal graph data can be generated from a variety of domains, such as cybersecurity [92, 97], system management [124, 146], medical healthcare [32, 171], and many others [119, 131, 186]. Applications in these domains (*e.g.,* malware detection in cybersecurity, root cause analysis in system management, and treatment effectiveness prediction in medical healthcare) desire the insights concealed in the collected data. Meanwhile, a common set of data mining tasks over temporal graphs are able to serve the knowledge demand from different applications. These tasks include pattern mining [39], anomaly detection [20], ranking [209], link prediction [137], and so on [17, 25, 172]. For example, in the application of malware detection, a key task is to build signatures for malware. In this case, temporal graphs are collected from system call logs generated computer systems, where nodes are basic system entities (*e.g.,* processes, files, sockets, etc.) and edges suggest at which time what kind of interactions happened between these system entities. By performing pattern mining tasks over the data, we can find discriminative patterns that are unique for malware and then these patterns can serve as signatures for malware. In sum, the knowledge discovered by mining temporal graphs benefit various applications.

Another track of applications rely on querying and managing temporal graphs. Real-life tasks, such as forensic analysis in cybersecurity [188], service placement in datacenter management [77], and disease detection in medical healthcare [138], can be formulated as querying problems against temporal graph data, including pattern matching [207], similarity search [91], reachability [206], optimal path [187], and so on [170]. Take forensic analysis in cybersecurity as an example. The goal of this task is to detect the existence of suspicious activities in computer systems. In this sense, forensic analysis can be served as a pattern matching problem: a query is a small temporal graph indicating how system entities interact when a suspicious activity happens, a database stores a large temporal graph recording a history of system entity interactions, and the goal is to find matches in the database for the specified pattern in the query. If any matches are found, suspicious activities exist. To serve queries on temporal graphs and retrieve knowledge for different applications, we need to provide efficient management tools that enable fast query performance.

While dynamics in temporal graphs have brought us new opportunities, we are also facing new technical challenges.

First, it is difficult to discover or retrieve knowledge from complex temporal graph data.

- For mining, the key questions are what kind of new knowledge we can deliver from temporal graph data and how to model such temporal information in graphs so that we can deliver useful knowledge for real-life applications. The answers to these questions remain unknown.

- For management, the key challenge is how to build meaningful queries such that users can obtain the desired knowledge. In other words, to retrieve knowledge, we have to submit proper questions; however, over complex temporal graph data, it is burdensome to formulate right queries.

Second, due to the high-level dynamics, we also face new scalability problems.

- For mining, because of the extra time dimension, the underlying search space becomes much larger. Existing mining algorithm cannot scale or even deal with the time dimension. New mining algorithms are desired to scale with temporal graphs.

- For management, we usually need techniques, such as graph indexing, compression, or partitioning, to speed up query processing. Existing data management techniques mainly focus on static graphs, and cannot deal with dynamics in graphs. Therefore, it is critical to develop flexible data structures that efficiently manage temporal graphs.

My research work aims to address the new challenges raised by mining and managing large-scale temporal graphs. The statement of this dissertation is as follows.

To discover and retrieve knowledge from large-scale temporal graphs, we need to understand how to utilize temporal structural information and develop cost-effective algorithms that scale with both dynamics and size of graphs.

Driven by the statement, we have developed algorithms and tools to mine and manage temporal graphs.

In terms of temporal graph mining, we have investigated two important problems, including ranking problems and link prediction. These two problems support critical applications in system management and online social networks. In this series of study, we investigate what kind of new knowledge are brought by temporal information and how these new knowledge benefits real-life applications. Moreover, we analyze how temporal dimension in graphs raises computation difficulties and develop mining algorithms to overcome these problems.

In terms of temporal graph management, we have tackled management problems such as subgraph matching, temporal reachability, and graph partitioning. In particular, we studied these problems in the background of datacenter networks, mobile networks, and stream processing systems, respectively. In these works, we unveil the importance of managing temporal graphs, and demonstrate how temporal information can help us address the scalability problems in temporal graph management.

Next, we briefly introduce the works included in this dissertation.

1.1 Mining Temporal Graphs

Node ranking and link prediction are our recent focus in the category of mining temporal graphs.

Node ranking in temporal graphs. Datacenters are computation facilities used for hosting users' services and data. They are powerful but complex. In general, it is impossible for human beings to manually check whether a datacenter performs normally. What we usually do is we plant sensors into datacenters to monitor their performance. When these monitoring data suggests there are anomalies in the systems, alerts will be generated. A big headache for system admins is there are too many alerts, and they have no time to check the alerts one by one. We also notice that among the large amount of alerts, some alerts are critical and can trigger many other alerts. System admins should first fix problems behind the critical alerts, and then other alerts will automatically disappear. In this work, our goal is to help system admins find the most critical alerts so that they can work on those alerts first. To address this problem, we first build temporal graphs on alerts representing their dependencies over time. Because of temporal dependency among alerts, the resulting temporal graphs are directed and acyclic. This property inspires us to develop efficient inference algorithms that identify alerts that have high probability to trigger a large number of other alerts. Note that the idea in this work is not restricted to datacenters. It can also be applied to managing other complex systems, like electricity power plants, aircraft systems, and so on. Moreover, the proposed algorithms can also be applied in online social networks for influence maximization problems.

Link prediction in temporal graphs. In online social networks, information cascades are temporal graphs that record traces of information propagation. While information cascades provide valuable materials for studying the processes governing information propagation, in practice it is difficult to obtain the complete structures for information cascades, because of data privacy policies and noise. In this work, we study a cascade inference problem: Given partially observed cascades and a social network of users, the goal is to recover the structures for the partially observed cascades. The search space of this problem is extremely large because of the extra temporal dimension and pure graph size (*i.e.,* the size of online social networks). To tackle the search scalability problem, we propose to use both temporal and structural information in partial observations to identify infeasible information flows between users and prune the search space. The resulting algorithms improve the inference accuracy and scale well with large graphs of millions of nodes and billions of edges.

From these studies we have learned valuable lessons. First, we have found concrete evidences showing how knowledge discovered from temporal graphs benefits applications from different areas. Second, we made an interesting observation from these two problems. When we deal with temporal data, we usually think they will complicate mining processes. But what we found in our study is if we wisely utilize temporal information, we can even simplify mining algorithms, which is counter-intuitive.

1.2 Managing Temporal Graphs

In the direction of temporal graph management, we have investigated subgraph matching, temporal reachability, and graph partitioning.

Subgraph Matching in temporal graphs. In a cloud datacenter, a routine task is to find a set of servers that can host users' services. A user's service may include multiple resource requirements. For example, one user may want to rent 6 virtual machines. First, each machine may require different amount of computation resources, like memory, CPU, and bandwidth. Second, the user may want these 6 machines to be connected in a specific way, like a star, a ring or even more complex topology. When a user's service arrives, we need to find qualified servers as soon as possible in order to guarantee the system's throughput. In this work, we represent users' services as small graphs, nodes represent virtual machines in services, edges represent the connections between machines, and attributes on nodes and edges represent required resources. Cloud datacenters are modeled as large temporal graphs. Nodes represent available servers, edges represent possible connections, attributes represent available computation resources. In such graphs, the attributes will change over time because of the dynamics in cloud. To this end, the cloud service placement becomes a graph querying problem, and the goal is to find subgraphs in the large graph that can match the query. We observe that network structures in datacenters are relatively stable. Based on this observation, we develop indexing techniques to speed up subgraph matching, and this index can be efficiently updated when node/edge attributes in the large graph evolve. In this work, we identify a key application for graph queries, and develop the first graph index that can handle numerical attributes and their dynamic evolution.

Temporal reachability. In this work, we consider a set of moving entities like buses or soldiers. When two entities are close enough to each other, they can communicate; otherwise, they will be disconnected. Therefore, we can use temporal graphs to model their dynamic connections over time. In this work, we focus on information routing in such mobile networks. In particular, we aim to minimize network communication cost when information are required to be sent to a subset of entities within a time window. To solve this problem, we need to check temporal reachability: whether one entity can send information to another within a time window. In general, one information routing task could generate a large number of temporal reachability queries. If we process those queries one by one, it will be very slow. In the study, we found entities in mobile networks usually follow some periodic movement patterns, and develop indexing techniques based on these patterns. This index can process temporal reachability queries in a batch, which significantly improves the speed of reachability information collection. With the reachability information, we can quickly find the optimal routing strategy by existing linear programming solvers.

Partitioning in temporal graphs. In a stream processing system, data sources and queries form a bipartite graph representing their subscription relationships. While data sources continuously generate data as streams, queries subscribe to one or multiple data sources to obtain the desired knowledge. This subscription graph is temporal, as queries can dynamically arrive and leave. The number of queries in the system could be huge. Therefore, it is difficult to host all the queries in one single server. An intuitive idea is to distribute the queries into multiple servers, but distributing queries will bring two problems. First, query distribution will result in extra networking traffic, because we might need to send the stream data from the same data sources to multiple servers. Second, balanced workload among servers is preferred, which minimizes wasted computing resources. Inspired by these constraints, we formulate a query placement problems: given the data sources, queries, and servers, we want to place queries into servers so that workload is balanced and the overall network traffic is minimized. We propose a full set of algorithms to tackle this problem. First, for the case of static queries, we develop bounded approximation algorithms. Second, for queries with dynamic arrival and leaving, we find the popularity distribution of data sources is usually stable in practice, and propose a probabilistic model to randomly assign queries with performance guarantee. Finally, for dynamic queries where the popularity distribution of data sources changes over time, we develop heuristic algorithms that empirically work well.

The following is the key insight drawn from our works on managing temporal graphs. In many cases, we can identify relatively stable components from temporal graphs (*e.g.,* stable network structure, periodic movement patterns, or stable interest/popularity distribution). These stable parts can be very useful: they form the backbone data structures in data management; on top of the backbones, we can further build light-weight data structures that handle the dynamics.

1.3 Contributions

In the following, we summarize the key contributions of this dissertation.

- We identify key applications of mining and managing temporal graphs in multiple domains. In terms of mining, Chapters 2 and 3 demonstrate how mining temporal graphs discovers critical alerts in system management and recovers the missing cascade structure in online social networks. In terms of management, Chapters 4, 5, and 6 reveals the critical applications of temporal graph management including service placement in datacenter management, information routing in mobile networks, and query assignment in stream processing systems.

- We propose scalable mining algorithms to extract meaningful knowledge from large-scale temporal graphs. (1) On critical alert mining, by leveraging the directed acyclic property caused by temporal dependency among alerts, we first develop fast approximation algorithms that is able to find near-optimal solutions, and then develop highly efficient sampling algorithms that empirically work well on real-life data. (2) On information cascade inference, we propose consistent trees as the model to infer the missing cascade structures, and use both temporal and structural constraints obtained from partial observations to prune the underlying search space. The resulting algorithm improves inference accuracy and scales with large-scale graphs with millions of nodes and billions of edges.

- Cost-effective algorithms are developed to manage large-scale temporal graphs. We identify relatively stable components in temporal graphs, and make use of the components to build backbone data structures that efficiently deal with the dynamics in temporal graphs. (1) In dynamic subgraph matching, the topologies of temporal graphs are quite stable, but node/edge attributes are highly dynamic. We propose a graph index based on stable topologies, and then develop grid-based indexes inside of the graph index to handle dynamically changing node/edge attributes. The proposed index can scale with millions of attribute updates per second, and process subgraph matching queries in a few seconds. (2) For temporal reachability in mobile networks, topologies of temporal graphs are highly dynamic, and the evolution of topologies follows periodic patterns. Based on this observation, we develop a graph index that processes temporal reachability queries in a batch, which significantly improves the speed of finding optimal information routing strategy in mobile networks. (3) In terms of stream query assignment, topologies of temporal graphs are also dynamic, but the degree distribution in the graphs is relatively stable in practice. This insight helps us develop a probabilistic model that randomly assign queries with performance guarantee.

1.4 Thesis Organization

The rest of the dissertation is organized as follows. We start with mining problems, where critical alert mining and information cascade inference are discussed in Chapters 2 and 3, respectively. Next, we move to management problems in Chapters 4, 5, and 6, covering dynamic subgraph matching, temporal reachability, and stream query assignment. At the end of this dissertation, we summarize our works, and discuss future directions.

Chapter 2

Node Ranking in Temporal Graphs

2.1 Introduction

System monitoring and analysis in datacenters and cybersecurity applications produces alert sequences to capture abnormal events. For example, performance metrics are posed on hosts in datacenters to measure the system activities, and capture alerts such as high CPU usage, memory overflow, or service errors. Understanding the causal and dependency relations among these alerts is critical for datacenter management [74, 134], cyber security [99], and device network diagnosis [124], among others.

While there exists a variety of approaches for modeling and deriving causal relations [26, 158, 164], another important step is to efficiently suggest critical alerts from a huge amount of observed alerts. Intuitively, these critical alerts indicate the "root causes" that account for the observed alerts, such that if fixed, we may expect a great reduction of other alerts without blindly addressing them one by one. We consider several real-life applications below.

Datacenters. System monitoring and analysis providers seek efficient and reliable techniques to understand a large number of system performance alerts in datacenters. According to LogicMonitor[1], a SaaS network monitor company, a datacenter of 122 servers generates more than $20,000$ alerts per day. While it is daunting for domain experts to manually check these alerts one by one, it is desirable to automatically suggest a small set of alerts that are potentially causes for a large amount of alerts, for further verification. These critical alerts also help in determining key control points for datacenter infrastructures [134].

Intrusion detection [22, 98]. State-of-the-art intrusion detection systems produce large numbers of alerts from cyber network sensors, over tens of thousands of security metrics, *e.g.,* Host scan or TCP hijacking [22]. As suggested in [98], it is observed that a few critical alerts generally account for over 90% of the alerts that an intrusion detection system triggers. By handling only a small number of critical alerts, a huge amount of effort and resource can be reduced. On the other hand, critical alerts can reduce the number of "false alerts" and improve alarm quality [22].

Network performance diagnosis [124]. Large-scale IP networks (*e.g.,* North America IPTV network) contain millions of devices, which generate a great number of performance alarms from customer call records and provider logs. Scalable mining of critical alerts for a given set of symptom events benefits fast network diagnosis [124].

[1] http://www.logicmonitor.com/

These highlight the need for efficient algorithms to mine critical alerts, given the sheer size of observed ones. In this chapter, we investigate efficient critical alert mining techniques. We focus on a general framework with desirable performance guarantees on alert quality and scalability.

(1) We formulate the *critical alert mining* problem: Given a set of alerts and a number k, it aims to find a set of k critical alerts, such that the number of alerts that are potentially caused by them is maximized. We introduce a generic framework for mining critical alerts. In this framework, we learn and maintain a temporal graph over alerts (referred to as an alert graph), a graph representation of causal relations among alerts. Upon users' requests, top critical alerts are mined from alert graphs.

(2) We show that the critical alert mining problem is NP-complete. Nonetheless, we provide an algorithm with approximation ratio $1-\frac{1}{e}$, in time $O(k|V||E|)$, where $|V|$ and $|E|$ are the number of alerts and the number of their causal relations, respectively. To further improve the efficiency of the algorithm, we propose a bound and pruning algorithm that effectively reduces the size of alerts to be verified as critical ones. In addition, we identify a special case: when alert graphs are trees, it is in $O(k|V|)$ time to find k critical alerts, with the same approximation ratio.

(3) The quadratic time approximation may still be expensive for large alert graphs. We further propose two fast heuristics for large-scale critical alert mining. These algorithms induce trees that preserve the most probable causal relations from large alert graphs, and estimate top critical alerts and their impact by only accessing

the trees. The first one induces a single tree, while the second algorithm balances alert quality and mining efficiency with multiple sampled trees.

(4) We experimentally verify our critical alert mining framework. Over real-life datacenter datasets, our algorithms effectively identify critical alerts that trigger a large number of other alerts, as verified by domain experts. We found that our approximation algorithms mine a top critical alert from up to $270,000$ causal relations (one day's alert sequences) in 5 seconds. On the other hand, while our heuristics preserve more than 80% of solution quality, they are up to $5,000$ times faster than their approximation counterparts. The heuristics also scale well over large synthetic alert graphs, with up to in total 1 million alerts and 10 million relations.

In contrast to conventional causality modeling and mining, our algorithms leverage effecitive pruning and sampling methods for fast critical alert mining. In addition, we do not assume the luxury of accessing rich semantics from the alerts that helps in improving mining efficiency, although our methods immediately benefit from the semantics in specific applications [98, 124, 134], as well as domain experts. Taken together with domain knowledge and causality mining tools, these algorithms are one step towards large-scale critical alert analysis for datacenters, intrusion detection systems, and network diagnosis systems.

2.2 Problem definition

We start with the notions of alert sequences and alert graphs. Then we introduce the critical alert mining problem.

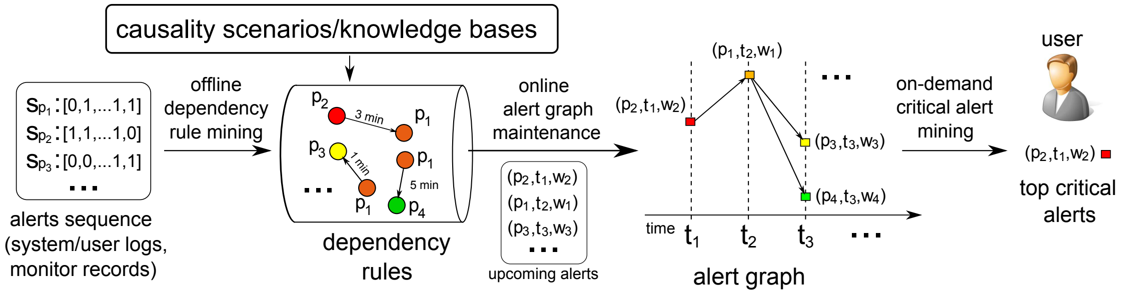


Figure 2.1: Critical alert mining: pipeline

Performance metrics. A performance metric measures an aspect of system performance. For datacenters, common types of performance metrics include CPU and memory usage for virtual machines, error rate of disk writes for a service, or communication time between two hosts. The same type of metrics over different hosts, virtual machines, or services are considered as distinct performance metrics.

In practice, system service providers e.g., LogicMonitor may cope with 2 million metrics from a datacenter with $5,000$ hosts. These metrics could correlate with and cause each other due to functional or resource dependencies.

Alert and alert sequences. For a set of performance metrics $\mathbb{P}$, alerts are determined by aggregating the metric values of interest. For example, in datacenters, an alert is raised when the value of a performance metric (*e.g.,* CPU usage) goes beyond a pre-defined threshold (*e.g.,* $> 75\%$). In this work, we define an alert as a triple $u = (p_u, t_u, w_u)$, where $p_u \in \mathbb{P}$ is a performance metric u corresponds to, t_u denotes the timestamp when the alert u happened, and w_u is the weight of u, representing the benefit if u is fixed.

We use a sequence of alerts to characterize abnormal events for a specific performance metric. Indeed, in practice the performance metrics are typically periodically monitored to capture the abnormal events as alerts. We denote as

$\vec{s}_p$ an alert series (an ordered sequence of alerts following their timestamps), for a specific performance metric $p \in \mathbb{P}$. Each entry of $\vec{s}_p$ is either 0 (normal) or 1 (alert).

To characterize causal relations between two alerts, we next introduce a notion of *dependency rule*. We also introduce *alert graph* as an intuitive graph representation for multiple dependency rules.

Dependency rule. Let p and q be two distinct performance metrics. A dependency rule $p \xrightarrow{l_{pq}} q$ denotes an alert issued on q at some time t is caused by an alert issued on p at $t' \in [t - l_{pq}, t - 1]$, where l_{pq} is a *lag* from p to q (*e.g.,* 5 minutes). Note that we do not specify the time t, as a dependency rule describes a statistical rule for all the observed alerts. Intuitively, a dependency rule indicates that alerts on q occurs if and only if alerts on p occurs as the cause of the alerts on q; that is, the alerts on p will trigger the alerts on q. If certain trouble shooting action is taken to fix p, q is addressed accordingly [22, 124].

Dependency rules can be automatically learned from alert series [26, 158]. They can also be suggested by experts and existing knowledge bases [59]. To smoothen the noise or error brought by rule generation process, we associate an uncertainty to each dependency rule. In particular, we denote the uncertainty by $\Pr(p \xrightarrow{l_{pq}} q)$, which is the probability that the corresponding dependency rule holds.

Alert graph. An alert graph over a set of alerts V is a directed acyclic graph $G = (V, E, f_e)$:

- V is the set of vertices in G, where each vertex $v \in V$ is an alert from V.

- E is a set of edges in G. Let $u = (p_u, t_u, w_u)$ and $v = (p_v, t_v, w_v)$ be two alerts in V. There is an edge $(u, v) \in E$ if and only if there exists a dependency rule $p_u \xrightarrow{l_{p_u p_v}} p_v$, where $t_u < t_v$, and $t_v - t_u \leq l_{pq}$.
- f_e is a function that assigns for each edge (u, v) the probability that u *causes* v, *i.e.*, $\Pr(p_u \xrightarrow{l_{p_u p_v}} p_v)$.

We shall use the following notations. Abusing the notions from tree topology, we say u (resp. v) is a parent (resp. child) of v (resp. u) if $(u, v) \in E$, and the edge (u, v) is an incoming edge of v. The *topological order* r of an alert u in G is defined as follows. (a) $r(u) = 0$ if u has no parent, and (b) $r(u) = 1 + \max r(v)$, for all its parents v.

Following the convention of causal relation and cascading models [142], we assume that an alert is caused by a single alert issued earlier, if any. Intuitively, a path from an alert u to another alert v in the alert graph indicates a potential "causal chain" from u to v, indicated by e.g., the actual dependencies among the vulnerabilities of the servers [43].

Critical alerts. We next introduce a metric to characterize critical alerts, in terms of how many alerts are potentially caused by them via a cascading effect (and hence are addressed if the critical ones are fixed). Given $G = (V, E, f_e)$, a set of fixed alerts $S \subseteq V$, and an alert $u \in V$, we use a notion of *alert-fixed probability* P_f to characterize the probability that u is fixed if S is fixed. More specifically,

- $P_f(S, u) = 1$ if $u \in S$,
- otherwise,

$$P_f(S, u) = 1 - \prod_{(u', u) \in E} \Big(1 - P_f(S, u') f_e(u', u)\Big).$$

Based on the alert-fixed probability, we next define a set function, denoted as Gain, to characterize critical alerts. Given an alert graph $G = (V, E, f_e)$ and $S \subseteq V$, the gain of S is a set function

$$\mathsf{Gain}(S) = \sum_{u \in V} w_u \cdot P_f(S, u).$$

As remarked earlier, here w_u refers to the weight of u, *i.e.*, the benfit if u is fixed. Intuitively, $\mathsf{Gain}(S)$ computes the total expected benefits induced via fixing a set of alerts S and subsequently addressing the alerts caused by S. The larger $\mathsf{Gain}(S)$ is, the more "critical" S is.

We next introduce the critical alert mining problem.

Definition 1. *Given an alert graph G and an integer k, the critical alert mining problem (referred to as* CAM*) is to find a set of k critical alerts $S \subset V$ such that* $\mathsf{Gain}(S)$ *is maximized.*

Finding the best set of k alerts which maximize the gain is desirable albeit intractable.

Theorem 1. *For a given alert graph G and an integer k, the problem* CAM *is NP-complete.*

Proof. We prove the NP-completeness of the decision version of CAM as follows. (1) CAM is in NP. Indeed, given an alert graph $G = (V, E)$ and a set of vertices $S \subseteq V$, one can evaluate Gain by computing $P_f(S, v)$ of each alert v in polynomial time. (2) To show that CAM is NP-hard, we construct a reduction from the maximum coverage problem, which is known to be NP-hard [180]. An instance of a maximum coverage problem consists of a set of sets $\mathcal{S}$ and an integer k. It selects at most k of these sets such that the number of elements that are covered is no less than a bound B. A maximum coverage instance can be constructed as a bipartite alert graph, with each "upside" node as a set in $\mathcal{S}$, each "downside" node a distinct element in these sets, and there is an edge from upside node to downside node if the corresponding element is in the set denoted by the upside node. In addition, the weights on edges are uniformly 1. Given the bound B, one may verify that there is a solution for the maximum converage problem if and only if there is a set S of k critical alerts with $\mathsf{Gain}(S) \geq B$. Therefore, CAM is at least as hard as maximum coverage problem, and is NP-hard. Hence, CAM is NP-complete. □

2.3 Mining framework

In this section, we present a framework for critical alert mining. It consists of three components as illustrated in Figure 2.1: (1) offline dependency rule mining; (2) online alert graph maintenance; and (3) on-demand critical alert mining.

Offline dependency rule mining. Given a set of observed alert sequences, the system mines the alerts of interest and their causal relations offline, and represent them as a set of dependency rules. As there are a variety of methods to model a causal relation, in this work we adopt Granger causality [26, 158], which can naturally be represented by dependency rules. An alert sequence X is said to Granger-cause another sequence Y if it can be shown, via certain statistic tests on lagged values of X and Y, that the values of X provide statistically significant information to predicate the future values of Y. More specifically,

(1) We collect alert sequences for all performance metrics of interest as training data, following two criteria as follows: (a) the alerts in training data should be the latest ones such that the latest dependency patterns among performance metrics can be captured; and (b) the alert information should be rich enough such that learned dependency rules would be more robust. In our work, we treat the latest one week alert data as the training data.

(2) We apply existing Granger causality analysis tools [158] to mine the dependency rules, and apply conditional probabilities to estimate the uncertainty of the rules [106].

The learned dependency rules are stored in knowledge bases to support online alert graph maintenance. Moreover, existing knowledge bases such as event causality scenarios [59], or vulnerabilities exploitation among cyber assets [43] can also be "plugged" into our critical causal mining framework. The dependency rules are then shipped to the next stage in the system to maintain alert graphs.

Online alert graph maintenance. Using dependency rules, our system constructs and maintains an alert graph G online from a range of newly issued alerts. Upon an alert u from performance metric q is detected at time t, it first marks u as a new alert in G. It then checks (1) if there exists dependency rules in the form of $p \xrightarrow{l_{pq}} q$, and (2) whether there are alerts detected on performance metric p during the time period $[t - l_{pq}, t)$. If there exists such an alert v on p, an directed edge from v to u is inserted, and the rule uncertainty $\Pr(p \xrightarrow{l_{pq}} q)$ is associated to the edge (v, u). Following the above steps, it maintains G online for newly detected alerts.

On-demand critical alert mining. The major task (and the focus of this work) in the pipeline is to identify k critical alerts from alert graphs. In practice, a user may specify a time window of interest, which induces an alert graph from the maintained alert graph. It contains all the alerts detected during the time window. However, the induced alert graphs can still be huge.

In this paper, we propose three algorithms to address the scalability issue: (1) a quadratic time approximation with performance guarantees on the quality of critical alerts, (2) a linear time approximation, which guarantees the alert quality for tree-structured alert graphs; and (3) sampling-based heuristics which can be *tuned* to balance the alert quality and response time. The critical alerts are then returned to users for further analysis and verification.

2.4 Bound and pruning algorithm

Theorem 1 tells us that it is unlikely to find a polynomial time algorithm to find the best k alerts with the maximum gain. All is not lost: we can find polynomial time algorithms that approximately identify the most critical alerts. The main result in this section is as follows.

Theorem 2. *Given an alert graph $G = (V, E, f_e)$ and an integer k, (1) there exists an algorithm in $O(k|V||E|)$ time with approximation ratio $1 - \frac{1}{e}$, where e is the base of natural logarithm, and (2) there exists a $1 - \frac{1}{e}$ approximation algorithm in $O(k|V|)$ time, when G is a tree.*

Proof. We focus on showing the function $\mathsf{Gain}(\cdot)$ has diminishing return as follows.

(1) One could verify when $u \in S_1$, $u \in S_2$, or $u = v$, $P_f(S_1 \cup \{v\}, u) - P_f(S_1, u) \geq P_f(S_2 \cup \{v\}, u) - P_f(S_2, u)$;

(2) For $u \notin S_2 \cup \{v\}$, we prove the diminishing return by mathematical reduction.

(a) Assume that all u's parents u' satisfy $P_f(S_1 \cup \{v\}, u') - P_f(S_1, u') \geq P_f(S_2 \cup \{v\}, u') - P_f(S_2, u')$.

(b) When u has only one parent, $P_f(S \cup \{v\}, u) - P_f(S, u) = f_e(u', u)(P_f(S \cup \{v\}, u) - P_f(S, u))$, and it is easy to see the diminish return for u.

(c) Assume that when u has m parents, we have $a - b \geq c - d$, where $a = \prod_{u' \in N_p(u)}(1 - f_e(u', u)P_f(S_1, u'))$, $b = \prod_{u' \in N_p(u)}(1 - f_e(u', u)P_f(S_1 \cup \{v\}, u'))$, $c = \prod_{u' \in N_p(u)}(1 - f_e(u', u)P_f(S_2, u'))$, and $d = \prod_{u' \in N_p(u)}(1 - f_e(u', u)P_f(S_2 \cup \{v\}, u'))$. Note that $b \geq d$. Consider the case when u has $m + 1$ parents, and *w.l.o.g.*, u'' is the $m + 1$-th parent satisfying $x_3 - x_1 \geq x_4 - x_2$, where $x_3 = P_f(S_1\{v\}, u'')$, $x_1 = P_f(S_1, u'')$, $x_4 = P_f(S_2 \cup \{v\}, u'')$, and $x_2 = P_f(S_2, u'')$, where $x_1 \leq x_2$. Therefore, we have $a(1 - x_1) - b(1 - x_3) = a - b - ax_1 + bx_3 = (1 - x_1)(a - b) + (x_3 - x_1)b$, and $c(1 - x_2) - d(1 - x_4) = c - d - cx_2 + dx_4 = (1 - x_2)(c - d) + (x_4 - x_2)d$. Thus, we obtain $a(1 - x_1) - b(1 - x_3) \geq c(1 - x_2) - d(1 - x_4)$, which is $P_f(S \cup \{v\}, u) - P_f(S, u) = f_e(u', u)(P_f(S \cup \{v\}, u) - P_f(S, u))$.

In all the cases, when $u \notin S_2 \cup \{v\}$, its diminishing return holds. Hence, $\mathsf{Gain}(\cdot)$ is a submodular function. It is known that for maximizing a submodular function, a greedy strategy achieves $1 - \frac{1}{e}$ approximation ratio [139]. Theorem 2 hence follows. $\square$

Here e refers to Euler's number (approximately 2.71828). Denote the optimal k alerts as S^*, we present an efficient algorithm to identify k alerts S where $\mathsf{Gain}(S) \geq (1 - \frac{1}{e})\mathsf{Gain}(S^*)$, in quadratic time.

We start with a greedy algorithm, denoted as Naive.

Naive greedy algorithm. Given an alert graph $G = (V, E, f_e)$ and an integer k, Naive finds k critical alerts in k iterations as follows. (1) It initializes a set S_0 to store the selected alerts. (2) At the ith iteration, Naive checks each alert in V, and greedily picks the alert s_i that maximizes the *incremental gain* $\mathsf{Gain}(S_{i-1} \cup \{s_i\})$, where S_{i-1} is the set of critical alerts found at iteration $i - 1$. (3) It repeats the above step until k alerts are identified.

One may verify that Naive is a $1 - \frac{1}{e}$ approximation algorithm. To see this, observe that the set function $\mathsf{Gain}(\cdot)$ is a *monotonically submodular function.* A function $f(S)$ over a set S is called *submodular* if for any subset $S_1 \subseteq S_2 \subset S$ and $x \in S \setminus S_2$, $f(S_1 \cup \{x\})$ - $f(S_1) \geq f(S_2 \cup \{x\})$ - $f(S_2)$. It is known that for maximizing a submodular function, a greedy strategy achieves $1-\frac{1}{e}$ approximation ratio [139]. Hence it suffices to show that the function Gain is a monotonically submodular function. Indeed, (1) one may verify that Gain is monotonic: for any $S_1 \subseteq S_2 \subseteq V$, $\mathsf{Gain}(S_1) \leq \mathsf{Gain}(S_2)$; (2) the diminishing return of Gain can be shown by mathematical reduction.

For complexity, Naive requires k iterations, and in each iteration, it scans all the vertices u and computes $P_f(S_{k-1}, u)$, which takes in total $O(k|V||E|)$ time.

Naive provides a polynomial time algorithm to approximate CAM within $1-\frac{1}{e}$. Nevertheless, the scalability issue of Naive makes it difficult to use in practice for large alert graphs. For instance, when an alert graph of around $20K$ vertices and

$200K$ edges, Naive mines 6 critical alerts in more than 800 seconds. We next present a faster approximation algorithm with the same approximation ratio. By using pruning and verification, the algorithm is 30 times faster than Naive, as verified in our experimental study.

2.4.1 Pruning and verification

To select a most promising alert at each iteration, Naive evaluates the incremental gain for each alert in $V \setminus S$, and then selects the one of the highest incremental gain, which runs in $O(|V||E|)$ time. Instead of blindly processing every alert, we may efficiently filter "unpromising" alerts, and then evaluate the exact gain for the remaining vertices. In particular, at each iteration i, for two alerts v' and $v \in V \setminus S_{i-1}$, we compute upper bounds U'_v, U_v and lower bounds L'_v, L_v for $\mathsf{Gain}(S_{i-1} \cup \{v\})$ and $\mathsf{Gain}(S_{i-1} \cup \{v'\})$, respectively. If v' is already not a critical alert, all the alerts v with $L'_v > U_v$ can be safely skipped without losing the alert quality.

We next derive an upper and lower bound for $\mathsf{Gain}(\cdot)$, and present algorithms to compute them efficiently. Instead of visiting each alert and causal relation in G, these algorithms compute the bounds by visiting only local information of each alert in G. This enables a fast estimation of $\mathsf{Gain}(\cdot)$.

2.4.2 Upper bound

We introduce a notion of *sum gain* (denoted as SGain) to characterize the upper bound for $\mathsf{Gain}(\cdot)$. Given an alert graph $G = (V, E, f_e)$, an alert $v \in V$, and a set of selected critical alerts $S \subseteq V$, an upper bound is computed as $\mathsf{SGain}(S \cup \{v\}) = \sum_{u \in V} w_u \cdot \hat{P}_f(S \cup \{v\}, u)$, where

- $\hat{P}_f(S \cup \{v\}, u) = 1$, if $u \in S$;
- $\hat{P}_f(S \cup \{v\}, u) = \sum_{(u',u) \in E} \hat{P}_f(S \cup \{v\}, u') f_e(u', u)$, if $u \notin S$.

The sum gain SGain (as illustrated in Figure 2.2) is an upper bound for $\mathsf{Gain}(\cdot)$. Better still, it can be efficiently computed.

Proposition 1. *Given an alert graph $G = (V, E, f_e)$, a set of critical alert $S \subseteq V$, and an alert $u \in V \setminus S$, (1) $\mathsf{Gain}(S \cup \{u\}) \leq \mathsf{SGain}(S \cup \{u\})$; and (2) SGain can be computed for* all *alerts in V in $O(|E|)$ time.*

We first prove Proposition 1 (1). We remark that SGain is built upon the following generalization of Bernoulli's inequality [129]. Given $x_i \leq 1$, we have

$$1 - \prod_{i=1}^{n} (1 - x_i) \leq \sum_{i=1}^{n} x_i.$$

We next conduct a mathematical induction over the topological order (Section 2.2) of the alerts in G as follows.

- Consider the alerts $u_1 \in V$ with topological order 0: (1) if $u_1 \in S$, $\hat{P}_f(S, u_1) = 1$ and $P_f(S, u_1) = 1$; (2) otherwise, $\hat{P}_f(S, u_1) = 0$ and $P_f(S, u_1) = 0$, since u_1 has no parents. In both cases, $P_f(S, u_1) \leq \hat{P}_f(S, u_1)$.

- Assume that alert $u_i \in V$ with topological order i satisfies $P_f(S, u_i) \leq \hat{P}_f(S, u_i)$. For an alert $u_{i+1} \in V$,

$$\begin{aligned} P_f(S, u_{i+1}) &= 1 - \prod_{(u', u_{i+1}) \in E} \big(1 - P_f(S, u') f_e(u', u_{i+1})\big) \\ &\leq \sum_{(u', u_{i+1}) \in E} P_f(S, u') f_e(u', u_{i+1}) \\ &\leq \sum_{(u', u_{i+1}) \in E} \hat{P}_f(S, u') f_e(u', u_{i+1}) \\ &= \hat{P}_f(S, u_{i+1}) \end{aligned}$$

Therefore, for any $u \in V$, $P_f(S, u) \leq \hat{P}_f(S, u)$. By definition, $\mathsf{Gain}(S \cup \{u\}) \leq \mathsf{SGain}(S \cup \{u\})$. Hence, SGain is indeed an upper bound for $\mathsf{Gain}(\cdot)$.

Upper bound computation. As a constructive proof for Proposition 1 (2), we present a procedure (denoted as computeUpperBound) for SGain to compute the upper bounds for all vertices in $O(|E|)$ time.

The algorithm (not shown) follows a "bottom up" computation, starting from the alerts with the highest topological order in G. (1) It first computes the topological order for all the alerts in G. (2) Starting from the alert with the highest topological order, it computes SGain for each alert $u \in V \setminus S$ as follows: (a) $\mathsf{SGain}(S \cup \{u\}) = \mathsf{SGain}(S \cup \{u\}) + w_u$, and (b) for each $u' \in N_i(u)$, it updates $\mathsf{SGain}(S \cup \{u'\})$ by $\mathsf{SGain}(S \cup \{u'\}) + f_e(u', u)\mathsf{SGain}(S \cup \{u\})$. (3) It repeats step (2) until all the alerts are processed.

It takes $O(|E|)$ time for computeUpperBound to obtain the topological order by depth-first search in step (1). Each edge in G is visited exactly once in step (2) and (3). Therefore, the algorithm runs in $O(|E|)$ time.

The above analysis completes the proof of Proposition 1.

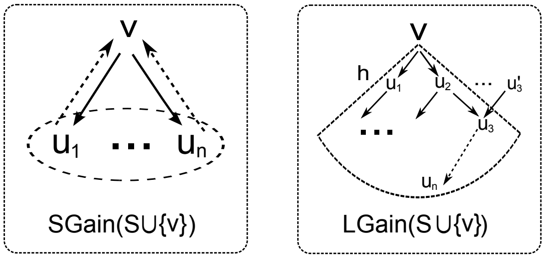


Figure 2.2: Algorithm BnP: Upper and lower bound

2.4.3 Lower bound

To compute the lower bound of $\mathsf{Gain}(\cdot)$, we introduce a notion *local gain* (denoted as LGain). Given an alert graph $G = (V, E)$, an alert $v \in V$, a set of selected alerts $S \subseteq V$, and an integer h, LGain of $S \cup \{v\}$ is defined as follows.

$$\mathsf{LGain}(S \cup \{v\}) = \sum_{u \in V_v^h} w_u \cdot P_f(S \cup \{v\}, u),$$

where h is a tunable integer, and $V_v^h \subseteq V$ is a set of vertices that can be reached from v in no more than h hops. Intuitively, LGain estimates a lower bound of $\mathsf{Gain}(S)$ with the impact of an alert to its local "nearby" alerts in G (as illustrated in Figure 2.2). One may verify the following.

Proposition 2. *Given $G = (V, E)$, $S \subseteq V$, for any alert $u \in V \setminus S$, (1) $\mathsf{Gain}(S \cup \{v\}) \geq \mathsf{LGain}(S \cup \{v\})$, and (2) LGain can be computed in $O(\sum_{v \in V} |E_v^h|)$ time, where E_v^h is the set of incoming edges in G of the alerts in V_v^h.*

We present a procedure computeLowerBound to compute LGain. For each alert $v \in V \setminus S$ (*e.g.,* u_3 in Figure 2.2), the algorithm visits the alerts in V_v^h and their incoming edges (*e.g.,* (u_3', u_3)) once, and computes LGain following the definition, in $O(\sum_{v \in V} |E_v^h|)$ time.

2.4.4 Algorithm BnP

Based on the upper and lower bounds, we propose an approximation, denoted as BnP. BnP enables faster critical alert mining while achieving the approximation ratio $1 - \frac{1}{e}$. The algorithm follows Naive's greedy strategy: given an integer k, it conducts k iterations of search, each determines a top critical alert. The difference is that in each iteration, it invokes a procedure Prune to identify a set C of candidate alerts for consideration.

The procedure Prune (as illustrated in Figure 2.3) invokes computeUpperBounds and computeLowerBounds to dynamically update the lower and upper bounds for each alert by accessing their local information (lines 1-2), and filters the alerts that are not critical:

1. it scans the lower bounds LGain of each alert, and find the maximum one as *bar* (line 3);

2. it scans the upper bounds SGain of each alert, and prunes those with $\mathsf{SGain}(u) < bar$, adding the rest to a candidate alert set C.

Correctness and Complexity. The algorithm BnP achieves approximation ratio $1-\frac{1}{e}$, as it follows the same greedy strategy as Naive. Note that the pruning procedure Prune does not affect the approximation ratio.

For complexity, let C_m be the maximum set of candidate sets in all the iterations after pruning. For the alerts in C_m, it takes BnP $O(|C_m||E|)$ time to find a best alert. The total time for pruning is $O(k(\sum_{u \in V} |E_u^h| + |E|))$. Hence, it takes BnP in total $O(k(\sum_{u \in V} |E_u^h| + |C_m||E|))$ time. Moreover, $|E_u^h|$ is typically small, and is *tunable* by varying h, as indicated by Proposition 2. For example, when

```
Input:   An alert graph $G = (V, E, f_e)$;
a set of critical alert $S$.
Output: a set of candidate alerts $C$.

computeUpperBound ($G$, $S$);
computeLowerBound ($G$, $S$);
set bar as the largest LGain over alerts in $V \setminus S$;
$C \leftarrow \emptyset$;
for each  $u \in V \setminus S$
    if  SGain($u$) $\geq$ bar
        $C \leftarrow C \cup \{u\}$;
return $C$;
```

Figure 2.3: The pruning procedure Prune

$h = 1$, LGain can be computed in $O(d_m|E|)$ for all the alerts, where d_m is the largest in-degree in G. As h gets larger, the computation complexity gets higher, leading to tighter lower bound LGain. In our experimental study, by setting $h = 3$, 95% of the alerts are pruned, which makes BnP 30 times faster than Naive without losing alert quality.

Mining Alert Trees. When G is a directed tree, the algorithm BnP identifies k critical alerts in $O(k|V|)$ as follows. (1) Starting from the alerts $u \in V$ of the highest topological order, it computes $\text{Gain}(u) = \text{Gain}(u) + w_u$, and makes an update by $\text{Gain}(u') = \text{Gain}(u') + f_e(u', u)\text{Gain}(u)$, if u' is the parent of u. (2) It repeats (1) on the alerts following the decreasing topological order, until all the alerts are processed. One iteration over (1) and (2) identifies a critical alert. (3) BnP repeats (1) and (2) to find k critical alerts.

Following the correctness analysis, BnP preserves the approximation ratio $1-\frac{1}{e}$ over trees. Moreover, each edge in G is visited once in a single iteration. Hence, it takes $O(k|V|)$ time of BnP over G as trees. Theorem 2 (2) hence follows.

2.5 Tree approximation

Algorithm BnP needs to process all the candidates and their causal relations, which may not be efficient for a large amount of alert sequences. In extreme cases where few alerts are pruned, BnP degrades to its naive greedy counterpart.

As indicated by Theorem 2(2), fast approximation exists for alert graphs as trees. Following this intuition, we may make large alert graphs "small", by sparsifying them into directed trees, which "preserve" most of alert dependency information in an alert graph. This enables both fast algorithms and low quality loss.

2.5.1 Single-tree approximation

We start by introducing a heuristic algorithm ST. The basic idea is to induce a *maximum directed tree* (forest) T from a given alert graph G, such that for any set of alerts S in G, $\mathsf{Gain}(S)$ in T is "close" as much as possible to $\mathsf{Gain}(S)$ in G, and a fast approximation can be performed over T without much quality loss.

Maximum directed tree. Given an alert graph $G = (V, E, f_e)$, a maximum directed tree of G is a spanning tree $T = (V, E')$, where $E' \subset E$, such that (1) for any $u \in V$, u has at most one incoming edge, and (2) $\sum_{\langle u,v\rangle \in E'} f_e(u,v)$ is maximized. Intuitively, T depicts a "skeleton" of an alert graph G, where causal relations always follow the most likely dependency rules.

Algorithm ST. Given an alert graph $G = (V, E, f_e)$, the single-tree approximation ST mines k critical alerts as follows. (1) ST first finds the maximum directed tree T. To construct T, an algorithm simply selects, for each alert u in G, the incoming edge (u', u) with the maximum $f_e(u', u)$ among all its incoming edges. (2) ST searches the k critical alerts following the algorithm BnP over T.

One may verify that it is in $O(|E|)$ time to construct T. From Theorem 2(2), it is in $O(k|V|)$ time to find k critical alerts in T (as either a tree or a forest). Hence, the algorithm ST takes in total $O(|E| + k|V|)$ time. Note that the induced T can be a set of disjoint trees, where the above complexity still holds.

2.5.2 Multi-tree sampling

Single-tree approximation provides fast mining method for large scale alerts. On the other hand, using induced trees to approximate causal structures may lead to biased results. For example, more dependency information could be lost for alerts with more incoming edges. To rectify this, we propose a heuristic, denoted as MTS, based on multi-tree sampling.

Algorithm MTS. The algorithm MTS is as illustrated in Figure 2.4. Given an alert graph $G = (V, E, f_e)$, integer k and a sample number N, MTS starts by initializing a set S_0 as $\emptyset$, the alert-fixed probability for each node as 0 (line 1), and identify the topological orders of the alerts in G.

Algorithm MTS then finds a set S critical alerts in k iterations as follows. Denote the selected critical set at iteration $i-1$ as S_{i-1}. At each iteration i, (1) MTS updates the alert-fixed probability $P_f^{(i-1)}(S_{i-1}, u)$ for each alert $u \in V$ in G, fixing S_{i-1} as the critical alerts (lines 3-4). (2) It then invokes procedure sampleTree to sample N trees from G (lines 6-7), according to the updated alert-fixed probability in (1). (3) For each alert u, MTS computes the weighted sum of u's descendants $D(u, l)$ in each sampled tree $T_{u,l}$, and takes the average $D(u, l)$ over all sampled tress as an estimation of $\mathsf{Gain}(S_{i-1} \cup \{u\})$ (lines 8-9). It selects the alert u that introduces the maximum improvement, and update S_{i-1} as S_i by adding u (lines 10-11), which is used to update $P_f(\cdot)$ in G in the next iteration.

Procedure sampleTree. Given an alert graph G and an integer N, the procedure sampleTree (line 7) samples N trees (forest) from G at iteration i. More specifically, it generates a single tree (or forest) $T_{i,l}$ as follows. (1) It first samples a set of alerts $V_{i,l}$ as the nodes for tree $T_{i,l}$ following Bernoulli distributions. For each alert $u \in V$ and the updated $P_f^{(i-1)}(u)$, MTS selects u with probability $1 - P_f^{(i-1)}(u)$, and inserts it to $V_{i,l}$. (2) MTS then samples an edge for each alert $u \in V_{i,l}$. It randomly orders u's parents. Starting from the first parent, u tries to build an edge (u', u) to its parent with probability $f_e(u', u)$, where u' ranges over all the parents of u, until an edge is selected (and attached to u), or all the parents are visited. (3) MTS repeats (2) until all the alerts $u \in V_{i,l}$ are visited.

It takes in total $O(k * N|E|)$ time for MTS to find k critical alerts. (a) MTS takes in total $O(k|E|)$ time to update P_f in G; (b) the total sampling time is in $O(k * N|E|)$; and (c) it takes in total $O(k * N|V|)$ time to select the critical alerts.

```
Input:  Alert graph $G = (V, E, f_e)$,
        integer $k$, the number of sampled trees $N$.
Output: A set $S$ of $k$ critical alerts.

$S \leftarrow \emptyset$; initializes $P_f(\cdot)$; $i \leftarrow 0$;
while $i \leq k$ Do
    for each alert $u$ in $G$ Do
        update $P_f^{(i-1)}(S_{i-1}, u)$;
    $l \leftarrow 0$;
    while $l \leq N$ Do
        $T_{i,l} \leftarrow \mathsf{sampleTree}(G)$;
    for each alert $u$ in $G$ Do
        $\mathsf{Gain}(S_{i-1} \cup \{u\}) \leftarrow \mathsf{Gain}(S_{i-1}) + \frac{\sum_{l=1}^{N} D(u,l)}{N}$;
    select $u$ with the maximum $\mathsf{Gain}(S_{i-1} \cup \{u\})$;
    $S_i \leftarrow S_{i-1} \cup \{u\}$;
return $S_k$;
```

Figure 2.4: Algorithm MTS

In contrast to its single-tree counterpart, MTS leverages sampling to reduce the bias: alerts with more parents and larger probability are more likely to have a parent in a sampled tree. In addition, it synthesizes the gain estimation from multiple trees, such that the noise from a single tree is smoothed. Indeed, we found that using only 300 samples, MTS finds top 6 critical alerts with $\mathsf{Gain}(\cdot)$

90% as good as Naive, and is 80 times faster. It reduces 10% more loss on Gain(·) compared with ST (see Section 2.6).

2.6 Experiment

We applied both real-life and synthetic data to evaluate our algorithms. We first provide a case study (Section 2.6.2). Using real-life data, we next investigate (1) the efficiency and effectiveness of our algorithms (Section 2.6.3), (2) the impact of the number of explored hops to the performance of BnP (Section 2.6.4), and (3) how the number of samples affects MTS (Section 2.6.5). In addition, we evaluate the scalability of our algorithms, over large synthetic data (Section 2.6.6).

2.6.1 Setup

Real-life data. We use real-life datacenter performance data (referred to as LM), from LogicMonitor, an SaaS network monitoring company. The data spans 53 days from Nov. 23, 2013 to Jan. 14, 2014. It contains the sequences for 50,772 performance metrics from 9,956 services residing in 122 servers. Each metric is reported every 2 minutes. The alerts are identified by specified rules provided by LogicMonitor, where we assign a weight 1.0 to all the metrics.

Dependency rules and alert graphs. Dependency rules were mined from data collected in 7 consecutive days, and are used to construct alert graphs using the data from the following days. We used the tool developed by [158] to mine the Granger causality among performance metrics as dependency rules (with the p-value set to be 0.01 [158]). We then applied conditional probability to estimate the uncertainty of the rules [106]. From the dataset LM, we mined 46 sets of dependency rules, where each set contains on average $2,082$ rules. Each set of rules were mined in less than 60 minutes.

By applying the sets of dependency rules on the alert detected in the next single day, we obtained 46 alert graphs, following the online alert graph construction (Section 2.3). The number of alerts (resp. edges) ranges from $20,248$ to $25,057$ (resp. $162,000$ to $270,370$) for a single graph.

Synthetic alert graphs. For scalability tests over large alert graphs, we applied the graph model proposed in [100] to generate large synthetic alert graphs (referred to as SYN). In particular, the node degree and edge weights follow the empirical distributions [179] learned from alert graphs over the real-life data LM. We ranged the number of alerts from 100K to 1M, and the average degree of SYN graphs is 9.

Evaluation. To measure the quality of the critical alerts identified by an algorithm A, we investigate a metric *loss ratio* of A defined as

$$\textbf{loss ratio}(A) = 1 - \frac{Gain(S_A)}{Gain(S_{\mathsf{Naive}})},$$

where S_{Naive} (resp. S_A) is the set of critical vertices returned by the algorithm Naive (resp. algorithm A). As Naive guarantees the alert quality within a bound, loss ratio suggests how “close” the quality of the alerts from heuristic algorithms and the optimal ones is. The less, the better.

Implementation. In addition to the proposed algorithms BnP, ST, and MTS, we implemented the following baseline algorithms: (1) Naive, the greedy algorithms without pruning strategy; (2) $\mathsf{BnP}_{\mathsf{UB}}$, a simplified version of BnP, which only uses upper bound to filter unpromising alerts: it skips those alerts with upper bound smaller than an alert with computed Gain(·) in each iteration (Section 2.4). (3) MaxDeg, a simple strategy that returns the top k alerts with the largest weighted sum of outgoing edges.

All the algorithms were implemented in C++, and all experiments were executed on a machine powered by an Intel Core i7-2620M 2.7GHz CPU and 8GB of RAM, using Ubuntu 12.10 with GCC 4.7.2. Each experiment was run 10 times, and their average results are presented.

2.6.2 Case study

Using real-world data LM, our algorithms suggest reasonable critical alerts that are indeed the source of a range of large amount of alerts, as verified by the domain experts from LogicMonitor. We illustrate three "causality patterns" induced by top two critical alerts and their descendants following the weighted dependency rules in Figure 2.5. (1) Our algorithms suggest that StorageUsed, a critical alert that indicates insufficient memory, leads to poor performance of Web servers (Apache), which typically triggers delayed Ping round-trip time (Ping-avgrtt) from other servers. In another set of hosts, it leads to insufficient shared memory over a range of servers, which typically triggers slower Shared Data Access write time (SDA writetime) on their own. (2) A second critical alert DiskReadLatency suggests I/O bottleneck for a range of abnormal status of database applications. The disk access speed alert often triggers the unsolved back up requests from another server, which leads to poor performance of CPU and database servers, and further affects a range of database related requests from more outside servers.

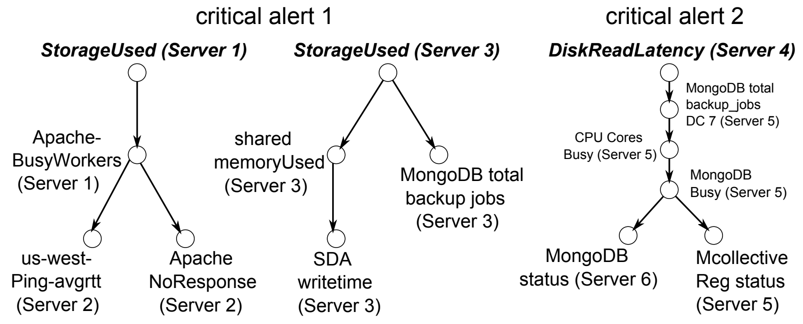

critical alert 1
critical alert 2
StorageUsed (Server 1)
StorageUsed (Server 3)
DiskReadLatency (Server 4)
Apache-BusyWorkers (Server 1)
shared memoryUsed (Server 3)
MongoDB total backup jobs (Server 3)
MongoDB total backup_jobs DC 7 (Server 5)
CPU Cores Busy (Server 5)
MongoDB Busy (Server 5)
us-west-Ping-avgrtt (Server 2)
Apache NoResponse (Server 2)
SDA writetime (Server 3)
MongoDB status (Server 6)
Mcollective Reg status (Server 5)

Figure 2.5: Critical alerts over LM

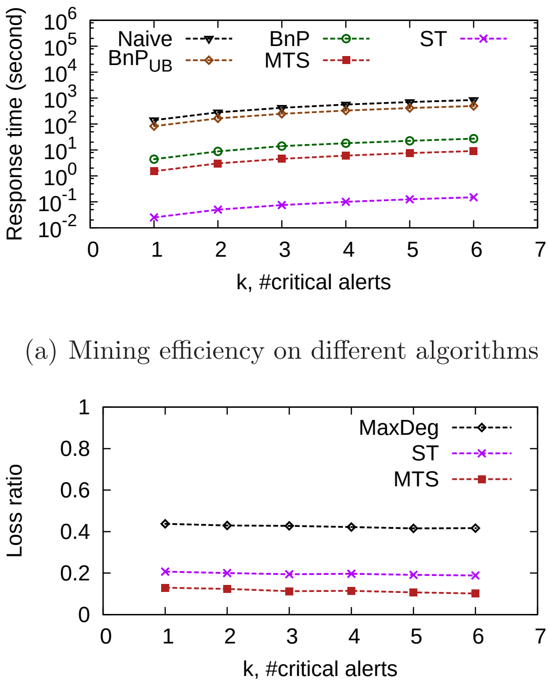
These causal patterns are consistent with the workflow of datacenters at LogicMonitor.

Our algorithms do not assume prior domain knowledge. On the other hand, external knowledge and rules enable our algorithms to further improve the quality of the critical alerts and causal patterns.

2.6.3 Overall performance evaluation

We first investigate the efficiency and effectiveness of the proposed algorithms, using alert graphs from LM. In the following tests, we fixed the number of explored hops in BnP as 3, and the number of sampled trees in MTS as 300.

As illustrated in Figure 2.6(a), the proposed algorithms BnP, MTS, and ST consistently outperform the baseline algorithms Naive and BnP$_{\text{UB}}$ in efficiency, while varying k, the number of required critical alerts. They introduce different levels of efficiency improvement. Compared with Naive and BnP$_{\text{UB}}$, BnP is 30 times and 17 times faster, respectively, without quality loss on solutions. With some quality loss, ST is 5000 times and 3000 times faster than Naive and BnP$_{\text{UB}}$,



(a) Mining efficiency on different algorithms

(b) Loss ratio comparison

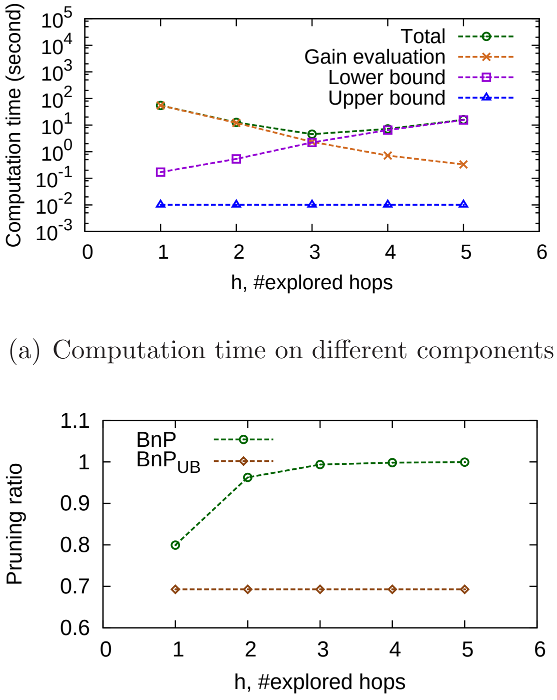
Figure 2.6: Mining performance on LM alert graphs

respectively, and MTS results in 80 times and 50 times speedup. In addition, all the algorithms take more time when k varies from 1 to 6, as expected.

Figure 2.6(b) shows the loss ratio of ST and MTS, where k varies from 1 to 6. Compared with MaxDeg, MTS and ST obtain significant improvement on loss ratio. As k increases, the loss ratio of MaxDeg is consistently more than 0.4; meanwhile, the loss ratio of MTS and ST is around 0.1 and 0.2, respectively. Compared with MTS, ST receives higher efficiency at the cost of solution quality loss. When the number of required critical alerts varies from 1 to 6, MTS and ST share the same trend: the loss ratio decreases. Compared with BnP that returns critical alerts without quality loss, ST and MTS are 180 and 3 times faster, respectively, at the cost of small quality loss.

In all cases, we observe that the total Gain(·) increases with larger k with diminish return (not shown). This is consistent with its submodularity.

2.6.4 Performance evaluation of BnP



(a) Computation time on different components

(b) Pruning ratio comparison

Figure 2.7: BnP performance on LM alert graphs

In this set of experiments, we focus on the impact of the number of hops h (for lower bound computation) to the performance of BnP. We fixed k as 1. Besides running time, we investigate the *pruning ratio* of BnP, defined as $\frac{|V|-|C|}{|V|}$, where

$|V|$ is the total alert number in an alert graph G, and $|C|$ is the average size of the candidate set C (Section 2.4) after pruning, for all the k iterations.

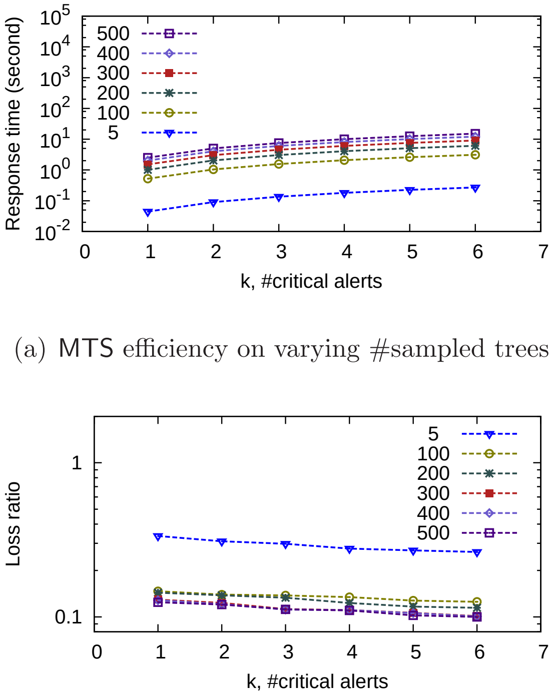
Figure 2.7(a) and Figure 2.7(b) illustrate how the computation time of different components in BnP varies, and how the pruning ratio varies, respectively, while the number of explored hops h varies from 1 to 5. The result tells us the following. (1) When h increases from 1 to 3, the response time of BnP drops. Indeed, as observed from Figure 2.7(b), the efficiency improvement comes from the increasing number of pruned alerts. With more alerts pruned, the amount of time taken on Gain evaluation, which is the dominating cost, drops accordingly. (2) When the number of hops increases from 3 to 5, the response time of BnP increases. As the number of hops grows from 3 to 5, we can see that the pruning ratio of BnP marginally is improved from Figure 2.7(b); however, the amount of computation time for lower bound in BnP dramatically increases, which becomes the dominating computation cost. According to our result, when the number of explored hops is set to be 3, BnP achieves the best performance on LM alert graphs.

In addition, as shown in Figure 2.7(b), BnP consistently outperforms $\mathsf{BnP}_{\mathsf{UB}}$ in terms of pruning ratio, since the upper and lower bounds in BnP introduce more powerful pruning to reduce unnecessary computation.

2.6.5 Performance evaluation of MTS

In this set of experiments, we demonstrate how the number of sampled trees affect the performance of MTS.

Figure 2.8(a) tells us the following. (1) While the number of required critical alerts is fixed, the response time of MTS is proportional to the number of sampled



(a) MTS efficiency on varying #sampled trees

(b) MTS effectiveness on varying #sampled trees

Figure 2.8: MTS performance on LM alert graphs

trees (varies from 5 to 500). (2) When the number of samples is fixed, the response time of MTS grows linear to the number of required critical alerts. In all cases, MTS takes no more than 15 seconds.

Figure 2.8(b) illustrates how the number of sampled trees influences the effectiveness of MTS. When the number of sampled trees increases, the loss ratio of MTS decreases, while the reduction of loss ratio diminishes. As the number of sampled trees changes from 5 to 100, the loss ratio of MTS is significantly improved; meanwhile, as the number of sampled trees changes from 100 to 500, the loss ratio is marginally improved. In addition, fixing the number of sampled trees, when the number of required critical alerts is increased, the loss ratio of all MTS variants decreases.

2.6.6 Scalability

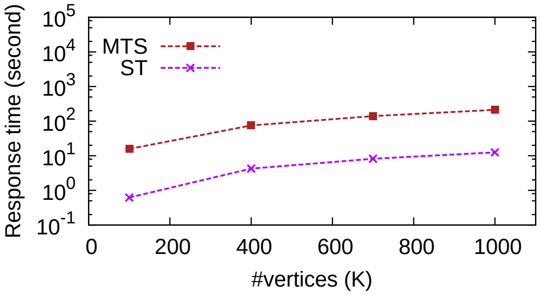


Figure 2.9: Scalability results on SYN graphs

On SYN alert graphs, we fixed the number of required critical alerts to be 3, and evaluate the scalability of BnP, MTS, ST, Naive, and $\mathsf{BnP}_{\mathsf{UB}}$. Note that the number of hops explored in BnP is fixed to be 3, and the number of sampled trees in MTS is fixed to be 300.

Figure 2.9 reports the scalability results. When the number of alerts in SYN graphs increases from 100K to 1000K, the response time of MTS and ST linearly grows. In particular, when a SYN graph has 1M alerts and more than 90M edges, MTS and ST return 3 critical alerts in 4 minutes and 13 seconds, respectively. On the other hand, Naive, $\mathsf{BnP}_{\mathsf{UB}}$, and BnP cannot finish the computation in an hour, even for alert graphs with $100K$ alerts (hence are not shown). Indeed, the efficiency of BnP relies on the amount of alerts it can prune. In the worst case, it works as slow as Naive. In contrast, MTS and ST are much less sensitive to the growth of graph size, and are more promising for large alert graphs.

2.6.7 Summary

We found the following. (1) With pruning strategy, BnP outperforms baseline algorithms in terms of efficiency up to 30 times, without loss of solution quality. (2) While MTS is up to 80 times faster than baseline algorithms, the resulting loss ratio is around 0.1. (3) ST is up to 5000 times faster than baseline algorithms, with loss ratio around 0.2.

2.7 Related work

Causality models and analysis. Causal relations among time series data have been modeled with Granger causality [175], lagged correlation [124], Bayesian networks [142, 136], among others. Granger causality measures a cause in terms of whether it passes Granger Test, *i.e.,* whether it helps in predicating the future events, beyond what can be predicted by using only the historical events. Lagged correlation characterizes causal relations with the correlation between two time series shifted in time relative to one another. Causal Bayesian networks interprets causal relations with graphical models, in which the predecessors of a node are interpreted as directly causing the variable associated with that node.

A variety of causality mining techniques have been studied [26, 158, 164], varied with causality models. Silverstein et al. [164] proposed algorithms to mine causal relations in large databases by estimating the conditional probability of rules of interest. For Granger causality, Arnold et al. [26] applied Lasso Granger method to find a set of events that are conditionally dependent with regression, without exhaustively performing pairwise Granger Test. A toolbox for detecting Granger causality is developed [158]. These methods stop at identifying causal relations. Our work, on the contrary, efficiently identifies the most critical alerts rather than suggesting all possible causal relationships. On the other hand, efficient causality mining techniques, as well as existing knowledge bases on event causality scenarios [59] serve as preprocessing in our critical causal mining framework.

Root cause analysis. We are aware of a range of domain-specific studies that aims to find the "root causes". Given a set of observed symptom events, the problem is to identify the set of root causes that can best explain the symptom. In intrusion detection, Julisch [98] leveraged alert clustering techniques to indicate root causes for system alarms. A hierarchical clustering process is iteratively performed over groups of similar alarms, until the top causes are identified. In network performance diagnosis, Mahimkar et al. [124] proposed methods to identify potential root causes as the events that have statistically significant (lagged) correlations with a set of known symptom events. In contrast, we propose a general computational framework for efficient root cause analysis over large-scale alert sequences in networks. While we do not have the luxury to assume the access of rich domain-specific semantics that benefit event filtering, any such knowledge serves as preprocessing to reduce the input size of our problem.

Influence maximization. Node influence evaluation aims to select a group of nodes with maximized influence, under various information diffusion models, such as independent cascade model [113], linear threshold model [101], competing model [41, 90], continuous-time model [65, 155], and credit distribution model [81]. The problem is, however, highly intractable (#P-hard). Sampling methods such as Monte Carlo simulations are usually applied to estimate node influence. Nonetheless, these approaches typically take massive amount of computation time and are hard to scale over large graphs [120]. To improve the scalability, various pruning algorithms have been proposed to reduce the number of Monte Carlo simulations [56, 65, 82, 205], and heuristic algorithms have been studied to estimate node influence [49, 51, 52, 141]. In contrast to these works, we identify efficient algorithms for critical alert mining, with desirable performance guarantees on alert quality and efficiency. Striking a balance between mining quality and efficiency, these algorithms suggest scalable mining for large scale alert analysis.

2.8 Summary

In this chapter, we study the critical alert mining problem. Despite its intractability, we develop approximation algorithms with quality guarantees, as well as fast heuristics that preserve at least 80% of solution quality, and perform up to $5,000$ times faster than their approximation counterparts.

This work is a first step towards large-scale critical alerts mining. We are conducting experiments over various large real-life datasets and causality models. One topic is to extend our techniques for distributed network monitoring systems and datacenters. Another topic is to dynamically maintain the alert graphs and mined critical alerts. In addition, to further improve the alert quality, one wants to combine the mining framework with external semantics and knowledge bases, and to automatically interpret the critical alerts for various application domains.

Chapter 3

Link Prediction in Temporal Graphs

3.1 Introduction

In various real-life networks, users frequently exchange information and influence with each other. The information (*e.g.,* messages, articles, recommendation links) is typically created from a user and spreads via links among users, leaving a trace of its propagation. Such traces are typically represented as temporal graphs, namely, *information cascades*, where (a) each node in a cascade is associated with the time step at which it receives the information, and (b) an edge from a node to another indicates that a user propagates the information to and *influences* its neighbor [38, 78].

A comprehensive understanding and analysis of cascades benefit various emerging applications in social networks [44, 102], viral marketing [27, 64, 153], and recommendation networks [121]. In order to model the propagation of informa-

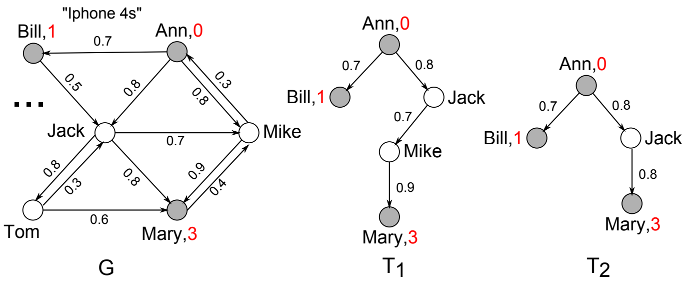


Figure 3.1: A cascade of an Ad (partially observed) in a social network G from user Ann, and its two possible tree representations T_1 and T_2.

tion, various *cascade* models have been developed [60, 167, 183]. Among the most widely used models is the *independent cascade model* [102], where each node has only one chance to influence its inactive neighbors, and each node is influenced by at most one of its neighbors independently. Nevertheless, it is typically difficult to observe the entire cascade in practice, due to the noisy graphs with missing data, or data privacy policies [110, 156]. It is important to develop techniques that can *infer* the cascades using partial information. Consider the following example.

Example 1. *The temporal graph G in Figure 3.1 depicts a fraction of a social network (*e.g.*, Twitter), where each node is a user, and each edge represents an information exchange. For example, edge $(Ann, Bill)$ with a weight* 0.7 *represents that a user Ann sends an advertisement (Ad) about a released product (*e.g.*, "Iphone 4s") with probability* 0.7*. To identify the impact of an Ad strategy, a company would like to know the complete cascade starting from their agent Ann. Due to data privacy policies, the observed information may be limited: (a) at time step* 0*, Ann posts an Ad about "Iphone 4s"; (b)* **at** *time step* 1*, Bill is **influenced** by Ann and retweets the Ad; (c)* **by** *time step* 3*, the Ad reaches Mary, and Mary retweets it. As seen, the information diffuses from one user to his or her neighbors* *with different probabilities, represented by the weighted edges in G. Note that the cascade unfolds as a* ***tree****, rooted at the node Ann.*

To capture the entire topological information of the cascades, we need to make inferences on the temporal graph. Given the above partially observed information, two such inferred cascades are shown as trees T_1 and T_2 in Figure 3.1. T_1 illustrates a cascade where each path from the source Ann to each observed node has a length that exactly equals to the time step, at which the observed node is influenced, while T_2 illustrates a cascade where any path in T_2 from Ann to an observed node has a length no greater than the observed time step when the node is influenced, due to possible delay in observation, e.g., *Mary is known to be influenced by (instead of exactly at) time step* 3*. The inferred cascades provide useful information about the missing links and users that are important in the propagation of the information.*

The above example highlights the need to make reasonable inference about the cascades, according to only the partial observations of influenced nodes and the time at or by which they are influenced. Although cascade models and a set of related problems, *e.g.,* influence maximization, have been widely studied, much less is known on how to infer the cascade structures, including complexity bounds and approximation algorithms.

In this chapter, we investigate the cascade inference problem, where cascades follow the widely used *independent cascade model.* To the best of our knowledge, this is the first work towards inferring cascades as *general trees* following independent cascade model, based on the partial observations. The rest of this chapter are organized as follows.

- We introduce the notions of *(perfect and bounded) consistent trees* in Section 3.2. These notions capture the inferred cascades by incorporating connectivity and time constraints in the partial observations. To provide a quantitative measure of the quality of inferred cascades, we also introduce two metrics, based on the size of the consistent trees and the likelihood when a diffusion function of the network graph is taken into account, respectively. These metrics give rise to two optimization problems, referred to as the *minimum consistent tree* problem and *minimum weighted consistent tree* problem.

- We investigate the problems of identifying perfect and bounded consistent trees, for given partial observations, in Section 3.3 and Section 3.4, respectively. These problems are variants of the inference problem. We show that these problems are all NP-complete. Worse still, the optimization problems are hard to approximate: unless P = NP, it is not possible to approximate the problems within any *constant* ratio. Nevertheless, we provide approximation and heuristic algorithms for these problems. For bounded trees, the problems are $O(|X| * \frac{\log f_{min}}{\log f_{max}})$-approximable, where $|X|$ is the size of the partial observation, and f_{min} (resp. f_{max}) are the minimum (resp. maximum) probability on the graph edges. We provide such polynomial approximation algorithms. For perfect trees, we show that it is already NP-hard to even find a feasible solution. However, we provide an efficient heuristics using a greedy strategy. Finally, we address a practical special case for perfect tree problems, which are $O(d * \frac{\log f_{min}}{\log f_{max}})$-approximable, where d is the diameter of the graph, which is typically small in practice.

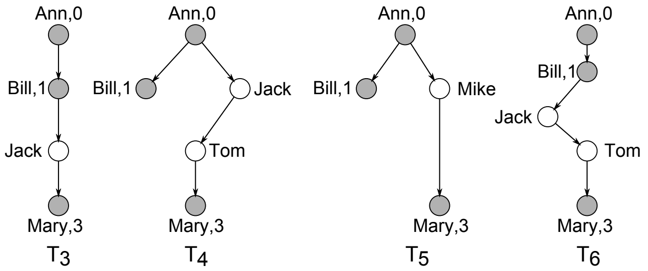


Figure 3.2: Tree representations of a partial observation $X = \{(\text{Ann}, 0), (\text{Bill}, 1), (\text{Mary}, 3)\}$: T_3, T_4 and T_5 are consistent Trees, while T_6 is not.

- We experimentally verify the effectiveness and the efficiency of our algorithms in Section 3.5, using real-life data and synthetic data. We show that our inference algorithms can efficiently infer cascades with satisfactory accuracy.

- We discuss the related work in Section 3.6 and conclude this chapter in Section 3.7.

3.2 Consistent Trees

We start by introducing several notions.

Diffusion graph. We denote a social network as a *directed* graph $G = (V, E, f)$, where (a) V is a finite set of nodes, and each node $u \in V$ denotes a user; (b) $E \subseteq V \times V$ is a finite set of edges, where each edge $(u, v) \in E$ denotes a social connection via which the information may diffuse from u to v; and (c) a *diffusion function* $f : E \to R^+$ which assigns for each edge $(u, v) \in E$ a value $f(u, v) \in [0, 1]$, as the probability that node u *influences* v.

Cascades. We first review the *independent cascade model* [102]. We say an information propagates over a graph G following the *independent cascade model* if (a) at any time step, each node in G is exactly one of the three states {*active*, *newly active*, *inactive*}; (b) a cascade starts from a *source node* s being *newly active* at time step 0; (c) a *newly active* node u at time step t has only one chance to influence its *inactive* neighbors, such that at time $t+1$, (i) if v is an inactive neighbor of u, v becomes *newly active* with probability $f(u,v)$; and (ii) the state of u changes from *newly active* to *active*, and cannot influence any neighbors afterwards; and (d) each *inactive* node v can be influenced by at most one of its *newly active* neighbors independently, and the neighbors' attempts are sequenced in an arbitrary order. Once a node is *active*, it cannot change its state.

Based on the independent cascade model, we define a *cascade* C over graph $G = (V, E, f)$ as a *directed tree* $(V_c, E_c, s, \mathcal{T})$ where (a) $V_c \subseteq V$, $E_c \subseteq E$; (b) $s \in V_c$ is the *source node* from which the information starts to propagate; and (c) $\mathcal{T}$ is a function which assigns for each node $v_i \in V_c$ a *time step* t_i, which represents that v_i is *newly active* at time step t_i. Intuitively, a cascade is a tree representation of the "trace" of the information propagation from a specified source node s to a set of influenced nodes.

Indeed, one may verify that any cascade from s following the independent cascade model is a tree rooted at s.

Example 2. *The graph G in Figure 1 depicts a social graph. The tree T_1 and T_2 are two possible cascades following the independent cascade model. For instance, after issuing an ad of "Iphone 4s", Ann at time* 0 *becomes "newly active". Bill and Jack retweet the ad at time* 1*. Ann becomes "active", while Bill and Jack are* *turned to "newly active". The process repeats until the ad reaches Mary at time step* 3. *The trace of the information propagation forms the cascade* T_1.

As remarked earlier, it is often difficult to observe the entire structure of a cascade in practice. We model the observed information for a cascade as a *partial observation*.

Partial observation. Given a cascade $C = (V_c, E_c, s, \mathcal{T})$, a pair (v_i, t_i) is an *observation point*, if $v_i \in V$ is known (observed) to be *newly active at* or *by* time step t_i. A *partial observation* X is a set of observation points. Specifically, X is a *complete observation* if for any $v \in V_c$, there is an observation point $(v, t) \in X$. To simplify the discussion, we also assume that pair $(s, 0) \in X$ where s is the source node. The techniques developed in this paper can be easily adapted to the case where the source node is unknown.

We are now ready to introduce the idea of consistent trees.

3.2.1 Consistent trees

Given a partial observation X of a graph $G = (V, E, f)$, a *bounded consistent tree* $T_s = (V_{T_s}, E_{T_s}, s)$ *w.r.t.* X is a directed subtree of G with root $s \in V$, such that for *every* $(v_i, t_i) \in X$, $v_i \in V_{T_s}$, and s reaches v_i by t_i hops, *i.e.,* there exists a path of length *at most* t_i from s to v_i. Specifically, we say a consistent tree is a *perfect consistent tree* if for every $(v_i, t_i) \in X$ and $v_i \in V_{T_s}$, there is a path of length *equals to* t_i from s to v_i.

Intuitively, consistent trees represent possible cascades which conform to the independent cascade model, as well as the partial observation. Note the following: (a) the path from the root s to a node v_i in a bounded consistent tree T_s is not necessarily a shortest path from s to v_i in G, as observed in [111]; (b) the perfect consistent trees model cascades when the partial observation is accurate, *i.e.,* each time t_i in an observation point (v_i, t_i) is exactly the time when v_i is newly active; in contrast, in bounded consistent trees, an observation point (v, t) indicates that node v is newly active at the time step $t' \leq t$, due to possible *delays* in the information propagation, as observed in [44].

Example 3. *Recall the graph G in Figure 1. The partial observation of a cascade in G is $X = \{(Ann, 0), (Bill, 1), (Mary, 3)\}$. The tree T_1 is a perfect consistent tree* w.r.t. *X, where T_2 is a bounded consistent tree* w.r.t. *X.*

Now consider the trees in Figure 3.2. One may verify that (a) T_3, T_4 and T_5 are bounded consistent trees w.r.t. *X; (b) T_3 and T_4 are perfect consistent trees* w.r.t. *X, where T_5 is not a perfect consistent tree. (c) T_6 is not a consistent tree, as there is no path from the source Ann to Mary with length no greater than 3 as constrained by the observation point $(Mary, 3)$.*

3.2.2 Cascade inference problem

We introduce the general cascade inference problem. Given a social graph G and a partial observation X, the *cascade inference problem* is to determine whether there exists a consistent tree T *w.r.t.* X in G.

There may be multiple consistent trees for a partial observation, so one often wants to identify the best consistent tree. We next provide two quantitative metrics to measure the quality of the inferred cascades. Let $G = (V, E, f)$ be a social graph, and X be a partial observation.

Minimum weighted consistent trees. In practice, one often wants to identify the consistent trees that are most likely to be the real cascades. Recall that each edge $(u, v) \in E$ in a given network G carries a value assigned by a diffusion function $f(u, v)$, which indicates the probability that u influences v. Based on $f(u, v)$, we introduce a *likelihood function* as a quantitative metric for consistent trees.

Likelihood function. Given a graph $G = (V, E, f)$, a partial observation X and a consistent tree $T_s = (V_{T_s}, E_{T_s}, s)$, the *likelihood* of T_s, denoted as $L_X(T_s)$, is defined as:

$$L_X(T_s) = \mathbb{P}(X \mid T_s) = \prod_{(u,v)\in E_{T_s}} f(u, v). \tag{3.1}$$

Following common practice, we opt to use the log-likelihood metric, where

$$L_X(T_s) = \sum_{(u,v)\in E_{T_s}} \log f(u, v)$$

Given G and X, a natural problem is to find the consistent tree of the maximum likelihood in G *w.r.t.* X. Using log-likelihood, the *minimum weighted consistent tree* problem is to identify the consistent tree T_s with the minimum $-L_X(T_s)$, which in turn has the maximum likelihood.

Minimum consistent trees. Instead of weighted consistent trees, one may simply want to find the *minimum* structure that represents a cascade [125]. The minimum consistent tree, as a special case of the minimum weighted consistent tree, depicts the smallest cascades with the fewest communication steps to pass the information to all the observed nodes. In other words, the metric favors those consistent trees consist with the given partial observation with the fewest edges.

Given G and X, the *minimum consistent tree* problem is to find the minimum consistent trees in G *w.r.t.* X.

In the following sections, we investigate the cascade inference problem, and the related optimization problems using the two metrics. We investigate the problems for perfect consistent trees in Section 3.3, and for bounded consistent trees in Section 3.4, respectively.

3.3 Cascades as perfect trees

As remarked earlier, when the partial observation X is accurate, one may want to infer the cascade structure via *perfect consistent trees*. The *minimum (resp. weighted) perfect consistent tree* problem, denoted as $\mathsf{PCT}_{\mathsf{min}}$ (resp. $\mathsf{PCT}_{\mathsf{w}}$) is to find the perfect consistent trees with minimum size (resp. weight) as the quality metric.

Though it is desirable to have efficient polynomial time algorithms to identify perfect consistent trees, the problems of searching $\mathsf{PCT}_{\mathsf{min}}$ and $\mathsf{PCT}_{\mathsf{w}}$ are nontrivial.

Proposition 3. *Given a graph G and a partial observation X, (a) it is* NP-*complete to determine whether there is a perfect consistent tree* w.r.t. *X in G; and (b) the* $\mathsf{PCT}_{\mathsf{min}}$ *and* $\mathsf{PCT}_{\mathsf{w}}$ *problems are* NP-*complete and* APX-*hard.*

One may verify Proposition 3(a) by a reduction from the Hamiltonian path problem [180], which is to determine whether there is a simple path of length

$|V|-1$ in a graph $G = (V, E)$. Following this, one can verify that the $\mathsf{PCT}_{\mathsf{min}}$ and $\mathsf{PCT}_{\mathsf{w}}$ problems are NP-complete as an immediate result.

Proposition 3(b) shows that the $\mathsf{PCT}_{\mathsf{min}}$ and $\mathsf{PCT}_{\mathsf{w}}$ problems are hard to approximate. The APX class [180] consists of NP optimization problems that can be approximated by a polynomial time (PTIME) algorithm within *some* positive constant. The APX-hard problems are APX problems to which every APX problem can be reduced. Hence, the problem for computing a minimum (weighted) perfect consistent tree is among the hardest ones that allow PTIME algorithms with a constant approximation ratio.

It is known that if there is an *approximation preserving reduction (*AFP-reduction*)* [180] from a problem Π_1 to a problem Π_2, and if problem Π_1 is APX-hard, then Π_2 is APX-hard [180]. To see Proposition 3(b), we may construct an AFP-reduction from the minimum directed steiner tree (MST) problem. An instance of a directed steiner tree problem $I = \{G, V_r, V_s, r, w\}$ consists of a graph G, a set of *required* nodes V_r, a set of *steiner* nodes V_s, a source node r and a function w which assigns to each node a positive weight. The problem is to find a minimum weighted tree rooted at r, such that it contains all the nodes in V_r and a part of V_s. We show such a reduction exists. Since MST is APX-hard, $\mathsf{PCT}_{\mathsf{min}}$ is APX-hard.

3.3.1 Bottom-up searching algorithm

Given the above intractability and approximation hardness result, we introduce a heuristic WPCT for the $\mathsf{PCT}_{\mathsf{w}}$ problem. The idea is to (a) generate a "backbone network" G_b of G which contains all the nodes and edges that are possible to form a perfect consistent tree, using a set of *pruning rules*, and also rank the observed nodes in G_b with the descending order of their time step in X, and (b) perform a bottom-up evaluation for each time step in G_b using a local-optimal strategy, following the descending order of the time step.

Backbone network. We consider pruning strategies to reduce the nodes and the edges that are not possible to be in any perfect consistent trees, given a graph $G = (V, E, f)$ and a partial observation $X = \{(v_1, t_1), \ldots, (v_k, t_k)\}$. We define a backbone network $G_b = (V_b, E_b)$, where

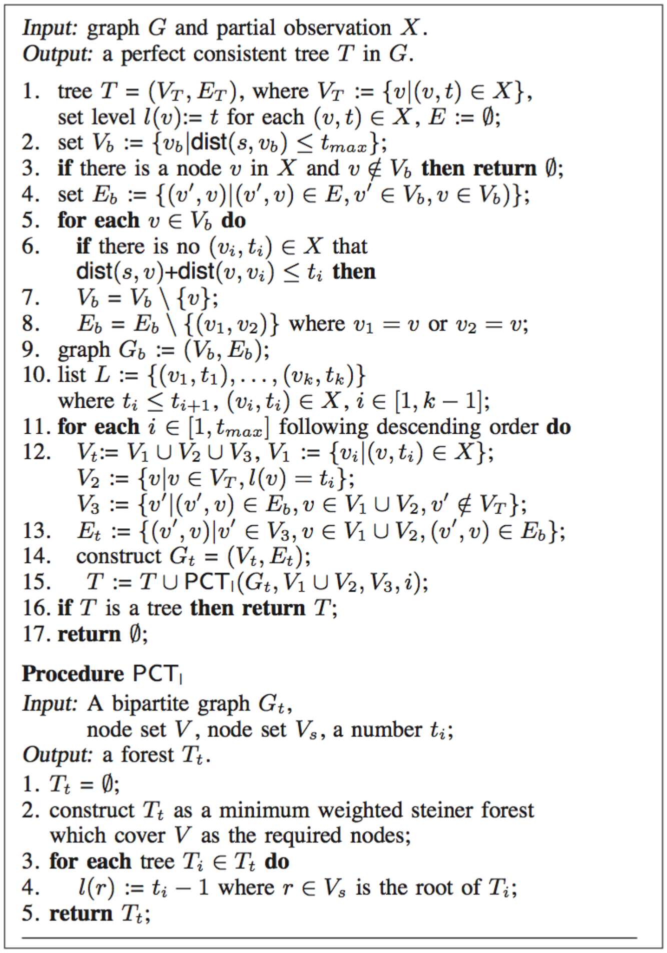
- $V_b = \bigcup\{v_j | \text{dist}(s, v_j) + \text{dist}(v_j, v_i) \leq t_i\}$ for each $(v_i, t_i) \in X$; and
- $E_b = \{(v', v) | v' \in V_b, v \in V_b, (v', v) \in E\}$

Intuitively, G_b includes all the possible nodes and edges that may appear in a perfect consistent tree for a given partial observation. In order to construct G_b, a set of *pruning rules* can be developed as follows: if for a node v' and each observed node v in a cascade with time step t, $\text{dist}(s, v') + \text{dist}(v', v) > t$, then v' and all the edges connected to v' can be removed from G_b.

Algorithm. Algorithm WPCT, as shown in Figure 3.3, consists of the following steps:

Initialization (line 1). The algorithm WPCT starts by initializing a tree T, by inserting all the observation points into T. Each node v in T is assigned with a *level* $l(v)$ equal to its time step as in X. The edge set is set to empty.

Pruning (lines 2-10). The algorithm WPCT then constructs a backbone network G_b with the pruning rules (lines 2-9). It initializes a node set V_b within t_{max} hop of the source node s, where t_{max} is the maximum time step in X (line 2). If there exists some node $v \in X$ that is not in V_b, the algorithm returns $\emptyset$, since there is no



```
Input: graph $G$ and partial observation $X$.
Output: a perfect consistent tree $T$ in $G$.
tree $T = (V_T, E_T)$, where $V_T := \{v|(v,t) \in X\}$,
  set level $l(v) := t$ for each $(v,t) \in X$, $E := \emptyset$;
set $V_b := \{v_b|\mathsf{dist}(s, v_b) \leq t_{max}\}$;
if there is a node $v$ in $X$ and $v \notin V_b$ then return $\emptyset$;
set $E_b := \{(v', v)|(v', v) \in E, v' \in V_b, v \in V_b)\}$;
for each $v \in V_b$ do
  if there is no $(v_i, t_i) \in X$ that
  $\mathsf{dist}(s, v)$+$\mathsf{dist}(v, v_i) \leq t_i$ then
  $V_b = V_b \setminus \{v\}$;
  $E_b = E_b \setminus \{(v_1, v_2)\}$ where $v_1 = v$ or $v_2 = v$;
graph $G_b := (V_b, E_b)$;
list $L := \{(v_1, t_1), \ldots, (v_k, t_k)\}$
  where $t_i \leq t_{i+1}$, $(v_i, t_i) \in X$, $i \in [1, k-1]$;
for each $i \in [1, t_{max}]$ following descending order do
  $V_t := V_1 \cup V_2 \cup V_3$, $V_1 := \{v_i|(v, t_i) \in X\}$;
  $V_2 := \{v|v \in V_T, l(v) = t_i\}$;
  $V_3 := \{v'|(v', v) \in E_b, v \in V_1 \cup V_2, v' \notin V_T\}$;
  $E_t := \{(v', v)|v' \in V_3, v \in V_1 \cup V_2, (v', v) \in E_b\}$;
  construct $G_t = (V_t, E_t)$;
   $T := T \cup \mathsf{PCT}_\mathsf{l}(G_t, V_1 \cup V_2, V_3, i)$;
if $T$ is a tree then return $T$;
return $\emptyset$;

Procedure $\mathsf{PCT}_\mathsf{l}$
Input: A bipartite graph $G_t$,
       node set $V$, node set $V_s$, a number $t_i$;
Output: a forest $T_t$.
$T_t = \emptyset$;
construct $T_t$ as a minimum weighted steiner forest
  which cover $V$ as the required nodes;
for each tree $T_i \in T_t$ do
    $l(r) := t_i - 1$ where $r \in V_s$ is the root of $T_i$;
return $T_t$;
```

Figure 3.3: Algorithm WPCT: initialization, pruning and local searching

path from s reaching v with t steps for $(v, t) \in X$ (line 3). It further removes the redundant nodes and edges that are not in any perfect trees, using the pruning rules (lines 5-8). The network G_b is then constructed with V_b and E_b at line 9. The partial observation X is also sorted *w.r.t.* the time step (line 10).

Bottom-up local searching (lines 11-17). Following a bottom-up greedy strategy, the algorithm WPCT processes each observation point as follows. For each i in $[1, t_{max}]$, it generates a (bipartite) graph G_t. (a) It initializes a node set V_t as the union of three sets of nodes V_1, V_2 and V_3 (line 12), where (i) V_1 is the nodes in the observation points with time step t_i, (ii) V_2 is the nodes v in the current perfect consistent tree T with level $l(v) = t_i$, and (iii) V_3 is the union of the parents for the nodes in V_1 and V_2. (b) It constructs an edge set E_t which consists of the edges from the nodes in V_3 to the nodes in V_1 and V_2. (c) It then generates G_t with V_t and the edge set E_t, which is a bipartite graph. After G_t is constructed, the algorithm WPCT invokes procedure $\mathsf{PCT_l}$ to compute a "part" of the perfect tree T, which is an *optimal* solution for G_t, a part of the graph G_b which contains all the observed nodes with time step t_i. It expands T with the returned partial tree (line 15). The above process (lines 11-15) repeats for each $i \in [1, t_{max}]$ until all the nodes in X are processed. Algorithm WPCT then checks if the constructed T is a tree. If so, it returns T (line 16). Otherwise, it returns $\emptyset$ (line 17). The above procedure is as illustrated in Figure 3.4.

Procedure $\mathsf{PCT_l}$. Given a (bipartite) graph G_t, and two sets of nodes V and V_s in G_t, the procedure $\mathsf{PCT_l}$ computes for G_t a set of trees $T_t = \{T_1, \ldots, T_i\}$ with the minimum total weight (line 2), such that (a) each T_i is a 2-level tree with a root in V_s and leaves in V, (b) the leaves of any two trees in T_t are disjoint, and (c) the

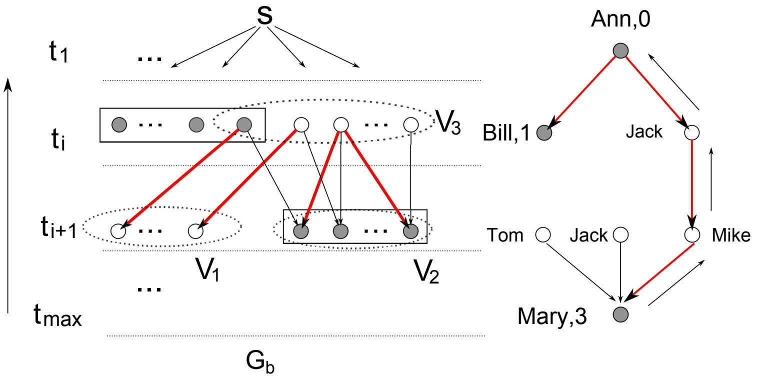


Figure 3.4: The bottom-up searching in the backbone network

trees contain all the nodes in V as leaves. For each T_i, $\mathsf{PCT_l}$ assigns its root r in V_s a level $l(r) = t_i - 1$ (line 4). T_t is then returned as a part of the entire perfect consistent tree (line 5). In practice, we may either employ linear programming, or an algorithm for MST problem (*e.g.*, [154]) to compute T_t.

Example 4. *The cascade T_1 in Figure 1, as a minimum weighted perfect consistent tree, can be inferred by algorithm* WPCT *as illustrated in Figure 3.4.* WPCT *first initializes a tree T with the node Mary. It then constructs G_t as the graph induced by edges (Tom, Mary), (Jack, Mary), and (Mike, Mary). Intuitively, the three nodes as the parents of Mary are the possible nodes which accepts the message at time step* 2*. It then selects the tree with the maximum probability, which is a single edge (Mike, Mary), and adds it to T. Following Mike, it keeps choosing the optimal tree structure for each level, and identifies nodes Jack. The process repeats until* WPCT *reaches the source Ann. It then returns the perfect consistent tree T as the inferred cascade from the partial observation X.*

Correctness. The algorithm WPCT either returns $\emptyset$, or correctly computes a perfect consistent tree *w.r.t.* the partial observation X. Indeed, one may verify that (a) the pruning rules only remove the nodes and edges that are not in any perfect consistent tree *w.r.t.* X, and (b) WPCT has the loop invariant that at each iteration i (lines 11-15), it always constructs a part of a perfect tree as a forest.

Complexity. The algorithm WPCT is in time $O(|V||E|+|X|^2+t_{max}*\mathcal{A})$, where t_{max} is the maximum time step in X, and $\mathcal{A}$ is the time complexity of procedure PCT_l. Indeed, (a) the initialization and preprocessing phase (lines 1-9) takes $O(|V||E|)$ time, (b) the sorting phase is in $O(|X|^2)$ time, (c) the bottom-up construction is in $O(|t_{max} * \mathcal{A}|)$, which is further bounded by $O(|t_{max} * |V|^3)$ if an approximable algorithm is used [154]. In our experimental study, we utilize efficient linear programming to compute the *optimal* steiner forest.

The algorithm WPCT can easily be adapted to the problem of finding the minimum perfect consistent trees, where each edge has a unit weight.

Perfect consistent SP trees. The independent cascade model may be an overkill for real-life applications, as observed in [50, 107]. Instead, one may identify the consistent trees which follow the shortest path model [107], where cascades propagate following the shortest paths. We define a *perfect shortest path (*sp*) tree* rooted at a given source node s as a perfect consistent tree, such that for each observation point $(v, t) \in X$ of the tree, $t = \text{dist}(s, v)$; in other words, the path from s to v in the tree is the *shortest path* in G. The PCT_w (resp. $\mathsf{PCT}_\mathsf{min}$) problem for sp trees is to identify the sp trees with the maximum likelihood (resp. minimum size).

Proposition 4. *Given a graph G and a partial observation X, (a) it is in* PTIME *to find a* sp *tree* w.r.t. *X; (b) the* $\mathsf{PCT}_\mathsf{min}$ *and* PCT_w *problems for perfect* sp *trees are* NP*-hard and* APX*-hard; (c) the* $\mathsf{PCT_w}$ *problem is approximable within* $O(d*\frac{\log f_{min}}{\log f_{max}})$*, where* d *is the diameter of* G*, and* f_{max} *(resp.* f_{min}*) is the maximum (resp. minimum) probability by the diffusion function* f*.*

We next provide an approximation algorithm to the $\mathsf{PCT_w}$ problem for sp trees. Given a graph G and a partial observation X, the algorithm, denoted as $\mathsf{WPCT_{sp}}$ (not shown), first constructs the backbone graph G_b as in the algorithm WPCT. It then constructs node sets $V_r = \{v|(v,t) \in X\}$, and $V_s = V \setminus V_r$. Treating V_r as required nodes, V_s as steiner nodes, and the log-likelihood function as the weight function, $\mathsf{WPCT_{sp}}$ approximately computes an undirected minimum steiner tree T. If the directed counterpart T' of T in G_b is not a tree, $\mathsf{WPCT_{sp}}$ transforms T' to a tree: for each node v in T' with more than one parent, it (a) connects s and v via the shortest path, and (b) removes the redundant edges attached to v. It then returns T' as an sp tree.

One may verify that (a) T' is a perfect sp tree *w.r.t.* X, (b) the weight $-L_X(T')$ is bounded by $O(d * \frac{\log f_{min}}{\log f_{max}})$ times of the optimal weight, and (c) the algorithm runs in $O(|V^3|)$ time, leveraging the approximation algorithm for the steiner tree problem [180]. Moreover, the algorithm $\mathsf{WPCT_{sp}}$ can be used for the problem $\mathsf{PCT_{min}}$ for sp trees, where each edge in G has the same weight. This achieves an approximation ratio of d.

3.4 Cascades as bounded trees

In this section, we investigate the cascade inference problems for bounded consistent trees. In contrast to the intractable counterpart in Proposition 3(a), the problem of finding a bounded consistent tree for a given graph and a partial observation is in PTIME.

Proposition 5. *For a given graph G and a partial observation X, there is a bounded consistent tree in G* w.r.t. *X if and only if for each $(v, t) \in X$, dist$(s, v) \leq t$, where dist(s, v) is the distance from s to v in G.*

Indeed, one may verify the following: (a) if there is a node $(v_i, t_i) \in X$ where $\text{dist}(s, v_i) > t_i$, there is no path satisfies the time constraint and T is empty; (b) if $\text{dist}(s, v_i) \leq t_i$ for each node $(v_i, t_i) \in X$, a BFS tree rooted at s with each node v_i in X as its internal node or leaf is a bounded consistent tree. Thus, to determine whether there is a bounded consistent tree is in $O(|E|)$ time, via a BFS traversal of G from s.

Given a graph G and a partial observation X, the *minimum weighted bounded consistent tree* problem, denoted as $\mathsf{BCT_w}$, is to identify the bounded consistent tree T_s^* *w.r.t.* X with the minimum $-\log L_X(T_s^*)$ (see Section 3.2).

Theorem 3. *Given a graph G and a partial observation X, the $\mathsf{BCT_w}$ problem is*

(a) NP-complete and APX-hard; and

(b) approximable within $O(|X| * \frac{\log f_{min}}{\log f_{max}})$, where f_{max} (resp. f_{min}) is the maximum (resp. minimum) probability by the diffusion function f over G.

We can prove Theorem 3(a) as follows. First, the $\mathsf{BCT_w}$ problem, as a decision problem, is to determine whether there exists a bounded consistent tree T with $-L_X(T)$ no greater than a given bound B. The problem is obviously in NP. To show the lower bound, one may show there exists a polynomial time reduction from

```
Input: graph $G$ and partial observation $X$.
Output: a bounded consistent tree $T$ in $G$.

1.  tree $T = (V_t, E_t)$, where $V_t := \{s|(s, 0) \in X\}$, $E_t := \emptyset$;
2.  compute $t_k$ bounded BFS DAG $G_d$ of $s$ in $G$;
3.  foreach $t_i \in [t_1, t_k]$ Do
4.      foreach node $v$ where $(v, t_i) \in X$ and $l(v) = i$ Do
5.          if$i > t_i$ return $\emptyset$;
6.          find a path $\rho$ from $s$ to $v$ with the
            minimum weight $w(\rho) = -\Sigma \log f(e)$ for each $e \in \rho$;
7.          $T = T \cup \rho$;
8.  return $T$ as a bounded consistent tree;
```

Figure 3.5: Algorithm WBCT: searching bounded consistent trees via top-down strategy

the exact 3-cover problem (X3C). Second, to see the approximation hardness, one may verify that there exists an AFP-reduction from the minimum directed steiner tree (MST) problem.

We next provide a polynomial time algorithm, denoted as WBCT, for the $\mathsf{BCT_w}$ problem. The algorithm runs in *linear time w.r.t.* the size of G, and with performance guarantee as in Theorem 3(b).

Algorithm. The algorithm WBCT is illustrated in Figure 3.5. Given a graph G and a partial observation X, the algorithm first initializes a tree $T = (V_t, E_t)$ with the single source node s (line 1). It then computes the t_k bounded BFS directed acyclic graph (DAG) [33] G_d of the source node s, where t_k is the maximum time step of the observation points in X, and G_d is a DAG induced by the nodes and edges visited by a BFS traversal of G from s (line 2). Following a top-down strategy, for each node v of $(v, t) \in X$, WBCT then (a) selects a path ρ with the minimum $\Sigma \log f(e)$ from s to v, and (b) extends the current tree T with the path ρ (lines 3-7). If for some observation point $(v, t) \in T$, $\text{dist}(s, v) > t$, then WBCT returns $\emptyset$ as the tree T (line 5). Otherwise, the tree T is returned (line 8) after all the observation points in X are processed.

Correctness and complexity. One may verify that algorithm WBCT either correctly computes a bounded consistent tree T, or returns $\emptyset$. For each node in the observation point X, there is a path of weight selected using a greedy strategy, and the top-down strategy guarantees that the paths form a consistent tree. The algorithm runs in time $O(|E|)$, since it visits each edges at most once following a BFS traversal.

We next show the approximation ratio in Theorem 3(b). Observe that for a single node v in X, (a) the total weight of the path w from s to v is no greater than $-|w| \log f_{min}$, where $|w|$ is the length of w; and (b) the weight of the counterpart of w in T^*, denoted as w', is no less than $-|w^*| \log f_{max}$. Also observe that $|w| \leq |w^*|$. Thus, $w/w^* \leq \frac{\log f_{min}}{\log f_{max}}$. As there are in total $|X|$ such nodes, $L_X(T)/L_X(T^*) \leq |X|\frac{w}{w^*} \leq |X|\frac{\log f_{min}}{\log f_{max}}$. Theorem 3(b) thus follows.

Minimum bounded consistent tree.

We have considered the likelihood function as a quantitative metric for the quality of the bounded consistent trees. As remarked earlier, one may simply want to identify the bounded consistent trees of the *minimum* size. Given a social graph G and a partial observation X, the *minimum bounded consistent tree problem*, denoted as $\mathsf{BCT}_{\mathsf{min}}$, is to identify the bounded consistent tree with the minimum size, *i.e.,* the total number of nodes and edges. The $\mathsf{BCT}_{\mathsf{min}}$ problem is a special case of $\mathsf{BCT}_{\mathsf{w}}$, and its main result is summarized as follows.

Proposition 6. *The* $\mathsf{BCT}_{\mathsf{min}}$ *problem is (a)* NP*-complete, (b)* APX*-hard, and (c) approximable within* $O(|X|)$*, where* $|X|$ *is the size of the partial observation* X*.*

Proposition 6(a) and 6(b) can both be shown by constructing reductions from the MST problem, which is NP-complete and APX-complete [180].

Despite of the hardness, the problem can be approximated within $O(|X|)$ in polynomial time, by applying the algorithm WBCT over an instance where each edge has a unit weight. This completes the proof of Proposition 6(c).

3.5 Experiment

We next present an experimental study of our proposed methods. Using both real-life and synthetic data, we conduct three sets of experiments to evaluate (a) the effectiveness of the proposed algorithms, (b) the efficiency and the scalability of WPCT and WBCT.

Experimental setting. We used real-life data to evaluate the effectiveness of our methods, and synthetic data to conduct an in-depth analysis on scalability by varying the parameters of cascades and partial observations.

(a) Real-life graphs and cascades. We used the following real-life datasets. (i) *Enron email cascades.* The dataset of *Enron Emails* [1] consists of a social graph

[1] http://www.cs.cmu.edu/ enron/

of $86,808$ nodes and $660,642$ edges, where a node is a user, and two nodes are connected if there is an email message between them. We tracked the *forwarded* messages of the same subjects and obtained 260 cascades of depth no less than 3 with more than 8 nodes. (ii) *Retweet cascades* (RT). The dataset of *Twitter Tweets* [2] [193] contains more than 470 million posts from more than 17 million users, covering a period of 7 months from June 2009. We extracted the retweet cascades of the identified *hashtags* [193]. To guarantee that a cascade represents the propagation of a single hashtag, we removed those retweet cascades containing multiple hashtags. In the end, we obtain 321 cascades of depth more than 4, with node size ranging from 10 to 81. Moreover, we used the EM algorithm from [157] to estimate the diffusion function.

(b) Synthetic cascades. We generated a set of synthetic cascades unfolding in an anonymous Facebook social graph [3], which exhibits properties such as power-law degree distribution, high clustering coefficient and positive assortativity [186]. The diffusion function is constructed by randomly assigning real numbers between 0 and 1 to edges in the network. The generating process is controlled by size $|T|$. We randomly choose a node as the source of the cascade. By simulating the diffusion process following the independent cascade model, we then generated cascades *w.r.t.* $|T|$ and assigned time steps.

(c) Partial observation. For both real life and synthetic cascades, we define *uncertainty* of a cascade T as $\sigma = 1 - \frac{|X|}{|V_T|}$, where $|V_T|$ is the size of the nodes in T, and $|X|$ is the size of the partial observation X. We remove the nodes from the

[2] http://snap.stanford.edu/data/twitter7.html

[3] http://current.cs.ucsb.edu/socialnets

Algorithms	**Precision**	**Enron**		**Twitter**	
		d=3	d=4	$d = 4$	$d = 5$
WPCT	prec_v	100%	100%	97.2%	93.2%
	prec_e	78.2%	82.4%	86.1%	82.6%
WBCT	prec_v	100%	70.1%	73.6%	66.1%
	prec_e	69%	55.7%	60.6%	41.7%

Table 3.1: prec_v and prec_e over real cascades

given cascades until the uncertainty is satisfied, and collect the remaining nodes and their time steps as X.

(d) Implementation. We have implemented the following in C++: (i) algorithms WPCT, and WBCT; (ii) two linear programming algorithms $\mathsf{PCT}_{\mathsf{lp}}$ and $\mathsf{BCT}_{\mathsf{lp}}$, which identify the optimal weighted bounded consistent trees and the optimal perfect consistent trees using linear programming, respectively; (iii) two randomized algorithms $\mathsf{PCT}_{\mathsf{r}}$ and $\mathsf{BCT}_{\mathsf{r}}$, which are developed to randomly choose trees from given graphs. $\mathsf{PCT}_{\mathsf{r}}$ is developed using a similar strategy for WPCT, especially for each level the steiner forest is randomly selected (see Section 3.3); as WBCT does, $\mathsf{BCT}_{\mathsf{r}}$ runs on bounded BFS directed acyclic graphs, but randomly selects edges. (iv) to verify various implementations of WPCT, an algorithm $\mathsf{PCT}_{\mathsf{g}}$ is developed by using a greedy strategy to choose the steiner forest for each level (see Section 3.3). We used LP_solve 5.5 [4] as the linear programming solver.

We used a machine powered by an Intel(R) Core 2.8GHz CPU and 8GB of RAM, using Ubuntu 10.10. Each experiment was run by 10 times and the average is reported here.

Experimental results. We next present our findings.

[4] http://lpsolve.sourceforge.net/5.5/

Effectiveness of consistent trees. In the first set of experiments, using real life cascades, we investigated the accuracy and the efficiency of our cascade inference algorithms.

(a) Given a set of real life cascade $\mathbf{T} = \{T_1, \ldots, T_k\}$, for each cascade $T_i = (V_{T_i}, E_{T_i}) \in \mathbf{T}$, we computed an inferred cascade $T_i{}' = (V_{T_i{}'}, E_{T_i{}'})$ according to a partial observation with uncertainty σ. Denote the nodes in the partial observation as V_X. We evaluated the *precision* as $\mathsf{prec} = \frac{\Sigma(|(V_{T_i{}'} \cap V_{T_i}) \setminus V_X|)}{\Sigma(|V_{T_i{}'} \setminus V_X)|}$, and $\mathsf{rec} = \frac{\Sigma(|(V_{T_i{}'} \cap V_{T_i}) \setminus V_X|)}{\Sigma(|V_{T_i} \setminus V_X)|}$. Intuitively, prec is the fraction of inferred nodes that are missing from T_i, while rec is the fraction of missing nodes that are inferred by $T_i{}'$.

For Enron email cascades, Figure 3.6(a) and Figure 3.6(b) show the accuracy of WPCT, $\mathsf{PCT_g}$ and $\mathsf{PCT_r}$ for inferring cascades, while σ is varied from 0.25 to 0.85. $\mathsf{PCT_{lp}}$ does not scale over the Enron dataset and thus is not shown. (i) WPCT outperforms $\mathsf{PCT_g}$ and $\mathsf{PCT_r}$ on both prec and rec. (ii) When the uncertainty increases, both the prec and rec of the three algorithms decrease. In particular, WPCT successfully infers cascade nodes with prec no less than 70% and rec no less than 25% even when 85% of the nodes in the cascades are removed. Using the same setting, the performance of WBCT, $\mathsf{BCT_{lp}}$ and $\mathsf{BCT_r}$ are shown in Figure 3.6(c) and Figure 3.6(d), respectively. (i) Both $\mathsf{BCT_{lp}}$ and WBCT outperform $\mathsf{BCT_r}$, and their prec and rec decrease while the uncertainty increases. (ii) $\mathsf{BCT_{lp}}$ has better performance than WBCT. In particular, both $\mathsf{BCT_{lp}}$ and WBCT successfully infer the cascade nodes with the prec no less than 50% and with the rec no less than 25%, even when 85% of the nodes in the cascades are removed.

For retweet cascades, the prec and the rec of WPCT, $\mathsf{PCT_g}$ and $\mathsf{PCT_r}$ are shown in Figure 3.6(e) and in Figure 3.6(f), respectively. While the uncertainty increases from 0.25 to 0.85, (i) WPCT outperform $\mathsf{PCT_r}$ and $\mathsf{PCT_g}$, and (ii) the performance of all the algorithms decreases. In particular, WPCT successfully infers the nodes with the prec more than 80% and the rec more than 35%, while the uncertainty is 25%. Similarly, the prec and the rec of WBCT and $\mathsf{BCT_r}$ are presented in Figure 3.6(g) and Figure 3.6(h), respectively. As $\mathsf{BCT_{lp}}$ does not scale on retweet cascades, its performance is not shown. While the uncertainty σ increases, the prec and the rec of the algorithms decrease. For all σ, WBCT outperforms $\mathsf{BCT_r}$; in particular, WBCT correctly infers the nodes with prec no less than 60% and rec no less than 25%, when σ is 25%.

(b) To further evaluate the structural similarity of T_i and $T_i{}'$ as described in (a), we also evaluate (i) $\mathsf{prec}_v = \frac{|V''|}{|V'|}$ for nodes $V' = (V_{T_i{}'} \cap V_{T_i}) \setminus V_X$, where $V'' \in V'$ are the nodes with the same *topological order* in both T_i' and T_i, and (ii) $\mathsf{prec}_e = \frac{|E'|}{|E_{T_i}{}'|}$ for $E' = E_{T_i} \cap E_{T_i}{}'$, following the metric for measuring graph similarity [150]. The average results are as shown in Table 3.1, for σ =50%, and the cascades of fixed depth. As shown in the table, for WPCT, the average prec_v is above 90%, and the average prec_e is above 75% over both datasets. Better still, the results hold even when we set $\sigma = 85\%$. For WBCT, prec_v and prec_e are above 65% and above 40%, respectively. For WPCT, prec_v and prec_e have almost consistent performance on both datasets; however, for WBCT, the prec_v and prec_e of the inferred Enron cascades are higher than those of the inferred retweet cascades. The gap might result from the different diffusion patterns between these two datasets: we observed that there are more than 70% of cascades in the Enron dataset whose structures are contained in the BFS directed acyclic graphs of WBCT, while in the Twitter Tweets there are less than 45% of retweet cascades following the assumed graph structures of WBCT.

Efficiency over real datasets. In all the tests over real datasets, $\mathsf{PCT_r}$, $\mathsf{BCT_r}$, $\mathsf{PCT_g}$ and WBCT take less than 1 second. $\mathsf{BCT_{lp}}$ does not scale for retweet cascades, while $\mathsf{PCT_{lp}}$ does not scale for both datasets. On the other hand, while WPCT takes less than 0.4 seconds in inferring all the Enron cascades, it takes less than 20 seconds to infer Twitter cascades where d=4, and 100 seconds when $d = 5$. Indeed, for Twitter network the average degree of the nodes is 20, while the average degree for Enron dataset is 7. As such, it takes more time for WPCT to infer Twitter cascades in the denser Twitter network. In our tests, the efficiency of all the algorithms are not sensitive *w.r.t.* the changes to σ.

Efficiency and scalability over synthetic datasets. In the second set of experiments, we evaluated the efficiency and the scalability of our algorithms using synthetic cascades.

(a) We first evaluate the efficiency and scalability of WPCT and compare WPCT with $\mathsf{PCT_r}$ and $\mathsf{PCT_g}$.

Fixing uncertainty $\sigma = 50\%$, we varied $|T|$ from 30 to 240. Figure 3.7(c) shows that WPCT scales well with the size of the cascade. Indeed, it only takes 2 seconds to infer the cascades with 300 nodes.

Fixing size $|T| = 100$, we varied the uncertainty σ from 0.25 to 0.85. Figure 3.7(d) illustrates that while all the three algorithms are more efficient with larger σ, WPCT is more sensitive. All the three algorithms scale well with σ.

As $\mathsf{PCT}_{\mathsf{lp}}$ does not scale well, its performance is not shown in Figure 3.7(c) and Figure 3.7(d).

(b) Using the same setting, we evaluated the performance of WBCT, compared with $\mathsf{BCT}_{\mathsf{lp}}$ and $\mathsf{BCT}_{\mathsf{r}}$.

Fixing σ and varying $|T|$, the result is reported in Figure 3.7(a). First, WBCT outperforms $\mathsf{BCT}_{\mathsf{lp}}$, and is almost as efficient as the randomized algorithm $\mathsf{BCT}_{\mathsf{r}}$. For the cascade of 240 nodes, WBCT takes less than 0.5 second to infer the structure, while $\mathsf{BCT}_{\mathsf{lp}}$ takes nearly 1000 seconds. Second, while WBCT is not sensitive to the change of $|T|$, $\mathsf{BCT}_{\mathsf{lp}}$ is much more sensitive.

Fixing $|T|$ and varying σ, Figure 3.7(b) shows the performance of the three algorithms. The figure tells us that WBCT and $\mathsf{BCT}_{\mathsf{r}}$ are less sensitive to the change of σ than $\mathsf{BCT}_{\mathsf{lp}}$. This is because WBCT and $\mathsf{BCT}_{\mathsf{r}}$ identify bounded consistent tree by constructing shortest paths from the source to the observed nodes. When the maximum depth of the observation point is fixed, the total number of nodes and edges visited by WBCT and $\mathsf{BCT}_{\mathsf{r}}$ are not sensitive to σ.

Summary. We can summarize the results as follows. (a) Our inference algorithms can infer cascades effectively. For example, the original cascades and the ones inferred by WPCT have structural similarity (measured by prec_e) of higher than 75% in both real-life datasets. (b) Our algorithms scale well with the sizes of the cascades, and uncertainty. They seldom demonstrated their worst-case complexity. For example, even for cascades with 240 nodes, all of our algorithms take less than two seconds.

3.6 Related work

We categorize related work as follows.

Cascade Models. To capture the behavior of cascades, a variety of cascade models have been proposed [31, 79, 83, 103, 104], such as *Suscepctible/Infected (SI) model* [31], *decreasing cascade model* [103], *triggering model* [102], *Shortest Path Model* [107], and the *Susceptible/Infected/Recover (SIR) model* [104]. In this paper, we assume that the cascades follow the *independent cascade model* [79], which is one of the most widely studied models (the shortest path model [107] is one of its special cases).

Cascade Prediction. There has been recent work on cascade prediction and inference, with the emphasis on global properties (*e.g.,* cascade nodes, width, size) [42, 70, 108, 114, 156, 167, 183] with the assumption of missing data and partial observations. The problem of identifying and ranking influenced nodes is addressed in [108, 114], but the topological inference of the cascades is not considered. Wang et al. [183] proposed a *diffusive logistic* model to capture the evolution of the density of active users at a given distance over time, and demonstrated the prediction ability of this model. Nevertheless, the structural information about the cascade is not addressed. Song et al. [167] studied the probability of a user being influenced by a given source. In contrast, we consider a more general inference problem where there are multiple observed users, who are influenced at different time steps from the source. Fei et al. [70] studied social behavior prediction and the effect of information content. In particular, their goal is to predict actions on an article based on the training dataset. Budak et al. [42] investigated the optimization problem of minimizing the number of the possible influencing nodes following a specified cascade model, instead of predicting cascades based on partial observations.

All the above works focus on predicting the nodes and their behavior in the cascades. In contrast, we propose approaches to infer both the nodes and the topology of the cascades in the graph-time domain.

Network Inference. Another host of work study network inference problem, which focuses on inferring network structures from observed cascades over the unknown network, instead of inferring cascade structures as trees [66, 80]. Manuel et al. [80] proposes techniques to infer the structure of a network where the cascades flow, based on the observation over the time each node is affected by a cascade. Similar network inference problem is addressed in [66], where the cascades are modeled as (Markov random walk) networks. The main difference between our work and theirs is (a) we use consistent trees to describe possible cascades allowing partial observations; (b) we focus on inferring the structure of cascades as trees instead of the backbone networks.

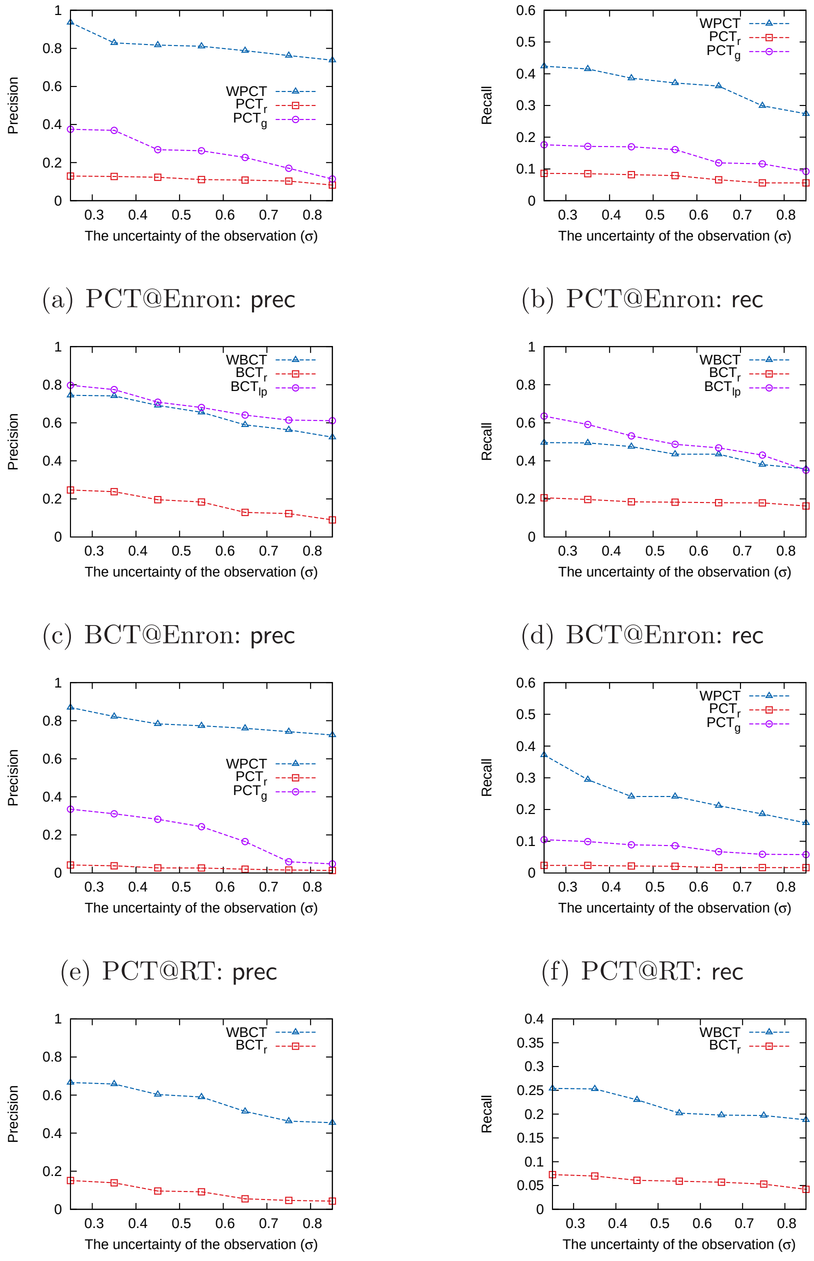
Closer to our work is the work by Sadikov et al. [156] that consider the prediction of the cascades modeled as k-trees, a balanced tree model. The global properties of cascades such as size and depth are predicted based on the incomplete cascade. In contrast to their work, (a) we model cascades as general trees instead of k-balanced trees, (b) while Sadikov et al. [156] assume the partial cascade is also a k-tree and predict only the properties of the original cascade, we infer the nodes as well as topology of the cascades only from a set of nodes and their activation time, using much less available information. (c) The temporal information (*e.g.,* time steps) in the partial observations is not considered in [156].

Problem	Complexity	Approximation	time
$\mathsf{BCT}_{\mathsf{min}}$	NP-C, APX-hard	$\|X\|$	$O(\|E\|)$
$\mathsf{BCT}_{\mathsf{w}}$	NP-C,APX-hard	$\|X\| * \frac{\log f_{max}}{\log f_{min}}$	$O(\|E\|))$
$\mathsf{PCT}_{\mathsf{min}}$ (sp tree)	NP-C, APX-hard	d	$O\|V^3\|$
$\mathsf{PCT}_{\mathsf{w}}$ (sp tree)	NP-C, APX-hard	$d * \frac{\log f_{max}}{\log f_{min}}$	$O\|V^3\|$
$\mathsf{PCT}_{\mathsf{min}}$	NP-C, APX-hard	–	$O(\|t_{max} * \|V\|^3)$
$\mathsf{PCT}_{\mathsf{w}}$	NP-C, APX-hard	–	$O(\|t_{max} * \|V\|^3)$

Table 3.2: Complexity and approximability

3.7 Summary

In this chapter, we investigat cascade inference problem on temporal graphs based on partial observation. We propose the notions of consistent trees for capturing the inferred cascades, namely, bounded consistent trees and perfect consistent trees, as well as quantitative metrics by minimizing either the size of the inferred structure or maximizing the overall likelihood. We establish the intractability and the hardness results for the optimization problems as summarized in Table 3.2. Despite the hardness, we develop approximation and heuristic algorithms for these problems, with performance guarantees on inference quality. We verify the effectiveness and efficiency of our techniques using real life and synthetic cascades. Our experimental results show that our methods are able to efficiently and effectively infer the structure of information cascades.



(a) PCT@Enron: prec

(b) PCT@Enron: rec

(c) BCT@Enron: prec

(d) BCT@Enron: rec

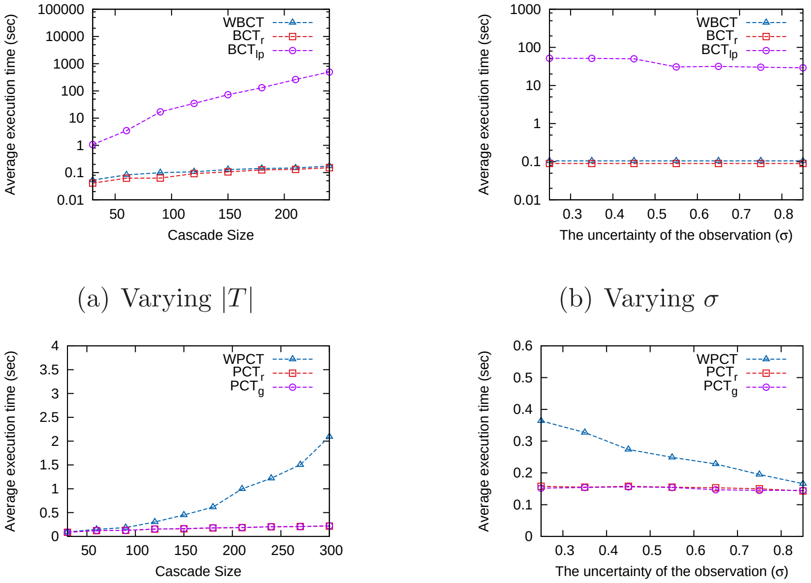
(e) PCT@RT: prec

(f) PCT@RT: rec

(g) BCT@RT: prec

(h) BCT@RT: rec

Figure 3.6: The prec and rec of the inference algorithms over Enron email cascades and Retweet cascades



(a) Varying $|T|$

(b) Varying σ

(c) Varying $|T|$

(d) Varying σ

Figure 3.7: Efficiency and scalability over synthetic cascades

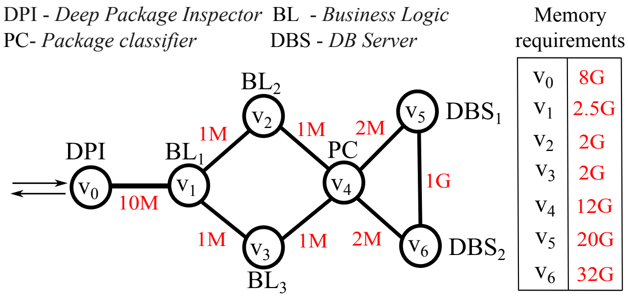
Chapter 4

Subgraph Matching in Temporal Graphs

4.1 Introduction

Temporal graphs have been applied to model frequently updated datacenter networks [149]. Node/edge on these graphs contain numerical values describing network states, such as machines' CPU/memory usage and links' available bandwidth. These numerical values are frequently updated to reflect network dynamics [149].

Given a cloud residing in a datacenter network, it is important to place services into the cloud so that users' requirements are satisfied [35, 77]. Cloud service placement can be naturally formulated as *dynamic subgraph matching* queries: Given a large temporal graph G with numerical node/edge labels and a smaller query graph Q with user-specified numerical node/edge labels (*e.g.,* required computation and communication resources), the goal is to return a set of subgraphs



Memory requirements	
v$_0$	8G
v$_1$	2.5G
v$_2$	2G
v$_3$	2G
v$_4$	12G
v$_5$	20G
v$_6$	32G

Figure 4.1: A user-defined accounting service with diverse memory and bandwidth requirements on nodes and edges

of G, each of which is structurally isomorphic to Q, and whose node/edge labels are compatible with Q (*i.e.*, the corresponding nodes/edges can provide enough computation and network resources). Consider the following example.

Example 5. *In Figure 4.1, an accounting service is defined as a query graph. Numerical labels on nodes represent the amount of memory required for diverse types of servers, while labels on edges represent the amount of bandwidth required among servers. Given such a query graph, a cloud administrator is obliged to find a subgraph from a temporal cloud graph to place the service. A qualifying subgraph should be structurally isomorphic to the query graph, and its nodes/edges should have enough resources to satisfy the specified requirements.*

The aforementioned subgraph matching problem not only brings a new, critical graph query application, but also new challenges to existing techniques. First, existing graph indexing techniques, *e.g.*, [67, 89, 176, 191, 198, 201], focus more on graph structure and fixed node/edge labels, while datacenter networks usually have quite stable structure, but more frequent label updates. Although some incremental graph indexing algorithms are available [191], they are not designed to accommodate frequent label updates (*e.g.,* $10-50\%$ node/edge labels are updated every 10 seconds [149]). Second, existing techniques supporting approximate or probabilistic graph matching can hardly handle partially ordered numerical labels in service placement. For example, a server with 32G free memory can accommodate a service requiring 1G memory, even when these two values are very different from each other. These challenges motivate us to develop a new graph indexing mechanism that is specific for datacenter networks and service placement.

In this chapter, we propose a graph index framework Gradin (Graph index for dynamic (temporal) graphs with numerical labels) to address the above challenges. Gradin encodes subgraphs into multi-dimensional vectors and organizes them such that it can efficiently search the matches of a query's subgraphs and combine them into full matches of the query graph. First, we propose a multi-dimensional index that supports vector search and is able to handle frequent updates. Different from existing indexes that are efficient for index updates but suffer from low pruning power, we develop a search algorithm that preserves the pruning power. Moreover, we present a theoretical analysis of how index parameters affect update and search performance. Second, we propose pruning techniques to enable a fast combination of partial matches of a query graph. A naive solution is costly, when the number of matches for a query's subgraphs is large. Using a minimum cover of subgraphs in a query and subgraphs' *fingerprints*, our method is able to significantly improve query response time.

Our work identifies a key application of graph query in cloud computing. We develop a new graph index framework that accelerates subgraph matching on temporal graphs of numerical labels. To the best of our knowledge, this is the first study on this topic. Using both real and synthetic datasets, we demonstrate that Gradin outperforms the baseline approaches up to 10 times.

4.2 Problem definition

For the sake of simplicity, we examine undirected graphs where only nodes have single labels, and assume that labels are normalized [35, 151]. Our work can be extended to general graphs where both nodes and edges have multiple labels.

Data graph. A *data graph* G is represented by a tuple (V, E, A), where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges; and (3) $A : V \to [0, 1]$ is a function that assigns a numerical label to each node $u \in V$.

Query graph. A *query graph* is defined as $Q = (V', E', A', p)$, where (1) V' and E' are a node set and an edge set, respectively; (2) $A' : V' \to [0, 1]$ is a labeling function; and (3) p is a predicate function that assigns a predicate for each node $u' \in V'$. In other words, p specifies search conditions: $p(u', u)$ defines a predicate $A'(u')$ op $A(u)$, where (1) u' is a node in a query graph; (2) u is a matched node of u' in a data graph; and (3) op is a comparison operator drawn from the set $\{<, \leq, =, \neq, \geq, >\}$. Here, we focus on the predicate $A'(u') \leq A(u)$.

Compatibility. Given two graphs $H_1 = (V_1, E_1, A_1)$ and $H_2 = (V_2, E_2, A_2)$, H_2 is *compatible* with H_1, if (1) H_1 is structurally graph isomorphic to H_2 by a bijective function $f : V_1 \to V_2$; and (2) $\forall u \in V_1,\ A_1(u) \leq A_2(f(u))$.

Graph update in datacenters. In datacenter networks, the most frequent updates come from numerical values on nodes and edges. (1) Update frequency is close to, or even higher than query frequency, and (2) a large portion $(10 - 50\%)$

of nodes/edges in a data graph are frequently updated. In contrast, the physical connections of nodes are relatively stable. Thus, topological update is not the focus of this study.

Definition 2 (Dynamic subgraph matching)**.** *Given a data graph G with its node/edge labels frequently updated, a query graph Q, and an integer r, Dynamic subgraph matching for cloud service placement is to find up to r compatible subgraphs of Q from G.*

The number of returned subgraphs r is decided by applications. To place a service into a cloud, we might need more than one compatible subgraphs in order to optimize the performance of the whole cloud [127]: (1) some compatible subgraphs might not be available due to the network dynamics and the query processing delay; and (2) cloud administrators might be interested in optimizing other performance metrics, such as network congestion [34] and transmission cost [127].

Dynamic subgraph matching is a hard problem. By a reduction from the well-known *subgraph isomorphism* problem [73], the problem can be shown to be NP-complete. On the other hand, for small query graphs and sparse datacenter networks, it is possible to build indices to solve the matching problem in a practical manner. In the following sections, we investigate the feasibility and principles of building a graph index on networks with partially ordered numerical labels, and study how to speed up index update while preserving search speed. We will also discuss how to optimize query processing.

4.3 An overview of Gradin

Fragment. A **fragment** $h = (V_h, E_h, A_h)$ is a connected subgraph from a graph $H = (V, E, A)$, where (1) $V_h \subset V$ and $E_h \subset E$ are a node set and an edge set, respectively; and (2) $\forall u \in V_h$, $A_h(u) = A(u)$.

In particular, we use g to denote a fragment extracted from a data graph G, referred as a *graph fragment*, and use q to denote a fragment extracted from a query graph Q, referred as a *query fragment*. To facilitate index building and searching, we represent fragments by *fragment coordinates*.

Fragment coordinate. Given the *canonical labeling* [190] of a fragment h of k nodes, the fragment coordinate, denoted by $x(h)$, is a k-dimensional vector, where the i-th dimension contains the information about the i-th visited node in its canonical labeling. In particular, the i-th dimension of a fragment coordinate could either be the *id* or the *label* of the i-th visited node in the canonical labeling.

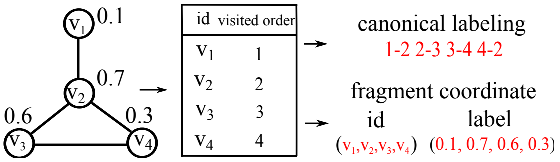
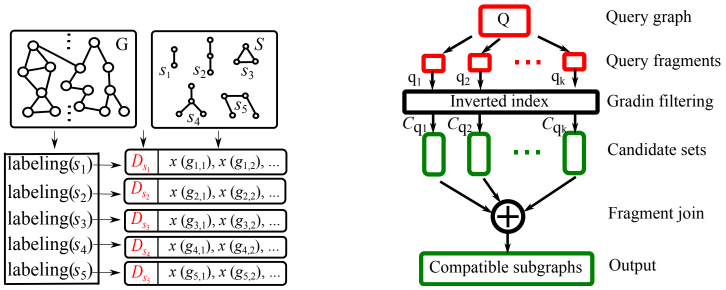


Figure 4.2: A fragment with its canonical labeling (top right) and fragment coordinates (bottom right)

Figure 4.2 shows an example of fragment coordinates. (1) The canonical labeling of the fragment is shown in the top-right corner, where i denotes the i-th visited node, and $i - j$ denotes a visit from the i-th visited node to the j-th visited node. (2) The coordinate (v_1, v_2, v_3, v_4) stores node id information, and the coordinate $(0.1, 0.7, 0.6, 0.3)$ stores node label information. Note that each graph has a unique canonical labeling, and the fragment coordinates specify how to assign ids and labels to nodes and edges.

In addition, we refer to the coordinate of node id information as an *id coordinate*, and the coordinate of label information as a *label coordinate*. When the context is clear, the term fragment coordinate is used without ambiguity.

Let Q be a query graph. A region in a data graph is worth searching, if for any query fragment q, the region contains graph fragments that are compatible with q; otherwise, we can safely exclude that region from search. Gradin implements this idea in two components: *offline index construction* and *online query processing*.



(a) Offline index building (b) Online query processing

Figure 4.3: Gradin consists of two parts: (1) offline index building and (2) online query processing

Offline index construction. Let G be a data graph and $\mathcal{S}$ be a graph structure set that is decided by existing structure selection algorithms [191]. For each structure $s \in \mathcal{S}$, we use subgraph mining technique [190] to search $\mathcal{D}_s$, which contains all the graph fragments in G of the structure s. Using the canonical labeling of structures as keywords, graph fragments are organized by inverted indices. To further optimize search and storage, graph fragments in the inverted indices are denoted by their fragment coordinates.

Figure 4.3(a) illustrates an example of the offline index construction. G is a data graph at the top-left corner, and $\mathcal{S}$ at the top-right corner is a set of structures we aim to index. First, $\mathcal{D}_{s_i}$ – all graph fragments corresponding to structure s_i – are mined from the data graph. Using an inverted index, the canonical labeling of s_i points to $\mathcal{D}_{s_i}$. In particular, the inverted index stores fragment coordinates.

Online query processing. As shown in Figure 4.3(b), given a query graph Q, Gradin searches the compatible subgraphs of Q in three steps. (1) **Decomposition**. Gradin decomposes Q into query fragments, whose structures have been indexed. (2) **Filter**. For each query fragment q, Gradin first finds the set of graph fragments sharing the same structure, and only returns those fragments that are compatible with q. The returned set of graph fragments is also referred to as *candidate set* $\mathcal{C}_q$. (3) **Join**. Gradin conducts fragment join among candidate sets. By stitching fragments in candidate sets, Gradin returns compatible subgraphs for Q.

To provide a good query processing performance, Gradin needs to enable fast search and join at the filter and the join phase. We need to address two challenges: (1) frequent updates and (2) the large search space for fragment join.

Frequent updates. It is preferable to build a search index for $\mathcal{D}_s$ (the graph fragments of structure s), especially when the size of $\mathcal{D}_s$ is very large (*e.g.,* the number of 3-star, a star structure with three branches, in a medium-size datacenter network reaches 10M). However, when node labels are frequently updated, it is non-trivial to build the desired search index. On the one hand, a sophisticated search index (*e.g.,* R-tree [88]) offers strong pruning power; however, it is costly to perform updates [165]. On the other hand, a simple search index (*e.g.,* an inverted index) provides faster index update speed; however, the index's pruning power decreases and longer time is needed for filtering the remaining candidates. In Section 4.4, we discuss how we address this challenge.

The large search space for fragment join. This challenge includes two aspects. First, since the size of a candidate set can be very large, a naïve join algorithm will be extremely slow. Second, since a query graph might be decomposed into dozens of query fragments (*e.g.,* a 10-star query graph contains 110 query fragments of no more than 2 edges), it is preferable to select a subset of query fragments that covers the query graph and minimizes the amount of redundant intermediate results. In Section 4.5, we propose a two-step algorithm that prunes the large search space for fragment join.

4.4 Fragment index

In this section, we present an index FracFilter that efficiently processes frequent updates, and preserves search speed.

4.4.1 Naive solutions

Let $\mathcal{D}_s$ be the set of graph fragments of structure s. There are three basic options to build a search index for $\mathcal{D}_s$: (1) R-tree variant, (2) inverted index, or (3) grid index.

R-tree variant. One might build R-tree variants [48, 88] based on fragment label coordinates to offer good pruning power. However, when node labels are frequently updated, the search trees will process a massive number of update operations. Update operations on R-tree variants are costly [165]. Even though with sophisticated insertion strategies R-tree-like search structures can process around $16,000$ updates per second [165], they will spend a considerable amount of time in processing index updates. For example, in a datacenter network of $3,000$ nodes, when 30% node labels are updated, in the case of graph fragments of the structure 3-star, more than 5M label coordinates need to be updated. Therefore, the state-of-the-art R-tree variant might take more than 5 minutes to update the index. As queries need to wait on index update, the throughput of the whole system will suffer.

Inverted index. One might consider fragment id coordinates, and build inverted indices on id coordinates with the canonical labeling as keywords. To prune unpromising graph fragments for a query fragment q_s, we have to verify all graph fragments in $\mathcal{D}_s$. Since updates on node labels never change id coordinates, these indices take little index update cost; however, the size of $\mathcal{D}_s$ is usually large (*e.g.,* tens of millions), so a thorough scan will slow down query processing.

Grid index. One might apply grid indices to allow affordable update operations [45, 163]. The general idea is as follows. (1) The multi-dimensional space of label coordinates are partitioned into grids. (2) The fragments in the same grid are managed with a light-weight data structure (*e.g.,* list). (3) If a fragment label coordinate is updated, update operations will be conducted only when the updated coordinate moves out of the original grid. (4) When a query fragment arrives, we issue a range query based on its label coordinate: (a) mark those fragments in the grids that are fully covered by the range query as candidates; and (b) verify those fragments in the grids that are partially covered by the range query. Note that the search speed depends on the amount of time taken by verification. If we apply a naive method that compares the targeted graph fragments with the query fragments, it will take a considerable amount of time. Consider the following example.

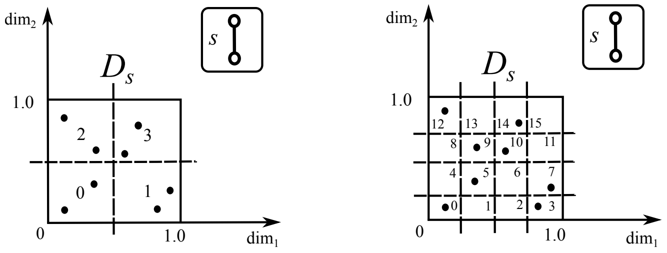
Example 6. *10M graph fragments of the structure 3-star (*i.e., *4D coordinates) are managed by a grid index with 10,000 grids. Suppose graph fragments are uniformly distributed, each grid covers 1,000 fragments. A 10-star query has 720 query fragments of the structure 3-star. Suppose a query fragment is uniformly distributed, it will partially cover 505 grids in average. Therefore, a query fragment needs up to* $4 * 505 * 1000 = 2.02M$ *comparisons to complete verification, and in total we need up to* $720 * 2.02M = 1454.4M$ *comparisons for one indexed structure.*

In Example 6, we consider the case of node labels and single label per node. In a general case of edge labels and multiple labels per node/edge, the number of comparisons will proportionally increase. Therefore, the verification phase will slow down by a large amount of comparisons.

In the following discussion, we introduce an index FracFilter that addresses both update and search issues. FracFilter is a grid-based index that inherits the merit of update efficiency; moreover, we propose a verification algorithm that accelerates search by avoiding redundant comparisons.

4.4.2 FracFilter construction

We start with the construction of FracFilter. For an indexed structure, a FracFilter is constructed by two steps: (1) it partitions the label coordinate space into grids, and (2) maps graph fragments into the corresponding grids.



(a) $\lambda = 2$

(b) $\lambda = 4$

Figure 4.4: FracFilters of density 2 (left) and 4 (right): s in the top right corner is the structure of $\mathcal{D}_s$, points are label coordinates, and the integer in each grid is the grid id.

Partition the space into grids. Let λ be a positive integer, called *grid density*, n_s be the number of fragments with structure s, and d be the number of dimensions for fragments' label coordinates. One partition strategy is to uniformly slice label coordinate space into λ parts for each dimension; however, this strategy might result in unacceptable index searching and updating performance when label coordinate distribution is skew. Therefore, we consider to use the empirical distribution of label coordinates in each dimension to partition the space: (1) slice each dimension into λ parts, and each part contains $\frac{n_s}{\lambda}$ fragments; and (2) independently repeat this procedure in each dimension. In this way, we obtain λ^d d-dimensional grids in total. Moreover, each grid is associated with a grid id represented by a base-λ integer. Suppose the i-th dimension of a grid falls into the j-th partition, then the i-th bit of the grid id is j. The advantages of using the above way to assign a grid id include (1) the ease of designing fragment mapping functions, and (2) the ease of avoiding redundant comparisons (discussed in Section 4.4.3).

Figure 4.4 demonstrates two FracFilters of density 2 and 4, respectively, on a 2-dimensional space: (1) grids are disjoint, (2) a point (fragment) is covered by one and only one grid, and (3) the whole space is covered.

Map fragments into grids. Let g be a graph fragment, $x(g) = (x_1, x_2, ..., x_k)$ be g's fragment coordinate, and *gid* be the id of the grid to which g should be mapped. The mapping function is designed as follows. (1) Starting with x_1, if x_1 falls into the j_1-th partition, we set the first bit of *gid* to be j_1. (2) At the i-th dimension, if x_i falls into the j_i-th partition, we set the i-th bit of *gid* to be j_i. (3) Repeat this process for all dimensions. we use lists to manage fragments in grids.

The pseudo code of the construction algorithm is shown in Figure 4.5 for reference. The above construction algorithm shows that a FracFilter can be constructed in linear time, and the following result indicates the computation complexity.

Proposition 7. *Let λ be the grid density, n_s be the number of fragments, and d be the number of dimensions of a label coordinate space. We can construct a* FracFilter *in $O(max\ (\lambda^d, dn_s))$.*

Remark. (1) The construction algorithm will be executed once for each indexed structure. In other words, if we index 5 structures and obtain 5 sets of graph fragments, the construction algorithm will be executed for 5 times, and each run builds a FracFilter for the corresponding structure. (2) We define the grid density λ and divide each dimension into λ partitions for the ease of discussion. Indeed, with

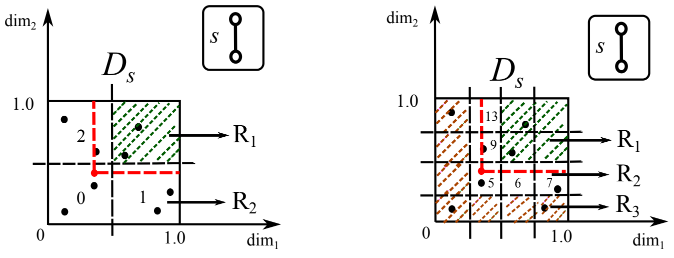
```
Input:  (1) grid density $\lambda$;
        (2) the number of dimensions $d$;
        (3) a list of label coordinates of $\mathcal{D}_s$, $frag$;
Output: a FracFilter, $filter$.

$grid$.resize($\lambda^d$)
$locX$.resize(len($frag$)), $locY$.resize(len($frag$))
for $i$ in range(0, len(frag))
    $gid$ = gridID($frag[i]$)
    $grid[gid]$.append($frag[i]$)
    $locX[i] = gid$
    $locY[i]$ = len($grid[gid]$) $- 1$
return $filter$ ($grid$, $locX$, $locY$, $d$, $\lambda$)
```

Figure 4.5: The Algorithm sketch for constructing a FracFilter

little modification, the construction algorithm along with its theoretical results works in the cases where each dimension is divided into a variable number of partitions.

4.4.3 Searching in FracFilter

In this section, we present how a FracFilter avoids unnecessary comparisons and accelerates search for a query fragment. When a query fragment arrives, we formulate a range query that is a multi-dimensional box with the query fragment's label coordinate as the bottom corner and $(1.0, 1.0, ..., 1.0)$ as the top corner. The range query divides the label coordinate space into three regions R_1, R_2, and R_3: (1) R_1 contains the grids that are fully covered by the range query; (2) R_2 contains the grids that are partially covered by the range query; and (3) R_3 contains rest of the grids. Figure 4.6 gives examples of R_1, R_2, and R_3. In particular, we mark fragments in R_1 as candidates, discard fragments in R_3, and verify fragments in R_2.



(a) $\lambda = 2$

(b) $\lambda = 4$

Figure 4.6: The same query fragment (the red dot) requests fragment searching on two FracFilters of density 2 (left) and 4 (right).

For graph fragments in R_2, a naive verification algorithm blindly makes comparisons in every dimension; however, for some dimensions, comparisons are unnecessary. Consider an example in Figure 4.6(b). A query fragment falls into grid 5 (11_4 in base-4 form), and grid $5(11_4)$, $6(12_4)$, $7(13_4)$, $9(21_4)$, and $13(31_4)$ are in R_2. For fragments in grid $5(11_4)$, we have to take all dimensions into consideration for verification. However, for fragments in grid $6(12_4)$ and $7(13_4)$, we only need to consider dim_2 since their label values in dim_1 is surely greater; and similarly, for fragments in grid $9(21_4)$ and $13(31_4)$, we only need to consider dim_1.

Let c_q be the grid where a query fragment falls, and c be a grid in R_2. We obtain the following pruning rule.

Lemma 1. *Comparisons at the i-th dimension are necessary, only if the i-th bit of c_q's id equals the i-th bit of c's id.*

A natural question is how many comparisons we can avoid from this rule. Suppose that (1) label coordinates of graph fragments are uniformly distributed in grids and (2) a query fragment's label coordinate is uniformly distributed in grids as well, the following result shows the expected number of comparisons a FracFilter makes for verification.

Theorem 4. *Given λ, grid density, d, the number of dimensions, and n_s, the number of graph fragments, the expected number of extra comparisons for an arbitrary query fragment is $\frac{dn_s}{\lambda^d}\left(\frac{\lambda+1}{2}\right)^{d-1}$.*

Proof. We show the expected number of comparisons a FracFilter requires for an arbitrary query fragment. (1) The probability that a query fragment falls into grid c_q with a base-λ id $a_d a_{d-1} \cdots a_1$ is $\frac{1}{\lambda^d}$. (2) The number of grids in R_2 that need d more comparisons for each graph fragment is 1 (c_q itself), the number of grids that need $d-1$ more comparisons is $\sum_{j=1}^{d}(\lambda - a_j - 1)$, and in general, the number of grids that need $d-k$ more comparisons for each graph fragment is $\sum_{(j_1,j_2,\cdots,j_k)}\prod_{i=1}^{k}(\lambda - a_{j_i} - 1)$ where $(j_1, j_2, \cdots, j_k)$ enumerates all k-combinations of $(1, 2, \cdots, d)$. Taking summation over all possible c_q, the number of grids that need $d-k$ more comparisons is $\binom{d}{k}\lambda^{d-k}\left[\frac{\lambda(\lambda-1)}{2}\right]^k$. Thus, the expected number of comparisons is derived by

$$\begin{aligned} EX &= \frac{n_s}{\lambda^d}\frac{1}{\lambda^d}\sum_{k=1}^{d} k \cdot \binom{d}{k} \lambda^k \Big[\frac{\lambda(\lambda-1)}{2}\Big]^{d-k} \\ &= -\frac{n_s}{\lambda^d}\frac{(\lambda-1)^{d+1}}{2^d}\left[\Big(\frac{2}{\lambda-1}+1\Big)^{d}\right]' \\ &= \frac{dn_s}{\lambda^d}\Big(\frac{\lambda+1}{2}\Big)^{d-1}. \end{aligned}$$

Therefore, Theorem 4 is proved. □

Consider an inverted index that scans the whole set of graph fragments taking dn_s comparisons, and a naive verification algorithm on a grid index that scans $\frac{dn_s}{\lambda^d}\big[\big(\frac{\lambda+1}{2}\big)^d - \big(\frac{\lambda-1}{2}\big)^d\big]$ in average (the derivation is similar to the proof of Theorem 4). The ratio from the number of comparisons made by a FracFilter to the number by the inverted index is $\frac{d}{\lambda^d}\big(\frac{\lambda+1}{2}\big)^{d-1}$; similarly, the ratio from a FracFilter to the naive verification algorithm is $\frac{2(\lambda+1)^{d-1}}{(\lambda+1)^d-(\lambda-1)^d}$. In other words, when $d = 7$ (a 3-star fragment with single node and edge labels), and $\lambda = 25$, this ratio is lower than 0.005 for the inverted index, and lower than 0.18 for the naive verification algorithm.

Remark. (1) Theorem 4 demonstrates that a FracFilter of a larger grid density has a faster pruning speed in average. In particular, when $\lambda^d \leq \frac{n_s}{2}$, the first derivatives of the above ratios will be negative so that the *efficiency* will increase if λ increases; moreover, when $\lambda^d \leq \frac{n_s}{2}$, the second derivatives of the above ratios will be positive so that the *efficiency gain* will diminish if λ increases. (2) Although in practice graph fragments might not strictly uniformly distributed in grids, our experimental results show that FracFilter performs well with both real and synthetic fragment distributions in Section 4.6.

4.4.4 Index update in FracFilter

In this section, we discuss the update operations in FracFilter. When a graph fragment is updated, it triggers one of the two events in FracFilter: *bounded* or *migration*. (1) If an update triggers *bounded*, the fragment stays in the same grid. (2) If an update triggers *migration*, the fragment moves out of the old grid, and moves into another one.

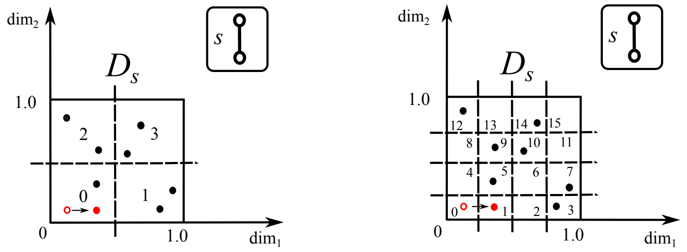
Given a fragment update, FracFilter is updated in two steps: (1) find which grid should accommodate the updated fragment; and (2) decide which event this update triggers and take the corresponding action to update FracFilter. In the second step, if the update triggers event *bounded*, it takes no update operation; if the update triggers event *migration*, it takes two operations: (a) delete the fragment from the old grid's fragment list, and (b) insert the updated fragment into the right grid.

Suppose the label coordinate of an updated fragment is uniformly distributed in the space, we obtain the following complexity result for index update in FracFilter.

Theorem 5. *Given d is the number of dimensions and λ is the grid density,* FracFilter *takes $2(1 - \frac{1}{\lambda^d})$ operations per update in average.*

Proof. For an arbitrary update, the probability of staying in the original grid is $\frac{1}{\lambda^d}$. Thus, the expected number of operations an update takes is $0 \cdot \frac{1}{\lambda^d} + 2 \cdot (1 - \frac{1}{\lambda^d})$. □

Remark. Theorem 5 suggests that a FracFilter of smaller grid density λ is more likely to take fewer update operations. In Figure 4.7, we make the same update to the fragment in the bottom-left corner: (1) the update in the FracFilter of $\lambda = 2$



(a) $\lambda = 2$

(b) $\lambda = 4$

Figure 4.7: An update on FracFilter of density 2 (left) and 4 (right), respectively. When a fragment in bottom left corner is updated, it triggers a bounded event on the left, but a migration event on the right.

triggers a *bounded* event, and requires no operation (Figure 4.7(a)); however, (2) the update in the FracFilter of $\lambda = 4$ triggers a *migration* event requiring a deletion and an insertion on lists (Figure 4.7(b)). Indeed, if an update triggers a *migration* event in a FracFilter of smaller grid density, it must triggers a *migration* event in a FracFilter of larger grid density; on the other hand, if an update triggers a *migration* event in a FracFilter of larger grid density, it might only trigger a *bounded* event in a FracFilter of smaller grid density. Note that a larger grid density λ brings better search performance at the cost of higher memory usage and slower update performance. In practice, one can try different λ, and use the one that best fit the application.

4.5 Optimize query processing

In this section, we discuss how to accelerate subgraph matching at the fragment join phase. Various heuristics have been proposed to join paths or two-level trees in a selected order such that the amount of redundant intermediate results is reduced [173, 201]. Different from these studies, we deal with general subgraphs in this work. In particular, we need to address two critical issues. First, we need to define the join selectivity of general subgraphs, and propose an algorithm to find a set of query fragments that minimizes redundancy. Second, in a considerable number of cases, no matter which join order we apply, a naive join will take a long time. Consider the following example.

Example 7. *C_{q_1}, C_{q_2}, and C_{q_3} are the candidate sets of query fragment q_1, q_2, and q_3, and the size of these candidate sets is uniformly 10^6. In the data graph, there is no matched subgraph, but we do not know it when we conduct the join. In this case, no matter how we place the join order, a naive join for the first two candidate sets will take 10^{12} comparisons.*

In this paper, we propose a two-step method to address the above issue: (1) we use *minimum fragment cover* to find a set of query fragments whose candidate sets potentially involve the minimum amount of redundant intermediate results; and (2) *fingerprint based pruning* is applied to prune redundant comparisons between a pair of candidate sets.

4.5.1 Minimum fragment cover

Minimum fragment cover finds a small set of selective query fragments with small candidate sets to reduce computation cost at the join phase, with the constraint that the result of fragment join over this small set of query fragments is equivalent to that over the whole set of query fragments. There are two intuitions behind minimum fragment cover. (1) We only need a subset of query fragments that jointly cover all nodes and edges in the query graph, and we refer to such a subset as a *fragment cover*. (2) As there are multiple ways to select fragment covers, one might prefer to take the one of a smaller search space for fragment join. In the following, we define an optimization problem that implements the above intuition.

Given a fragment cover $\{q_1, q_2, ..., q_k\}$ and their corresponding candidate sets $\{C_{q_1}, C_{q_2}, ..., C_{q_k}\}$, the joint search space size is bounded by $\exp(J)$,

$$J = \log(\prod_{i=1}^{k} |C_{q_i}|) = \sum_{i=1}^{k} \log(|C_{q_i}|).$$

Indeed, one may prefer a fragment cover that optimizes the upper bound J. Therefore, with J as the objective function, we define the minimum fragment cover problem as follows.

Definition 3 (Minimum fragment cover)**.** *Given a query graph Q with its whole set of query fragments $\{q_1, q_2, ..., q_l\}$ and their candidate sets $\{C_{q_1}, C_{q_2}, ..., C_{q_l}\}$, a* minimum fragment cover *is a subset of query fragments $\{q_{i_1}, q_{i_2}, ..., q_{i_k}\}$ such that (1) this subset is a fragment cover, and (2) its corresponding J is minimum.*

However, the following result indicates that it is difficult to find the *minimum fragment cover* in polynomial time. Instead, one can obtain an approximated solution in polynomial time with approximation guarantee.

Theorem 6. *The* minimum fragment cover *problem is NP-complete; however, there exist greedy algorithms with an approximation ratio $O(\ln n)$, where n is the sum of the number of nodes and the number of edges.*

Proof. To prove the NP-completeness of the minimum fragment cover problem, one could reduce an arbitrary set cover instance [73] to a minimum fragment cover instance by (1) constructing a depth-1 tree where each leaf maps to an element in the ground set and (2) constructing a smaller depth-1 tree for each set where a leaf maps to an element contained by the set. One can verify that this transformation runs in polynomial time, and a solution to the transformed instance is a solution to the original set cover instance. Since the minimum fragment cover problem is NP, the NP-completeness of the problem follows. Moreover, an arbitrary minimum fragment cover instance can be easily transformed into a set cover instance. Therefore, the approximation guarantee $O(\ln n)$ for set cover [73] can be applied to minimum fragment cover. In sum, the correctness of Theorem 6 is proved. □

4.5.2 Fingerprint based pruning

The intuition of fingerprint based pruning includes two aspects. (1) Given a fragment cover, it is preferable to join fragments in an order such that it results in connected subgraphs at every intermediate step. Indeed, disconnected subgraphs at intermediate step will lead to an explosion of the search space. To obtain a connected intermediate subgraph, two query fragments at any intermediate step have to share a set of nodes. In other words, these overlapping and non-overlapping nodes form the join conditions for graph fragments. (2) Suppose join operations are conducted between two candidate sets C_{q_1} and C_{q_2} where q_1 and q_2 share several common nodes. Let g_i be a graph fragment from C_{q_1}. It is very likely that only a small portion of graph fragments in C_{q_2} share the required common nodes with g_i; meanwhile, only this small portion of graph fragments are worth checking. Therefore, instead of linearly scanning C_{q_2}, it is preferable for g_i to only check those graph fragments of the required common nodes. With this spirit, we propose *fingerprint based pruning* that (1) extracts the required common nodes for fragment join, (2) makes *fingerprints* based on these common nodes, and (3) prunes redundant search if two fragments have different fingerprints.

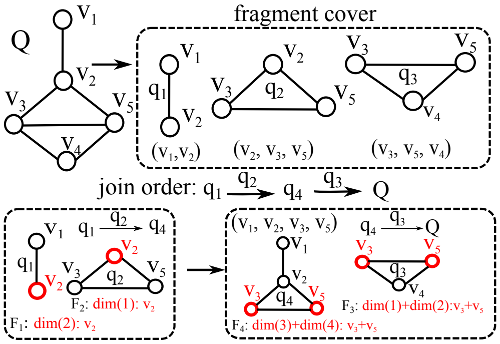


Figure 4.8: An example of fingerprint based pruning

To illustrate how we perform fingerprint based pruning, an example is presented in Figure 4.8.

First, for a query graph Q, three query fragments q_1, q_2, and q_3 form a fragment cover, and their id coordinates are (v_1, v_2), (v_2, v_3, v_5), and (v_3, v_5, v_4), respectively.

Second, the order of fragment join is as follows. (a) Join the candidate sets of q_1 and q_2; and (b) join the intermediate subgraphs from (a) with the candidate set of q_3.

Third, starting with q_1 and q_2, they share v_2 that is the *second* dimension of q_1's id coordinate, and the *first* dimension of q_2's id coordinate. Thus, for each graph fragment g_i in the candidate set of q_2, the fingerprint of g_i is constructed by the node id at the *first* dimension of its id coordinate. Using fingerprints as keys, graph fragments from q_2's candidate set are organized by an inverted index. Given a graph fragment g_j from C_{q_1}, we first extract its fingerprint by the node id at the *second* dimension of its id coordinate. With its fingerprint, we search q_2's inverted index, and only check those graph fragments sharing g_j's fingerprint.

Fourth, at intermediate steps, a similar procedure is conducted. As shown in Figure 4.8, the intermediate query graph q_4 is obtained by joining q_1 and q_2, and we next join q_4 with q_3. The common nodes of q_4 and q_3 are node v_3 and node v_5. v_3 and v_5 are the nodes at the *third* and the *fourth* dimension of q_4's id coordinate; meanwhile, they are at the *first* and *second* dimension of q_3's id coordinate. Similarly, we obtain the fingerprints of q_3's candidate fragments, and organize them by an inverted index. For a graph fragment g_i of q_4, we first extract g_i's fingerprint, locate those graph fragments sharing its fingerprint by q_3's inverted index, and only check those promising graph fragments.

In addition, given a fragment cover, the join order is determined by a heuristic algorithm. We start with the query fragment of the smallest candidate set in the fragment cover. At each step, we select the query fragment that shares common nodes with the query fragments that have been joined. If there exist multiple such query fragments, we select the one with the smallest candidate set. We repeat this process until all the query fragments in a fragment cover are joined.

4.6 Experiments

Using real and synthetic data, we conducted three sets of experiments to evaluate the performance Gradin. Both real and synthetic data are used to evaluate Gradin's performance on query processing and index construction/update. The synthetic data is used to study its scalability.

Gradin is implemented in C++. All experiments were executed on a machine powered by an Intel Core i7-2620M 2.7GHz CPU and 8GB of RAM, using Ubuntu 12.10 with GCC 4.7.2. Each experiment was run 10 times. In particular, for query processing, each run includes 100 query graphs. For all experiments, their average results are presented.

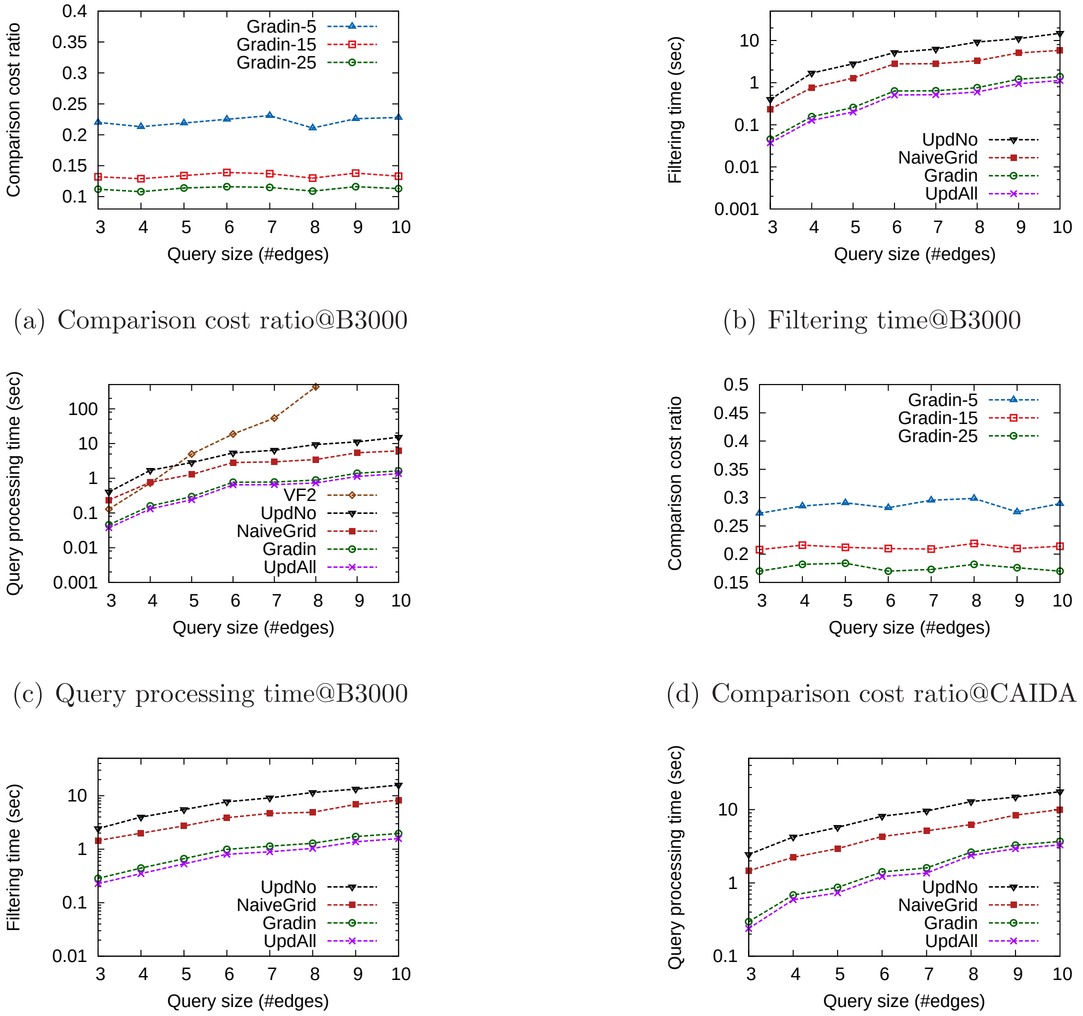
4.6.1 Experiment setup

Data graphs. We used the following network topologies as data graphs. (1) BCUBE is a network architecture for datacenters [87]. We generated BCUBE networks as follows. (a) The number of nodes in the networks ranges from $3,000$ to $15,000$ with step $2,000$; and (b) the average degree of a network is between 18 and 20. In particular, a BCUBE network of $3,000$ nodes was used to compare Gradin with its baselines, and the rest networks were used to evaluate Gradin's scalability. (2) CAIDA[1] dataset contains 122 Autonomous System (AS) graphs, from January 2004 to November 2007. In particular, we used the largest AS graph of $26,475$ nodes and $106,762$ edges to compare query processing and indexing performance between Gradin and its baselines.

Numerical labels and updates. We obtained the numerical labels and their updates from the ClusterData[2]. It contains the trace data from about 11k machines over about a month-long period in May 2011 [151]. For each machine, we extracted its CPU and memory usage traces, and each trace is represented as a sequence of normalized numerical values between 0 and 1. Moreover, we

[1] http://snap.stanford.edu/data/as-caida.html

[2] http://code.google.com/p/googleclusterdata/wiki/ClusterData2011_1



(a) Comparison cost ratio@B3000

(b) Filtering time@B3000

(c) Query processing time@B3000

(d) Comparison cost ratio@CAIDA

(e) Filtering time@CAIDA

(f) Query processing time@CAIDA

Figure 4.9: Query processing performance on B3000 and CAIDA with 100 compatible subgraphs returned

randomly mapped a cluster machine to a node in a data graph, and obtained numerical labels on nodes along with their updates. In particular, we applied the node labels/updates from ClusterData to the evaluation for *query processing* (Section 4.6.2) and *indexing performance* (Section 4.6.3).

In order to explore how our technique performs on different numerical label distributions, we generated labels and updates from statistical distributions of estimated parameters (using ClusterData as sample data). The generated labels are applied to the evaluation for *scalability* in Section 4.6.4.

Query graphs. As query graphs are usually small [184, 201], we consider all possible connected graphs of 3 to 10 edges as possible queries, and the numerical labels on nodes are randomly drawn from 0 to 1.

Baselines. Five baselines are considered: VF2, UpdAll, NaiveGrid, NaiveJoin, and UpdNo. (1) VF2 [58] is a state-of-the-art subgraph search algorithm without any index. (2) UpdAll indexes fragment label coordinates with multi-dimensional search trees. This index enables fast candidate search; however, update operations are costly on the search tree. In particular, we implemented two versions of search trees: (a) a multi-dimensional binary tree of better update processing performance is used to compare indexing performance in Section 4.6.3; and (b) an R-tree [88] of faster query processing performance is used to compare query processing performance in Sections 4.6.2 and 4.6.4. (3) NaiveGrid is a grid index with a naive verification algorithm. (4) NaiveJoin uses FracFilter at the filtering phase, but a naive join is applied for fragment join. (5) UpdNo is an inverted index where each entry points to a list of fragments of the same structure. It never updates index; however, it requires a large amount of comparisons for searching candidate fragments.

4.6.2 Query processing

In the first set of experiments, we investigated the query processing performance of Gradin on the BCUBE graph of 3,000 nodes (B3000) and the largest CAIDA graph (CAIDA).

Since a service placement task might require multiple compatible subgraphs to be returned [34, 127], in the experiments, the number of returned compatible subgraphs r is set to be 5, 10, or 100. Formally, given a graph G, taking a query graph Q as input, we find up to r compatible subgraphs.

Pruning power. We evaluate the pruning power of Gradin, UpdAll, NaiveGrid, and UpdNo, by measuring *comparison cost ratio* and *filtering time*. (1) *Comparison cost ratio* defines the ratio from the amount of label comparisons executed by Gradin to the amount by NaiveGrid. In particular, the *comparison cost ratio* is 100% for NaiveGrid. (2) *Filtering time* defines the total amount of time spent in searching candidate fragments. All the evaluations are conducted by varying query graph size (the number of edges).

On B3000, Figure 4.9(a) and Figure 4.9(b) present the pruning power of UpdAll, UpdNo, NaiveGrid, and Gradin: Figure 4.9(a) presents the comparison cost ratio of Gradin with grid density 5, 15, and 25 (referred to as Gradin-5, Gradin-15, and Gradin-25), respectively, while Figure 4.9(b) only presents the filtering time of the Gradin with grid density 25 (it always outperforms the other two variants). On the one hand, when query size increases, the Gradin variant of a greater grid density receives better pruning power; however, the power gain is diminishing when grid density grows. On the other hand, the filtering time of all algorithms increases, when graph query size increases. In particular, (1) as query size increases from 3 to 10, the *filtering time* of Gradin increases from 0.046 to 1.387 seconds, which is close to the performance of UpdAll (later, we will show that UpdAll is 4-10 times slower in index construction and update.); and (2) in terms of filtering time, Gradin is up to 10 times faster than UpdNo, and is around 5 times faster than NaiveGrid.

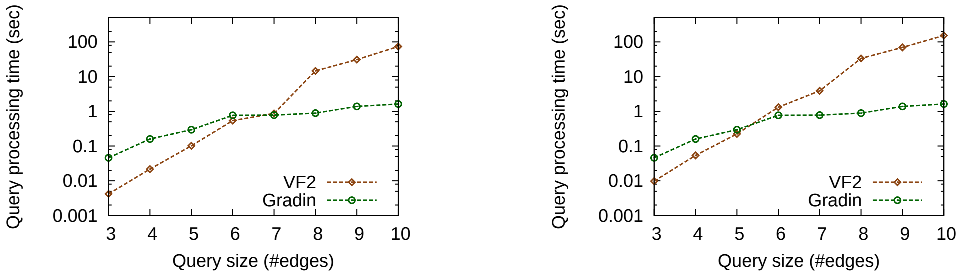
On CAIDA, Figure 4.9(d) and Figure 4.9(e) present consistent results: (1) Gradin with grid density 25 has the best pruning power, but the power gain is diminishing when grid density grows; and (2) in terms of filtering time, Gradin is close to UpdAll, is 10 times faster than UpdNo, and is up to 5 times faster than NaiveGrid.

In sum, the above results verify our theoretical analysis in Theorem 4.

Query processing time. We evaluate the total *query processing time* of Gradin, UpdAll, UpdNo, NaiveGrid, and VF2. The total *query processing time* includes query decomposition time, filtering time, and fragment join time. In addition, all the evaluations are conducted by varying query graph size (the number of edges).

First, we studied the total query processing time of different techniques, and set the number of returned compatible subgraphs to be 100. Figure 4.9(c) and 4.9(f) present the total query processing time on B3000 and CAIDA, respectively. Since Gradin with grid density 25 always outperforms Gradin with density 5 or 15, we only present the query processing time of Gradin with grid density 25. On both data graphs, we observe two common trends: (1) Gradin processes queries as fast as UpdAll does; (2) Gradin outperforms UpdNo up to 10 times; and (3) Gradin is up to 5 times faster than NaiveGrid. In particular, on B3000, as the query size grows from 3 to 10, the query processing time of Gradin increases from 0.046 to 1.62

seconds, while UpdNo's query processing time increases from 0.40 to 15.2 seconds, and NaiveGrid's query processing time increases from 0.27 to 6.12 seconds, which achieves up to 10 times and 5 times speedup, respectively; meanwhile, on CAIDA, as the query size grows, the query processing time of Gradin increases from 0.29 to 1.98 seconds, while UpdNo's query processing time increases from 2.44 to 17.5 seconds, and NaiveGrid's query processing time increases from 1.46 to 9.98 seconds, which is around 8 times and 5 times speedup, respectively. Moreover, VF2 cannot scale on both B3000 and CAIDA: (1) on B3000, when the query size is more than 8, VF2 cannot process 100 queries within 12 hours; and (2) on CAIDA, even when the query size is 3, VF2 cannot process 100 queries within 6 hours. In the case of NaiveJoin, within 6 hours, it cannot process 100 queries of size 4 on B3000, and cannot process 100 queries of size 3 on CAIDA (not shown).



(a) 5 subgraphs returned

(b) 10 subgraphs returned

Figure 4.10: Query processing on B3000 returning 5 or 10 matches

Second, we investigated how VF2, NaiveJoin, and Gradin perform when the number of returned subgraphs is smaller by setting the number of returned subgraphs to be 5 or 10. The query processing time on B3000 is presented in Figure 4.10(a) and Figure 4.10(b). Two observations are made: (1) VF2 is faster when the query size is small; and (2) Gradin performs better when the query size is relatively large (*e.g.,* $\geq$ 8 edges). When a query graph is smaller, the number of matched subgraphs in a data graph is larger in general, and the amount of redundant search for VF2 is smaller as well. Since VF2 does not conduct a global pruning as Gradin does, VF2 earns a better performance when the query size is small. However, when a query graph is relatively large, the number of matched subgraphs decreases, and VF2 cannot successfully skip the redundant search. Meanwhile, the pruning power of Gradin is pronounced, especially when the query size gets larger. In particular, when the query size is 10, Gradin performs around 50 times better than VF2 if 5 subgraphs are returned, and around 100 times better if the number of returned subgraphs is 10. On CAIDA, even when the number of returned subgraphs is 5, VF2 cannot process 100 queries of size 3 within 6 hours (not shown); however, Gradin is able to process 100 queries of size 10 within 6 minutes (*i.e.,* 3.6 seconds per query). In the case of NaiveJoin, even when the number of returned subgraphs is 5, within 6 hours, it cannot process 100 queries of size 4 on B3000, and cannot process 100 queries of size 3 on CAIDA (not shown).

Remarks. (1) When a query graph's structure is indexed, Gradin can directly answer the query without fragment join. For example, we can answer query graph of size 3 on B3000 after the filtering phase. (2) We indexed fragments of no more than 3 edges on B3000, while the indexed fragments on CAIDA have no more than 2 edges. On B3000, as the query size increases from 4 to 10, the fragment join time increases from 0.004 to 0.20; meanwhile, on CAIDA, the fragment join time increases from 0.011 to 1.72, when the query size grows from 3 to 10. This

Table 4.1: Index construction time (sec)

Data graph	UpdAll	Gradin-5	Gradin-15	Gradin-25
B3000	85.3	18.5	19.6	21.1
CAIDA	61.6	17.1	17.4	17.5

shows that our fragment join algorithms effectively prune redundant search. The experiments also show that as less sophisticated structures are indexed, it usually takes more time on fragment join, since we need more query fragments to cover a query graph, which results in a larger number of join operations. (3) As the fragment join time is largely reduced by our optimization techniques, the importance of reducing *filtering time* is highlighted. Using FracFilter, Gradin successfully reduces filtering time.

4.6.3 Indexing performance

In the second set of experiments, we investigated the indexing performance of Gradin on the BCUBE of 3000 nodes (B3000) and the largest CAIDA graph (CAIDA). In particular, we built Gradin variants of grid density 5, 15, and 25, referred to as Gradin-5, Gradin-15, and Gradin-25.

Index construction. We evaluate *index construction time* and *index size* of UpdAll and Gradin variants. *Index construction time* measures the amount of time spent in building index after graph fragments are mined, and we separately report the amount of time for mining graph fragments.

On B3000, we built Gradin variants and UpdAll based on fragments of no more than 3 edges. It took us 370 seconds to mine fragments from B3000, obtaining

Table 4.2: Index size (MB)

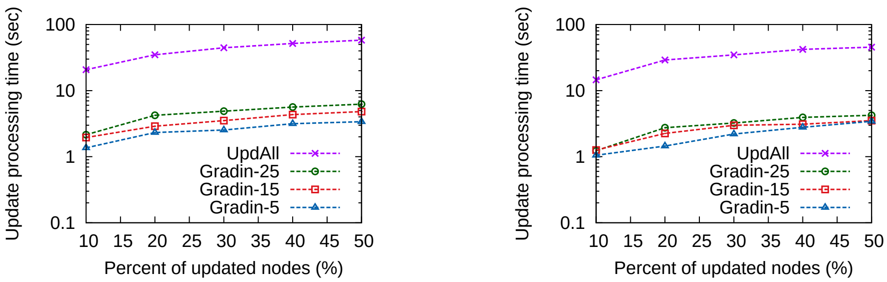
Data graph	UpdAll	Gradin-5	Gradin-15	Gradin-25
B3000	1644	607	610	615
CAIDA	1486	583	585	589

fragments of 5 different structures. Index construction time and index size of UpdAll and Gradin variants are shown in Tables 4.1 and 4.2, respectively. All Gradin variants take less time and space to build indices on B3000.

On CAIDA, we built Gradin variants and UpdAll based on fragments of no more than 2 edges, and mined fragments of 2 different structures including one-edge and two-edge paths in 268 seconds. The performance results are reported in Tables 4.1 and 4.2, respectively. All Gradin variants take less time and space to build indices on CAIDA as well.

Index update. We evaluate index update performance of Gradin and UpdAll by measuring their *update processing time* while varying the *percent of updated nodes*. Given a data graph, the *percent of updated nodes* is the percentage of nodes whose labels are updated. Since a node might appear in multiple fragments, if the label of a node is updated, multiple fragments might be simultaneously updated as well. Given a set of updated nodes, we first find the corresponding updated fragments, and then apply those fragment updates to a graph index. The time spent in updating a graph index is referred to as *update processing time*.

Figure 4.11(a) presents the update processing time of UpdAll, Gradin-5, Gradin-10, and Gradin-25 on B3000: (1) Gradin variants outperform UpdAll, and (2) when the percent of updated nodes increases, the update processing time of all algo-



(a) Index update@B3000

(b) Index update@CAIDA

Figure 4.11: Update processing time on B3000 and CAIDA

rithms increases. In sum, Gradin variants are about 10-20 times faster than UpdAll on B3000.

Figure 4.11(b) shows update processing time on CAIDA. The trends are consistent with those shown in Figure 4.11(a), and Gradin variants are 13-20 times faster than UpdAll on CAIDA.

Remarks. (1) As we consider more complex structures to build graph indices, it usually takes more time on index update. For both UpdAll and Gradin, a more complex structure increases the number of dimensions of the corresponding search index, which in turn increases update complexity. (2) When Gradin-5 processes updates, about 40% of updates trigger *bounded* events, which requires no update operations on Gradin. However, this ratio drops from 40% to 15%, as we apply Gradin-25. Even though Gradin-25 takes more time to process updates that trigger *migration* events, it is still efficient since update operations are merely insertions and deletions on lists. In contrary, UpdAll processes all updates with costly insertions and deletions on multi-dimensional search trees. It is Gradin's partial update strategy and inherent low-cost update operations that make Gradin more efficient on index update.

4.6.4 Scalability

In the third set of experiments, we investigated the scalability of Gradin by varying the size of a BCUBE graph.

Constrained by security policies and communication latency, a service placement usually considers a few racks in a datacenter[3] [77, 127]. Suppose there are 40 machines in each rack [4], we ranged the number of racks in a datacenter from 125 to 375[4], and generated six BCUBE graphs of $5,000$, $7,000$, $9,000$, $11,000$, $13,000$, and $15,000$ nodes.

Node labels and updates were generated from beta distributions $B(a, b)$ of estimated parameters. The reasons why we employed beta distributions include (1) per machine attributes (*i.e.,* available CPU time/memory) in ClusterData are normalized between 0 and 1, while beta distribution is one of the widely-used distributions that could generate random numbers in such an interval; and (2) there exist efficient algorithms to estimate parameters a and b from sample data[5]. In particular, the estimated parameters based on machine CPU time are $a \approx 7.91$ and $b \approx 7.03$, and the estimated parameters based on machine memory are $a \approx 2.42$ and $b \approx 2.73$. Moreover, the corresponding distributions are referred to as SYNCPU and SYNMEM, respectively.

In addition, the grid density of Gradin is set to 25, and the indexed structures have no more than 2 edges.

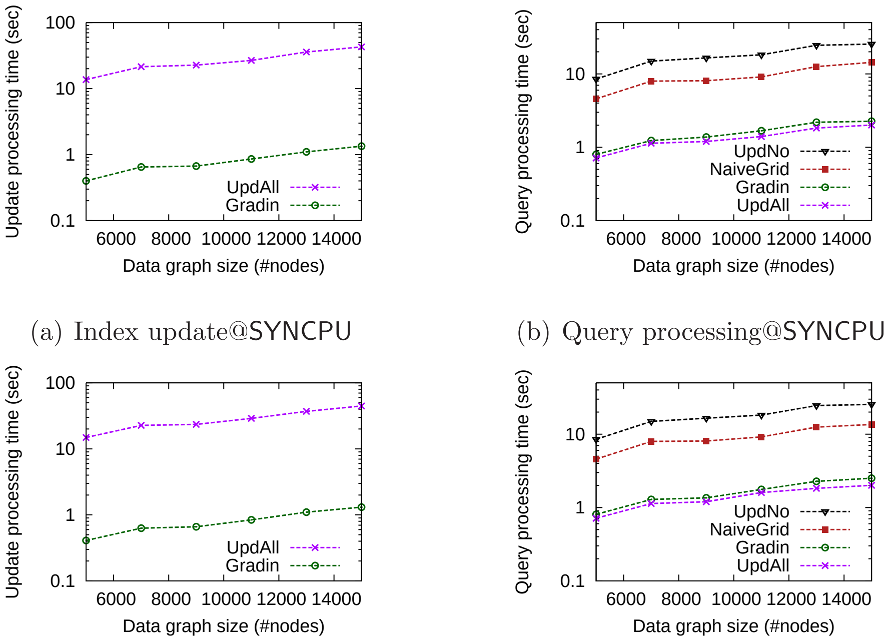
[3] http://msdn.microsoft.com/en-us/library/windowsazure/jj717232.aspx

[4] http://news.cnet.com/8301-10784_3-9955184-7.html

[5] http://en.wikipedia.org/wiki/Beta_distribution

Table 4.3: Construction time and index size of Gradin

Graph size	5K	7K	9K	11K	13K	15K
T_{mine} (sec)	49	89	142	227	357	513
T_{index} (sec)	2.8	4.0	4.3	6.2	7.0	7.8
Size (MB)	65	100	111	176	202	229



(a) Index update@SYNCPU

(b) Query processing@SYNCPU

(c) Index update@SYNMEM

(d) Query processing@SYNMEM

Figure 4.12: Scalability on BCUBE graphs of 5K - 15K nodes

Index construction. We evaluated index construction time and index size of Gradin by varying data graph size. As shown in Table 4.3, When the graph size increases, for a data graph of 15,000 nodes with more than 150,000 edges, Gradin is constructed in less than 10 minutes (including fragment mining), taking 229MB memory.

Index update. We evaluated the index update performance of Gradin by varying data graph size. In particular, the percent of updated nodes is fixed to 30%. As seen in Figure 4.12(a) and 4.12(c), Gradin takes less than 2 seconds to process more than 9M updates, outperforms UpdAll in all cases, and the resulting speedup is up to 20 times. Indeed, when data graph size gets larger, Gradin can efficiently process updates.

Query processing. We evaluated the query processing performance of Gradin by varying data graph size. In particular, the query size (the number of edges) is fixed to 7, and Gradin returns the first 100 distinct compatible subgraphs. Since VF2 and NaiveJoin cannot scale on the query graphs of size 7 (they cannot process 100 query graphs within 6 hours), Figure 4.12(b) and 4.12(d) only reports the query processing time of Gradin, UpdAll, NaiveGrid, and UpdNo. When a data graph has 15K nodes with more than 150K edges, Gradin can process a query graph of size 7 within 3 seconds in average. In terms of query processing time, Gradin is close to UpdAll, and outperforms UpdNo and NaiveGrid in all cases, with up to 8 times speedup.

4.6.5 Summary

We summarize the experimental results as follows. (1) Gradin is efficient for index updates. Gradin outperforms the baseline algorithm UpdAll, and the speedup is up to 10 times on our datasets. (2) The search algorithm and fragment join algorithms in Gradin accelerate query processing. Gradin outperforms the baseline algorithms VF2, NaiveGrid, NaiveJoin, and UpdNo. While VF2 cannot scale on larger query graphs, Gradin processes all query graphs in 4 seconds, matches the query processing speed of UpdAll, and is up to 10 times and 5 times faster than UpdNo and NaiveGrid, respectively.

4.7 Related Work

Subgraph matching. Subgraph matching is one of the most critical primitives in many graph applications, such as pattern search in protein-protein interaction networks [194, 210], chemical compounds [89, 191], program invocation graphs [198, 201], and communication networks [77, 184]. In these applications, subgraph matching queries have been defined in their own ways, and a variety of techniques have been proposed to resolve the corresponding challenges.

Subgraph matching queries are usually defined by the NP-hard *subgraph isomorphism* [73]. Although branch-and-bound based algorithms, such as Ullmann's algorithm [178] and VF2 [58], were proposed to improve the search efficiency, these algorithms still cannot scale on large graphs [67, 191].

To accelerate subgraph-isomorphism based subgraph matching, a variety of graph indices have been proposed. Among them, substructure based indexing is the most widely adopted framework. Although the indexed substructures could be diverse, such as paths [76, 201], trees [159, 194], or general subgraphs [54, 95, 173, 189, 191], they follow a common spirit: a small data graph, or a small region in a large data graph, is worth searching, if it contains a query graph's substructures. In addition, He et al. [89] proposed *closure-tree*, an R-tree-like search index, which only checks the data graphs that are similar to a query graph and prunes unpromising search for dissimilar data graphs.

In addition to exact matches, an inexact subgraph matching query is a more general and flexible graph primitive. Given a query graph, it aims to find similar subgraphs in data graphs, where similarity is usually defined by *graph edit distance* [192]. To speed up query processing, substructure based indices are widely adopted, such as Grafil [192], SAPPER [198], and many others [105, 132, 176, 185].

Note that all the graph indices discussed above are designed for static data graphs with discrete node/edge labels; however, in our problem setting, node/edge labels are dynamically updated, and these labels are numerical values. The above indices cannot efficiently process subgraph matching queries on dynamic graphs with numerical labels: (1) frequent graph updates result in serious index maintenance issues; and (2) numerical node/edge labels cannot be naturally supported by most of those indices. In this paper, we propose a graph index addressing these issues.

Meanwhile, variants of subgraph matching queries have been studied. Tong et al. [177] proposed a proximity-based score function, and the top-k subgraphs of the highest score are returned as output. Cheng et al. [55] relaxed matching conditions: instead of an adjacent node pair, a pair of reachable nodes in a data graph is also eligible to match an edge on a query graph. Similarly, in [210], a pair of nodes that satisfy a pre-defined distance constraint is eligible to match an edge on a query graph as well. Moreover, Fan et al. [68, 69] defined subgraph matching by *graph simulation*, and Yuan et al. [197] studied the subgraph matching problem on uncertain graphs with categorical labels. Different from them, our work focuses subgraph-isomorphism based queries where data graphs have dynamically changing numerical labels.

Closer to our work are the studies from Wang et al. [47], Mondal et al. [130] and Fan et al. [67]. In [67], incremental algorithms are proposed to answer a fixed set of queries on dynamic graphs. In this paper, we consider a different setting and aim to serve arbitrary subgraph matching queries on dynamic graphs. Wang et al. [47] also considered the dynamic nature of data graphs; however, our work is different from theirs as follows: (1) in [47], the proposed technique aims to answer approximate matching, while we propose a solution for exact matching; and (2) although node/edge uncertainty represented by numerical values are also considered, the proposed indexing technique is still based on categorical node/edge labels [182]; thus it cannot solve the challenge in our problem setting. Mondal et al. [130] studied how to make node replication decisions based on dynamically changing workload to optimize the performance for queries such as reading neighbors' data. Instead, our work focuses on how to improve the performance for more sophisticated subgraph matching queries on dynamic graphs with numerical labels.

Multi-dimensional index. Multi-dimensional indices have been studied for applications that monitor moving objects. Early works [48] used variants of R-tree to index moving objects and accelerate range/kNN queries; however, the update operations in these techniques are usually costly, while data points are frequently updated in the applications [163, 166]. To address the issue from frequent data updates, grid based indices are proposed [45, 135, 162, 163, 196]. In particular, Chakka et al. [45] used a grid based index to manage a large number of trajectories, Mouratidis et al. [135] and Yu et al. [196] discussed how to use grid based indices to accelerate kNN queries, Šidlauskas et al. [163] conducted an experimental study to compare grid based indices with variants of R-tree in update-intensive applications, and the possibility of incorporating parallelism into grid based indices is investigated in [162]. Our grid based index shares this spirit. It differs in the following aspects. (1) We show *theoretical* results on a more *general* setting: instead of two-dimensional space widely discussed in the above works, we focus on indexing points in a general multi-dimensional space, and derive theoretical bounds for update and search performance. (2) We discuss how index parameters affect update and query performance with both theoretical and experimental results. (3) We investigate the feasibility and principles of applying multi-dimensional indices to prune redundant search for dynamic subgraph matching queries.

4.8 Summary

In this chapter, we identify a new important application of graph query in cloud computing and define a general subgraph matching problem on temporal graphs of numerical labels. We introduce a new method called Gradin to index graphs of numerical labels. Gradin can efficiently process frequent index updates and prune unpromising matches. In particular, FracFilter, one important component of Gradin, is proposed to keep the cost of update operations low and enable fast search. Minimum fragment cover and fingerprint based pruning are proposed for fast query processing. Our experimental results show that Gradin has better index update and query processing performance in comparison to the baseline algorithms.

Chapter 5

Temporal Reachability

5.1 Introduction

Delay Tolerant Networks (DTNs) are communication networks that lack continuous connectivity due to node mobility, failures or other factors. They experience frequent partitioning, and end-to-end paths between two nodes may even never exist. Routing in DTNs uses a *store-carry-forward* approach [94], where intermediate nodes delay the transmission of messages until new links are available and the messages are "eventually" delivered with some delay. When the lack of connectivity is due to node mobility, the movement of nodes can be exploited to carry the messages.

In recent years, routing protocols for multicast in DTNs have received considerable attention [72, 116, 203]. Multicast protocols optimize the transmission cost by sharing routing paths among multiple destinations. Recent advances allow us to achieve a good tradeoff between minimization of the transmission cost and maximization of the delivery rate. However, due to the nature of DTNs, proper delivery cannot always be guaranteed.

To guarantee the connectivity, nodes can be equipped with *long-range communication* devices [202] which would be used if end-to-end communication cannot be otherwise achieved. Since long-range communication is expensive, its utilization should be limited as much as possible. Providing an extra long-range communication to guarantee delivery introduces the new challenging problem of minimizing its cost.

In this paper we formulate the problem of optimizing the long-range communication cost in a network of moving nodes as a reachability problem on temporal graphs, called *demand cover*.

- Our model considers a set of moving nodes (*e.g.,* people or vehicles) that are equipped with devices providing two kinds of communication. A short-range communication (*e.g.,* radio), considered non-costly, can occur between two nodes when they are close to each other. A remote (long-range) communication (*e.g.,* cellular or satellite), which involves a cost, can occur at any time independently of the node positions. Therefore, the dynamic short-range connections between nodes form temporal graphs that evolve over time.

- In our model, a set of continuously-updated data objects need to be shared among nodes. Each data object needs to be received by a subset of destinations. For each destination, its deadline (time instant at which the object is needed) is specified. To avoid receiving non-recent copies of objects, a latency is also specified. In other words, if two nodes are reachable on the temporal graph formed by short-range connections within a time window which guarantees both freshness of the data object and arrival by the deadline, the data request can be fulfilled with small cost.

- When no such time windows exist for a pair of nodes, we have to use costly remote transmissions to accomplish the request. To this end, our goal is to find the set of remote transmissions that minimizes the communication cost subject to the aforementioned delay constraints.

To our knowledge, we are the first to analyze the problem of optimizing the long-range communication cost for multicast in DTNs.

The described problem has several practical applications.

- Consider a network of city buses, in which the transportation agency wants to provide passengers with personalized news that depends on their position, traveling plan etc. Each bus can obtain the updates by the cellular network (costly). However, it is more convenient to share the information among various buses via radio communication (non-costly).

- Another example considers soldiers or military vehicles that move following a specific strategy. They need to access certain information related to their location (*e.g.,* satellite images). In this case, the only options available are satellite communication (highly costly) and node-to-node communication.

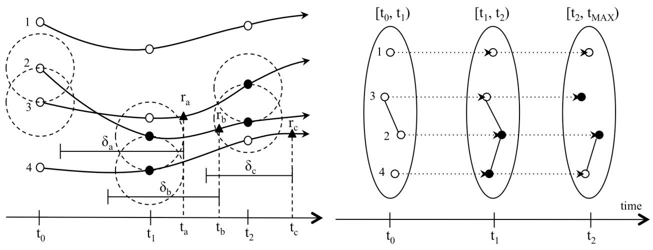
- The proposed approach also has applications in data ferrying [145, 202]. A set of moving nodes (ferries) is charged for gathering data. Depending on time constraints on data delivery, the ferries may decide whether to use short-range or long-range communication.

Note that in the above examples, the node trajectories and the traffic demands are known in advance.

Solving the demand cover problem introduces new challenges due to the temporal constraints. A data object may need to be transmitted remotely more than once, due to either lack of connectivity or the need of satisfying a latency constraint. For example, consider the four nodes in Figure 5.1(a) that move following certain trajectories. Initially, nodes 2 and 3 are in contact. At time t_1, nodes 2 and 4 enter in each other's radio range and a new contact begins. At time t_2, the contact between nodes 2 and 3 ends since they move away from each other. Three of these nodes (shown with triangles) need to receive the same data object. Each of them has a given deadline (t_a, t_b and t_c, respectively) and a latency (δ_a, δ_b and δ_c, respectively). A remote transmission to node 3 at time t_1 covers the data needs r_a and r_b. Although r_c can be reached by transmitting the object to node 4 at any time after t_1, the latency δ_c cannot be satisfied. Therefore, an updated copy of the object needs to be transmitted after $t_c - \delta_c$.

In this paper, we prove that the demand cover problem is NP-hard and present a baseline graph-based approach for it. In order to make this problem feasible on large datasets, we formulate it as a temporal reachability problem and develop a novel graph-indexing-based solution. Due to the index, we are able to handle thousands of destinations on a network with millions of encounters in less than 10 seconds, with an improvement of up to 100 times compared to a naive approach.

The rest of this chapter is organized as follows. We start with the preliminaries in Section 5.2. In Section 5.3, we define the temporal reachability problem *demand cover* that formalizes the problem of optimizing the long-range communication cost in a network of moving nodes, demonstrate its NP-hardnes, and develop a



(a) (b)

Figure 5.1: An example of a DTN among moving nodes. (a) The four solid lines represent four trajectories. To simplicity we use the x-axis indicating the time. The big dashed circles represent the radio range of nodes. Nodes that are involved in a transition (contact beginning or contact end) are filled. Three data needs (r_1, r_2 and r_3) are represented with filled triangles. Their deadlines are t_a, t_2 and t_3, respectively while their latencies are δ_a, δ_2 and δ_3, respectively. (b) The corresponding temporal graph. Snapshots of the connectivity graph at three different times are depicted within big ovals. Temporal links joining contiguous snapshots are represented with dotted lines.

baseline algorithm. In Section 5.4, we propose a novel indexing system for quickly solving demand cover on graphs optimally. Our system compresses temporal graphs and uses an efficient filtering approach to retrieve a small portion of nodes that are relevant for achieving an optimal solution. In Section 5.5, we evaluate the proposed approach on two real and one synthetic datasets, and show that an exact solution can be found in reasonable time in datasets with millions of encounters. Finally, we discuss related work in Section 5.6 and summarize this work in Section 5.7.

5.2 Preliminaries

In this section we describe some basic concepts concerning DTNs that introduce our approach. We consider a network of moving nodes whose *trajectories* are known in advance or can be predicted. When two nodes enter each other's radio coverage area, a link between them is formed and a *contact* (or encounter) begins. A contact between two nodes terminates when they lose radio connectivity as they move away from each other. Contact beginnings and contact ends are also called *transitions*. The status of the network at a certain time instant can be described by a *connectivity graph*, whose vertices represent moving nodes and a link is placed between two nodes if their distance is within a given threshold d, called *radio range*. The network dynamics can be described by a series of snapshots of the connectivity graph over the time horizon [40, 94]. All the snapshot graphs are aggregated in a unique composite graph (named *space-time graph*) where vertices corresponding to the same moving node in two consecutive connectivity graphs are joined by a *temporal link*. In contrast to *spatial links*, temporal links are directed. A message can travel across a so called *space-time path*. If some spatial links toward the destination are available, the message is forwarded, otherwise the message is *carried* by the moving node (a temporal link is traversed) and forwarded when another suitable node is encountered. In the following, we refer to *route* for indicating the space-time path that a message traverses.

Figure 5.1(b) shows the space-time graph corresponding to Figure 5.1(a). Three snapshot graphs are represented, each of them describing the connectivity of the network at the time intervals $[t_0, t_1)$, $[t_1, t_2)$ and $[t_2, t_{MAX})$, respectively. Contiguous snapshots are joined by temporal links (in dotted line). Each snapshot is associated with its *lifetime*, i.e. the extent in time that it refers to. Message routes travel across paths of the space-time graph that can include both spatial and temporal links.

5.3 Problem Statement

In this section we define formally the demand cover problem and give some baseline approaches for it. We consider a set of n *nodes* (numbered $1, 2, \ldots, n$) that move following certain *trajectories* ($T_1, T_2, \ldots, T_n$, respectively). A trajectory associates each time instant t in the range $[t_0, t_{MAX})$ with the position in the space (typically a plane) that the corresponding node occupies at time t. At a specific time, two nodes can communicate with each other through a so called *contact transmission* (short-range, typically radio) if they are *in contact*, i.e. their Euclidean distance is within a fixed threshold d. The contact transmission (between nodes) does not involve any cost. Each node can also communicate at any time with a *data source* (Internet or a central server) through a costly *remote transmission* (cellular or satellite), with a certain cost. Our approach can be extended to a decentralized scenario where a central server is not available and data are distributed among nodes.

We consider the problem of delivering *data objects* to multiple *destinations*. In contrast to other multicast approaches in which static messages are sent to multiple destinations, we consider the problem of sharing data objects that are continuously updated over time. The dynamic character of data objects introduces new constraints: each destination needs to receive the object before a given *deadline* and with a delay that is limited by a given *latency*. We define the *data demand* I of a data object as a set of *data needs*, i.e. triples of the form (i, t, δ), where i, t and δ represent the destination, the deadline and the latency, respectively. We call the instant $t - \delta$ *release time*. It represents the earliest time instant in which an object can leave the data source.

The data flow is modeled by two kinds of transmissions: a *remote transmission* is denoted by a pair (i, t), where i represents the node that receives the object and t is the time instant in which the transmission occurs; a *contact transmission*, is represented by a triple (i_1, i_2, t) where i_1, i_2, t represent the node that transmits the object, the node that receives the object and the time instant at which the transmission occurs, respectively. For simplicity, all the transmissions are considered instantaneous. Although it may not be true in all cases, the movement between nodes is usually very slow compared to the speed of transmission. Therefore, in most cases all the necessary objects can be transmitted before the contact terminates. We say that a remote transmission (i_s, t_s) *covers* a data need (i_d, t_d, δ) if there exists a sequence of contact transmissions (i_0, i_1, t_1), (i_1, i_2, t_2), ..., (i_{k-1}, i_k, t_k) with $i_0 = i_s$ and $i_k = i_d$, such as $t_s \leq t_1 \leq t_2 \leq \ldots \leq t_k \leq t_d$ and $t_d - t_s \leq \delta$. The set of data needs covered by a remote transmission is also called *coverage* of the remote transmission. The demand cover problem is defined as follows:

Problem definition: Given a set of trajectories and a data object with demand I, find the minimum set of remote transmissions that covers all the data needs in I.

The formulated problem can be shown to be NP-hard (even in the 2D plane) by reduction from the well known Set-cover problem. Given a family of sets $\mathcal{S} = \{S_1, S_2, ..., S_m\}$ of elements taken from a set C, Set-cover calls for finding the minimum sub-family of S that covers all the elements of C.

Theorem 7. *Any instance of Set-cover can be reduced in polynomial time to an instance of the demand cover problem.*

Proof. Let the family $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ be an instance of Set-cover where elements are taken from a universe set C (i.e. $S_i \in C$ for each $i = 1, 2, \ldots, n$) of size m. Choose d, t_s and t_d arbitrarily and set $\delta = t_d - t_s$. For each element c of C, consider a fixed (non-moving) node j_c (called destination) and a data need (j_c, t_d, δ). Set the positions of these nodes along a line so that any two contiguous nodes are at a distance $2 \cdot d$ from each other. Consider n points $p_1, p_2, \ldots, p_n$ in a parallel line at a distance $dl = 2 \cdot d \cdot \max(m, n)$ from the first line. Given a set S_i, for each element c of S_i consider a node with a straight trajectory that starts from p_i and ends to the location of j_c. Informally, these nodes are introduced to carry the data object from the initial location to the destination. Note that dl has been chosen so as to guarantee that any two nodes are always at a distance greater then d from each other, except for the destination. The number of moving nodes is $z = \sum_{i=1}^{n} |Si|$. The speed of each moving node is assigned in the following way. Divide the time interval $[t_s, t_d]$ in z slots of the same length, each of them associated to a moving node. Each node remains without moving until its time slot is reached. Then it moves with a constant speed that allows it to reach the destination before the time slot terminates ($speed = 4 \cdot \frac{z}{\delta} \cdot d \cdot \max(m, n)$ or higher). Then it stops again.

A data object transmitted by a remote transmission at time t_s to a node in p_i is shared among all the nodes at position p_i and carried to the destinations that correspond to elements of S_i. No other nodes receive the data object. Therefore an optimal set of remote transmissions corresponds to an optimal sub-family of $\mathcal{S}$ for Set-cover. □

An example of reduction is given in Figure 5.2. Points in the left-hand side correspond to sets, while nodes in the right-hand side model elements. The minimal sub-family that covers all destinations is $\{S_2, S_4\}$, since an object can be carried from point p_2 to j_1 and j_3 and from point p_4 to j_2, j_4 and j_5.

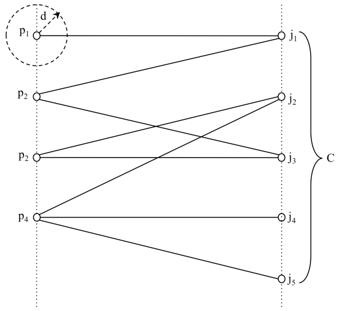


Figure 5.2: (a) An example of reduction from Set-cover. Each set S_i of the family $\mathcal{S}$ is associated to a point p_i in the left-hand side. The fixed nodes $j_1, j_2, \ldots, j_5$ in the right-hand side are associated to elements. The dashed circle delimits the radio range, of length d. Moving nodes follow the trajectories depicted by solid lines. The minimal sub-family that covers all destinations is $\{S_2, S_4\}$, corresponding to points p_2, p_4.

5.3.1 ILP formulation

The demand cover problem can be formulated in ILP (Integer Linear Programming) and solved by a standard solver. Here we give an ILP formulation and show that solving it on large datasets is infeasible.

We consider a set of n moving nodes numbered 1 through n and a special node that represents the central server, numbered 0. We write $i \rightarrow_t j$ if node i can communicate with node j at time t (i.e. they are within distance d or $i = 0$). We also consider a discrete set $\mathcal{T}$ of time instants that correspond to transitions or deadlines of data needs. This restriction does not compromise the result. In fact, given an optimal solution for demand cover, it is always possible to modify this solution in such a way that each transmission between two nodes is delayed until the contact between them ends (right before the link breaks) or a data need that involves one of the nodes expires, without increasing the cost. Since communication is assumed to be instantaneous, the contact length is not important.

We employ two classes of boolean variables. The first class contains variables of the kind $x_{i,j,t,r}$, where i and j represent nodes, t represents a time instant and $r = (i_r, t_r, \delta_r) \in I$ represents a data need, which models the flow of data objects. The variable $x_{i,j,t,r}$ has value 1 if i send a message to j at time t to satisfy the data need r. Variables of this kind are considered for $i \rightarrow_t j$ and $t_r - \delta_r \leq t \leq t_r$. The second class of variables, of the kind $y_{i,t}$, counts the number of remote transmissions. Each variable says whether a remote transmission between the central server and a particular node occurs at a certain time or not. The complete formulation follows.

$$\begin{aligned}
&min && \sum_{t \in \mathcal{T}} \sum_{i=1}^{n} y_{i,t} \\
&s.t. && \sum_{\substack{t \in \mathcal{T} \\ t \geq t_r - \delta_r \\ t \leq t_r}} \sum_{\substack{i=0 \ldots n \\ i \rightarrow_t i_r}} x_{i,i_r,t,r} \geq 1 \quad \forall r = (i_r, t_r, \delta_r) \in I && (5.1) \\
& && \sum_{\substack{t' \in \mathcal{T} \\ t_r - \delta_r \leq t' \leq t}} \sum_{\substack{i=0 \ldots n \\ i \rightarrow_{t'} j_1}} x_{i,j_1,t',r} - x_{j_1,j_2,t,r} \geq 0 && (5.2) \\
& && \qquad \forall r = (i_r, t_r, \delta_r) \in I, \forall j_1, j_2, t \mid j_1 \rightarrow_t j_2 && \\
& && y_{i,t} \geq x_{0,i,t,r} \quad \forall r \in I, i = 1 \ldots n, t \in \mathcal{T} && (5.3) \\
& && x_{i,j,t,r}, y_{i,t} \in \{0, 1\} &&
\end{aligned}$$

Constraint 5.1 models the fact that for each data need, the data object must be sent to the destination at a time instant between the release time and the deadline. Constraint 5.2 models the propagation of data objects. It says that if a data object is transmitted from a source j_1 to j_2 at time t for satisfying a data need r, then j_1 must receive the object before t and after the release time. Finally, constraint 5.3 assigns value 1 to each variable of the kind $y_{i,t}$ if a message is transmitted from the central server to node i at time t in order to satisfy some data need.

Solving this formulation with standard solvers is infeasible on large instances. The main problem concerns the number of variables and constraints. Even considering a sparse network with 100 nodes, 100 encounters per node and 100 data needs, we have hundreds of millions of variables and constraints. In our experiments we obtained hundreds of billions of variables and constraints, therefore executing it was not possible.

5.3.2 A naive approach for the demand cover problem

An improvement on executing the ILP program can be obtained by reducing the problem to Set-cover. Each candidate remote transmission can cover a set of data needs. The minimum set of remote transmissions that covers all the data needs corresponds to the minimum Set-cover in the family of associated sets. Since remote transmissions can occur at any time, the number of sets for the Set-cover family is huge. However, not all time instants need to be considered. To guarantee that all the data needs are covered, one can consider only time instants that correspond to the release time of a data need. We note that the release time is the earliest time instant in which the data object needs to be sent for a data need to be satisfied. Delaying a remote transmission after the release time does not help with serving the data needs, unless some other release times are overtaken. Given a candidate remote transmission, the set of covered data needs can be computed by exploring the space-time graph (through depth-first search or breadth-first search).

5.3.3 A compact graph representation

The naive approach requires exploring a space-time graph, whose size can be huge. However, this graph can be compacted thanks to two observations. First, in each snapshot graph, all the vertices that are in the same connected component have the same reachability properties, so one vertex can be taken as a representative of all the others. Second, when a transition occurs, connected components of the snapshot graph that do not contain any nodes involved in the transition are not influential.

To generate the *compressed graph*, we focus on two class of transitions. A *split transition* causes a connected component to be divided in two connected components. A *merge transition* causes two components to merge into a single connected component. We generate a space-time graph considering only these two kinds of transitions. Then, for all snapshot graphs, each connected component is collapsed into one single vertex. At this point, all the edges of the graph are directed and can be classified as follows: (i) *split edges*, which connect splitting components with their partitions; (ii) *merge edges*, which connect merging components with the resulting components and (iii) *non-influential edges*, which connect components that do not change. Finally, each non-influential edge is removed by collapsing its endpoints and each vertex is labeled with its lifetime (note that a vertex can span several snapshots), which we call *component lifetime*.

One example of a compressed graph is shown in Figure 5.3, which refers to the example in Figure 5.1. Boxes represent vertices of the compressed graph. Edges of the connected graph are represented by solid lines. The extent of a box in time represents the component lifetime. For instance, the extent of the component c_1 is $[t_0, t_{MAX})$ (the whole time horizon) since this component is never involved in any split or merge. The naive approach can be executed on the compressed graph in place of the cumbersome space-time graph. The family for Set-cover is obtained by building a set for each vertex of the compressed graph whose lifetime contains the release time of some data needs. Each set can be computed by exploring the compressed graph.

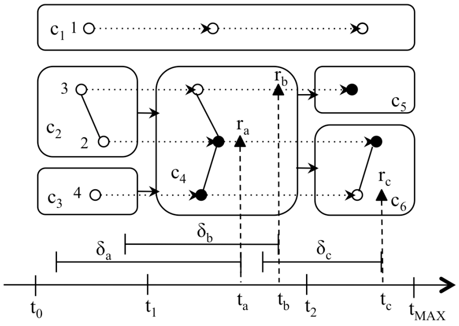


Figure 5.3: The compressed graph representation of the example in Figure 5.1. A compressed graph is depicted over the space-time graph. Boxes and solid lines represent vertices and edges of the compressed graph, respectively. The extent of a box in time represents the component lifetime. Three data needs are represented (by filled triangles) with their extent in time. From left to right: $r_a = (2, t_a, \delta_a)$, $r_b = (3, t_b, \delta_b)$, $r_c = (4, t_c, \delta_c)$.

5.4 An indexing system for the demand cover problem

Solving the demand cover problem efficiently raises several challenges. First, for each vertex v of the compressed graph, the set of data needs that can be covered by v needs to be retrieved. This operation may be very expensive when the size of the graph is large. Second, Set-cover is NP-hard, therefore no polynomial-time solutions exist (unless P=NP) in the general case. For small instances, Set-cover can be solved optimally in acceptable time by pruning techniques as *branch-and-bound*. In our case, however, the number of sets generated is usually very high, since a data need can potentially be covered by many vertices. Many of these sets are redundant, i.e. they are fully contained in other sets. For example in Figure 5.3 the set of data needs covered by c_6 contains only r_c. The vertex c_4 covers the set $\{r_a, r_b, r_c\}$, which contains the data need covered by c_6. Therefore, c_6 can be excluded by the computation since all the data needs that can be covered by it can also be covered by c_4. Removing redundant sets leads to a considerable reduction of the Set-cover instance. However, removing the redundancy by traditional methods is expensive, since it requires one to find *maximal sets* [195]. Additionally, in a typical application, a large number of data objects are requested and each data object has its own set of data needs. Solving the demand cover problem for each data object can be extremely expensive.

We propose a novel approach, *Path-wise indexing* (PIE, for short), which builds an index of the set of trajectories with the purpose of efficiently performing queries of the form: *given a set of data needs, return the minimum set of remote transmissions that covers all the data needs.* We use a preprocessing-filtering-optimization scheme to solve the demand cover problem. Given a database of trajectories, a *preprocessing* phase generates a compact data index. When a query (represented by a set of data needs) has to be performed, we use the data index to generate a lightweight instance of Set-cover (*filtering* phase). Set-cover is then solved optimally (*optimization* phase) and the solution is returned.

The proposed indexing system has several advantages. First, the index is much more compact than the compressed graph, and hence requires less memory and is much more efficiently manageable. Second, the set of vertices in the compressed graph from which the data needs are reachable can be identified fast. Note that current reachability indexes cannot be efficiently applied to our problem since many reachability tests need to be performed. Finally, we can efficiently prune nodes of the compressed graph that are not promising and generate a small instance for Set-cover. Next, we introduce the proposed index and describe the three phases of our indexing system: preprocessing, filtering and optimization.

5.4.1 Index structure

The key idea is that the set of data needs covered by a node in a path p of the compressed graph includes the set of data needs covered by other subsequent nodes p. Therefore, a node can be taken as a representative of a portion of the path. Moreover, a node of the compressed graph can be uniquely determined by a path and a time instant. This implies that we can use the coverage of a pair (p, t) in place of the coverage of the corresponding compressed node. We denote the coverage of (p_i, t) as $C(p_i, t)$. Based on these considerations, we partition the compressed graph into a set of disjoint paths and build a compact graph, named a *PIE graph*, whose vertices represent disjoint paths and edges preserve the connectivity across paths. Each vertex of the PIE graph is labeled with a time interval (named *lifetime*) that is the union of the lifetimes of its composing vertices. Instead of exploring all the nodes of a path, we can determine a set of time instants that is representative of the whole path by exploring the compact PIE graph.

Figure 5.4(a) shows an example. The small circles and thin edges form the compressed graph, while the big ovals represent disjoint paths. Consider the path p_3. The set of data needs covered by p_3 at time t_a is $C(p_3, t_a) = \{r_1, r_2\}$. Since no other data needs (i, t, δ) have $(t - \delta) \in [t_a, t_b)$, (p_3, t_a) is representative of the interval $[t_a, t_b)$. t_b coincides with the *release time* of r_3 (i.e. $(t_{r3} - \delta_3)$). Therefore, its coverage ($C(p_3, t_b) = \{r_3\}$) cannot be contained in $C(p_3, t_a)$. The pair (p_3, t_b) is instead representative of the remaining part of the path. The path p_3 produces only two sets ($C(p_3, t_a)$ and $C(p_3, t_b)$) for Set-cover. In general, up to 4 sets would be produced without indexing, since we may have many other non reachable data needs whose release times falls within all vertices of p_3. Next we describe in details the three steps of our method: preprocessing, filtering and optimization.

5.4.2 Preprocessing

Given the set of trajectories, first a compressed graph (G_C) is generated. The graph is then decomposed into a disjoint set of paths. There is a large number of possible ways to partition the graph into disjoint paths. A suitable partition strategy should satisfy the following properties: (i) the number of disjoint paths is minimal and (ii) the number of edges across two paths is minimal. In general,

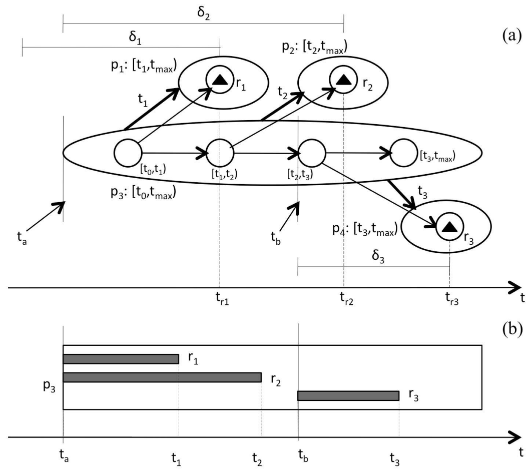


Figure 5.4: (a) An example of PIE graph. The small circles and thin arrows form the compressed graph. Each path is circumscribed by an oval and its lifetime is reported. Links between paths are represented by thick arrows. They are labeled by the ends of the lifetimes of their source vertices. Solid triangles within circles represent data needs. (b) Validity intervals of a set of data needs in a path p_3. Bars represent the extent of validity intervals of data needs. The minimal family of sets for this path is $\{C(p_3, t_a), C(p_3, t_b)\}$.

finding the minimum set of disjoint paths that covers a graph is a non-trivial problem [62]. However, since the compressed graph is a DAG and is generated by a simple split-merge model, we can use optimally the following simple strategy: pick one vertex a time (proceeding in time order) and elongate it by random walk until a vertex without outgoing edges is reached.

We can prove that across two paths no more than one edge exists in each direction. Indeed, each edge of the compressed graph comes from a merge or a split between two components. In the case of merge, the source vertex cannot have other outgoing edges, while in the case of split, the target vertex cannot have other incoming edges. This implies that no edges can exist between internal nodes of two different paths, and hence each edge connecting two paths can be either outgoing from the last vertex of the source path or incoming to the first vertex of the target path. Since the compressed graph is a DAG, and each path is elongated as much as possible, at most two edges can connect two paths, one in each direction.

We associate each edge (p_i, p_j) of the PIE graph with the end of the lifetime of the source vertex in p_i. We denote this time instant as $ft(p_i, p_j)$. It represents the time in which a data object can traverse the edge (p_i, p_j). Figure 5.4 depicts an example of PIE graph. The small vertices and thin edges form the compressed graph, while the big vertices and thick edges represent the PIE graph.

5.4.3 Filtering

For each vertex p of the PIE graph, our filtering algorithm finds a set of time instants TI_p that are representative of the whole path p, and the family $\mathcal{S}$ of corresponding sets. Our strategy guarantees that the coverage of each vertex of the compressed graph is fully contained in at least one set in $\mathcal{S}$. Since the PIE graph is much smaller than the compressed graph, exploring the former is much more advantageous in terms of elaboration time and memory consumption.

The filtering procedure considers two steps: *backflow* and *prune*. *Backflow* propagates the data needs in reverse order from the destination paths to all the possible source paths. For each path, we compute the *validity interval* of a data need, which defines the time interval in which the data object must reach the path for the data need to be covered. At the end, each path is associated with a set of data needs that it can cover with their validity intervals. The coverage of a pair (p, t) can be identified by the set of data needs such as their validity intervals in p

include t. After the validity intervals are generated, the *prune* procedure computes the family of sets for Set-cover. It collects the family of maximal coverage sets of each path, i.e. the set of time instants whose coverage is not strictly contained in the coverage of any other time instant in the path.

Before describing these two procedures in detail, we give an example. Figure 5.4(b) shows the path p_3 of the example in Figure 5.4(a) and the validity interval of each data need in it. The validity intervals of r_1 and r_2 start at the beginning of the path, since their release time precede it. These intervals end at times t_1 and t_2, respectively, times associated to outgoing edges (see Figure 5.4). Each of them represents the last time instant in which the data object must leave the path to be able to reach the respective data need. For the data need r_1 (r_2 resp.), if the data object leaves the path after t_1 (t_2 resp.), the destination cannot be reached. The validity interval of r_3 starts at time $t_b = t_{r3} - \delta_3$, corresponding to the release time of r_3, and ends at time t_3, time associated to the unique outgoing edge that can reach r_3. The representative time instants for this path are t_a and t_b, corresponding to maximal sets of data needs. Therefore, the minimal family of sets for this path is $\mathcal{S} = \{C(p_3, t_a), C(p_3, t_b)\}$. Note that no other time instants have a coverage that is not included in at least one set of the family.

Backflow

We define the *validity interval* of a data need $r = (i, t, \delta)$ in a path p (named $valid\_int(r, p)$) recursively in the following way:

If p has lifetime $[b, e)$ and is the destination path of r (i.e. $t \in [b, e)$), we have: $valid\_int(r, p) = [\max(b, t - \delta), t)$.

If p is a non-destination path with lifetime $[b, e)$ that links to a set of paths $p_1, p_2, \ldots, p_k$ with validity intervals $[b_1, e_1)$, $[b_2, e_2), \ldots$, $[b_k, e_k)$, respectively:

$$valid\_int(r,p) = \begin{cases} \Phi \quad \text{if } ft(p,p_i) \notin [b_i, e_i) \; \forall i = 1 \ldots k \\ [t_1, t_2) \quad \text{otherwise} \end{cases}$$

where $t_1 = \max(b, t - \delta)$ and t_2 is the maximum t' such as $t' = ft(p, p_i)$ for some $i = 1 \ldots k$ and $t' \in [b_i, e_i)$.

Intuitively the end of a validity interval in a path is given by the last time instant in which the data object can flow in another path that has a compatible validity interval, while the beginning of a validity interval is limited by $t - \delta$ and the starting time of the path.

The coverage of a pair (p, t) can be identified by the set of data needs whose validity intervals include t. Intuitively, if validity intervals are represented by horizontal bars (as in Figure 5.4(b)), the coverage of a pair (p, t) can be easily identified by drawing a vertical line and taking all the data needs whose validity intervals are intersected. For instance, in Figure 5.4b) a vertical line drawn at time t_a intersects the validity intervals of r_1 and r_2. Therefore, the coverage of (p, t_a) is $\{r_1, r_2\}$. This property is formally stated by the following lemma:

Lemma 2. *Let (T, I) be an instance of demand cover, where T is the set of trajectories and I is the set of data needs, and G_P be the corresponding PIE graph. Given a vertex p of G_P and a time instant t, the coverage of p at time t is:*

$$C(p,t) = \{r \in I \mid t \in valid\_int(r,p)\}$$

$valid\_int(r, p)$ can be computed for all paths in a breadth-first search fashion, by starting from the path containing r and exploring the PIE edges in reverse time order until the release time is reached. When a new vertex is visited, the validity interval of r in it is updated. The complexity is $O(|E_P|)$, where E_P is the set of edges in the PIE graph.

Prune

For each path p, we identify the minimum-size set TI_p of time instants that is representative of the whole path, i.e. such that for all $t \in lifetime(p)$ we have $C(p,t)$ contained in at least one set $C(p,t')$ with $t' \in TI_p$. This problem corresponds to the problem of finding the maximal sets in the family of all possible coverage sets of p (i.e. $\{C(p,t)|t \in lifetime(p)\}$).

Figure 5.5 shows an example path with the validity intervals of five data needs. The coverage of a time instant can be easily identified by drawing a vertical line and taking all the validity intervals that it intersects. The representative time instants for this path are t_1, t_2 and t_3, corresponding to the maximal sets of data needs. Note that no other time instants have a coverage that is not included in the coverage of at least one of the time instants t_1, t_2 or t_3.

In general, the maximal sets can be found in time $O(mn)$, where m is the number of maximal sets and n is the size of the input [195]. In our case, since each element corresponds to a contiguous interval, we can find the maximal sets in linear time. Our procedure slides a vertical line across the path in reverse time order, and takes all the time instants that correspond to maximal sets. Each position t of the line corresponds to a coverage $C(p,t)$. As the line is slid, the coverage is modified, by either adding or deleting data needs. Whenever a deletion follows an addition, the current coverage is taken as a maximal set. Note that additions correspond to the end of validity intervals, while deletions correspond to

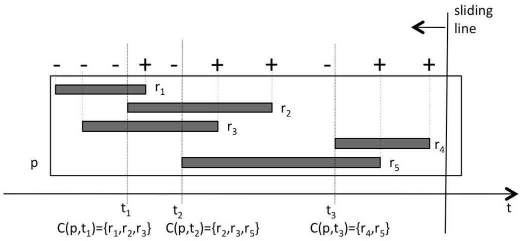


Figure 5.5: An example of maximal coverage sets in a path. Bars represent the extent of validity intervals of data needs. The coverage of the time instants t_1, t_2 and t_3 are maximal sets among all the coverage sets in the path. The family of maximal sets can be found by sliding a vertical line in reverse time order and taking each time instant that corresponds to the beginning of a validity interval (indicated by the symbol "-" at the top) that occurs right after the end of the same or another validity interval (indicated by the symbol "+"). This family has minimum size.

the beginning of validity intervals. In Figure 5.5, the coverage associated with the sliding line is initially empty. When the line intersects the validity interval of r_4, r_4 is added to the coverage (additions are indicated by the symbol "+" at the top). The interval of r_5 is then encountered and r_5 is also added to the coverage. When the beginning of the validity interval of r_4 is encountered (at time t_3), the current coverage is taken as maximal set and r_4 is deleted (indicated by the symbol "-"). Other two additions are then encountered (r_2 and r_3) followed by a deletion (r_5). The coverage at time t_2 (before deleting r_5) is then taken as another maximal set. The last maximal set is taken at time t_1, after another addition and another deletion are encountered. The following lemma states that this procedure finds all and only the maximal sets in the family of coverage sets.

Lemma 3. *Let (T, I) be an instance of demand cover and G_P be the PIE graph built from (T, I). Given a path p of G_P, consider the sequence of time instants $t_1, t_2, \ldots, t_k$ corresponding to extremes (beginnings or ends) of validity in-*

tervals in reverse time order and the set $TI_p = \{t_i \mid t_i$ *is a beginning time and* t_{i-1} *is an ending time*$\}$.

1. *For each time instant* $t \in lifetime(p)$ *we have:* $\exists t' \in TI_p \mid C(p,t) \subseteq C(p,t')$*;*

2. *For each time instant* $t' \in TI_p$ *we have:* $\nexists t \in lifetime(p) \mid C(p,t') \subset C(p,t)$*;*

3. *For each pair of distinct time instants* $t', t'' \in TI_p$ *we have* $C(p,t') \neq C(p,t'')$.

A clear consequence of this lemma is that the family of maximal sets generated by our procedure has minimum size. TI_p can be built in time $O(|I| \cdot \log(|I|))$.

5.4.4 Optimization

After the filtering process, a *post-pruning* (in short PP) phase is applied in order to remove sets that are fully contained in other sets. Note that although the purpose of the filtering procedure is to remove these sets, this procedure is not guaranteed to be exhaustive, since redundant sets can occur across different paths. The post-pruning phase can be applied to the naive approach as well.

We use an Integer Linear Program to solve Set-cover optimally. Finally, the optimal set of remote transmissions is extracted from the optimal subfamily returned by Set-cover.

5.4.5 Adaptive extension

In real world, it is difficult for many applications to guarantee that moving objects travel with known trajectories over a long time interval. However, it is reasonable to assume that moving objects stick to known traveling plans in near future. In this case, the time dimension is partitioned into discrete time slots, where trajectories of moving objects are updated after each time slot. PIE can adapt to this variation without much modification. In the following, we briefly introduce two possibilities: *null-initial-state* and *adaptive* extensions.

The most straightforward way is to build an independent index for each time slot. We call it null-initial-state extension because this method simply ignores previous knowledge and treats each time slot as a new start. One weakness of this method is that it neglects information objects transmitted during the prior time slots, producing more remote transmissions. An alternative is to apply the adaptive extension. To reuse data objects transmitted before, we keep track of the distribution of the objects over the nodes, together with the remote transmission time of each object, and use them as initial state for the new time slot. Some data needs can be satisfied without any additional remote transmissions and will not be considered in the computation.

5.5 Experiment

5.5.1 Dataset

Cabs Mobility [144] (CAB, for short) contains mobility traces of taxi cabs in San Francisco, USA. It consists of GPS coordinates of 536 taxis collected over 23 days in the San Francisco Bay Area. The average time interval between two consecutive location updates is less than 10 seconds.

GeoLife GPS Trajectories [7] (GeoLife, for short) is a GPS trajectory dataset collected in (Microsoft Research Asia) GeoLife project by 165 users in a period of over two years.

Synthetic trajectories (SYN, for short) consists of $10K$ nodes that move randomly on a 2-D plane with size 3600 km^2 over 10 days. Starting from a uniformly random position, the speed of each node is updated periodically with normal distribution ($\mu = 1.2$ m/s and $\sigma = 1$) as well as its direction ($\mu =$ current direction and $\sigma = 1$ radiant). The updating rate is generated with exponential distribution ($\mu = 60$ sec).

For all datasets, the radio range is set to 100 meters. The data needs are generated by the following process. First, for each moving node, the number of data needs is generated with a Poisson distribution. Then, each data need is generated with a deadline uniformly distributed and a latency normally distributed ($\mu = 15$ min and $\sigma = 1$).

5.5.2 Implementation

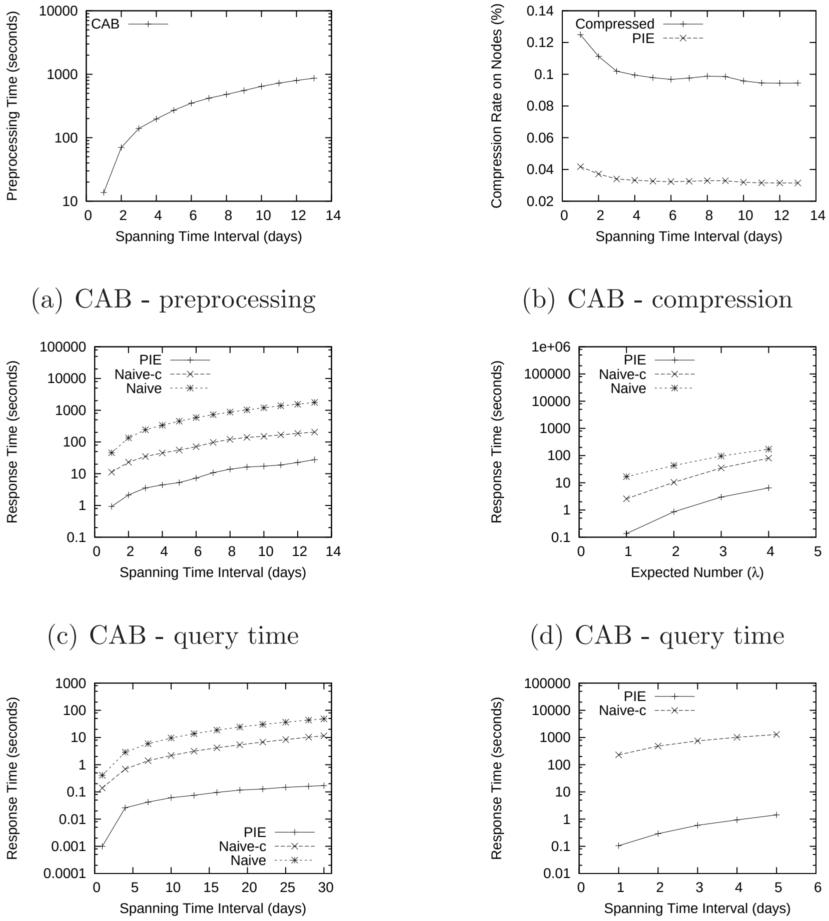
We implemented the naive approach described in Section 5.3.2 on the space-time graph. We also implemented the naive approach on the compressed graph (called *naive-c* for short) and the PIE indexing system (Section 5.4). All methods include the post-pruning phase described in Section 5.4.4. We tried a version without the post-pruning phase, obtaining a slight degradation of performances in each method. We also tried to run the ILP program described in Section 5.3.1, but it did not terminate due to the huge number of variables and constraints (hundreds of billions) considered. All the approaches were implemented in C++ (Dev C++ IDE ver. 4.9.9.2). The experiments were performed on a DELL Intel core I7 CPU with 2 Gb of memory. For the ILP solver we use *lp_solver* 5.5.2.0 [37], an open source tool based on branch-and-bound.

5.5.3 Response time

Each dataset is first preprocessed and its PIE index is generated. Figure 5.6(a) reports the preprocessing time on CAB, concerning a number of datasets spanning from 1 to 13 days. Depending on the dataset size, the preprocessing phase take tens through thousands of seconds. Although the preprocessing phase is sometimes expensive, it is executed only once. The rate of compression of the PIE graph and the compressed graph with respect to the space-time graph is shown in Figure 5.6(b). The compressed graph is about 16 times smaller than the space-time graph and PIE further reduces the size of about three times.

On CAB, the execution time for demand cover queries is shown in Figure 5.6(c) and 5.6(d). Figure 5.6(c) shows the execution time for a number of datasets spanning from 1 to 13 days. The average number of data needs per cab per day is set to 2. The reported times represent an average over 10 queries. PIE performs about two times faster than naive-c and four times faster than naive in almost all cases. In order to evaluate the scalability over the size of the query, we generate queries by varying the expected number of data needs per cab per day from 1 to 4. The results over 1 day are reported in Figure 5.6(d). For more than 4 expected data needs, the naive method is unable to answer queries in acceptable time.

We also execute the adaptive extension (Section 5.4.5) on CAB, for one day with time slot 15 minutes. Over a total number of 1089 data needs, null-initial-states method returns 675 remote transmissions, while the adaptive method returns 617 ones, with approximately a 10% improvement. For reference, the number of transmissions suggested by using full knowledge is 480. All the results of the adaptive extension refer to an average over 10 executions.



(a) CAB - preprocessing

(b) CAB - compression

(c) CAB - query time

(d) CAB - query time

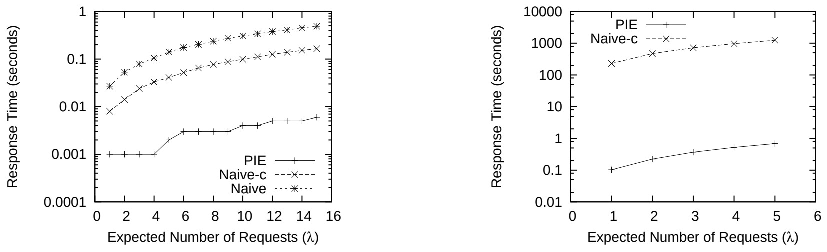
(e) GeoLife

(f) SYN

Figure 5.6: Performances as functions of the size of the dataset (number of days) and the number of data needs.

Figure 5.6(e) show the execution time for demand cover queries on GeoLife. The expected number of data needs per person per day is set to 10. The results refer to a set of datasets, each of them spanning a time interval ranging from 1 to 30 days. As for CAB, the reported times represent an average over 10 queries. In this dataset, PIE scales better than naive and naive-c with length of the spanning interval. For SYN, the results are reported in Figure 5.6(f). They refer to one data need per node per day. The naive approach here is not able to terminate in acceptable time even for one day, therefore we report only PIE and naive-c. PIE performs about three orders of magnitude faster then naive-c.

5.5.4 Scalability over query size

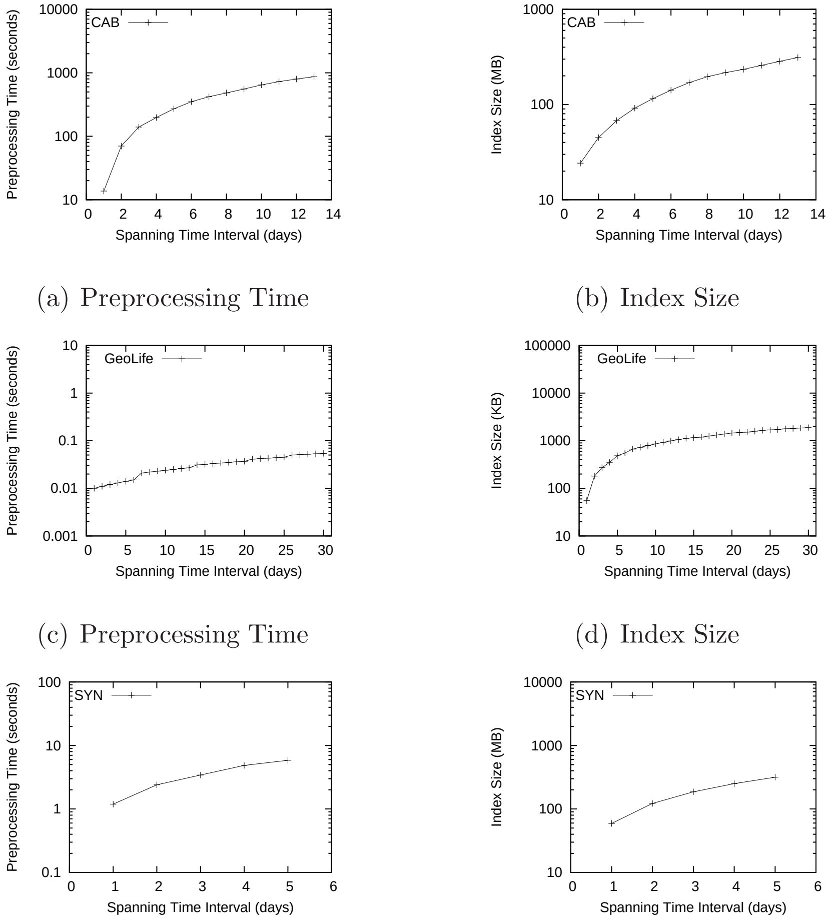


(a) GeoLife

(b) SYN

Figure 5.7: Scalability over query size

Figure 5.7 shows the execution time at varying the expected number of data needs per node per day, on GeoLife and SYN datasets (we refer to Section 5.5 for CAB). On both datasets the time interval spans one day. On GeoLife (a), PIE is over one order of magnitude faster than naive-c and about two orders of magnitude faster than naive when the time interval spans more than three days. Increasing the number of data needs, the degradation of performance is less pronounced in PIE. On SYN (b), PIE is about three orders of magnitude faster than naive-c. The results for naive are not reported since it was not able to terminate in acceptable time even for one data need.



(a) Preprocessing Time

(b) Index Size

(c) Preprocessing Time

(d) Index Size

(e) Preprocessing Time

(f) Index Size

Figure 5.8: Preprocessing time and index size produced by CIDP on different datasets. The first row refers to CAB, the second row refers to GeoLife and the last row refers to SYN.

5.5.5 Preprocessing

The preprocessing time and the index size of PIE for all datasets are reported in Figure 5.8. Since the preprocessing phase does not apply to naive, we do not compare with it. In order to verify the scalability, we evaluate the algorithm on a number of datasets. For CAB, each dataset spans a number of days ranging from

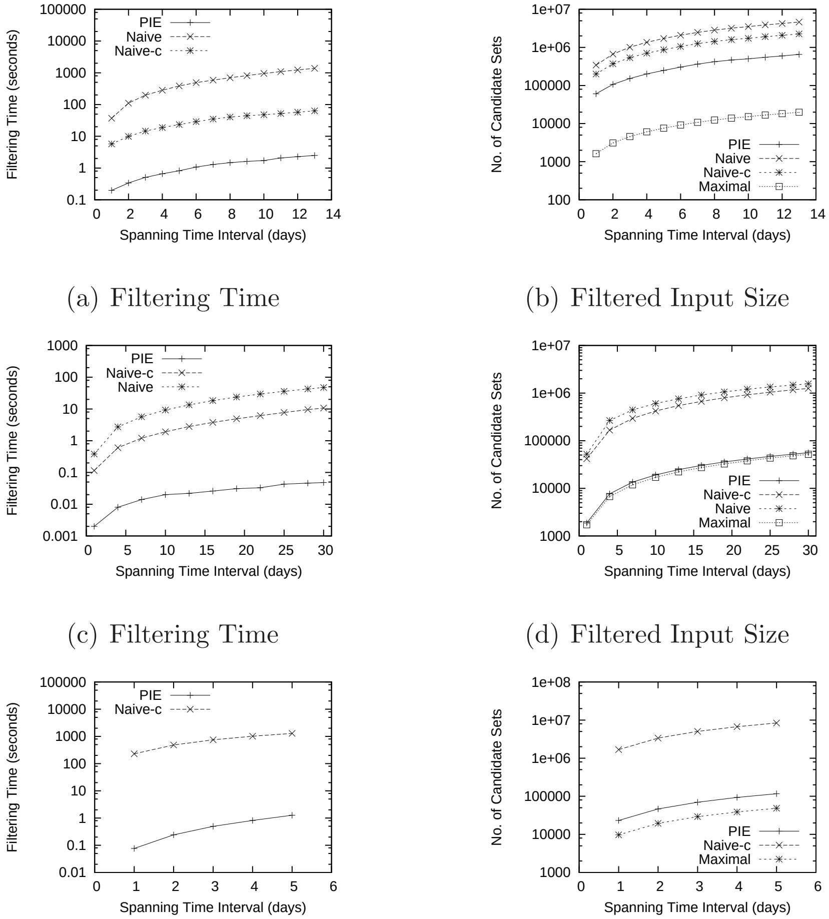
1 to 13. For GeoLife, each dataset spans a number of days from 1 to 30. For SYN, the spanned interval ranges from 1 day to 10 days.

For CAB, the time for preprocessing and the size of index are presented in Figure 5.8(a) and 5.8(b), respectively. For GeoLife, the preprocessing time and the index size of PIE are shown in Figure 5.8(c) and 5.8(d), respectively. In GeoLife, the preprocessing time is less than that for CAB and the index size is smaller. The main reason is that the number of events (contact beginnings and contact ends) captured in GeoLife is much smaller than the one in CAB. In GeoLife, the average number of events per day is $1,401$, while in CAB it is $809,558$.

For SYN, the preprocessing time and the index size is shown in Figure 5.8(e) and Figure 5.8(f), respectively. This experiment performs on a number of datasets, each of them spanning a number of days ranging from 1 to 5. The number of events captured in SYN is $904,818$ per day, which is comparable to that in CAB. Since SYN contains $10,000$ moving nodes, the average number of events per node per day is 90.5 while for CAB it is $1,510$. Therefore a moving node in CAB has more opportunities to connect to other nodes and it takes more time to identify connected components.

5.5.6 Filtering capability

In Figure 5.9, the filtering time and the size of the input family for Set-cover obtained by naive, naive-c and PIE are reported, respectively for all datasets. For naive, the filtering time refers to the time for generating the family of sets. PIE strongly outperforms naive on both filtering time and size of the input family



(a) Filtering Time

(b) Filtered Input Size

(c) Filtering Time

(d) Filtered Input Size

(e) Filtering Time

(f) Filtered Input Size

Figure 5.9: Evaluation of filtering capability. The first row refers to CAB, the second row refers to GeoLife and the last row refers to SYN.

generated for Set-cover in all cases. Figure 5.9(a), 5.9(c) and 5.9(e) show that PIE performs more than 100 times faster than naive. PIE performs more than two times faster than naive-c in CAB and up to three order of magnitude faster than naive-c in SYN. Figure 5.9(b), 5.9(d) and 5.9(f) show that the size of the input family filtered by PIE is much smaller than the one generated by naive and naive-c. The size of the input family after post-pruning (referred as Pruned) is also reported and show that in SYN and GeoLife the input family generated by PIE is almost the most compact one. For CAB, the high number of encounters per node degrades the performances. However, PIE produces a family eight times smaller than one generated by naive-c.

5.5.7 Communication cost

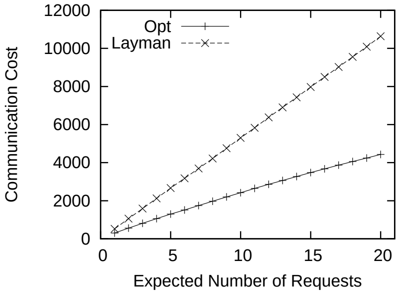


Figure 5.10: Communication cost with varying number of requests per cab per day

In Figure 5.10 we show the communication cost obtained by our solution, compared to a layman approach that performs a remote transmission for each data need. We do not distinguish between naive and PIE since they produce the same result. We perform this experiment on CAB. The queries are generated by varying the expected number of data needs per cab per day λ from 1 to 20. With $\lambda = 20$ our solution reduces the number of transmissions by more than 50% and the gain increases with λ.

5.6 Related Work

Previous works on DTNs focus on three types of contacts including scheduled, predicted and opportunistic contacts. Scheduled contacts result from applications of known trajectories, such as deep-space communication and data service in developing regions [21, 94, 122, 145]. Predicted contacts are considered when there exist mobility patterns in applications [23, 84, 117, 123, 168, 169, 200]. Opportunistic contacts deal with completely uncertain circumstances where mobile nodes meet each other by random chance. Our work falls into the category of scheduled contacts [94, 145].

Graph representations are widely applied in studying routing strategies. In [40, 128, 145], evolving graphs are employed to model topological mutations in DTNs. In our work, we use compression and indexing techniques to efficiently explore evolving graphs with the purpose of minimizing the communication cost.

Multicast for DTNs has recently drawn considerable attention. In [203], semantic models are proposed to unambiguously describe multicast in the context of DTNs. The throughput of multicast is discussed in [116] and mobility-assisted routing is used to improve the throughput bound of wireless multicast. In [72], multicast problems in DTNs are considered in a social network setting where centrality and community in DTNs are employed to help determine the appropriate selection of relays, with the objective of minimizing the delay of multicast message transmissions. In this paper, we study a novel optimization problem which is similar to the traditional multicast problems. However, instead of minimizing the delay of message transmissions, we are interested in minimizing the communication cost subject to some time constraints. To this end, the question of whether a node is reachable from another node is more important than the question of how a message flows in the network. In our problem setting, the DTN is a medium to propagate information, and our goal is to maximize the role it plays in information sharing.

Graph indexing systems have been widely studied by the database community. The most common approaches aim to efficiently solve problems as graph matching [132, 199] or reachability test [53, 96]. The closest to our work are systems for reachability tests, which aim to efficiently check if two vertices are reachable from each other (a path that connects them exists) in a directed graph. Some systems use chains [93] (generalization of paths) decomposition or path-tree [96] decomposition. The underlying idea is that if a vertex u of a chain (or a tree-of-paths) is reachable from another vertex v, all the vertices downstream in that chain are reachable from v. We use a similar idea, but our system is designed to fast identify the regions of the graph that can reach a given destination instead of verifying the reachability between pairs. Moreover, we give a method for quickly identifying a small subset of representative vertices that allows us to solve the demand cover problem optimally and with reasonable efficiency on large datasets.

5.7 Summary

In this chapter, we present a new approach that optimizes the long-range communication cost for multicast in DTNs. By formalizing the optimization problem as a temporal reachability problem and showing its NP-hardness, we provide a graph-indexing-based solution. Our system can solve the problem optimally on large real instances (dataset with million of events and queries with thousands of nodes) in less than 10 seconds in most cases. There are two potential extensions to our work. First, we can take into account the uncertainty in mobility and data needs. For this, we need to fit stochastic mobility models in our framework and optimize the expected communication cost. Finally, we can consider the problem of scheduling new trajectories with the purpose of guaranteeing the connectivity, in the case when the communication with a central data source is not always available.

Chapter 6

Partitioning in Temporal Graphs

6.1 Introduction

Complex Event Processing (CEP) has been popular for many applications [115]. Platforms such as S4 [12, 140], Storm [14], Photon [24], MillWheel [19] and Amazon's Kinesis [3], enable scalable data analytics over data streams. More recently, we are observing the rise of stream processing platforms that ingest slower-rate data streams, but allow more expressive operations. For example, the back-end that supports the Cortana personal assistant [10] executes, on behalf of its users, queries that monitor and process traffic, weather, news, and other events, and forwards events of interest to its users (Figure 6.1). Such queries typically hold the following properties.

(a) They usually bring low network and computation overheads (*e.g.,* processing weather updates for a region);

(b) They may join multiple streams, such as a personal event stream (*e.g.,* calendar updates) with a global stream (*e.g.,* traffic updates), to generate user events (*e.g.,* reminder to leave to catch next appointment);

(c) They support a very expressive programming model (*e.g.,* using UDF's expressed in LINQ [9], or JavaScript as in Amazon's Lambda [4]).

Property (a) and (c) suggest that it is natural to execute each query in a single server (unlike systems that scale-out processing to deal with high stream data rates), but due to (b) the processing overhead is often not trivial to estimate from the query. Even though each query is rather "small", the expectation is that there will be a very large number of queries that need to be supported with low network and computational resources. Hence, the main challenge in building a platform to support a vast number of "small" queries is to allocate queries to servers such as to minimize network and compute overheads.

Figure 6.1 depicts a typical platform that hosts queries on behalf of users. External to the system, there are event sources that generate events of interest to the users; these can be events of general interest, such as news, weather, stocks, traffic, flight updates, and personalized events, such as calendar events, user location events, etc. Event gateways ingest events from each source, and then forward them to the internal processing nodes (*query evaluators*). In practice, user-defined queries can dynamically arrive and leave; therefore, events are essentially flowing in a temporal graph.

It is important to observe that there is one or very few event gateways per event source, but then the gateway(s) forward the stream to all query evaluators that host user queries that depend on that event source, potentially by replicating

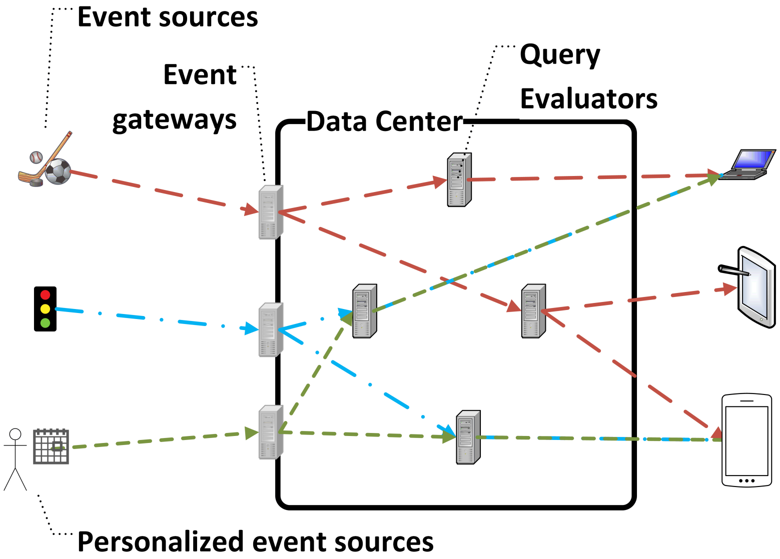


Figure 6.1: Example of events flowing from the event sources to the query evaluators, and finally to users. In addition to global events, there are also personalized event sources; in this example, a query combines traffic updates with calendar events to generate notifications for the user (*e.g.,* time to leave to be on time for next appointment given current traffic). Note that as user queries can dynamically come and go, the data flow graph is temporal.

the stream multiple times. The query evaluators execute their queries and forward the results to the end-users. In this work, we study specifically the problem of minimizing the network overhead for forwarding the events from the event gateways to query evaluators while balancing the load among the query evaluators. We assume that the service is hosted in a generic compute platform, such as Azure Compute [15], Amazon's EC2 [2], or Rackspace [11]; this is a typical requirement for many modern services as it separates the management of the platform from the operation of the service. Hence, we cannot control the assignment of the query evaluators to the underlying platform, and cannot rely on an efficient transport (*e.g.,* multicast) of the events from the gateways to the evaluators. However, we can assume that each stream will be delivered at most once to each query evaluator, even when the evaluator serves multiple queries operating on that stream. Hence, there is a unique network connection per stream between the gateway and each of the query evaluators interested in the stream, and we want to minimize the network overheads incurred by those connections.

As a simple example, consider that all streams have the same popularity and that their aggregate rate is W (this is the rate from the event sources to the gateways). The aggregate rate W can be many 10's of Gbps, even though each individual stream is much smaller. The rate from the gateways to the evaluators will be at least W (best case). However, if there are k query evaluators, typically k is many 100's, a naive allocation of queries of evaluators can result into W incoming rate per evaluator, which can violate the evaluator's capacity, and a total rate inside the platform of $k \cdot W$, which incurs significant network load.

Our approach strives to minimize total network load, and at the same time be mindful of capacity constraints and processing overheads incurred when executing the queries. These are non-trivial because the platform allows a flexible query processing programming model. In other words, a solution that places all queries using the same stream to the same query evaluator does not scale for popular queries and streams. One approach to capture this requirement would be to associate capacity limits for all critical resources of the system (*e.g.,* network bandwidth, processing and memory demands per server). However, this approach requires a priori knowledge of those limits, and in our experience may result in very unbalanced allocations that are not desirable from an operational point of view. Instead, our allocation strives to balance the load in the system and reduce network overheads.

The main insight that enables us to optimize the allocation of queries to evaluators is the observation that many queries will use the same input streams. For example, many users may be interested in traffic updates from the same city, or weather updates for the same region. This is akin to the existence of many applications for *e.g.,* smart-phones that present weather, news, and other information to end-users using the same sources, and the large number of users for many of these similar applications. Hence, we anticipate large benefits by co-locating similar queries to the same query evaluator, and transporting the relevant network streams once.

Three practical requirements complicate query assignment.

1. Queries should be able to subscribe to more than one stream. This is required, for example, to support joins, and it is a common feature in many CEP systems.

2. The assignment of queries to servers should be semi-permanent, which means that the platform should avoid moving queries between servers. For example, to reduce overheads, as this requires moving (query) state while guaranteeing that the query does not miss any stream updates. Obviously, queries will be re-assigned when their server fails, but such events should be an exception. Hence, the platform must make a good decision when assigning a query to a server upon query arrival.

3. We expect churn both in the queries (queries have limited lifetime) and in servers (due to server failures and re-cycles). The queries arrive and depart dynamically, and the assignment of queries to servers should be robust to query and server dynamics.

In this work, we propose and study the problem of assigning streaming queries to query evaluators (servers), under the requirements and assumptions described above. We model the data flow in the system as temporal graphs, use both analysis and simulations to understand the complexity of the problem, and design efficient algorithms for assigning queries to servers.

The remaining of this chapter are organized as follows. In Section 6.2, we formulate the problem of assigning streaming queries to servers in the context of an online platform that hosts queries on behalf of the users as a service. We show that the problem of reducing network load while balancing server load is NP complete in Section 6.3, and provide approximation bounds in Section 6.4. In Section 6.5, we use the offline algorithms to reason about the performance of the assignment process, and to draw inspiration to develop effective algorithms for online cases. Using analysis and simulation in Section 6.6, we identify the online heuristic that gives the best performance, even under query and server churn, and is often up to four times better than (naive) random assignment. We discuss related work in Section 6.7 and summarize this work in Section 6.8

6.2 System Model and Assumptions

We consider a stream processing platform with the following three key components (see also Figure 6.1): (event) *sources*, *queries*, and *servers* (query evaluators).

Sources. A source is a publisher that generates new events as a data stream at some rate. $S = \{s_1, s_2, \ldots, s_m\}$ denotes a set of sources, and $w(s)$ is the rate of events published by source $s \in S$. The total event rate for a subset of sources

$S' \subseteq S$ is denoted by $w(S') = \sum_{s \in S'} w(s)$. For convenience, and without loss of generality, we shall treat sources as if located in the event gateways.

Queries. A query subscribes to one or multiple sources and processes the events originating from the sources. A set of queries is denoted by $Q = \{q_1, q_2, \ldots, q_n\}$, and the set of sources subscribed by a query $q \in Q$ is represented as S_q, which is a subset of the set of sources S.

If two queries subscribe to identical sets of sources, we say that they are of the same *query type*. Let $T \subseteq 2^S$ be the complete set of distinct query types. Each query $q \in Q$ has a query type $t \in T$, where $t = S_q$. Given n queries, we have $n = \sum_{t \in T} n_t$, where n_t is the number of queries of query type t.

Moreover, $N(s)$ denotes the subset of queries that subscribe to source s, that is, $N(s) = \{q \in Q \mid s \in S_q\}$.

Servers. A server is a container that evaluates queries. We assume there are $k \geq 1$ servers in a stream processing platform. (The terms “server” and “query evaluator” are used interchangeably; we mostly use “server” for brevity.)

6.2.1 Query Assignment Problem

We consider optimizing query assignment with respect to the following two criteria: network traffic and server load.

Network Traffic. We are interested in minimizing network traffic between sources and servers. If a server hosts at least one query that subscribes to a source s, then this contributes $w(s)$ to the network traffic cost. This implies that it is desirable to co-locate queries that use the same source(s). Formally, the total network traffic cost of a server that hosts queries $Q' \subseteq Q$ is defined as

$$f(Q') = \sum_{s \in S} w(s) \cdot \mathbb{1}\{s \text{ is required by some } q \in Q'\}.$$

The total network traffic of an assignment of queries to servers according to the sets of queries $Q_1, Q_2, \ldots, Q_k$ assigned to respective servers $1, 2, \ldots, k$ is given by: $\sum_{j=1}^{k} f(Q_j)$.

Server Load. A query assignment is feasible if it approximately balances the processing load of servers up to a given slackness. In practice, it is non trivial to quantify the capacity of a server. Servers are typically hosted by virtual machines in a cloud service platform. The capacity of a server depends on several factors including the processing requirements of queries, and other factors such as the load of virtual machine to which the server is assigned. Therefore, we aim at balancing the load over different servers which does not require knowing exact capacities of individual servers. The underlying assumption is that the system operates at a load that allows for a feasible query assignment. In this case, balancing the load across different servers is a natural objective.

In this work, we assume that each query contributes a fixed processing load to the server it is assigned to. For simplicity of exposition, we assume queries contribute identical processing loads, say of unit value, but our results naturally generalize to non identical query processing loads. In this way, the processing load of a server corresponds to the number of queries assigned to this server.

Given a *slackness parameter* $\nu \geq 0$ for balancing the load across servers, a query assignment to k servers is specified by a partitioning of the set of queries Q into k disjoint subsets $Q_1, Q_2, \ldots, Q_k$. A query assignment is said to be *ν-load* *balanced*, if it satisfies the following condition:

$$|Q_j| \leq (1+\nu)\frac{n}{k}, \text{ for } j = 1, 2, \ldots, k. \tag{6.1}$$

Throughout the paper, we interchangeably refer to the parameter ν as the slackness parameter or *relative relaxation* parameter.

Query Assignment Problem (QA) Let $\mathcal{P}(Q)$ be the set of all possible k-partitions $Q_1, Q_2, \ldots, Q_k$ of the set of queries Q such that (a) $Q_1 \cup Q_2 \cup \cdots \cup Q_k = Q$; and (b) $Q_i \cap Q_j = \emptyset$ for every $i \neq j$.

The QA problem is defined as follows: given a set of sources S, a set of queries Q, a set of k servers, and a slackness parameter $\nu \geq 0$, find $(Q_1, Q_2, \ldots, Q_k) \in \mathcal{P}(Q)$ which minimizes the network traffic

$$\sum_{j=1}^{k} f(Q_j)$$

subject to the server load balancing constraints (6.1).

6.3 Hardness and Benchmark

In this section, we first discuss the computational complexity of the query assignment problem (QA), and then characterize the inefficiency of a standard load balancing strategy that assigns each query by sampling a server uniformly at random.

6.3.1 NP Hardness

In general, it is computationally hard to find an optimal solution for an arbitrary QA instance in polynomial time as showed in the following theorem.

Theorem 8. *Query assignment problem is NP-complete.*

Proof. The proof consists of two steps: (a) we first prove QA's decision problem is NP-hard; and (b) show that it is NP.

First, we prove its NP-hardness by a reduction from the well-known bin packing problem [73].

The decision problem of QA is described as follows: Given the set of sources S, the set of queries Q, k servers, a slackness parameter $\nu \geq 0$, and a real number $\gamma \geq 0$, does there exist $(Q_1, Q_2, \ldots, Q_k) \in \mathcal{P}(Q)$ such that (a) $\sum_{j=1}^{k} f(Q_j) \leq \gamma$, and (b) every server is ν-load balanced?

Consider a special case of QA's decision problem, where (a) each query subscribes to exactly one source, (b) each source publishes events at a unit rate $w(s) = 1$ for every $s \in S$, and (c) $\gamma = |S|$. In other words, we need to find a feasible solution such that queries of the same type are assigned to the same server. We can reduce an arbitrary instance of bin packing problem to an instance of the special case under consideration: (a) we reduce an item to a query type, where the item's size is reduced to the number of queries for the corresponding type; (b) the number of bins is reduced to the number servers; and (c) the size of each bin is reduced to the upper limit for each server's load. Since bin packing problem is NP-hard, we conclude that QA's decision problem is NP-hard.

Second, given a solution to QA's decision problem, we can check whether it is feasible in polynomial time, so the QA problem is NP. Therefore, QA is NP-complete. □

The following competitive ratio holds for every feasible query assignment.

Proposition 8. *For every assignment of queries to servers, the network traffic cost is at most k times the optimum network traffic cost, where k is the number of servers.*

6.3.2 Random Query Assignment

A naive query assignment strategy is to assign each query to a server sampled independently, uniformly at random. This is a standard load balancing strategy that can be implemented by hash partition of query identifiers. This strategy can efficiently balance the number of queries over servers. Specifically, it is known to guarantee the maximum server load of $n/k + O\left(\sqrt{(n \log k)/k}\right)$ with probability $o(1)$ for $n \gg k \log^3 k$ [148]. However, this strategy can be grossly inefficient with respect to network traffic cost, which we show analytically below and also experimentally in Section 6.6.

Proposition 9. *Consider the set of sources S ($|S| = m$) and k servers, with d_s denoting the number of queries subscribed to source $s \in S$. The expected network traffic cost under uniform random query assignment strategy is*

$$\left(1 - \frac{1}{m} \sum_{s \in S} \left(1 - \frac{1}{k}\right)^{d_s}\right) km. \tag{6.2}$$

Proof. Consider an arbitrary source $s \in S$ and an arbitrary server $j \in K$. Under random query assignment, at least one query that requires input from source s is assigned to server j with probability $1 - (1 - 1/k)^{d_s}$. Summing over all servers j gives the expected number of servers to which the stream of source s need to be transferred. Summing further over all sources $s \in S$ gives the expected number of streams that need to be transferred from sources to servers, which corresponds to total network traffic. □

Proposition 9 implies that the naive strategy can easily achieve the upper bound in Proposition 8 and result in a large amount of network traffic cost. From Equation 6.2, the expected network traffic cost of random query assignment is nearly equal to the worst-case network cost whenever $\frac{1}{m}\sum_{s\in S}(1-1/k)^{d_s} \ll 1$. In fact, the worst-case network traffic cost is achievable under the random query assignment policy: consider m sources and n queries partitioned into k balanced pieces so that there are m/k sources and n/k queries in respective pieces $S_1, S_2, \ldots, S_k$ and $Q_1, Q_2, \ldots, Q_k$, and assume that each query in Q_j only subscribes to the sources in S_j and none in $S \setminus S_j$. The subscription between queries and sources corresponds to a collection of k disconnected complete bipartite graphs, each of which has m/k sources and n/k queries. In this case, the expected network traffic cost is

$$(1-(1-1/k)^{n/k})km,$$

which for large n tends to the worst-case network traffic cost of km. On the other hand, the best strategy in this case is to assign each piece of queries to a distinct server, which achieves minimal network traffic and perfect load balancing. Note that the inefficiency of the naive strategy can be made arbitrarily large by taking k large enough.

The high network traffic cost caused by naive strategies such as random query assignment asks for the design of more sophisticated query assignment algorithms. In the next section, we focus on offline QA problem, and propose approximation algorithms with better performance guarantees. In Section 6.5, we discuss online algorithms that irrevocably assign queries to servers at their arrival.

6.4 Offline Query Assignment

In this section, we investigate practical approximation algorithms for offline QA problem.

We first consider approximation algorithms for the general case where each query subscribes to one or multiple sources, which we refer to as *multi-source* QA. For sources with identical event rates, we propose an approximation algorithm that guarantees the network traffic cost of at most $2d_{max}(1+\log k)$ of the optimal network traffic cost, where d_{max} is the maximum number of sources required as input to a query, and k is the number of servers. Since the value of d_{max} is usually a small constant in practice [24], this is a much tighter bound compared with the worst-case bound of k. We also develop several heuristic query assignment algorithms that exhibit competitive performance in practice.

Moreover, we develop a 2-approximation algorithm for the case where each query subscribes to exactly one source, which we refer to as *single-source* QA.

6.4.1 Multi-Source Query Assignment

We first present an approximation algorithm and then introduce two heuristics for offline multi-source QA.

Minimum Query Type Packing

In this section, we establish the following main theorem.

Theorem 9. *Suppose that all sources have identical event rates. There exists a polynomial-time algorithm for multi-source* QA *with the approximation ratio*

$$2d_{max}(\log k + 1)$$

where $d_{max} = max_{q \in Q}|S_q|$ *is the maximum number of sources subscribed by a query. Furthermore, the same bound holds for sources of arbitrary rates with an extra factor* $\omega = \max_{s \in S} w(s) / \min_{s \in S} w(s)$*.*

Theorem 9 tells us that when queries subscribe to a number of sources that is bounded by a constant, we have an approximation guarantee of $O(\log(k))$ where k is the number of servers. This result is of practical interest in applications where each query subscribes to a few sources, and the asserted approximation ratio comes from an algorithm that approximately solves a *single-server minimum query type packing* problem, which we introduce shortly. We shall prove the correctness of Theorem 9 in the following three steps.

1. We show that if we can optimally solve k *single-server minimum query type packing (*MQP*)* problems, then we can approximate multi-source QA within factor $2(1 + \log k)$.

2. MQP is NP-complete.

3. MQP can be approximated within d_{max} in polynomial time.

We start with the definition of MQP.

Single-Server Minimum Query Type Packing (MQP). Given the set of queries Q, the set of sources S, and a real number $\theta > 0$, find a subset of query types $T' \subseteq T$ with $S' = \bigcup_{q \in Q \wedge S_q \in T'} S_q$ that minimizes $w(S')$, subject to the constraint: $\sum_{t \in T'} n_t \geq \theta$.

The multi-source QA can be approximated by sequentially solving MQP in k rounds, with the following performed in round j:

1. Select an empty server as the target for query assignment from the pool of remaining queries $Q^{(j)}$.

2. Suppose we find the optimal subset of query types $T' \subseteq T$ for the MQP problem with queries $Q^{(j)}$ and

$$\theta = n - (1+\nu)\frac{(k-1)n}{k}. \tag{6.3}$$

   Let $\hat{Q} \subseteq Q^{(j)}$ be the subset of queries of query types in T'. Assign $\hat{Q}$ to the selected server.

3. If $|\hat{Q}| > (1+\nu)\frac{n}{k}$, we arbitrarily select a query type $t' \in T'$, remove a number of queries of type t' to make the server ν-load balanced, and put the removed queries back to the query pool. Note that since $\theta < (1+\nu)n/k$, we only need to select one query type for query removal. If we need to select more than one query type, T' cannot be an optimal solution.

Since the constraints in the QA problem imply that $|Q_j| \geq \theta$, for every server $j = 1, 2, \ldots, k$, the MQP problem with θ can be seen as a relaxation of the QA problem.

Let $\hat{Q}_j^* \subseteq Q$ be the optimal solution for the single-server MQP problem on the j-th server, $f(\hat{Q}_j^*)$ be the network traffic cost, and OPT be the optimal solution for multi-source QA problem.

Lemma 4. *Given a multi-source* QA *problem with k servers, successive solving of k* MQP *problems yields a feasible solution for the* QA *problem. Moreover, if we can solve each* MQP *problem optimally with $\hat{Q}_1^*, \ldots, \hat{Q}_k^*$, we can guarantee*

$$\sum_{j=1}^{k} f(\hat{Q}_j^*) \leq 2(\log k + 1)\mathrm{OPT}.$$

Proof. he proof follows by upper bounding the cost incurred in each round where queries are assigned to a server by solving a single-server MQP problem. We first show the upper bound for the traffic cost of assigning queries to the first server, and then show how we bound the traffic cost for other servers.

Let $\mathrm{OPT}_j(n')$ be the optimal solution for a multi-source QA problem with n' queries, j servers, and the capacity constraint $(1+\nu)\frac{n}{k}$. Note that $\mathrm{OPT}_k(n) = \mathrm{OPT}$. For the first single-server MQP problem, let $\hat{Q}_1^*$ be an optimal subset of queries assigned to server 1 with traffic cost $f(\hat{Q}_1^*)$. Since single-server MQP problem is a relaxation of the QA problem, it holds $f(\hat{Q}_1^*) \leq f(Q_j^*)$, where Q_j^* is the subset of queries assigned to server j in OPT for every $j = 1, 2, \ldots, k$. Since $\mathrm{OPT}_k(n) = \mathrm{OPT}$, we obtain

$$f(\hat{Q}_1^*) \leq \frac{1}{k}\mathrm{OPT}_k(n) = \frac{1}{k}\mathrm{OPT}. \tag{6.4}$$

Consider now the j-th server. Let $\mathrm{OPT}_{k-j+1}(n')$ be the optimal solution given n' remaining queries, $k-j+1$ servers, and the capacity constraint $(1+\nu)\frac{n}{k}$ (note that this constraint remains the same throughout the execution of the algorithm). We claim that

$$f(\hat{Q}_j^*) \leq \frac{2}{k-j+1}\mathrm{OPT}, \text{ for } j = 2, 3, \ldots, k. \tag{6.5}$$

Suppose inequality (6.5) is true, then the proof of the lemma follows by summing up the upper bounds in (6.4) and (6.5) and using the fact that for the harmonic series H_k it holds $H_k \leq \log k + 1$. We prove inequality (6.5) as follows.

First, given n queries and the same capacity constraint per server, if there exist feasible solutions for a system of j and k servers, such that $j \leq k$, then we prove that $\mathrm{OPT}_j(n) \leq 2\mathrm{OPT}_k(n)$. For $\mathrm{OPT}_k(n)$, let $cost_i$ be the traffic cost for server i. Without loss of generality, suppose that the servers are enumerated such that $cost_1 \geq cost_2 \geq \ldots \geq cost_k$. Then, we have

$$\mathrm{OPT}_k(n) = \sum_{i=1}^{j} cost_i + \sum_{i=j+1}^{k} cost_i.$$

Using $\mathrm{OPT}_k(n)$, we can construct a feasible solution that requires only j servers by (1) arbitrarily selecting a server $a \leq j$ with available space, and (2) sequentially assigning queries on server $b > j$ to server a. If server a is full before all queries from server b are assigned, then arbitrarily select another server $a' \leq j$ with available space for the remaining queries from server b, and we repeat the procedure until all queries from server b are assigned. If all queries from server b are assigned but server a still has available space, we find another server $b' > j$, and assign queries from server b' to server a. By the above procedure, we can construct a feasible solution using only j servers. The resulting extra cost is no more than $j * cost_{j+1}$, since in the above procedure we break the sequential assignment at most j times, and each time add in no more than the cost of $cost_{j+1}$. Therefore,

$$\begin{aligned}\mathrm{OPT}_j(n) &\leq \sum_{i=1}^{j} cost_i + \sum_{i=j+1}^{k} cost_i + j * cost_{j+1} \\ &\leq 2\sum_{i=1}^{j} cost_i + \sum_{i=j+1}^{k} cost_i\end{aligned}$$

and, thus, it follows that

$$\frac{\mathrm{OPT}_j(n)}{\mathrm{OPT}_k(n)} \leq \frac{2\sum_{i=1}^{j} cost_i + \sum_{i=j+1}^{k} cost_i}{\sum_{i=1}^{j} cost_i + \sum_{i=j+1}^{k} cost_i} \leq 2$$

Hence,

$$\mathrm{OPT}_j(n) \leq 2\mathrm{OPT}_k(n), \text{ for all } 1 \leq j \leq k. \tag{6.6}$$

Second, for k servers and the same capacity constraint, if there exists feasible solutions for assigning n_i and n_j with $n_i \leq n_j$, then

$$\mathrm{OPT}_k(n_i) \leq \mathrm{OPT}_k(n_j). \tag{6.7}$$

Since $f(\hat{Q}_j^*) \leq \frac{1}{k-j+1}\mathrm{OPT}_{k-j+1}(n')$, (6.6) and (6.7), it follows:

$$f(\hat{Q}_j^*) \leq \frac{2}{k-j+1}\mathrm{OPT}, \text{ for } 1 < j \leq k.$$

□

In Lemma 4, we derived an approximate algorithm for the QA problem under assumption of the existence of an oracle that provides optimal solutions to MQP problems. We next show that MQP is NP-complete; therefore, it is hard to find a polynomial-time algorithm that solves MQP optimally.

Lemma 5. MQP *problem is NP-complete.*

Proof. We sketch the proof as follows. (a) To prove NP-hardness, we can reduce the NP-hard minimum k-union problem [181] to single-server MQP problem. (b) It is easy to verify a solution in polynomial time. □

Due to the NP-hardness, we propose the following algorithm to approximate MQP.

1. Order query types in decreasing order with respect to the number of queries;

2. Successively pick a query type with the largest number of queries until the number of assigned queries is at least θ.

Lemma 6. *Suppose sources have identical event rates. The above algorithm approximates* MQP *within* $d_{max} = max_{q \in Q}|S_q|$.

For arbitrary source rates with $\omega = \max_{s \in S} w(s)/\min_{s \in S} w(s)$, *the above algorithm approximates* MQP *within* $d_{max}\omega$.

Proof. In the above algorithm, we pick the query types with the largest number of queries to saturate a server. Suppose we eventually select h query types, we then conclude that the number of query types considered in an optimal solution is no less than h. Let $h + \Delta$ be the number of query types obtained from the optimal solution. For each query type, the above algorithm takes at most d_{max} times more of the network traffic rate compared with the optimal solution, when the source traffic rates are identical. In the case of arbitrary source traffic rates with the ratio of the maximum source traffic rate to the minimum source traffic rate at most ω, it takes at most $d_{max}\omega$ times more network traffic rate. This completes the proof of the lemma. □

In summary, Theorem 9 is proved using Lemma 4, 5, and 6.

Heuristics

In this section, we present two heuristics for the offline QA, including *incremental cost* (referred to as IC) and *min-max traffic cost per server* (referred to as MMS). These heuristics are observed to exhibit competitive performance in practice.

Incremental Cost Based Approach. IC assigns queries in successive rounds. At each round, it assigns queries to a server in three steps. (a) Given a query type $t \in T$ with non-zero number of unassigned queries and a server j of spare capacity to host more queries, we consider the *incremental traffic cost* resulting from assigning at least one query of type t to server j. (b) We select a pair of a query type and a server (t^*, j^*) that results in *the least incremental cost*. (c) We assign queries of type t^* to server j^* as many as possible until server j^* is full or there are no unassigned queries of type t^*.

Min-max Traffic Cost per Server. MMS aims to minimize the maximum traffic cost among servers in successive rounds. At each round, it assigns queries to servers in three steps. (a) Given a query type $t \in T$ with non-zero number of unassigned queries and a server j with spare capacity to host queries, we consider the *traffic cost* after assigning at least one query of type t to server j. (b) We select a pair of a query type and a server (t^*, j^*) that results in *the least traffic cost*. (c) We assign as many as possible queries of type t^* to server j^* until server j^* is full or there are no unassigned queries of type t^*.

6.4.2 Single-Source Query Assignment

In this section, we consider single-source QA, where each query subscribes to exactly one source. In this case, there is a one-to-one correspondence between query types and sources. Therefore, we use the term source and the term query type interchangeably.

We present an approximation algorithm for single-source QA that assigns queries to servers over successive rounds as shown in Figure 6.2. For server j, let d_j to be the spare capacity of server j. At the beginning of round 0, we initialize $d_j = \lfloor (1+\nu)n/k \rfloor$, where $\nu \geq 0$ is the slackness parameter. Let n_s be the number of unassigned queries that subscribe to source s.

```
Input:  A single-source QA instance;
Output: A query assignment.

while True
    Select server $j$ with the largest free capacity
    Select query type (source) $s$ of the largest event rate $w(s)$
    $b \leftarrow \min(d_j, n_s)$
    Assign $b$ type-$s$ queries to server $j$
    $d_j \leftarrow d_j - b$
    $n_s \leftarrow n_s - b$
    if there is no more queries
        return
```

Figure 6.2: 2-approximation for single-source QA.

Theorem 10. *The approximation algorithm given in Figure 6.2 has the following approximation guarantees:*

1. *The approximation ratio $1 + k/m \leq 2$, where m is the number of sources and k is the number of servers, for sources with identical event rates;*

2. *The approximation ratio 2, for sources with arbitrary event rates.*

Proof. We consider only the cases where $0 < n_s < d_j$ (with $d_j = \lfloor (1+\nu)n/k \rfloor$) for every $s \in S$ and $j = 1, 2, \ldots,$ or k as the other cases can be reduced to these cases. (If there exists s' with $n_{s'} \geq d_j$, then we reserve a server for query type s', reduce $n_{s'}$ by d_j, remove the server, and repeat; this assignment is optimal.) We also assume that $k < m$. (If $m \leq k$ and since $0 < n_s < d_j$ for every s and j, an optimal assignment is to allocate each query type to a distinct server.)

Identical Source Event Rates. Without loss of generality, we assume $w(s) = 1$, $\forall s \in S$. We show the lower bound for the optimal solution and the upper bound for the approximate solution.

The lower bound for the optimal solution is m, since every source is required by at least one server in the system.

The upper bound for the above algorithm is $k + m$. In each round, we either make a query type consume all the spare space of a server, or make a server host all the remaining queries of a query type. In other words, either the number of available servers or the number of available query types decreases by 1. It follows that the number of rounds is at most $k + m$. Since each round increases network traffic rate by at most 1, the total traffic rate cannot be larger than $k + m$.

Using the asserted lower bound and upper bound, we obtain the approximation ratio of $1 + k/m$.

Arbitrary Source Event Rates. Similarly, we demonstrate the lower bound for the optimal solution and the upper bound for the approximate solution.

The lower bound for the optimal solution is $w(S)$, since we have to send the data stream of each source to a server at least once.

The upper bound for the approximate solution is $2w(S)$. Let r_s be the number of rounds to assign queries of query type s, which in total introduces $r_s \cdot w(s)$ of network traffic cost into the system. By the same argument as for the case of identical source event rates, the total number of rounds is at most $m + k$, *i.e.*, $\sum_{s \in S} r_s \leq k + m$. From this, it follows that: $\sum_{s \in S}(r_s - 1) \leq k$. Since $0 < n_s < d_j$, we can guarantee that the top-k query types with respect to the traffic rate will be assigned at most once. Therefore,

$$\sum_{s \in S}(r_s - 1)w(s) \leq kw_{k+1} \leq w(S)$$

where w_{k+1} is the $k+1$-largest traffic rate among all query types. The total event rate satisfies

$$\sum_{s \in S} r_s w(s) \leq w(S) + \sum_{s \in S}(r_s - 1)w(s) \leq 2w(S).$$

Hence, the algorithm provides a 2-approximation. □

Remark. For single-source QA, there exists a constant factor approximation algorithm, and this guarantee holds for any number of sources m, number of queries n, and number of servers k. Moreover, if the source event rates are identical, then the approximation ratio of $1 + k/m$ can be guaranteed. Thus, this approximation ratio guarantee can be arbitrarily near to the optimal one whenever the number of sources relative to the number of servers is sufficiently large. In a practical system with single-source queries and many sources of approximately identical event rates, the proposed algorithm can guarantee nearly optimal performance.

6.5 Online Query Assignment

In this section, we consider online QA, where each query is irrevocably assigned to a server at its arrival time. We focus on the class of online algorithms that decide which server to host an incoming query based on (a) the set of sources required by an incoming query and (b) the queries that were previously assigned to servers and are still in the system.

The key to design such an online algorithm is to choose a metric for assigning queries to servers. Such a metric should consider both load balancing and network traffic cost. We introduce and discuss several greedy metrics for online query assignment in Section 6.5.1. Moreover, in Section 6.5.2, we discuss how to make use of the extra information (Section 6.5.2) or resources (Section 6.5.2) to improve the performance of online algorithms.

6.5.1 Greedy Online Algorithms

In this section, we present three greedy online algorithms, and describe the intuition behind the design of these online algorithms.

The three algorithms use different metrics to decide which server to host an incoming query; however, they share the common pipeline as shown in Figure 6.3. Given an incoming query q, k servers along with the corresponding set of queries a server is hosting, the relative relaxation ratio ν, and a predefined metric M, the server to host q is decided as follows. (a) From all k servers, we find the candidate servers C each of which will not violate the balance constraint if we add q into the server. (b) From C, we find the servers C^* each of which with the lowest cost in terms of M if we add q into the server. (c) If there is only one server in C^*,

```
Input:  (a) an incoming query $q$ requiring a set of sources $S_q$;
        (b) $k$ servers and the queries they are hosting $Q_1, \ldots, Q_k$;
        (c) relative relaxation ratio $\nu$;
        (d) a predefined metric $M$;
Output: the server that will host $q$.

Find candidate servers $C = \{i \mid |Q_i| + 1 < (1+\nu)\frac{\sum_{j=1}^{k} |Q_j|}{k}\}$
Find $C^* \subseteq C$ such that $\forall i \in C^*$,
    $M(Q_i, q) \leq M(Q_j, q), \forall j \in C$
if $|C^*| == 1$
    return the only server in $C^*$
else
    return the server $p = argmin_{i \in C^*}\{|Q_i|\}$
```

Figure 6.3: Greedy algorithms for online QA

we assign q to the server; otherwise, we select the least loaded server from C^* and then assign q to the server.

In this work, we propose three metrics to support online assignment decision: (a) *least incremental cost first* (referred to as LeastCost), (b) *least source cost per server first* (referred to as LeastSource), and (c) *least number of query types first* (referred to as LeastQT). Given a query q and a set of queries Q_j hosted by server j, the three metrics behave as follows.

$\mathsf{LeastCost}$. If $f(Q_j \cup \{q\}) - f(Q_j)$ results in a smaller incremental traffic cost, server j will host q with a lower cost defined as

$$M_{\mathsf{LeastCost}}(Q_j, q) = f(Q_j \cup \{q\}) - f(Q_j).$$

$\mathsf{LeastCost}$ is a natural metric for QA: since the ultimate goal of QA is to minimize traffic cost in a system, $\mathsf{LeastCost}$ attempts to achieve this goal by locally minimizing traffic cost at each query arrival.

$\mathsf{LeastSource}$. If $f(Q_j \cup \{q\})$ results in a smaller traffic cost, the cost of placing q into server j is lower defined as

$$M_{\mathsf{LeastSource}}(Q_j, q) = f(Q_j \cup \{q\}).$$

$\mathsf{LeastCost}$ has a potential issue: one server might subscribe to many sources because of locally optimal decisions such that many incoming queries are assigned to the server, the server gets full quickly, and eventually the server becomes unavailable for hosting incoming queries. If this effect propagates among servers, the overall traffic cost in the system can be very high. To mitigate this effect, we come up with $\mathsf{LeastSource}$ that aims to balance the traffic cost among servers such that no server will subscribe too many sources and result in too high traffic.

$\mathsf{LeastQT}$. Let $\mathcal{T}(Q_j)$ be the set of query types such that $\forall t \in \mathcal{T}(Q_j)$, there exists at least one query of type t hosted by server j. If $|\mathcal{T}(Q_j \cup \{q\})|$ is smaller, placing q into server j results in lower cost defined as

$$M_{\mathsf{LeastQT}}(Q_j, q) = |\mathcal{T}(Q_j \cup \{q\})|.$$

$\mathsf{LeastCost}$ and $\mathsf{LeastSource}$ have a common issue: a few servers might subscribe too many popular sources. If one server subscribes too many popular sources, it is able to host queries of various query types, and will be crowded quickly. If this effect propagates in the system, we have to make many servers subscribe those popular sources. One way to mitigate this effect is to limit the number of query types in a server such that no server can host too many query types and get crowded soon.

6.5.2 Discussion

In this section, we discuss how we develop online algorithms when we have more knowledge or more resources. The above metrics provide heuristic algorithms to solve online QA. Indeed, given the limited knowledge of only information about the incoming query and assigned queries in a system, we might not have much space to develop sophisticated algorithms. In practice, we might know some statistical information about queries, and may relax the load balancing constraint at specific conditions.

Known Query Type Distribution

When we deal with online QA, we might know the information about query type statistics. In particular, such statistics consist of the rate at which specific query types will arrive in the system, and may well be available in production systems that have been in operation for some time, which allows us to collect and maintain statistics about the query workload.

Concretely, with such statistical knowledge, we can develop an online algorithm as follows. Assume that popularity of query types is known: the probability that an incoming query is of type t follows a multinomial distribution with parameters

$(\lambda_t,\ t \in T)$, where $T \subseteq 2^S$ is the universe of query types. Knowing this distribution allows us to develop an online algorithm that makes reservations for query types in advance, and then assigns queries at their arrival times based on their query types. This allows one to emulate what an offline algorithm would do. Given $(\lambda_t,\ t \in T)$, we reduce an online QA to an offline QA as follows.

1. λ_t (the probability that a type-t query arrives) is reduced to n_t (number of type-t queries);

2. δ_j, the probability that server j receives a query, is reduced to the balance constraint, the number of queries a server could host at most, and in particular, we set $\delta_j = \frac{1}{k}$ in the algorithm;

3. $\pi_{t,j}$, the probability that a type-t query is assigned to server j, is reduced to the number of type-t queries in server j.

Therefore, with the statistical information on query type distribution, we can reuse the offline algorithms discussed in Section 6.4 to solve online QA.

Relaxed Load Balancing Constraints

QA problem is defined as a bi-criteria optimization problem where one of the criteria is balancing the load of servers. Specifically, the problem corresponds to finding a query assignment such that the maximum load is at most $(1 + \nu)$ of the mean load across different servers, for given input parameter $\nu \geq 0$. For an online QA, requiring to obey this condition at each query assignment instance may be too restrictive and result in sub-optimal query assignments with respect to the long-term network traffic cost.

Example 8. *Consider a system of* 10 *servers, and a fixed slackness parameter of* 0.05*. The first* 10 *queries will be distributed to* 10 *different servers, and that may result in* 10 *different copies of the same data stream, if those queries subscribe to the same source. This is because the average load times the slackness parameter is strictly less than* 1 *until the* 10*-th query. In general, during the initialization, the allocation of queries to servers will be grossly sub-optimal.*

To resolve the above problem, we relax the load balancing constraints as follows. (a) We define another balance constraint for a system in its initial phase, and use *absolute slackness parameter* to control the balance constraint. (b) In the initial phase, a system uses a balance constraint decided by absolute slackness parameter. When the system hosts more than n queries, we switch back to the balance constraint decided by relative slackness parameter. Let n be the number of queries in the system. The system is said to be α-absolutely balanced, if the number of queries in any server is no more than $\frac{n}{k}+\alpha$, where $\alpha \geq 0$ is the absolute slackness parameter. The system is said to be $(1+\nu)$-relatively balanced, if the number of queries in any server is no more than $(1+\nu) \cdot n/k$, where $\nu \geq 0$ is the relative slackness parameter.

In an online system, when the number of input queries is small, we apply absolute slackness parameter to balance the workload of servers; when the number of queries becomes sufficiently large, we switch to relative slackness parameter. In other words, given the values of parameters α and ν and the number of input queries n, the load balancing constraint for each server j is defined to be $d_j(n) \leq d(n)$, where $d_j(n)$ is the number of queries already assigned to server j and

$$d(n) = \max\left\{\frac{n}{k} + \alpha, (1+\nu) \cdot \frac{n}{k}\right\}. \tag{6.8}$$

The system switches to the relative load balancing as soon as the number of input queries satisfies $n \geq (\alpha/\nu)k$.

Configuring Relaxed Load Balancing

We provide guidelines on how to set the values of parameters α provided ν based on a probabilistic model. Assume that the probability of assigning a query to a server is according to a uniform random distribution across servers. Then, $(d_1(n), d_2(n), \ldots, d_k(n))$ is a random variable with multinomial distribution with parameters n and $(1/k, 1/k, \ldots, 1/k)$ where $\sum_{j=1}^{k} d_j(n) = n$. By the union bound, we have

$$\Pr\left(\cup_{j=1}^{k}\{d_j(n) > d(n)\}\right) \leq \sum_{j=1}^{k} \Pr(d_j(n) > d(n)). \tag{6.9}$$

Using Hoeffding's inequality, we obtain

$$\Pr(d_j(n) > d(n)) \leq \exp\left(-\frac{2(d(n)+1-\frac{n}{k})^2}{n}\right). \tag{6.10}$$

Combining with (6.8), we get: $\Pr(d_j(n) > d(n)) \leq \exp\left(-\frac{2\nu^2}{k^2}n\right)$. Therefore, $\Pr\left(\cup_{j=1}^{k}\{d_j(n) > d(n)\}\right) \leq ke^{-\frac{2\nu^2}{k^2}n}$. From this, it follows that $d_j(n) \leq d(n)$ for every $j = 1, \ldots, k$ with high probability provided that the following condition holds $k = O\left(\nu\sqrt{\frac{n}{\log n}}\right)$. Note that for given $\epsilon > 0$, $d_j(n) \leq d(n)$ for $j = 1, \ldots, k$ to hold with probability at least $1 - \epsilon$, it suffices that

$$n \geq \frac{k^2}{2\nu^2}\log\frac{1}{\epsilon}. \tag{6.11}$$

Suppose that given $\nu > 0$, we want the algorithm to switch to relative load balancing as soon as the probability of violation of the relative imbalance is guaranteed to be smaller or equal than given $\epsilon > 0$. By (6.8), the switch from the absolute to relative balancing constraints happens at the smallest integer n such that $n \geq \alpha k/\nu$. Combined with (6.11), we observe that it suffices to switch over when the number of queries n satisfies (6.11) and it suffices that the absolute relaxation parameter α is chosen such that: $\alpha \leq \frac{k}{2\nu}\log\frac{1}{\epsilon}$. An important insight from this is that the absolute relaxation ratio α should not be taken too large, and it should be at most a quantity that scales linearly with the number of servers k.

6.6 Experiment

In this section we present performance evaluation of the offline and online algorithms in Section 6.4 and Section 6.5 by an extensive set of simulations and using data from production system. Overall, our experimental evaluations provide support to the following claims:

1. Optimizing query assignment provides significant reduction of network traffic compared to random query assignment.

2. Specific online query assignment heuristic, namely LeastCost, consistently outperforms other online (and sometimes even offline) heuristics for a wide range of configurations.

3. LeastCost scales with respect to the number of queries, sources, and servers, and it is robust to dynamic arrival and departure of queries and servers.

6.6.1 Synthetic Workloads

We generated subscription of queries to sources according to a random bipartite graph model [85]. The subscriptions of queries to sources are represented by a bipartite graph $G = (S, Q, E)$, where S is the set of sources, Q is the set of queries, and there is an edge $(s,q) \in E$ if and only if query q receives input from source s. G is assumed to be a random bipartite graph with given degree distribution for the vertices that represent sources and the vertices that represent queries. Specifically, we consider (a) the degrees of source vertices according to Zipf distribution with power-law exponent $\beta > 0$, and (b) the degree of query vertices fixed to parameter $d > 0$. The popularity of sources typically follow a power-law distribution [85, 86], which is modeled by a Zipf distribution. In our experiments, we consider queries of unit processing costs.

Offline Algorithms. We evaluate performance of incremental cost based approach IC, minimum query packing based approach MQP, and min-max traffic cost per server MMS, which are defined in Section 6.4. As a baseline for comparison, we consider the following random offline assignment heuristic OffRand: (a) randomly select all queries of the same query type; (b) randomly select a server with available space to host queries; (c) assign queries to the selected server until either the server is saturated or all queries (of query type) have been assigned; (d) remove server or query type from further consideration; and (e) repeat from step (a) until all queries are assigned.

Online Algorithms. We evaluate the following online heuristics: (a) Least incremental traffic first LeastCost, (b) Least number of sources first LeastSource, and (c) Least query types first LeastQT. As a baseline for comparison, we consider the following random online query assignment OnRand: given an input query, we find a set of candidate servers that can accept the new query without violating the load balancing constraints, and then randomly select one of servers from this for assignment.

Parameters. In our experiments, we consider four parameters: (a) the power-law exponent β for the source degree distribution in the range 1.0 to 3.0 with a default value 2.0, (b) the number of sources per query in the range from 1 to 10 with a default value 2, (c) the number of servers in the range from 10 to 1000 with a default value 100; and (d) the number of queries in the range from $10,000$ to $1,000,000$ with a default value of $100,000$.

When considering query or server dynamics, we also have the following two parameters: (a) mean query life-time (Section 6.6.1), and (b) server departure rate (Section 6.6.1).

For all the offline algorithms considered, we fix the relative relaxation parameter ν to value 0.05. For all the online algorithms considered, we fix the relative relaxation parameter ν to value 0.05, and the absolute relaxation parameter α to value 10 (Section 6.5.2).

Performance metrics. We consider *replication factor* as the metric to evaluate the performance of algorithms. Let C be the resulting traffic cost of an algorithm, S be the set of sources with traffic cost w, and $f(S) = \sum_{s \in S} w(s)$. The replication factor of the algorithm is defined as $C/f(S)$. Intuitively, the replication factor represents normalized traffic cost under the given algorithm.

We run each configuration 10 times, and average the results.

Offline Algorithms

We examine the replication factor of the offline algorithms in Figure 6.4. These results are for sources with unit publishing rates; we examine heterogeneous source traffic rates in Section 6.6.1.

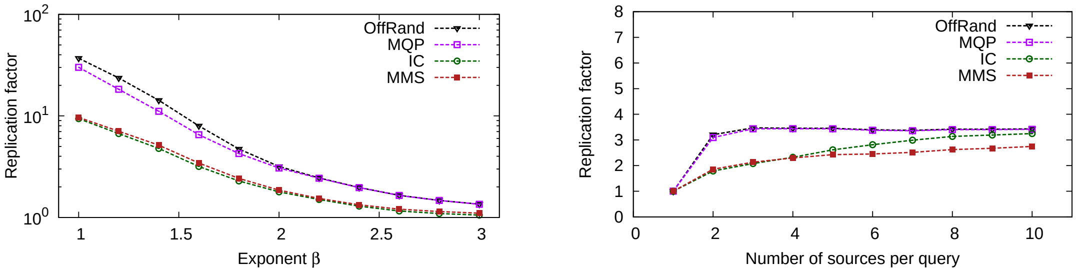


Figure 6.4: Performance of offline query assignment.

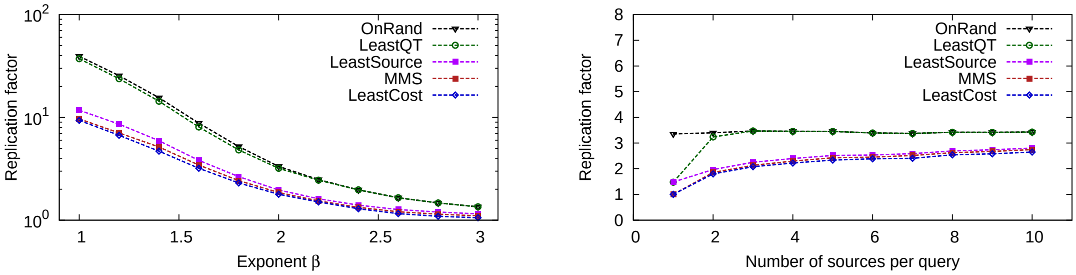


Figure 6.5: Performance of online query assignment without query departures.

In the top graph in Figure 6.4, we show the network traffic replication factor versus the power-law exponent of the Zipf distribution of the number of queries subscribed to a source (source popularity). The number of sources per query is fixed to 2, the number of queries is set to $100,000$, and the number of servers is fixed to 100. We observe that the larger the power-law exponent, the better the performance. In other words, the heavier the power-law distribution of the number of queries subscribed to a source, the larger the replication factor. This is intuitive as a heavier tail implies the existence of a few sources with many query subscribed to those sources, which would intuitively make the query assignment problem harder. The query assignment heuristics IC and MMS consistently exhibit the best performance, up to four times better than OffRand.

In the bottom graph in Figure 6.4, we show the network traffic replication factor versus the number of sources per query. In particular, the power-law exponent is fixed to value 2.0, the number of queries is set to value $100,000$, and the number of servers is set to value 100. We observe that the replication factor increases with the number of sources per query. For single-source queries, the replication factor is smaller than or equal to 2, which provides experimental confirmation of the theoretical guarantee in Theorem 10. The replication factor exhibits a diminishing returns increase with the number of sources per query. MMS exhibits the best performance, and this is matched by IC for sufficiently small number of sources per query.

We also examined the network traffic replication factor versus the number of servers, which is not presented for space reasons. The results suggest that the replication factor increases with the number of servers logarithmically. We also observed that the network traffic replication factor is largely invariant to the number of queries, which we also omit to show due to space reasons.

In summary, we observed that MMS consistently outperforms other offline algorithms, and results in a performance gain of up to factor 4 compared with random offline query assignment OffRand.

Online Algorithms

In this section, we examine the performance of online algorithms in the following settings: (a) online arrival of queries without query departure and a fixed number of servers, (b) online arrival of queries with query departure and a fixed number of servers, and (c) dynamic arrival and departure for both queries and servers.

Query Arrivals, No Departures In Figure 6.5, show the network traffic replication factor for online algorithms, in the same settings as in Figure 6.5. The two sets of graphs are overall qualitatively very similar, hence, we only discuss differences.

We observe that LeastCost exhibits the best performance, and sometimes even outperforms the best offline algorithm MMS. The performance of LeastSource is close to LeastCost. LeastCost performs up to four times better than OnRand, and the performance gain is close to what we observed between MMS and OffRand in the offline case. The performance of LeastQT is virtually identical to that of OnRand. Typically, LeastQT provides no benefits.

Query Arrivals and Departures A streaming query service hosts queries posted by users, and many such queries would be hosted only for a limited time, *e.g.,* the user may be interested in travel updates only while on the road. Hence, it is important to examine query assignment strategies in a system with query arrivals and departures. We consider query dynamics according to the following model: (a) at each time step, the number of query arrivals is a random variable with Poisson distribution; and (b) each arriving query has a lifetime, drawn independently from a given distribution. In particular, we consider two parametric families of distributions: (a) exponential distribution that models the cases of light-tailed query lifetimes, and (b) Pareto distribution that models the case of heavy-tailed query lifetimes. We found similar results for these two different families of distributions, so we only present the results for the exponential distribution.

Figure 6.6 shows the performance of online algorithms under dynamic query arrival and departure. The mean query lifetime ranges from 10 to $1,000,000$ time

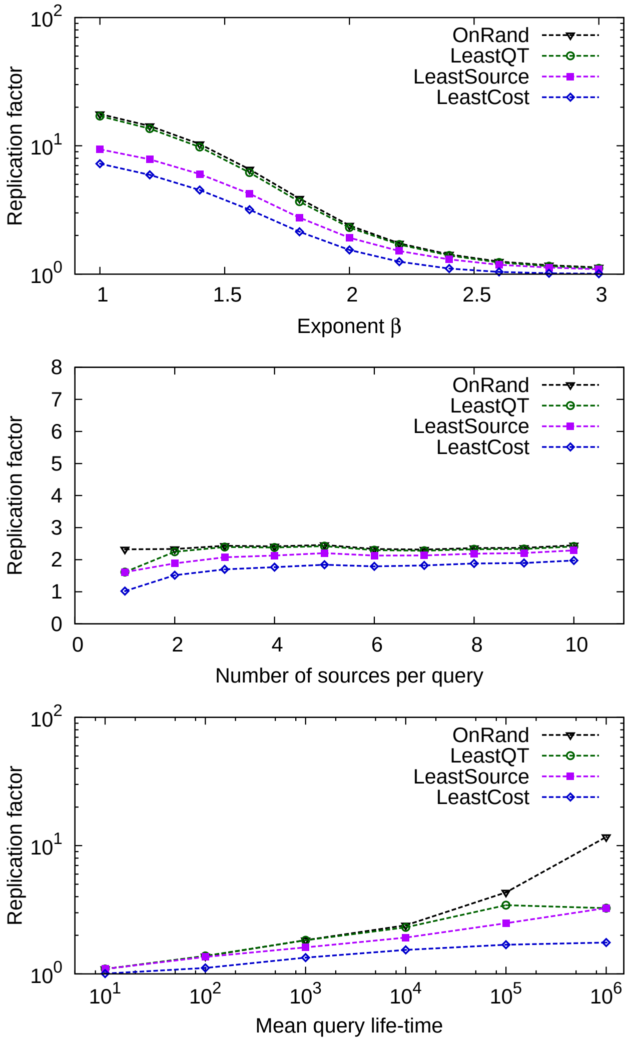


Figure 6.6: Performance of online algorithms with dynamic query arrivals and departures.

steps with the default value 100, 000, the average number of query arrivals per time step is 1, and the total number of discrete time steps τ is set to be 1, 000, 000 in all cases. Overall, we observe that LeastCost consistently yields the best performance. By Little's law, the mean number of queries in the system is the product of the query arrival rate and the mean query lifetime, thus, the given range covers the mean number of queries in the system from 10 to $1,000,000$ queries. Given that the number of servers is 100, we cover the system operating points of 0.1 to $10,000$ queries per server, which covers the range of lightly to highly loaded servers. The result indicate that the network traffic replication factor tends to increase with the load of the servers for all online algorithms. However, this increase is rather slow for LeastCost, which is sub-linear in the mean query lifetime.

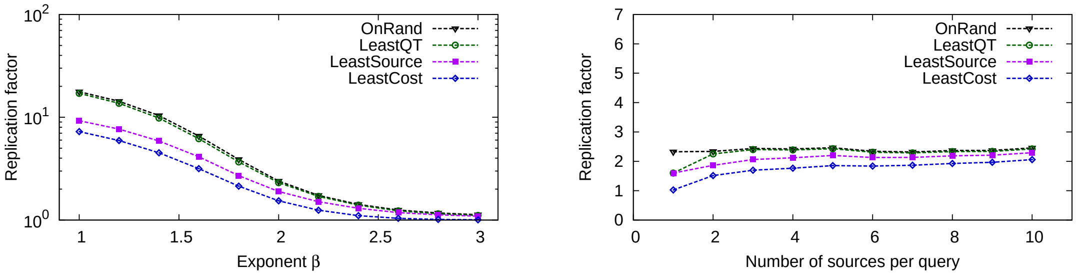


Figure 6.7: Performance of online algorithms with dynamic query and server arrivals/departures.

Server Arrivals and Departures In practice, we also expect some level of server churn; servers may fail, and hence queries need to be re-assigned, and new servers may be added to cope with increased demand, or after recycling failed servers. It is thus important to examine the robustness of different query assignment strategies with respect to arrivals and departures of servers.

We modeled server dynamics similarly to query dynamics: we start with k servers, and queries arrive in τ time steps. At each time step, the number of query arrivals is a random variable with Poisson distribution, and each query is associated with a lifetime that is a random variable with exponential distribution. Starting at time step 1, after every γ time steps (where γ is referred to as *server* *departure rate*), we make a Bernoulli trial: with probability 0.5, we add a new server; otherwise, we randomly delete a server, and re-assign its queries using the online algorithm (*i.e.,* assuming that they are new queries).

The results in Figure 6.7 demonstrate the performance of online query assignment under both query and server dynamics. By default, we consider 100 servers, the power-law exponent β for source popularity distribution of value 2.0, the number of sources per query of value 2, the total number of time steps is $1,000,000$, the mean number of query arrivals per time step of value 1, the mean query lifetime of value $100,000$, and the server departure rate γ is $10,000$. Consistent with the results presented earlier, we observe that LeastCost exhibits the best performance, and is robust with respect to the dynamics of the server arrival and departure.

Heterogeneous Source Traffic Rates Thus far, we have examined the performance of query assignment algorithms for the case of sources with identical traffic rates. We now examine the case of heterogeneous source traffic rates. Since many phenomena in nature follow a power-law distribution [86], we assume that the source traffic rates follow a power-law distribution. We consider the range of values of the power-law parameter that span the case of a fast decaying tail (exponent value of 3) and a slow decaying tail (exponent value of 1). To define the source traffic rates, we also need to decide the assignment of source traffic rates to sources, and how this assignment correlates with other factors such as the popularity of sources (measured by the number of query subscriptions to a source). To cover different possible scenarios, we consider the following three cases: (a) *random*: source traffic rates are assigned to sources independently of their popularity, (b) *positively correlated*: the traffic rate of a source is proportional to its

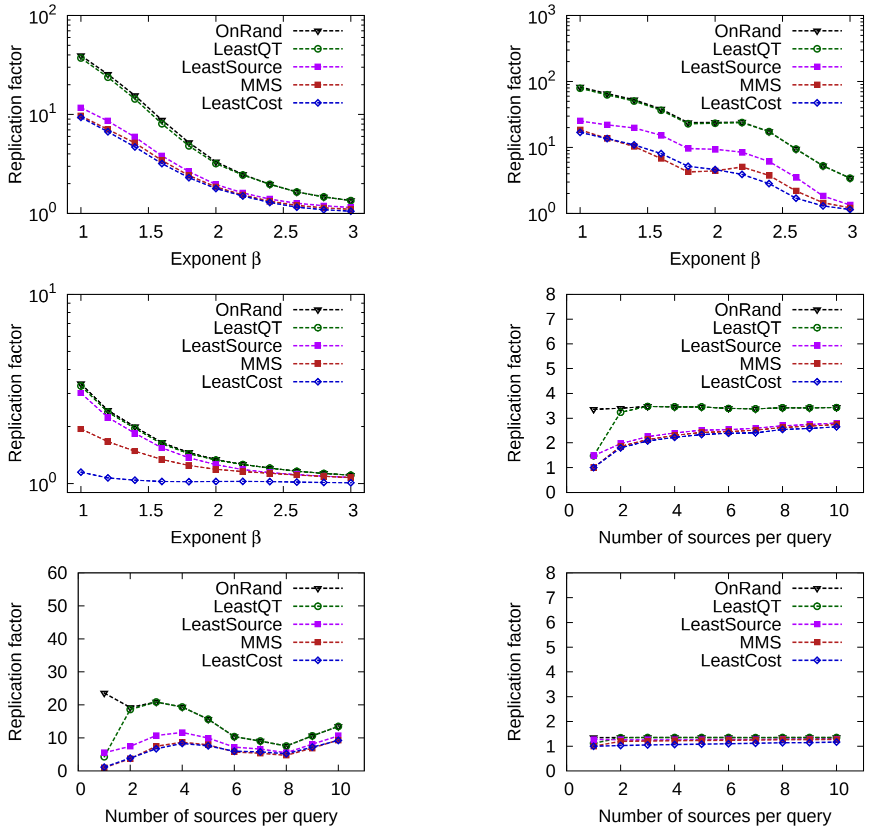


Figure 6.8: Performance of query assignment algorithms on heterogeneous source traffic rates: (left) random, (middle) positively correlated, and (right) negatively correlated.

popularity, and (c) *negatively correlated*: the traffic rate of a source is inversely proportional to its popularity.

The results are presented in Figure 6.8. In particular, we discuss the best offline algorithm MMS, all three online algorithms, and the online random algorithm OnRand, and note the following observations: (a) The more positive the correlation between the source traffic rates and the popularity of sources is, the larger the network traffic replication. (b) Typically, the best performing query assignment strategy is LeastCost. In the case when the source traffic rates are negatively correlated with popularity of sources, LeastCost is substantially better than MMS.

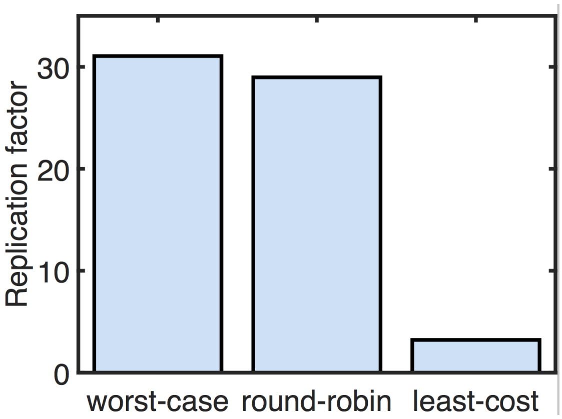


Figure 6.9: Performance of three different online query assignment strategies for a real-world workload.

6.6.2 Real-World Workloads

We compare the performance of LeastCost query assignment strategy using traces from a production environment with two alternative query assignment strategies: the "worst-case" that amounts to supplying each stream of a source to every server, and round-robin that assigns each query according to the round-robin policy (whose performance is expected to be essentially that of random query assignment, which we studied in Section 6.3.2). The trace contains information about query arrivals over a week long interval collected in April 2014 from a production deployment of a stream processing query platform with 100's of servers. The results in Figure 6.9 show that round-robin query assignment strategy performs nearly as badly as the worst-case, and that a significant reduction of the network traffic cost can be achieved by LeastCost query assignment strategy. Specifically, the network traffic cost under LeastCost query assignment is observed to be approximately only 11% of that under the round-robin query assignment.

6.7 Related Work

The query assignment problem studied in this paper aims at clustering similar queries together so as to minimize the network traffic and balance the load of servers. A variety of formulations of co-clustering of similar vertices of a graph was studied in previous work under various assumptions, e.g. see [71] for a survey. In particular, it was studied in the context of publish/subscribe systems, e.g. [147], [204], and [75]. However, to the best of knowledge, none of the previous work addressed the query assignment problem as formulated in this paper.

The problem of assigning tasks to machines to balance the load is a well-known problem, see [28] for a survey of online algorithms. Specifically, it is known that online greedy assignment provides a $2 - 1/k$ approximation. The problem of assigning *balls into bins* was also studied by various authors, *e.g.,* see [29, 36, 148] and the references therein. A standard objective here is to minimize the maximum load (aka minimize congestion), *e.g.,* [30], and packing under knapsack constraints, *e.g.,* [61, 161]. The uniform random assignment load balancing strategy is known to have the maximum load of $n/k + O(\sqrt{(n \log k)/k})$ with probability $o(1)$, for $n \gg k \log^3 k$ [148]. Other load balancing strategies have also been studied, e.g. power of two choices, where each ball is assigned to the least loaded out of two bins picked uniformly at random for each assignment of a ball: the maximum load is known to be $n/k + O(\log \log k)$, with high probability, for $n \gg k$ [36]. A related work is online bin packing where the bounds on the competitive ratio with respect to the offline solution were derived for input arrival order according to random permutation or independent and identically distributed sequence, *e.g.,* see [61] and the references therein. Our main difference is that we consider a bi-criteria optimization, where the server load balancing is only one of the two criteria.

The query assignment problem studied in this paper is an instance of a non-standard balanced graph partitioning problem. Standard balanced graph partitioning problem is defined as follows: given a graph with n vertices, a positive integer k and a parameter $\nu \geq 0$, the goal is to partition the set of vertices into k partitions such that each contains at most $(1+\nu)n/k$ vertices and the number of edges cut is minimized. The best known approximation ratio for this problem is $O(\sqrt{\log n \log k})$ [112]. The query assignment problem has the same form of constraints. However, the objective function is a different submodular function. The unconstrained problem of minimizing a submodular function in the context of graph partitioning was considered, *e.g.,*, by [46], who derived a 2-approximation algorithm. A notable difference with our work is that the query assignment problem minimizes a specific type of a submodular objective function subject to cardinality constraints. The problem of minimizing a submodular function subject to cardinality constraints was studied by [174]: they established a $\Theta(\sqrt{n/\log n})$ approximation ratio for this problem. The approximation algorithms in this paper provide much better approximation ratios whenever the maximum number of sources to which a query is subscribed is sufficiently small, e.g. much smaller than $O(\sqrt{n})$.

6.8 Summary

In this chapter, we propose and study the assignment of streaming queries to servers. This is important in the design of platforms that execute small streaming queries as a service. For such scenarios, where many streams need to be delivered to servers and the density of queries to servers is high, we need to minimize network traffic while balancing load among servers. We model this problem as a partition problem on temporal graphs, demonstrate this problem is NP complete, and derive approximation guarantees. We study, analytically and with simulations, offline and online heuristics for this multi-objective problem. In particular, we propose a heuristic that performs well under a wide range of scenarios, including query and server churn.

Chapter 7

Conclusion

In this section, we first summarize our works on mining and managing large-scale temporal graphs, then share the lessons and wisdom learned from our research, and finally discuss the directions we will pursue in future.

7.1 Summary

Temporal graphs are everywhere in our daily life. Unlike traditional graphs, temporal graphs are dynamic. While the dynamic properties have inspired new applications that rely on mining and managing temporal graphs, the dynamics also raise new challenges: (1) It is difficult to extract and retrieve knowledge from complex temporal graphs; and (2) existing algorithms on static graphs cannot scale with the extra temporal dimension. In this dissertation, we focus on multiple problems on mining and managing large-scale temporal graphs, and investigate the principles to tackle these challenges.

On mining temporal graphs, we have studied critical mining alert mining and information cascade inference. By leveraging the unique properties in temporal graphs, we develop effective and scalable algorithms to solve the problems.

- In critical alert mining, we aim to identify critical alerts that have high probability to trigger a large number of other alerts in system management applications. We first build temporal graphs over alerts to represent their dependencies. Because of temporal dependencies among alerts, the resulting temporal graphs are directed acyclic. Based on this fact, we develop fast approximation algorithm that can obtain near-optimal solutions as well as efficient sampling algorithms that empirically work well.
- In information cascade inference, our goal is to recover the structures of cascades given their partial observations. We propose consistent trees as the model to infer complete cascade structures, and the use temporal and structural constraints in partial observations to prune the search space. Our algorithms obtain improved inference accuracy and are able to scale with large graphs with millions of nodes and billions of edges.

On managing temporal graphs, our works include dynamic subgraph matching, temporal reachability, and stream query assignment. In these works, we identify relatively stable components, and build data management techniques on the components so that we can handle both query processing and dynamic updates in temporal graphs.

- In dynamic subgraph matching, we formulate the service placement problem in datacenter applications as subgraph matching queries on temporal graphs. In this problem, node/edge attributes evolve over time, but topologies of temporal graphs are relatively stable. Inspired by this observation, we develop a flexible graph index that enables fast query processing as well as efficient index updates. Compared with existing algorithms, our technique obtains 10 times speed up in terms of query processing and index updates.

- A large number of temporal reachability queries are generated from information routing tasks in mobile networks. In this case, topologies of temporal graphs change over time, and the topology evolution usually follows periodic patterns in many applications. Driven by this property, we develop a graph index that is able to process temporal reachability queries in a batch instead of processing them one by one. In terms of response time, our index brings up to 100 times of improvement compared with a naive approach.

- In stream processing system, the subscription relationships between data sources and queries form a temporal graph where queries can dynamically join and leave. In this work, our goal is to place queries into multiple servers so that workload is balanced and the resulting network traffic is minimized. Although topologies of temporal graphs in this problem can change over time, degree/popularity distribution on data sources is usually stable. Based on this observation, we develop a probabilistic model that randomly assign queries with performance guarantee.

7.2 Lessons

We have learned valuable lessons from our research work, and we summarize them as follows.

Mining temporal graphs. Mining graphs with temporal dimension may not necessarily increase computation complexity; on the contrary, we may reduce computation complexity if temporal information is wisely leveraged.

In the problem of critical alert mining, we use temporal information to infer dependencies between alerts, and the temporal dependencies determine the resulting alert graphs are directed acyclic. It is the unique topology in alert graphs that leads us to the fast approximation algorithms that are able to find near-optimal solutions and the highly efficiently sampling algorithms.

In information cascade inference, temporal and structural constraints together effectively limits the number of possible information flows among users, resulting in fewer ways to connect cascade pieces. The algorithm inspired by this intuition significantly prunes search space and enables fast inference over large graphs with million of nodes and billions of edges.

Managing temporal graphs. In real applications, temporal graphs may include relatively stable components. We can make use of the stable components to build backbone data structures that enable fast query processing; on top of the backbone data structure, we can further build flexible indexes that efficiently process dynamics in temporal graphs.

In dynamic subgraph matching, we model dynamic cloud as temporal graphs, and use subgraph matching query to guide service placement. In this case, the structures of cloud datacenters are relatively stable. Base on this property, we build a graph index that enables fast query processing. Moreover, we build grid indexes upon the graph index that efficiently process node/edge attribute updates as vector updates in multi-dimensional space.

In temporal reachability, we deal with millions of reachability queries generated from one single information routing task. If we process the reachability queries one by one, the whole routing process will be very slow. In our study, we find the movement of entities follow periodic patterns so that we can compress the periodically repeated graphs and build a graph index that can process reachability queries in a batch instead of one by one.

In stream query assignment, we develop partitioning algorithms to optimize query placement. In this case, data source popularity distributions are relatively stable. Based on the stable popularity distributions, we build a probabilistic model that randomly assign queries into servers with proved approximation guarantee.

7.3 Future Work

In this dissertation, we have introduced our effort towards mining and managing large-scale temporal graphs. Our next step is to uncover the true value of temporal graphs in critical real-life applications and explore novel solutions to multiple fundamental problems on temporal graph mining and management.

7.3.1 Data Analytics for Enterprise System Security

Enterprise computer system is one of the most important assets in companies, governments, and military organizations. Computers in the system collect data from outside sources, store business intelligence, and share with members inside of the organizations. As they host valuable and confidential information, the interest of their owners or even public safety heavily depends on secure computer systems.

Current enterprise computer systems usually employ security tools for protection, such as firewall [57], intrusion detection systems [126], and anti-virus software [143], but recent hacking events in renowned organizations (*e.g.,* Apple [1] and SONY [13]) suggest existing tools could not fully protect our systems and the loss is unaffordable. Indeed, attackers exploit various vulnerabilities in security tools to break into our systems [8, 16]. For example, faked source information will make firewall fail, manipulated network traffic and encrypted packets are hard to be detected by intrusion detection techniques, and zero-day malware is a big headache for anti-virus software.

One thing we may have to admit is advanced attacks always have a chance to break into our system. Now the question is how to fight against these break-in attacks. Currently, what we can do is really limited, because enterprise systems are so complex and they are almost like black boxes to us. If we have the visibility to check what is going on in these systems, we may have the hope to control the damage from break-in attacks.

How to obtain such visibility? We rely on system monitoring techniques. Among all possibilities, system call log [109] is a cost-effective way to get a comprehensive view of systems, as any software needs to make system calls to perform their tasks and the overhead to collect system calls is quite small. Each system call simply records what kind of interaction happened between system entities (*e.g.,* processes, files, and so on) at which time, so system call logs are essentially temporal graphs indicating how system entities interact over time. However, system call logs are not perfect: they cannot directly tell us control flow or causality between system entities, which are critical for cybersecurity applications. Therefore, mining and management systems on temporal graphs are desired to extract such knowledge for cybersecurity applications.

In our vision, the system that can automatically discovers cybersecurity intelligence includes three layers: syscall collection, data storage and management, and data analytics. We summarize the expected function of each layer by a bottom-up manner.

Syscall collection. Syscall collection is at the bottom layer of the whole system. The key challenge on this layer is to develop tools that crawl a complete set of system calls with minimized overhead.

Data storage and management. The middle layer is to store and manage syscall data (*i.e.,* temporal graphs). When monitoring data are collected, we need to store the temporal graph data into uniformly accessible database systems. The key question is how to build such a system that clean, fuse, and manage dynamically arriving temporal graph data and also provide fast data access for various upper-layer data analytics.

Data analytics. In the upper layer, we need to develop data analytics to detect break-in attacks. In general, there are four tasks.

- The first task is to employ temporal graph queries to detect break-in attacks. The question is how to formulate useful queries. Our idea is to extract temporal graph patterns as signatures for break-in attacks, and use the patterns as skeletons to formulate graph queries against syscall logs. The query results suggest the existence of known attacks that have broken into our systems.

- The second task is to perform reasoning and inference over temporal graphs. Once something malicious is found, we have to reason the source of the attack, identify the vulnerabilities in our systems, and infer other compromised system components.
- The third one targets on unknown attacks. Even if an attack is new, our system will behave abnormally when it is under attack. By anomaly detection, we grab the chance to discover unknown attacks.
- The fourth task is about attack modeling. Our system will fight against sophisticated attacks. To evaluate the effectiveness of our approaches, we need to understand and model the attacks so that we can generate synthetic attacks and test the robustness of our techniques.

7.3.2 Data Decay in the Age of Internet of Things

In the age of Internet of Things, we can collect data from a wider range of areas [18], including medical healthcare, city infrastructures, home devices, and many others. It is not far away from us. Samsung claimed that their products will be all connected by 2020 [5], and IDC estimated we will have more than 30 billion connected devices by 2020 [6], which is five years later.

When the time comes, we have to face new challenges caused by data deluge. Every day, we will collect a huge amount of data (*e.g.,* IDC estimates we can collect 10^8 TB data per day by 2020 [6]), and quickly, the collected data will become history. Now the question is how to deal with these historical data?

There are two extreme strategies. One extreme is to keep all the data, which is impractical because we have limited resources. The other is to drop all the historical data resulting in the risk of losing business opportunities.

To this end, something in the middle may be more reasonable: for some data, we keep them intact for future study; for some data, we drop them completely; and for the rest, we might build a sketch to summarize the data for future usage. Now the question is how to make right decisions in different scenarios? Our angle is to develop data driven methods to measure and model how data value decays in applications. Then, we can adopt suitable strategies for data at a specific decay phase.

Bibliography

[1] 2014 celebrity photo hack. https://en.wikipedia.org/wiki/2014_celebrity_photo_hack.

[2] Amazon elastic compute cloud (ec2). http://aws.amazon.com/ec2/.

[3] Amazon kinesis. http://aws.amazon.com/kinesis/.

[4] Aws lambda. http://aws.amazon.com/lambda/.

[5] Ces 2015: The internet of just about everything.

[6] Facts and forecasts: Billions of things, trillions of dollars. http://www.siemens.com/innovation/en/home/pictures-of-the-future/digitalization-and-software/internet-of-things-facts-and-forecasts.html.

[7] Geolife gps trajectories. http://research.microsoft.com/.

[8] Intrusion detection system. http://en.wikipedia.org/wiki/Intrusion_detection_system.

[9] Microsoft developer network. system.linq.expressions namespace. `https://msdn.microsoft.com/en-us/library/system.linq.expressions(v=vs.110).aspx`.

[10] Microsoft. scalable information stream processing by bing in support of cortana scenarios. `http://channel9.msdn.com/posts/Scalable-Information-Stream-Processing-by-Bing-in-Support-of-Cortana-Scenarios`.

[11] Rackspace: the open cloud company. `www.rackspace.co.uk`.

[12] S4: distributed stream computing platform. `http://incubator.apache.org/s4/`.

[13] Sony pictures entertainment hack. `https://en.wikipedia.org/wiki/Sony_Pictures_Entertainment_hack`.

[14] Storm: Distributed and fault-tolerant realtime computation. `http://storm-project.net/`.

[15] Windows azure: Microsofts cloud platform. `http://www.windowsazure.com/en-us/`.

[16] Zero-day virus. `http://en.wikipedia.org/wiki/Zero-day_virus`.

[17] C. Aggarwal and P. Yu. Online analysis of community evolution in data streams. In *SDM*, 2005.

[18] C. C. Aggarwal, N. Ashish, and A. P. Sheth. The internet of things: A survey from the data-centric perspective. *Managing and Mining Sensor Data*, 2012.

[19] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *VLDB*, 2013.

[20] L. Akoglu, M. Mcglohon, and C. Faloutsos. Anomaly detection in large graphs. In *In CMU-CS-09-173 Technical Report*, 2009.

[21] I. F. Akyildiz, O. B. Akan, C. Chen, J. Fang, and W. Su. Interplanetary internet: state-of-the-art and research challenges. *Comput. Netw.*, 2003.

[22] S. O. Al-Mamory and H. Zhang. Intrusion detection alarms reduction using root cause analysis and clustering. *Computer Communications*, 32(2):419–430, 2009.

[23] E. Altman, G. Neglia, F. De Pellegrini, and D. Miorandi. Decentralized stochastic control of delay tolerant networks. In *INFOCOM*, 2009.

[24] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, 2013.

[25] A. Angel, N. Koudas, N. Sarkas, and D. Srivastava. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *VLDB*, 2012.

[26] A. Arnold, Y. Liu, and N. Abe. Temporal causal modeling with graphical granger methods. In *SIGKDD*, 2007.

[27] D. Arthur, R. Motwani, A. Sharma, and Y. Xu. Pricing strategies for viral marketing on social networks. *Internet and Network Economics*, 2009.

[28] Y. Azar. On-line load balancing. In *Online Algorithms*. 1998.

[29] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM journal on computing*, 1999.

[30] Y. Azar and L. Epstein. On-line load balancing of temporary tasks on identical machines. *SIAM Journal on Discrete Mathematics*, 2004.

[31] N. Bailey. *The mathematical theory of infectious disease and its applications*. 1975.

[32] A. Bakker, M. S. Albert, G. Krauss, C. L. Speck, and M. Gallagher. Response of the medial temporal lobe network in amnestic mild cognitive impairment to therapeutic intervention assessed by fmri and memory task performance. *NeuroImage: Clinical*, 2015.

[33] J. Bang-Jensen and G. Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. 2008.

[34] N. Bansal, K. Lee, V. Nagarajan, and M. Zafer. Minimum congestion mapping in a cloud. In *PODC*, 2011.

[35] T. Benson, A. Akella, A. Shaikh, and S. Sahu. Cloudnaas: a cloud networking platform for enterprise applications. In *SoCC*, 2011.

[36] P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: The heavily loaded case. *SIAM Journal on Computing*, 2006.

[37] M. Berkelaar et. al. Mixed integer linear programming (milp) solver. http://sourceforge.net/projects/lpsolve.

[38] S. Bikhchandani, D. Hirshleifer, and I. Welch. A theory of fads, fashion, custom, and cultural change as informational cascades. *Journal of political Economy*, 1992.

[39] P. Bogdanov, M. Mongiovi, and A. K. Singh. Mining heavy subgraphs in time-evolving networks. In *ICDM*, 2011.

[40] V. Borrel, M. H. Ammar, and E. W. Zegura. Understanding the wireless and mobile network space: a routing-centered classification. In *CHANTS*, 2007.

[41] C. Budak, D. Agrawal, and A. El Abbadi. Limiting the spread of misinformation in social networks. In *WWW*, 2011.

[42] C. Budak, D. Agrawal, and A. El Abbadi. Limiting the spread of misinformation in social networks. In *WWW*, 2011.

[43] H. Cam and P. Mouallem. Mission-aware time-dependent cyber asset criticality and resilience. CSIIRW, 2013.

[44] M. Cha, F. Benevenuto, Y.-Y. Ahn, and P. K. Gummadi. Delayed information cascades in flickr: Measurement, analysis, and modeling. *Computer Networks*, 2012.

[45] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with seti. In *CIDR*, 2003.

[46] C. Chekuri and A. Ene. Approximation algorithms for submodular multiway partition. In *FOCS*, 2011.

[47] L. Chen and C. Wang. Continuous subgraph pattern search over certain and uncertain graph streams. *TKDE*, 2010.

[48] S. Chen, C. Jensen, and D. Lin. A benchmark for evaluating moving object indexes. In *VLDB*, 2008.

[49] W. Chen, C. Wang, and Y. Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *SIGKDD*, 2010.

[50] W. Chen, C. Wang, and Y. Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *KDD*, 2010.

[51] W. Chen, Y. Wang, and S. Yang. Efficient influence maximization in social networks. In *SIGKDD*, 2009.

[52] W. Chen, Y. Yuan, and L. Zhang. Scalable influence maximization in social networks under the linear threshold model. In *ICDM*, 2010.

[53] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, 2008.

[54] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, 2007.

[55] J. Cheng, J. Yu, B. Ding, P. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, 2008.

[56] S. Cheng, H. Shen, J. Huang, G. Zhang, and X. Cheng. Staticgreedy: solving the scalability-accuracy dilemma in influence maximization. In *CIKM*, 2013.

[57] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet security: repelling the wily hacker*. 2003.

[58] L. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *TPAMI*, 2004.

[59] O. Dain and R. K. Cunningham. Fusing a heterogeneous alert stream into scenarios. In *ACM workshop on Data Mining for Security Applications*, volume 13, 2001.

[60] K. Dave, R. Bhatt, and V. Varma. Modelling action cascades in social networks. In *AAAI*, 2011.

[61] N. R. Devanur, K. Jain, B. Sivan, and C. A. Wilkens. Near optimal online algorithms and fast approximation algorithms for resource allocation problems. In *EC*, 2011.

[62] R. Diestel. *Graph theory*. 2006.

[63] B. Ding, J. Yu, and L. Qin. Finding time-dependent shortest paths over large graphs. In *EDBT*, 2008.

[64] P. Domingos and M. Richardson. Mining the network value of customers. In *KDD*, 2001.

[65] N. Du, L. Song, M. Gomez-Rodriguez, and H. Zha. Scalable influence estimation in continuous-time diffusion networks. In *NIPS*, 2013.

[66] M. Eslami, H. R. Rabiee, and M. Salehi. Dne: A method for extracting cascaded diffusion networks from social networks. In *ICSC*, 2011.

[67] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. In *SIGMOD*, 2011.

[68] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. In *VLDB*, 2010.

[69] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, 2012.

[70] H. Fei, R. Jiang, Y. Yang, B. Luo, and J. Huan. Content based social behavior prediction: a multi-task learning approach. In *CIKM*, 2011.

[71] S. Fortunato. Community detection in graphs. *Physics Reports*, 2010.

[72] W. Gao, Q. Li, B. Zhao, and G. Cao. Multicasting in delay tolerant networks: a social network perspective. In *MobiHoc*, 2009.

[73] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* 1990.

[74] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM*, 2011.

[75] S. Girdzijauskas, G. Chockler, Y. Vigfusson, Y. Tock, and R. Melamed. Magnet: practical subscription clustering for internet-scale publish/subscribe. In *DEBS*, 2010.

[76] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs. In *ICPR*, 2002.

[77] I. Giurgiu, C. Castillo, A. Tantawi, and M. Steinder. Enabling placement of virtual infrastructures in the cloud. In *Middleware*, 2012.

[78] J. Goldenberg, B. Libai, and E. Muller. Talk of the network: A complex systems look at the underlying process of word-of-mouth. *Marketing Letters*, 2001.

[79] J. Goldenberg, B. Libai, and E. Muller. Talk of the Network: A Complex Systems Look at the Underlying Process of Word-of-Mouth. *Marketing Letters*, 2001.

[80] M. Gomez-Rodriguez, J. Leskovec, and A. Krause. Inferring networks of diffusion and influence. *TKDD*, 2012.

[81] A. Goyal, F. Bonchi, and L. V. Lakshmanan. A data-based approach to social influence maximization. *VLDB*, 2011.

[82] A. Goyal, W. Lu, and L. V. Lakshmanan. Celf++: optimizing the greedy algorithm for influence maximization in social networks. In *WWW*, 2011.

[83] M. Granovetter. Threshold models of collective behavior. *American Journal of Sociology*, 1978.

[84] R. Groenevelt, P. Nain, and G. Koole. Message delay in manet. *SIGMETRICS Perform. Eval. Rev.*, 2005.

[85] J.-L. Guillaume and M. Latapy. Bipartite graphs as models of complex networks. *Physica A: Statistical Mechanics and its Applications*, 2006.

[86] D. Gunawardena, T. Karagiannis, A. Proutiere, and M. Vojnovic. Characterizing podcast services: publishing, usage, and dissemination. In *IMC*, 2009.

[87] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *SIGCOMM Review*, 2009.

[88] A. Guttman. *R-trees: a dynamic index structure for spatial searching*. 1984.

[89] H. He and A. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, 2006.

[90] X. He, G. Song, W. Chen, and Q. Jiang. Influence blocking maximization in social networks under the competitive linear threshold model. In *SDM*, 2012.

[91] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *SIGMOD*, 2014.

[92] L. Invernizzi, S. Miskovic, R. Torres, S. Saha, S. Lee, M. Mellia, C. Kruegel, and G. Vigna. Nazca: Detecting malware distribution in large-scale networks. In *NDSS*, 2014.

[93] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 1990.

[94] S. Jain, K. Fall, and R. Patra. Routing in a delay tolerant network. 2004.

[95] H. Jiang, H. Wang, P. Yu, and S. Zhou. Gstring: A novel approach for efficient search in graph databases. In *ICDE*, 2007.

[96] R. Jin et. al. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, 2008.

[97] J. R. Johnson, E. Hogan, et al. A graph analytic metric for mitigating advanced persistent threat. In *ICISI*, 2013.

[98] K. Julisch. Clustering intrusion detection alarms to support root cause analysis. *TISSEC*, 2003.

[99] K. Julisch and M. Dacier. Mining intrusion detection alarms for actionable knowledge. In *SIGKDD*, 2002.

[100] B. Karrer and M. E. J. Newman. Random graph models for directed acyclic networks. *Phys. Rev. E*, 2009.

[101] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *SIGKDD*, 2003.

[102] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. In *KDD*, 2003.

[103] D. Kempe, J. Kleinberg, and E. Tardos. Influential nodes in a diffusion model for social networks. In *ICALP*, 2005.

[104] W. O. Kermack and A. G. Mckendrick. A contribution to the mathematical theory of epidemics. *Proc R Soc Lond A*, 1927.

[105] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood based fast graph search in large networks. In *SIGMOD*, 2011.

[106] S. Kim and E. N. Brown. A general statistical framework for assessing granger causality. In *ICASSP*, 2010.

[107] M. Kimura and K. Saito. Tractable models for information diffusion in social networks. *PKDD*, 2006.

[108] M. Kimura, K. Saito, R. Nakano, and H. Motoda. Finding influential nodes in a social network from information diffusion data. *Social Computing and Behavioral Modeling*, 2009.

[109] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.

[110] G. Kossinets. Effects of missing data in social networks. *Social networks*, 2006.

[111] G. Kossinets, J. Kleinberg, and D. Watts. The structure of information pathways in a social communication network. In *KDD*, 2008.

[112] R. Krauthgamer, J. S. Naor, and R. Schwartz. Partitioning graphs into balanced components. In *SODA*, 2009.

[113] T. Lappas, E. Terzi, D. Gunopulos, and H. Mannila. Finding effectors in social networks. In *SIGKDD*, 2010.

[114] T. Lappas, E. Terzi, D. Gunopulos, and H. Mannila. Finding effectors in social networks. In *KDD*, 2010.

[115] N. Leavitt. Complex-event processing poised for growth. *Computer*, 2009.

[116] U. Lee, S. Y. Oh, K.-W. Lee, and M. Gerla. Relaycast: Scalable multicast routing in delay tolerant networks. In *ICNP*, 2008.

[117] J. Leguay, T. Friedman, and V. Conan. Dtn routing in a mobility pattern space. In *WDTN*, 2005.

[118] J. Leskovec, L. Backstrom, and J. Kleinberg. Meme-tracking and the dynamics of the news cycle. In *KDD*, 2009.

[119] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *TKDD*, 2007.

[120] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance. Cost-effective outbreak detection in networks. In *SIGKDD*, 2007.

[121] J. Leskovec, A. Singh, and J. Kleinberg. Patterns of influence in a recommendation network. *Advances in Knowledge Discovery and Data Mining*, 2006.

[122] Q. Li and D. Rus. Sending messages to mobile users in disconnected ad-hoc wireless networks. In *MobiCom*, 2000.

[123] C. Liu and J. Wu. An optimal probabilistic forwarding protocol in delay tolerant networks. In *MobiHoc*, 2009.

[124] A. A. Mahimkar, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and Q. Zhao. Towards automated performance diagnosis in a large iptv network. In *ACM SIGCOMM Computer Communication Review*, 2009.

[125] M. Mathioudakis, F. Bonchi, C. Castillo, A. Gionis, and A. Ukkonen. Sparsification of influence networks. In *KDD*, 2011.

[126] M. Md and M. Hassan. Current studies on intrusion detection system, genetic and fuzzy logic. *International Journal of Distributed and Parallel Systems*, 2013.

[127] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM*, 2010.

[128] S. Merugu, M. Ammar, and E. Zegura. Routing in space and time in networks with predictable mobility. Technical report, 2004.

[129] D. S. Mitrinović, J. E. Pečarić, and A. Fink. Bernoulli‘s inequality. In *Classical and New Inequalities in Analysis*, 1993.

[130] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. In *SIGMOD*, 2012.

[131] M. Mongiovi, P. Bogdanov, and A. K. Singh. Mining evolving network processes. In *ICDM*, 2013.

[132] M. Mongiovì, R. Di Natale, R. Giugno, A. Pulvirenti, A. Ferro, and R. Sharan. Sigma: a set-cover-based inexact graph matching algorithm. *BMC*, 2010.

[133] M. Mongiovì, A. Singh, X. Yan, B. Zong, and K. Psounis. Efficient multicasting for delay tolerant networks using graph indexing. In *INFOCOM*, 2012.

[134] J. Moore, J. Chase, K. Farkas, and P. Ranganathan. Data center workload monitoring, analysis, and emulation. In *Eighth Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2005.

[135] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, 2005.

[136] K. P. Murphy. *Dynamic bayesian networks: representation, inference and learning*. PhD thesis, University of California, 2002.

[137] S. A. Myers and J. Leskovec. Clash of the contagions: Cooperation and competition in information diffusion. In *ICDM*, 2012.

[138] M. Nagori, S. Mutkule, and P. Sonarkar. Detection of brain tumor by mining fmri images. *International Journal of advanced research in Computer and communication Engineering*, 2013.

[139] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions-I. *Mathematical Programming*, 14(1):265–294, 1978.

[140] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW*, 2010.

[141] H. Nguyen and R. Zheng. Influence spread in large-scale social networks–a belief propagation approach. In *PKDD*, 2012.

[142] U. H. Nielsen, J.-P. Pellet, and A. Elisseeff. Explanation trees for causal bayesian networks. In *UAI*, pages 427–434, 2008.

[143] J. Oberheide, E. Cooke, and F. Jahanian. Cloudav: N-version antivirus in the network cloud. In *USENIX Security Symposium*, 2008.

[144] M. Piorkowski, N. Sarafijanovic-Djukic, and M. Grossglauser. CRAWDAD data set epfl/mobility (v. 2009-02-24). http://crawdad.cs.dartmouth.edu/epfl/mobility, 2009.

[145] B. K. Polat, P. Sachdeva, M. H. Ammar, and E. W. Zegura. Message ferries as generalized dominating sets in intermittently connected mobile networks. In *MobiOpp*, 2010.

[146] H. Qiu, Y. Liu, N. A. Subrahmanya, and W. Li. Granger causality for time-series anomaly detection. In *ICDM*, 2012.

[147] L. Querzoni. Interest clustering techniques for efficient event routing in large-scale settings. In *DEBS*, 2008.

[148] M. Raab and A. Steger. balls into binsa simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science*. 1998.

[149] R. Raghavendra, J. Lobo, and K. Lee. Dynamic graph query primitives for sdn-based cloudnetwork management. In *HotSDN*, 2012.

[150] J. W. Raymond, E. J. Gardiner, and P. Willett. Rascal: Calculation of graph similarity using maximum common edge subgraphs. *The Computer Journal*, 2002.

[151] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[152] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historial evolving graph sequences. *VLDB*, 2011.

[153] M. Richardson and P. Domingos. Mining knowledge-sharing sites for viral marketing. In *KDD*, 2002.

[154] G. Robins and A. Zelikovsky. Tighter bounds for graph steiner tree approximation. *SIAM J. Discrete Math.*, 2005.

[155] M. G. Rodriguez and B. Schölkopf. Influence maximization in continuous time diffusion networks. In *ICML*, 2012.

[156] E. Sadikov, M. Medina, J. Leskovec, and H. Garcia-Molina. Correcting for missing data in information cascades. In *WSDM*, 2011.

[157] K. Saito, R. Nakano, and M. Kimura. Prediction of information diffusion probabilities for independent cascade model. In *KES*, 2008.

[158] A. K. Seth. A matlab toolbox for granger causal connectivity analysis. *Journal of neuroscience methods*, 2010.

[159] H. Shang, Y. Zhang, X. Lin, and J. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. In *VLDB*, 2008.

[160] E. Shlizerman, J. A. Riffell, and J. N. Kutz. Data-driven inference of network connectivity for modeling the dynamics of neural codes in the insect antennal lobe. *Frontiers in computational neuroscience*, 2014.

[161] D. B. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 1993.

[162] D. Šidlauskas, K. Ross, C. Jensen, and S. Šaltenis. Thread-level parallel indexing of update intensive moving-object workloads. *Advances in Spatial and Temporal Databases*, 2011.

[163] D. Šidlauskas, S. Šaltenis, C. Christiansen, J. Johansen, and D. Šaulys. Trees or grids? indexing moving objects in main memory. In *SIGSPATIAL GIS*, 2009.

[164] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. *Data Mining and Knowledge Discovery*, 2000.

[165] M. Song, H. Choo, and W. Kim. Spatial indexing for massively update intensive applications. *Information Sciences*, 2012.

[166] M. Song and H. Kitagawa. Managing frequent updates in r-trees for update-intensive applications. *TKDE*, 2009.

[167] X. Song, Y. Chi, K. Hino, and B. L. Tseng. Information flow modeling based on diffusion rate for prediction and ranking. In *WWW*, 2007.

[168] T. Spyropoulos, K. Psounis, and C. S. Raghavendra. Efficient routing in intermittently connected mobile networks: the multiple-copy case. *TON*, 2008.

[169] T. Spyropoulos, K. Psounis, and C. S. Raghavendra. Efficient routing in intermittently connected mobile networks: the single-copy case. *TON*, 2008.

[170] K. Sricharan and K. Das. Localizing anomalous changes in time-evolving graphs. In *SIGMOD*, 2014.

[171] C. J. Stam. Modern network science of neurological disorders. *Nature Reviews Neuroscience*, 2014.

[172] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu. Graphscope: parameter-free mining of large time-evolving graphs. In *KDD*, 2007.

[173] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. In *VLDB*, 2012.

[174] Z. Svitkina and L. Fleischer. Submodular approximation: Sampling-based algorithms and lower bounds. *SIAM Journal on Computing*, 2011.

[175] J. Tian and J. Pearl. Probabilities of causation: Bounds and identification. *Annals of Mathematics and Artificial Intelligence*, 2000.

[176] Y. Tian and J. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, 2008.

[177] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, 2007.

[178] J. Ullmann. An algorithm for subgraph isomorphism. *JACM*, 1976.

[179] A. W. Van der Vaart. *Asymptotic statistics*. 2000.

[180] V. Vazirani. *Approximation algorithms*. 2004.

[181] S. A. Vinterboa. A note on the hardness of the k-ambiguity problem. 2002.

[182] C. Wang and L. Chen. Continuous subgraph pattern search over graph streams. In *ICDE*, 2009.

[183] F. Wang, H. Wang, and K. Xu. Diffusive Logistic Model Towards Predicting Information Diffusion in Online Social Networks. *ArXiv e-prints*, 2011.

[184] T. Wang, M. Srivatsa, D. Agrawal, and L. Liu. Learning, indexing, and diagnosing network faults. In *KDD*, 2009.

[185] X. Wang, A. Smalter, J. Huan, and G. Lushington. G-hash: towards fast kernel-based similarity search in large graph databases. In *EDBT*, 2009.

[186] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In *EuroSys*, 2009.

[187] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *VLDB*, 2014.

[188] T. Wüchner, M. Ochoa, and A. Pretschner. Malware detection with quantitative data flow graphs. In *ASIACCS*, 2014.

[189] Y. Xie and P. Yu. CP-index: on the efficient indexing of large graphs. In *CIKM*, 2011.

[190] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, 2002.

[191] X. Yan, P. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, 2004.

[192] X. Yan, P. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD*, 2005.

[193] J. Yang and J. Leskovec. Patterns of temporal variation in online media. In *WSDM*, 2011.

[194] J. Yang, S. Zhang, and W. Jin. Delta: indexing and querying multi-labeled graphs. In *CIKM*, 2011.

[195] D. M. Yellin. Algorithms for subset testing and finding maximal sets. In *SODA*, 1992.

[196] X. Yu, K. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, 2005.

[197] Y. Yuan, G. Wang, H. Wang, and L. Chen. Efficient subgraph search over large uncertain graphs. In *VLDB*, 2011.

[198] S. Zhang, J. Yang, and W. Jin. Sapper: subgraph indexing and approximate matching in large graphs. In *VLDB*, 2010.

[199] S. Zhang, J. Yang, and W. Jin. Sapper: subgraph indexing and approximate matching in large graphs. *VLDB*, 2010.

[200] X. Zhang, J. Kurose, B. N. Levine, D. Towsley, and H. Zhang. Study of a bus-based disruption-tolerant network: mobility modeling and impact on routing. In *MobiCom*, 2007.

[201] P. Zhao and J. Han. On graph query optimization in large networks. In *VLDB*, 2010.

[202] W. Zhao, M. Ammar, and E. Zegura. A message ferrying approach for data delivery in sparse mobile ad hoc networks. In *MobiHoc*, 2004.

[203] W. Zhao, M. Ammar, and E. Zegura. Multicasting in delay tolerant networks: semantic models and routing algorithms. In *WDTN*, 2005.

[204] Y. Zhao, K. Kim, and N. Venkatasubramanian. Dynatops: A dynamic topic-based publish/subscribe architecture. In *DEBS*, 2013.

[205] C. Zhou, P. Zhang, J. Guo, X. Zhu, and L. Guo. Ublf: An upper bound based approach to discover influential nodes in social networks. In *ICDM*, 2013.

[206] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: a total order approach. In *SIGMOD*, 2014.

[207] B. Zong, R. Raghavendra, M. Srivatsa, X. Yan, A. K. Singh, and K.-W. Lee. Cloud service placement via subgraph matching. In *ICDE*, 2014.

[208] B. Zong, Y. Wu, A. K. Singh, and X. Yan. Inferring the underlying structure of information cascades. In *ICDM*, 2012.

[209] B. Zong, Y. Wu, J. Song, A. K. Singh, H. Cam, J. Han, and X. Yan. Towards scalable critical alert mining. In *KDD*, 2014.

[210] L. Zou, L. Chen, and M. Özsu. Distance-join: Pattern match query in a large graph database. In *VLDB*, 2009.