Efficient techniques for subgraph mining and query processing

Lin, Wenqing

2015

Lin, W. (2015). Efficient techniques for subgraph mining and query processing. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/62137>

<https://doi.org/10.32657/10356/62137>

NANYANG
TECHNOLOGICAL
UNIVERSITY

**EFFICIENT TECHNIQUES FOR SUBGRAPH
MINING AND QUERY PROCESSING**

LIN WENQING

SCHOOL OF COMPUTER ENGINEERING

2015

Efficient Techniques for Subgraph Mining and Query Processing

Lin Wenqing

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Doctor of Philosophy

2015

Abstract

Graph data has been so prevalent that efficiently obtaining useful information from them is highly demanded. Given massive amounts of graph data, people are often interested in a small portion, namely their subgraphs, by the processes of mining and querying. Due to the enormous number of subgraphs in the massive graph data, these processes are highly costly. In this thesis, we study three important problems on subgraph mining and query processing, i.e., frequent subgraph mining, network motif discovery, and generalized subgraph query processing. These problems find numerous applications in real world, whereas they are extremely challenging.

First, mining frequent subgraphs from a large collection of graph objects is an important problem in several application domains such as bio-informatics, social networks, computer vision, etc. The main challenge in subgraph mining is efficiency, as (i) testing for graph isomorphisms is computationally intensive, and (ii) the cardinality of the graph collection to be mined may be very large. We propose a two-step *filter-and-refinement* approach that is suitable to massive parallelization within the scalable MapReduce computing model. We partition the collection of graphs among worker nodes, and each worker applies the filter step to determine a set of candidate subgraphs that are *locally frequent* in its partition. The union of all such graphs is the input to the refinement step, where each candidate is checked against all partitions and only the *globally frequent* graphs are retained. We devise a statistical threshold mechanism that allows us to predict which subgraphs have a high chance to become globally frequent, and thus reduce the computational overhead in the refinement step. We also propose effective strategies to avoid redundant computation in each round when searching for candidate graphs, as well as a lightweight graph compression mechanism to reduce the communication cost between machines. Extensive experimental evaluation results on several real-world large graph datasets show that the proposed approach clearly outperforms the existing state-of-the-art and provides a practical solution to the problem of frequent subgraph mining for massive collections of graphs.

Second, the identification of network motifs has essential applications in numerous domains, such as pattern detection in biological networks and graph analysis in digital circuits. However, mining network motifs is computationally challenging, as it requires to enumerate subgraphs from a real-life graph, and compute the frequency of each subgraph in a large number of random graphs. In particular, existing solutions often require days

to derive network motifs from biological networks with only a few thousand vertices. To address this problem, this thesis presents a novel study on network motif discovery using Graphical Processing Units (GPUs). The basic idea is to employ GPUs to parallelize a large number of subgraph matching tasks in computing subgraph frequencies from random graphs, so as to reduce the overall computation time of network motif discovery. We explore the design space of GPU-based subgraph matching algorithms, with careful analysis of several crucial factors that affect the performance of GPU programs. Based on our analysis, we develop a GPU-based solution that (i) drastically differs from existing CPU-based methods, and (ii) exploits the strengths of GPUs in terms of parallelism while mitigating their limitations in terms of the computation power per GPU core. With extensive experiments on a variety of biological networks, we show that our solution is up to two orders of magnitude faster than the best CPU-based approach, and is around 20 times more cost-effective than the latter, when taking into account the monetary costs of the CPU and GPUs used.

Finally, we study a new type of graph queries, which injectively maps its edges to paths of the graphs in a given database, where the length of each path is constrained by a given threshold specified by the weight of the corresponding matching edge. We give important applications of the new graph query and identify new challenges of processing such a query. In particular, the new type of graph queries further explodes the already exponential search space compared with conventional graph queries. Besides, existing pruning techniques cannot be directly applied or they are simply not adequate, since our new type of query graphs is different from their indexed features. To address these issues, we devise the cost model of the branch-and-bound algorithm framework for processing the graph query, and propose an efficient algorithm to minimize the cost overhead. We also develop three indexing techniques to efficiently answer the queries online, each of which can be applied into different scenarios. Finally, we verify the efficiency of our proposed indexes with extensive experiments on large real and synthetic datasets.

Acknowledgments

First, I want to express my most sincere appreciation to my supervisor, Prof. Xiaokui Xiao, for his patience in guiding me towards the wonderland of research, his continuous support in my PhD study, and his enthusiasm about and profound knowledge in research that always inspire me a lot. Without his great guidance, this thesis would not have been finished.

I also want to thank all my collaborators: Prof. James Cheng, Prof. Sourav S. Bhowmick, Prof. Yufei Tao, Prof. Gabriel Ghinita, Prof. Yabo Xu, Dr. Xiaoli Li, Dr. Xing Xie, Dr. Lingkun Wu, Andy Diwen Zhu, and Sibor Wang. It has been a great pleasure of me to work with and learn from them. Especially, I thank Prof. James Cheng for his insightful advice and countless hours of discussion in the junior years of my PhD study.

Besides, I thank the members of the Data-Intensive Scalable Computing (DISCO) Lab and my friends at Singapore for spending joyful time together.

Finally, I thank my families for their consistent support and encouragement in my study.

Contents

Abstract	i
Acknowledgments	iii
List of Figures	viii
List of Tables	x
1 Introduction	1
1.1 Problems and Motivations	2
1.2 Approaches and Contributions	7
1.3 Thesis Organization	9
2 Literature Survey	11
2.1 Frequent Subgraph Mining	11
2.2 Network Motif Discovery	15
2.3 Subgraph Query Processing	18
2.4 Summary	21
3 Large-Scale Frequent Subgraph Mining	23
3.1 Preliminaries	23
3.1.1 Mining Frequent Subgraphs	24
3.1.2 MapReduce	25
3.2 Solution Overview	26
3.3 Filter Round	27
3.3.1 A Preliminary Approach	27
3.3.2 An Improved Approach	28
3.4 Refinement Round	32

3.4.1	The Top-Down Approach	33
3.4.2	The Bottom-Up Approach	33
3.5	Reducing Communication Overhead	35
3.5.1	Compression of Canonical Labeling	35
3.5.2	Theoretical Analysis	37
3.6	Experimental Evaluation	38
3.6.1	Experimental Setup	38
3.6.2	Tuning Probability Upper-Bound ρ	39
3.6.3	Comparison with competitor techniques	41
3.7	Summary	44
4	Network Motif Discovery	45
4.1	Preliminaries	46
4.1.1	Problem Definition	46
4.1.2	Graphics Processing Units	48
4.2	Difficulties in GPU Translations	49
4.3	Solution Overview	50
4.4	GPU-Based Subgraph Matching	51
4.4.1	Representation of Graphs	52
4.4.2	Construction of C_i	52
4.4.3	Avoiding Duplicates	57
4.4.4	Correctness	59
4.5	Optimizations	59
4.5.1	Handling Large Candidate Sets	59
4.5.2	Handling Multiple Graphs	61
4.5.3	Matching Tree	62
4.6	Experimental Evaluation	63
4.6.1	Experimental Settings	63
4.6.2	Comparisons with CPU-Based Techniques	65
4.6.3	Effects of Optimizations	67
4.7	Summary	68

5	Generalized Subgraph Query Processing	69
5.1	Problem Statement	69
5.2	Generalized Subgraph Matching Algorithm	70
5.2.1	Existing Algorithms for Subgraph Isomorphism	71
5.2.2	A Cost-based Approach for Generalized Subgraph Isomorphism	72
5.3	Indexing Techniques	74
5.3.1	Distance Index	75
5.3.2	Frequent Pattern Index	78
5.3.3	Star Index	79
5.4	Experimental Evaluation	83
5.4.1	Experimental Settings	83
5.4.2	Generalized Subgraph Isomorphism	84
5.4.3	Tuning the FP-index	85
5.4.4	Performance of Indices	86
5.4.5	Performance on Synthetic Dataset	88
5.5	Summary	89
6	Conclusions and Future Work	90
6.1	Conclusions	90
6.2	Future Work	91
6.2.1	Frequent subgraph mining in single-large-graph scenario	91
6.2.2	Dense subgraph discovery on GPU	92
6.2.3	Subgraph query processing in single-large-graph scenario	92
	Appendix - List of Publications	92
	References	94

List of Figures

1.1	A drug database, $\mathcal{G} = \{g_1, g_2, g_3\}$, and two query graphs, q_1 and q_2	5
2.1	Illustration of traversals on graph lattice, in which grey nodes denote visited graphs and white nodes denote unvisited graphs.	12
3.1	Frequent subgraph computation.	24
3.2	Proposed filter-refinement MapReduce workflow for frequent subgraph mining.	26
3.3	Subgraph lattice.	31
3.4	Top-Down vs. Bottom-Up strategy.	32
3.5	Vary probability threshold ρ	40
3.6	Vary support threshold θ	41
3.7	Vary number of workers m	42
3.8	Impact of compression.	43
3.9	Strategy comparison in refinement round.	44
4.1	A real graph G , two subgraphs g_1 and g_2 of G , and two random graphs G_1 and G_2 degree-equivalent to G	46
4.2	The general architecture of a GPU.	49
4.3	Edge and offset arrays of G_1 in Figure 4.1.	52
4.4	A graph g_3 with several automorphism groups.	58
4.5	Illustration of a matching tree.	63
4.6	Computation time on all datasets.	64
4.7	<i>NemoGPU</i> (with Q2000 and K20) vs. <i>DistributedNM</i>	66
4.8	Computation time vs. k	67

4.9	Effects of optimization vs. dataset.	68
5.1	Two star structures: $s_{g_3}(v_1)$ and $s_{g_3}(v_3)$	80
5.2	Efficiency of generalized subgraph isomorphism algorithms on AIDS dataset.	84
5.3	Space and pre-computation costs of the FP-index on PubChem dataset.	85
5.4	Index performance v.s. dataset size on PubChem dataset.	86
5.5	Index performance v.s. parameters of the query graphs on PubChem.40K dataset.	87
5.6	Index performance v.s. graph density on synthetic dataset.	88

List of Tables

1.1	A graph query in ALADDIN language.	6
3.1	Table of datasets.	39
3.2	Experimental parameter values.	39
3.3	Table of running time ($\theta = 0.1$).	44
4.1	Specifications of E5645, Q2000, and K20C.	64
4.2	Table of datasets.	65
5.1	Weight multisets of g_3	80
5.2	Compressed weight multisets of g_3	81

Chapter 1

Introduction

Graph is a powerful data model that can naturally represent various entities and their relationships. Graph data is ubiquitous today and being able to efficiently discover useful information from such data is beneficial to many applications. For example, in chemical database, chemical compounds can be naturally modeled as graphs. Identifying the molecules that frequently appear in the database are helpful to drug design and discovery of the common formations among the chemical compounds. Besides, in biological networks, the interactions between proteins form a graph which has the proteins as vertices and the interactions as edges. Biologists are interested in the subgraphs with statistical significance to predict protein interactions and study functional sub-units. Furthermore, social networks have users as vertices and the relationships between them as edges in the graphs. Finding graph patterns in the social network can help identify different groups of users, and help understand the mechanics of social behaviors and interactions. There are two categories of graph data, i.e., the first category has a single large graph, and the second one consists of a large number of small or medium-sized graphs. In the previous examples, the chemical database belongs to the first category, and the biological network and social network are often modeled a single large graph respectively.

Given massive amounts of graph data, people are often interested in a small portion, namely their subgraphs, by the processes of mining and querying. This thesis studies three important problems on subgraph mining and query processing, i.e., frequent subgraph mining, network motif discovery, and generalized subgraph query processing. All these problems find numerous applications in real world, whereas they are extremely challenging. They often require a large number of subgraph isomorphism tests, which are computationally demanding. Besides, the number of subgraphs in a given graph could be extensively large, which renders those problems even costly. On the other hand, the first two problems all aim at mining subgraphs, but differ with each other in the types of subgraphs to be discovered. In addition, the algorithms of frequent subgraph mining can be used in index construction for efficiently answering subgraph queries in Chapter 5.

In this thesis, the network motif discovery problem takes as input a single large graph, and the other problems focus on a collection of graphs. In the future work, we plan to extend the approaches proposed in this thesis to handle all categories of graph data. In what follows, this chapter will briefly introduce these problems and the main contributions of this thesis. More details will be presented in the subsequent chapters respectively.

1.1 Problems and Motivations

Frequent subgraph mining. We first study the problem of frequent subgraph mining, which has tremendous applications from several areas such as bio-informatics, computational chemistry, social networks, the semantic web and computer vision, which make use of large amounts of data encoded as graphs. For instance, in the bio-informatics domain, graphs can naturally model protein structures. By looking at large sample sets of such graphs and determining common formations among them, researchers are able to understand what the role of a certain protein-protein interaction network is. Frequent subgraph patterns in social networks can help identify relationships within different groups, and help understand the mechanics of social behavior and interactions. The necessity to search for patterns within massive amounts of graph data, coupled with the computationally-intensive nature of testing graph isomorphism relationships (the fundamental operation in graph mining) makes the graph mining problem a very challenging one from a performance standpoint.

There are two broad categories of large-scale frequent subgraph mining scenarios: in the first case, there is one single large graph of massive scale, in the order of terabytes of data, and frequent subgraph patterns must be found in different regions of the graph. In the second case, frequent subgraphs must be found within a large-scale collection of moderate-sized graphs. The former case is relevant to the social network domain, whereas the latter finds many applications in the areas of bio-informatics and computational chemistry. Both scenarios share a number of common challenges, such as large data input size, which may exceed the memory resources of a single machine, and vast amounts of CPU time required to compute frequent patterns. Given these characteristics, cloud computing and the widespread MapReduce framework represent a promising direction to solve these challenging problems.

Several solutions have been proposed for the single-graph scenario in either a sequential [42, 46, 57, 58] or parallel computing (MapReduce, MPI) framework [50, 73, 97]. However, our focus is on the equally-important case of mining a large collection of individual, moderate-sized graphs. This problem is also known in literature as subgraph mining in a *transaction setting* [40]. The objective is to find subgraphs that occur with

*support*¹ higher than a *threshold* θ expressed as a fraction of the collection cardinality, i.e., $0 \leq \theta \leq 1$. The early solutions to this problem were memory-based [41, 43, 56, 64, 70, 100], and assumed that the entire graph collection fits in memory. However, as data size increases, the assumption no longer holds. To address the limited amount of main memory, some disk-based graph database solutions have been proposed [92]. These approaches do solve the memory limitation, but they incur significant overhead for accessing the data, as the number of disk I/Os is very high. The work in [40] is the only one so far to employ MapReduce for mining a large collection of graphs. That solution takes an incremental approach, similar in concept to the *Apriori* algorithm [43] for graph mining. Specifically, in the first step, a fraction of the graph collection is mapped to each worker, which determines the local support (on its data partition) for all possible *single-edge* subgraphs. A subsequent reduction phase determines the global support for each such subgraph, and candidates that do not meet the global support threshold are discarded. The program continues to the next step where all *two-edge* subgraphs are generated from the set of retained candidates, and so on until all frequent subgraphs are found. Although this method uses MapReduce, the large number of resulting steps still results in significant performance problems, as we will prove experimentally in Section 3.6.

Network motif discovery. The second problem is network motif discovery. Given a graph G , a *network motif* in G is a subgraph g of G , such that g appears much more frequently² in G than in *random graphs* whose degree distributions are similar to that of G [68]. The identification of network motifs finds important applications in numerous domains. For example, network motifs are used (i) in system biology to predict protein interactions in biological networks [15] and discover functional sub-units [85], (ii) in electronic engineering to understand the characteristics of circuits [44], and (iii) in brain science to study the functionalities of brain networks [86].

Numerous techniques [32, 51, 54, 61, 71, 74, 76, 93, 95] have been proposed to identify network motifs from sizable graphs. Roughly speaking, all existing techniques adopt a common two-phase framework as follows:

- *Subgraph Enumeration:* Given a graph G and a parameter k , enumerate the subgraphs g of G with k vertices each;
- *Frequency Estimation:* For each subgraph g identified in the subgraph enumeration phase, estimate its expected frequency in a random graph with identical degree distribution to G ; if g 's frequency in G is significantly higher than the expected frequency in a random graph, then return g as a motif.

¹Given a set $\mathcal{G}$ of graphs, the support of g is the fraction of the number of g 's super-graphs in $\mathcal{G}$.

²The frequency of g in G is defined as the number of subgraphs of G that are isomorphic to g .

The above framework, albeit conceptually simple, is difficult to implement efficiently due to the significant computation overhead incurred by the frequency estimation phase. Specifically, to estimate the expected frequency of a subgraph g in a random graph, the standard approach [54] is to generate a sizable number r of random graphs (e.g., $r = 1000$), and then take the average frequency of g in those graphs as an estimation. To compute the frequency of g in a random graph G' , however, we need to derive the number of subgraphs of G' that are isomorphic to g – this requires a large number of *subgraph isomorphism tests* [65], which are known to be computationally expensive. The high costs of subgraph isomorphism tests, coupled with the large number r of random graphs, render the frequency estimation phase a computational challenge. Existing techniques attempt to resolve this issue by improving the efficiency of subgraph isomorphism test, but only achieve limited success. As shown in Section 4.6, even the state-of-the-art solutions require days to derive network motifs from biological networks with only a few thousand vertices.

Motivated by the deficiency of existing work, we present an in-depth study on efficient solutions for network motif discovery. Instead of focusing on the efficiency of individual subgraph isomorphism tests, we propose to utilize Graphics Processing Units (GPUs) to parallelize a large number of isomorphism tests, in order to reduce the computation time of the frequency estimation phase. This idea is intuitive, and yet, it presents a research challenge since there is no existing algorithm for testing subgraph isomorphisms on GPUs. Furthermore, as shown in Section 4.2, existing CPU-based algorithms for subgraph isomorphism tests cannot be translated into efficient solutions on GPUs, since the characteristics of GPUs make them inherently unsuitable for several key procedures used in CPU-based algorithms.

Generalized subgraph query processing. We finally study a new type of graph queries, called generalized subgraph query, which finds a large number of applications. For example, in bio-informatics and chemical informatics, graphs can model compounds and proteins, and graph queries can be used for screening, drug design, motif discovery in protein structures, and protein interaction analysis. In computer vision, graphs represent organization of entities in images and graph queries can be used to identify objects and scenes. In heterogeneous web-based data sources and e-commerce sites, graphs model schemas and graph matching can be applied to solve problems of schema matching and integration. There are also many other applications, such as program flows, software and data engineering, taxonomies, etc., where data is modeled as graphs and it is essential to search and query the graph data.

Existing research focuses on mainly two types of graph datasets, one consisting of a single large graph (e.g., an online social network or the entire citation graph in a certain domain) and the other consisting of a large set of small or medium-sized graphs. We

Figure 1.1: A drug database, $\mathcal{G} = \{g_1, g_2, g_3\}$, and two query graphs, q_1 and q_2 .

focus on the later, which is also very popular in real life (e.g., most of the examples we listed earlier belong to this type).

To query a graph database, $\mathcal{G}$, that consists of many small graphs, there are three types of queries commonly studied in the literature. Let q be a query graph. The first one is *subgraph query* [22, 38, 80, 83, 101, 104, 106], which finds the subset of graphs $\mathcal{A}$ of $\mathcal{G}$ such that q is a *subgraph* of any graph in $\mathcal{A}$. The second one is *supergraph query* [20, 21, 81, 105], which finds the subset of graphs $\mathcal{A}$ of $\mathcal{G}$ such that q is a *supergraph* of any graph in $\mathcal{A}$. The third one is *similarity query* [79, 102, 103, 107], which finds the subset of graphs $\mathcal{A}$ of $\mathcal{G}$ such that q is a *similar graph* of any graph in $\mathcal{A}$ according to a given similarity measure.

The three types of queries are useful in different applications. However, both subgraph queries and supergraph queries are too rigid and therefore similarity queries are proposed as an alternative. Existing similarity queries are mostly measured by the edit distance [103, 107] or maximum common subgraph [79, 102], which is reasonable for some applications but often fails to capture meaningful patterns or targets in applications where critical objects or entities may have to be matched or they may be within a distance from each other that is beyond the specified similarity distance (i.e., the similarity threshold). We show such an application, where both exact queries and similarity queries are not applicable, by the following example.

Example 1.1: Consider a drug design system, which supports the inventive process of finding new medications based on the knowledge of the biological target. Figure 1.1 shows some compounds in the database, i.e., g_1 , g_2 , and g_3 . A compound can be naturally modeled as a graph, where atoms are vertices, the chemical name of the atom is the label of the corresponding vertex, and the chemical bonds between any two atoms are modeled as edges in the graph. Among many drug design methods, the pharmacophore model is the most popular one whose goal is to find the substructures that are closely matched to

Table 1.1: A graph query in ALADDIN language.

POINT N ; POINT H ; POINT O ; POINT C ; DISTANCE (1, 2) 1 3 ; DISTANCE (1, 3) 1 2 ; DISTANCE (1, 4) 1 2 ; DISTANCE (3, 4) 1 1 ;

the objective. ALADDIN [91] is a computer program for the design and recognition of compounds that meet geometric, steric, and sub-structural criteria. ALADDIN also uses a precise geometric description language to define the properties of a designed molecule.

The query shown in Table 1.1 is written in the ALADDIN language, which is to find a graph pattern where

- (i) there are four atoms: N, H, O, and C, whose positions are at 1, 2, 3, and 4, respectively;
- (ii) the distance between N and H is 1 to 3, and similarly 1 to 2 between N and O, 1 to 2 between N and C, and exactly 1 between O and C.

Since the distance between all pairs of atoms can be estimated [48], the distance can be further modeled as the number of bonds that connect the atoms. Thus, the query in Table 1.1 can be converted to a graph query which finds all graphs in the database such that

- (i) there exist four vertices u_1 to u_4 in the graph whose labels are N, H, O, and C, respectively;
- (ii) let $P = \langle u_i, \dots, u_j \rangle$ be a path that connects u_i and u_j and $|P|$ be the length of the path:
there exist paths $P_1 = \langle u_1, \dots, u_2 \rangle$, $P_2 = \langle u_1, \dots, u_3 \rangle$, $P_3 = \langle u_1, \dots, u_4 \rangle$, and $P_4 = \langle u_3, \dots, u_4 \rangle$, respectively, where $1 \leq |P_1| \leq 3$, $1 \leq |P_2| \leq 2$, $1 \leq |P_3| \leq 2$, and $|P_4| = 1$.

Such a query can be naturally represented as the query graph³ q_1 shown in Figure 1.1, and the answer to this query is $\{g_3\}$. In such a query, subgraph query cannot be applied, while similarity query is also not suitable when the matching paths are long. $\square$

In this thesis, we study this new type of graph queries as described in Example 1.1, which will be formally defined in Section 5.1. But intuitively, the new query is a generalization of the subgraph query, which generalizes exact edge matching to path matching constrained by a path length; that is, instead of matching each edge as in a

³We assume that the length of a path cannot be negative.

subgraph query, we find a path with two matching end vertices for each edge in the query graph, where the length of the matching path must be within the specified edge weight. Thus, the new query has a much stronger expressive power than a subgraph query.

Such a query is also useful in many other applications. For example, in querying user online traversal graphs, one may be only interested in whether users have visited certain important sites within a certain number of clicks, while an exact or a quality similar matching may not exist. In searching pictures in an image database, it is often rare to find an exact or even similar matching due to the huge amount of irrelevant information in the background (note that similarity measure by edit distance or maximum common subgraph often counts all such irrelevant information in the matching); in this case, we can specify a few features to be focused in the matching while relaxing the links between the features by some reasonable edge weight.

1.2 Approaches and Contributions

The first two problems are in the domain of subgraph mining. They share the similarity on that they all need to compute the frequency of subgraphs. However, they are different since (i) the former computes the frequent subgraphs but the latter computes the subgraphs that exhibit statistical significance, and (ii) the former studies a large collection of graphs whereas the input of the latter is a large single graph. The last problem is in the domain of subgraph query processing, which can exploit the results of subgraph mining in index construction for efficient query processing. In what follows, we describe our approaches to efficiently address the aforementioned problems.

Frequent subgraph mining. In the first problem, we propose a two-step *filter-and-refinement* approach that uses MapReduce, but considers a completely different computation workflow than the work in [40]. The proposed workflow is more suitable to massive parallelization, and works as follows: first, in the *filter* step, the collection of graphs is partitioned among worker nodes, and each worker determines a set of *local frequent* subgraphs on its local partition. As opposed to [40], there is no restriction on the size of such subgraphs. Next, the union of all local candidates is processed in the *refinement* step, where each candidate is evaluated across *all* partitions, and only the *global frequent* subgraphs are retained. The benefit of the two-step approach is that it reduces the amount of communication among worker nodes, and at the same time it allows for a high degree of parallelism within each step.

Recall that, an important source of computational overhead in graph mining is testing for subgraph isomorphism (an NP-hard problem). To improve efficiency, we devise a statistical model that predicts which subgraphs have a high chance to become globally frequent, and thus reduce the overhead of redundant subgraph isomorphism testing in the

refinement step. Furthermore, we propose effective strategies for reusing computation at each worker in the process of isomorphism testing for subgraphs that share edges.

Communication cost is also an important concern, as large amounts of intermediate data may be generated and transferred among workers. Excessive network transmission increases the overall execution time of graph mining, and may also lead to bottlenecks and failures. To reduce the amount of communication, we devise a lightweight graph compression scheme which reduces the amount of information that needs to be transferred between machines, while at the same time keeping the encoding/decoding computational overhead low.

Extensive experimental evaluation results on several real-world large graph datasets show that the proposed approach clearly outperforms the existing state-of-the-art and provides a practical solution to the problem of frequent subgraph mining for massive collections of graphs.

Network motif discovery. Then, to address the challenges in the problem of network motif discovery, we propose a novel subgraph matching technique tailored for GPUs. Our technique adopts the filter-refinement paradigm, and is developed with careful considerations of three crucial factors that affect the performance of GPU programs, namely, load balancing on GPU cores, branch divergences in GPU codes, and memory access patterns on the GPU (see Section 4.1.2). Based on those considerations, we make design choices that (i) drastically differ from existing CPU-based methods, but (ii) lead to superior efficiency on GPUs. In addition, our technique incorporates several optimization methods that considerably improve scalability and efficiency. We experimentally evaluate our solution against the state-of-the-art CPU-based methods on a variety of biological networks using two machines, each of which has a 500-dollar CPU, a low-end 300-dollar GPU, and a high-end 2700-dollar GPU. We show that, when running with the high-end (resp. low-end) GPU, our solution outperforms the best CPU-based approach by two orders of magnitude (resp. one order of magnitude) in terms of computation efficiency. Furthermore, the *per-dollar* performance of our solution is roughly 20 times higher than that of the best CPU-based method. This not only establishes the superiority of our solution, but also demonstrates that, for network motif discovery, GPU-based methods are much more cost-effective than CPU-based ones.

Generalized subgraph query processing. In the last problem, processing the new query is significantly more challenging. For both subgraph and supergraph query processing, it involves subgraph isomorphism which is NP-hard. The relaxation in the new query from exact edge matching to approximate path matching essentially further explodes the already exponential search space. Existing pruning techniques cannot be directly applied or they are simply not adequate, since our generalized query graph is different from the

indexed features. Therefore, this thesis proposes new effective pruning techniques and efficient data structures to solve this challenging problem.

To address the above challenges, we devise a fast algorithm for generalized subgraph matching, which is a significantly more complicated matching problem than subgraph isomorphism. And then, we develop three indexes for the efficient processing of generalized subgraph queries, namely, a *distance-based* index (S-index), a *frequent-pattern-based* index (FP-index), and a *star-structure-based* index (S-index). We discuss in details the strengths and limitations of the indexes. Finally, we evaluate the performance of our matching algorithm (for candidate verification) and our indexes (for filtering) using both real and synthetic datasets. In particular, the experimental results verify that our method is efficient in query processing (in both filtering and candidate verification). Although some of the indexes have weaknesses, we show how the weaknesses are addressed by another index; in particular, our results show that S-index achieves both a low index construction cost and a short query response time.

1.3 Thesis Organization

The remainder of this thesis is organized as follows.

Chapter 2 conducts a detail survey on the above problems respectively. We first introduce the previous work on frequent subgraph mining in transaction setting, which can be classified into two categories based on the ways that they traverse the search space. After that, we review and summarize the existing approaches for network motif discovery, which typically consist of two phases, i.e., subgraph enumeration and frequency computation. Finally, we present the existing techniques for subgraph query processing in the transaction setting, which often exploit the results of frequent subgraph mining in index construction for fast query processing.

Chapter 3 presents the MapReduce algorithms for large-scale frequent subgraph mining. We first introduce the fundamental concepts and definitions. Then, we give an overview of the proposed algorithm, which consists of two steps: filter and refinement. After that, we provide the details of each step, and present the graph compression scheme to reduce the communication cost. Finally, we show an extensive experimental evaluation, followed by a review of related work and summary.

Chapter 4 illustrates our GPU algorithm for network motif discovery. We first formally define the problem of network motif discovery and introduce GPU architecture. Then we show the difficulties in translating the CPU-based approaches on the GPU. After that, we present an overview of our GPU solution, and then illustrate the details step by step. In addition, we describe several optimization techniques that significantly improve the performance of the GPU algorithm. Then, we evaluate the performance of

our GPU algorithm on a variety of biological datasets. Finally, we discuss the related work and summarize this chapter.

Chapter 5 studies the problem of generalized subgraph query processing. After formally defining the problem, we present the algorithm for generalized subgraph matching. Then, we describe three indexes in details that follow the filter-and-verification framework. Finally, we report the experimental results, and present the related work and the summary of this chapter.

Chapter 6 concludes the thesis by summarizing the problems studied in this thesis, and proposes some future work on subgraph mining and query processing.

Chapter 2

Literature Survey

In this chapter, we broadly survey the techniques developed for solving the problems: frequent subgraph mining, network motif discovery, and subgraph query processing. Finally, we summarize this chapter in Section 2.4

2.1 Frequent Subgraph Mining

As explained in Chapter 1, we focus our discussion in this section on frequent subgraph mining in the transaction setting, i.e., the graph database is a large set of individual, moderate-sized graphs.

Given a graph database $\mathcal{G} = \{G_1, G_2, \dots, G_n\}$ and a graph g , the set of g 's super-graphs in $\mathcal{G}$ is denoted by $\mathcal{A}(g) = \{G \in \mathcal{G} \mid g \subseteq G\}$ where $g \subseteq G$ means that g is a subgraph of G . We define the *frequency* of g in $\mathcal{G}$ as $f(g) = |\mathcal{A}(g)|$, and the support of g in $\mathcal{G}$ as $s(g) = f(g)/n$. In the problem of frequent subgraph mining, given $\mathcal{G}$ and a pre-defined threshold $\theta \in [0, 1]$, we are to find all graphs g such that $s(g) \geq \theta$.

A straightforward approach is to enumerate a set $\mathcal{C}$ of all possible graphs and compute $s(g)$ for each $g \in \mathcal{C}$ by *subgraph isomorphism tests*. All graphs in $\mathcal{C}$ can be organized in a lattice, where each node denotes a graph in $\mathcal{C}$, as shown in Figure 2.1. In the graph lattice, we add a *dummy root* above the single-edge graphs, and each node extends the graph of its parent node(s) by adding a single edge, i.e., a child node is a super-graph of all its parent nodes. However, this straightforward approach is extremely inefficient since (i) the size of $\mathcal{C}$ is exponentially large, and (ii) the computation of $s(g)$ is extremely expensive due to that subgraph isomorphism testing is NP-hard [31]. To avoid this deficiency, several techniques [18, 41, 43, 56, 64, 70, 92, 100] have been proposed in the literature. All these techniques can prune the enumeration of most graphs in $\mathcal{C}$ and reuse the computation to calculate the frequencies of graphs in $\mathcal{C}$. The pruning strategies are based on the *downward closure* property:

$$\text{If } g \subseteq g', \text{ we have } \mathcal{A}(g) \supseteq \mathcal{A}(g'), \text{ i.e., } f(g) \geq f(g').$$

Figure 2.1: Illustration of traversals on graph lattice, in which grey nodes denote visited graphs and white nodes denote unvisited graphs.

That is, the frequency of a graph g is not smaller than that of its super-graph g' . Therefore, we can compute the frequencies of *some* graphs in $\mathcal{C}$, based on which we are able to prune the rest graphs accordingly. To explain, given that $g \in \mathcal{C}$ is not a frequent subgraph of $\mathcal{G}$, i.e., $s(g) < \theta$, we can avoid the computation on all super-graphs g' of g since $s(g') \leq s(g) < \theta$.

As such, the strategy can be applied during the traversal on the graph lattice for pruning the search space. There are two ways for traversing the graph lattice: breadth-first (BFS) traversal [43, 56] and depth-first (DFS) traversal [18, 41, 64, 70, 100], as shown in Figure 2.1.

BFS-based approaches. In the BFS-based approaches, the graphs at the same level in the graph lattice are computed at a time. Then, we prune the graphs that are infrequent, and we obtain a set of frequent graphs in that level. Based on the set of frequent graphs, we continue to compute the graphs in the next level in the graph lattice, until the set of frequent graphs in the current level is empty.

AGM (Apriori Graph Mining) [43] is the first algorithm in this category. Given the set $\mathcal{F}$ of frequent graphs in a certain level in the graph lattice, *AGM* generates graph candidates by adding a vertex to each graph in $\mathcal{F}$. Denote the set of graph candidates by $\mathcal{C}'$. Duplicates are removed by computing the *graph canonical form* for each graph in $\mathcal{C}'$. Given two graphs g and g' , the graph canonical forms of g and g' are identical if and only if g and g' are isomorphic. Therefore, for any two graphs g and g' that share the same graph canonical form, *AGM* only keeps one of the graphs in $\mathcal{C}'$. After that, for each graph $g \in \mathcal{C}'$, *AGM* computes the frequency of g by scanning the graphs in the database.

The drawbacks of *AGM* are mainly twofold: First, it generates too many candidates for each level due to exponential number of combinations; Second, its frequency computation is highly costly especially when the number of graphs in the database is sufficiently large. To address the deficiencies, Kuramochi et al. [56] propose the *FSG* technique which

utilizes edge-growth mining and computes the frequency of candidates based on the frequent graphs that have been mined already. In particular, given the set $\mathcal{F}_k$ of k -edge frequent graphs, *FSG* obtains the set $\mathcal{C}'$ of $(k+1)$ -edge candidate graphs by joining each two graphs g and g' in $\mathcal{F}_k$ if g and g' are *joinable*. We say g and g' are joinable if g and g' share a $(k-1)$ -edge common subgraph. Then, it computes the frequency of each graph $g'' \in \mathcal{C}'$ as follows: Let g and g' be the two graphs in $\mathcal{F}_k$ whose join gets g'' . Besides, *FSG* maintains a *graph ID list* $\mathcal{A}(g)$ for each frequent graph g such that $\mathcal{A}(g)$ consists of the graph IDs of g 's super-graphs in $\mathcal{G}$. As such, *FSG* first intersects $\mathcal{A}(g)$ and $\mathcal{A}(g')$ to obtain a superset of $\mathcal{A}(g'')$ due to the downward closure property. Then, it checks each graph G in the superset to verify whether g'' is a subgraph of G , and it finally obtains $\mathcal{A}(g'')$. In such a way, *FSG* computes a smaller set of candidate graphs, which reduces the overhead in frequency computation.

DFS-based approaches. Since BFS-based approaches need to keep a large number of graphs in memory, their memory usage becomes the bottleneck for them to handle large datasets or a large number of candidate graphs. To address this issue, the DFS-based approaches are proposed. In the DFS-based approaches, only one graph for each level is kept in memory. That is, we begin with root, and traverse the graph lattice in the DFS manner. Specifically, when visiting a graph g in the lattice, if g is a frequent subgraph, we generate one super-graph g' of g by adding one edge to g , then we recursively visit g' in the same manner; otherwise, we avoid the visiting of all super-graphs of g . After visiting g' , we backtrack to the computation on g , and recursively process another one super-graph of g which is generated by adding one edge to g . The computation on g terminates when we have processed all g 's super-graphs.

In an early work [100], Yan et al. develop *gSpan*, which uses a novel graph canonical form to facilitate the pruning of the search space. The graph canonical form used in *gSpan* is called *DFS code*, which permutes the edges in the DFS visiting order of the graph. In particular, *gSpan* computes the *maximal DFS code* that finds a permutation of edges such that the ordering obtained is a maximal code. Based on that, *gSpan* can extend each frequent graph efficiently by adding one edge, and it only keeps those extensions that are maximal DFS codes. After obtaining the candidate graphs, the frequency computation of each candidate graph is identical to that in *FSG*.

Later on, Huan et al. [41] propose *FFSM*, that targets at large and dense graphs with a small number of labels. *FFSM* adopts another graph canonical form, *CAM code*, that computes a permutation of vertices such that the code produced from the adjacency matrix for that permutation is maximal. Besides, *FFSM* maintains the set of occurrences for each frequent graph, where the occurrences of g in each graph $G \in \mathcal{G}$ are the subgraphs of G that are isomorphic to g . Then, it extends the occurrences of g accordingly for the frequency computation of g 's super-graphs. As such, it can avoid explicit subgraph isomorphism tests.

As an improvement, the other work [64, 70] observe that most frequent graphs are trees, and mining trees is much easier than mining general graphs. Based on this observation, they propose *GASTON* which categorizes the graphs into paths, trees, and cyclic graphs. And they develop techniques for each category to speed up the running time.

External-memory approaches. All the solutions discussed so far are in-memory algorithms that load the entire dataset into main memory. Such techniques cannot deal with datasets that are larger than the main memory size. To address this problem, Wang et al. [92] proposed *ADI-Mine*, an approach to facilitate frequent subgraph mining from a disk-based graph database. Different from *ADI-Mine*, Nguyen et al. [69] propose a data partition approach, which was earlier introduced for frequent itemset mining [62, 77]. In this approach, the entire dataset is partitioned into several portions in the first scan, and each portion is independently processed to obtain a set of candidates whose frequency is computed in a second scan of the dataset.

Parallel approaches. Due to the computation and I/O intensive characteristic of data mining problems, more and more efforts are geared towards solving it with the aid of parallel techniques. Cheung et al. [23] developed an approach for mining association rules in a distributed system. In that approach, several iterations are performed, and each iteration requires a broadcast of locally frequent itemsets to all machines. In a multi-core system, Li et al. [59] proposed a parallelizable FP-Growth [34] method, which partitions the database based on the frequent items. More recently, Li et al. [60] propose to parallelize FP-Growth in MapReduce with the aim of finding the k most frequent itemsets. In the area of parallel frequent sequence mining, Miliaraki et al. [66] describe *MG-FSM* which is built on MapReduce and follows the ideas of *projected database*.

Similar to traditional data mining problems (e.g. frequent itemset mining and frequent subsequence mining), the problem of frequent subgraph mining also incurs intensive computation and I/O costs. As such, more and more efforts are contributed to solving it with the aid of parallel techniques. There are two kinds of parallel approaches for frequent subgraph mining: one is the incremental approach [19, 24, 40], the other is the data partition approach [24]. The incremental approach is similar to the BFS-based approach, except that it partitions the computation in each level of the BFS tree among machines and aggregates the outputs of all machines to obtain the (intermediate) results. As such, the incremental approach requires a large number of rounds to terminate, and it incurs significant overhead in communication between machines since each machine can generate an enormous number of subgraphs.

In contrast to the incremental approach, the data partition approach divides the graph database into even-sized subsets such that each subset can fit in the main memory. Then each machine receives a subset of the database, and computes the graphs that are locally frequent in the subset by any existing frequent subgraph mining algorithm.

Combining the locally frequent subgraphs in all machines, we obtain a set of candidate graphs that is a super-set of graphs that are (globally) frequent in the database. Finally, for each candidate graph, we compute its frequency in the database to verify whether it is a globally frequent subgraph. To compute the frequencies for the candidate graphs, most existing approaches perform the subgraph isomorphism tests for each graph in the database, which incurs significant overhead. This is because (i) subgraph isomorphism testing is NP-hard, and (ii) both the number of candidate graphs and the number of graphs in the database would be extremely large.

Graph mining in MapReduce. MapReduce [27, 88] has established itself as the candidate of choice in *big data* problems. To address the challenges that arise in large-scale graph mining, several techniques and systems have been proposed. *PEGASUS* [50] is a system based on MapReduce for graph pattern mining and graph analysis tasks in a large graph, e.g., computing the diameter [49] and counting triangles [89]. To count or enumerate the subgraphs in a large graph with MapReduce, Zhao et al. [108] proposed a *color coding* based approach, while Afrati et al. [13] devised a multi-way join method by decomposing the given graph. Xiang et al. [98] employ MapReduce to mine the maximum cliques from a large graph using a coloring based partitioning method. However, these techniques and systems cannot solve the frequent subgraph mining problem in the transaction setting, which is the focus of our work.

None of the existing work has a satisfactory solution to frequent subgraph mining in the transaction setting with MapReduce. The previous approach from [69] also uses data partitioning, but it is not able to efficiently process a large-scale dataset, as shown in our experiments. In addition, prior approaches are all evaluated on small-scale or synthetic datasets. In contrast, our technique can handle tens of millions of real-world graphs in a moderate-sized cloud environment.

2.2 Network Motif Discovery

In the problem of network motif discovery, given a real graph G , the frequency of another graph g is the number of G 's subgraphs that are isomorphic to g , denoted by $f(g, G)$. Then, we estimate the expected frequency of g in a random graph by generating a set $\mathcal{G}_r$ of r random graphs that are degree-equivalent to G and compute $f(g, G')$ in each $G' \in \mathcal{G}_r$. Denote the expected frequency of g in a random graph by $\tilde{f}(g)$. Meanwhile, we also obtain the sampled standard deviation of g 's frequency in the random graphs, denoted by $\tilde{\sigma}(g)$. Now, we introduce the problem of network motif discovery as follows: Given G , $r > 0$, a user-defined threshold $\theta > 0$ and a number k , we are to find all size- k subgraphs g of G such that (i) $\tilde{\sigma}(g) > 0$ and (ii) $\frac{f(g, G) - \tilde{f}(g)}{\tilde{\sigma}(g)} \geq \theta$, where the size of graph g is defined as the number of vertices of g .

As mentioned in Chapter 1, previous methods [51, 54, 61, 74, 76, 93, 95] for motif discovery typically run in two steps:

- (i) *Subgraph enumeration*: Compute the set S_k of all subgraphs in the input graph G , as well as the frequency $f(g, G)$ of each subgraph g in G .
- (ii) *Frequency estimation*: Generate a set $\mathcal{G}_r$ of r random graphs that are degree-equivalent to G . For each subgraph $g \in S_k$ and each random graph $G' \in \mathcal{G}_r$, compute the frequency $f(g, G')$ of g in G' , and then determine whether g is a motif of G .

Among the above two phases, *frequency estimation* incurs by far the highest overhead due to the large number of random graphs in $\mathcal{G}_r$ that need to be examined. To alleviate this overhead, existing methods construct indices on the subgraphs in S_k for efficient subgraph search. In what follows, we first clarify the indices utilized by existing methods, and then explain the details of each phase.

CL-Index and AM-Index. Consider a random graph G' in $\mathcal{G}_r$. To derive subgraph frequencies in G' , we need to identify, for any given subgraph g' in G' , whether g' appears in S_k . For this purpose, existing methods index the subgraphs in S_k as follows. First, for each subgraph g in S_k , they compute the *canonical labeling* [65] of g , which is a sequence of numbers that uniquely identifies a graph, i.e., two graphs have the same canonical labeling if and only if they are isomorphic. A hash index, referred to as the *canonical labeling index (CL-index)*, is built to map each canonical labeling to each subgraph $g \in S_k$. For example, Ribeiro et al. [74] computes the canonical labeling of the graphs in the form of string, which are indexed by a trie [55].

Given the CL-index, we can determine whether a subgraph g' appears in S_k , by first computing the canonical labeling of g' and then checking whether the labeling appears in the CL-index. However, deriving the canonical labeling of g' is often computationally expensive. To mitigate this issue, existing methods construct an additional hash index, referred to as the *adjacency matrix index (AM-index)*, that maps adjacency matrices to subgraphs in S_k . Specifically, for each subgraph g in S_k , the AM-index records the adjacency matrix of at least one graph that is isomorphic to g . (Note that two isomorphic graphs may have different adjacency matrices.) As such, when we are to check whether a subgraph g' is in S_k , we can first examine whether the adjacency matrix of g' is indexed by AM-index. If it is indexed, then we have $g' \in S_k$; otherwise, we proceed to compute the canonical labeling of g' , and check if it appears in the CL-index. This filter-refinement approach leads to higher efficiency, as the adjacency matrix of g' is much easier to compute than its canonical labeling.

Phase 1: Subgraph Enumeration. To enumerate all size- k subgraphs in a given graph G , a naive approach is to examine all possible combinations of k vertices in G . This

approach, however, incurs prohibitive overheads due to the enormous number of vertex combinations that need to be inspected. In fact, most of the vertex combinations do not induce connected graphs, and hence, could have been ignored. (Recall that we require any subgraph to be connected.) Motivated by this, existing methods adopt the following heuristic to avoid inspecting all vertex combinations in subgraph enumeration. For each vertex v in G , they first identify a set $nbr(v, k)$ of vertices that are at most k hops to v in G (regardless of the directions of the edges). Observe that if a size- k subgraph contains v , then all vertices in the subgraph must appear in $nbr(v, k) \cup \{v\}$. Accordingly, existing methods enumerate all size- k subgraphs containing v , by inspecting the combinations of k vertices in $nbr(v, k) \cup \{v\}$; if a vertex combination induces a connected graph g containing v , then g 's adjacency matrix and canonical labeling are computed and inserted into the AM-index and CL-index, respectively. Once the k -hop neighborhood $nbr(v, k)$ of each vertex v is processed, the subgraph enumeration step terminates.

Phase 2.1: Random Graph Generation. Existing methods adopt Monte Carlo techniques to generate random graphs that are degree-equivalent to G . The most well adopted technique is the *switching algorithm* [67], which runs in an iterative manner. In each iteration, the algorithm randomly selects two directed edges $\langle v_1, v_2 \rangle$ and $\langle v_3, v_4 \rangle$ from G , and then replaces them with two new edges $\langle v_1, v_4 \rangle$ and $\langle v_3, v_2 \rangle$, if the replacement does not result in a self-loop or two identical edges in G . The algorithm terminates after $\alpha|E|$ iterations, where α is a large pre-defined constant.

Phase 2.2: Frequency Computation. To compute the frequency of each subgraph $g \in S_k$ in each random graph $G' \in \mathcal{G}_r$, existing methods enumerate each size- k subgraph g' of G' , and then utilize the AM-Index and CL-Index to check if it appears in S_k . In addition, if g' appears in the CL-Index but not the AM-Index, then its adjacency matrix m is inserted into the AM-Index, so that any other subgraph with the same adjacency matrix can be efficiently processed without inspecting the CL-Index. Once all size- k subgraphs from all random graphs are examined, the estimated frequency of each subgraph in S_k (with respect to $\mathcal{G}_r$) is computed, based on which the motifs of G are identified.

To make the aforementioned approach scalable on multiple cores, Wang et al. [93] propose a method by first generating subgraphs induced by v 's k -hop neighbors for each v in V , and then computing the network motifs in each subgraph respectively using the aforementioned approach. However, the size of subgraph induced by the k -hop neighbors of each vertex v in V would be dramatically different, this method incurs significant overheads in load imbalance. To avoid this issue, Ribeiro et al. [76] devise a master-worker strategy, in which there is a master node that generates workloads and distributes the workloads to worker nodes. As such, most of the cost in subgraph enumeration solely

happens in the master node, and there requires lots of communication cost for all nodes to synchronize the workloads.

There also exists some other approaches that do not follow these two phases. Instead, they [32, 71] first enumerate the set S_k of all possible size- k graphs, and then compute the frequency of each graph $g \in S_k$ in both real graph and random graphs. However, this approach suffers from generating too many size- k graphs that are redundant. Therefore, these approaches cannot scale to handle large k , as compared in the work [75].

Besides the aforementioned algorithms, there exist a number of CPU-based algorithms [16, 17, 52, 94] for *approximate* network motif discovery. The basic idea is to heuristically sample subgraphs from the input graph G and the random graphs in $\mathcal{G}_r$, and then identify motifs from those samples. Those algorithms are generally more efficient than conventional methods for network motif discovery, but due to their heuristic nature, they fail to provide any quality guarantees on the results produced.

In addition, there are several recent studies [13, 82, 87] on *subgraph listing*, i.e., identifying the occurrences of a query graph g in a large graph G . Although this problem is closely related to network motif discovery, the techniques in [13, 82, 87] focus on the scenario where G is a sizable graph (with billions of nodes and edges) that does not fit in the main memory of a single machine, and they employ distributed systems (e.g., MapReduce) to address the scalability issues that arise from this particular scenario. In contrast, in network motif discovery, we focus on the case where (i) G is relatively small, but (ii) there exist a large number of random graphs whose subgraphs need to be compared with those in G . As a consequence, the techniques in [13, 82, 87] are not suitable for network motif discovery.

Finally, there exist numerous techniques for *frequent subgraph mining* (see [45] for a survey) and *significant subgraph mining* [39, 72, 99], but those two problems are considerably different from network motif discovery. In particular, in frequent subgraph mining, we are given a set of graphs $\mathcal{G}$, and we aim to identify the subgraphs that appear in a large portion of the graphs in $\mathcal{G}$, disregarding the number of occurrences of each subgraph g in each individual graph. Similarly, *significant subgraph mining* also focuses on the *portion* of graphs in $\mathcal{G}$ where each subgraph g appears, and it quantifies the significance of g based on this portion instead of the frequency of g in each graph in $\mathcal{G}$. Therefore, algorithms for frequent subgraph mining and significant subgraph mining are inapplicable for identifying network motifs.

2.3 Subgraph Query Processing

This section focuses on the review on subgraph query processing in the transaction setting, as discussed in Chapter 1. Given a set of n graphs $\mathcal{G} = \{G_1, G_2, \dots, G_n\}$, and a

query graph Q , we are asked to compute the set of all graphs $G \in \mathcal{G}$ such that Q is a subgraph of G . Denote the set of Q 's super-graphs by $\mathcal{A}(Q) = \{G \in \mathcal{G} \mid Q \subseteq G\}$.

Obviously, it is costly to perform subgraph isomorphism testing on each graph in $\mathcal{G}$ since (i) subgraph isomorphism testing is NP-hard and (ii) the number of graphs in $\mathcal{G}$ could be substantially large. To address this deficiency, almost all previous work adopt the *filter-and-verification* framework. That is, given a query graph Q , they first filter the graphs in $\mathcal{G}$ and obtain a set of candidate graphs $\mathcal{C}$ such that (i) $\mathcal{C}$ is a subset of $\mathcal{G}$ but a super-set of $\mathcal{A}(Q)$, and (ii) the size of $\mathcal{C}$ is much smaller than that of $\mathcal{G}$. Then, they verify each graph G in $\mathcal{C}$ to check whether Q is a subgraph of G by subgraph isomorphism testing algorithms [26, 90].

Note that, there exists a previous work [96] that does not follow the *filter-and-verification* framework. In particular, it decomposes each graph in $\mathcal{G}$ into several sub-graphs, which are stored in an index that allows fast retrieval. As such, any subgraph query can be immediately answered with the index. However, given the exponential number of subgraphs stored in the index, which renders the pre-processing phase of this approach extremely costly in terms of both time and space.

To facilitate the filter process in the filter-and-verification framework, existing approaches [22, 38, 80, 83, 101, 104, 106, 110] construct index structures in a pre-processing step. The index structures help prune a large number of graphs in $\mathcal{G}$, which saves a huge amount of computation cost. Most existing work use frequent subgraphs (FSG) in index construction, whereas the others do not. Therefore, we classify the existing index structures into two categories according to whether they use FSGs, and explain the approaches in each category as follows.

FSG-based approaches. Techniques [22, 80, 101, 104, 106] in this category exploit the frequent subgraph mining algorithms to extract a set $\mathcal{F}$ of graphs, such that for each graph $g \in \mathcal{F}$ we have $|\mathcal{A}(g)| \geq s$, where $s \geq 0$ is a pre-defined threshold. Then, the index structures are constructed on $\mathcal{F}$ where the graph canonical forms of graphs $g \in \mathcal{F}$ serve as key and $\mathcal{A}(g)$ the corresponding values. Note that, $\mathcal{A}(g)$ is stored as a set of graph IDs. In the query processing, given a query graph Q , these techniques decompose Q into several sub-queries such that (i) each sub-query is a subgraph of Q , and (ii) each sub-query exists in the index structure. Then, a set $\mathcal{C}$ of candidate graphs can be computed by intersecting $\mathcal{A}(g)$ for all sub-queries g of Q . Finally, the answer of Q is obtained by performing subgraph isomorphism tests with Q against the graphs in $\mathcal{C}$.

However, the size of $\mathcal{F}$ could be extremely large due to the exponential number of subgraphs in the database. To address this issue, Yan et al. [101] propose *G-index*, which generates all frequent subgraphs of size up to a pre-defined threshold $maxL$, and removes all *non-discriminative* graphs. Given a pre-defined threshold γ , a graph g is discriminative with respect to $\mathcal{F}$ if $\frac{|\cap_{g' \in \mathcal{F} \wedge g' \subseteq g} \mathcal{A}(g')|}{|\mathcal{A}(g)|} \geq \gamma$. Therefore, given a query graph

Q , G -index processes Q in the same manner as above, except that it decomposes Q into several sub-queries whose sizes are up to $maxL$.

To further reduce the cost in index construction, several approaches [80, 104, 106] observe that (i) most frequent graphs are trees and (ii) the cost of mining frequent trees are much smaller than that of mining general graphs. Based on the observations, they use frequent trees in index construction, and decompose the query graph into trees for query processing.

In contrast to the above approaches, Cheng et al. [22] propose the FG -index, which is a disk-based approach that alleviates the overhead in memory consumption by keeping a subset of frequent subgraphs in memory. Besides, FG -index generates all frequent subgraphs such that the query Q can be answered immediately if Q is a frequent subgraph.

Non-FSG-based approaches. Unlike the FSG-based approaches, techniques [38, 83, 110] in this category utilize very different features, which renders their indexing and query processing methods different from each other. In the following, we explain each approach respectively.

The first approach is *GraphGrep* [83], which enumerates all paths of size up to $maxL$ that are utilized in the index construction. Given a query graph Q , *GraphGrep* divides Q into paths to probe the index structure for computing a set of candidate graphs. Although *GraphGrep* has much smaller cost in index construction than that of FSG-based approaches, the pruning power of paths is much lower than that of FSG-based approaches, which renders *GraphGrep* incur more overhead in the query processing.

Then, He et al. [38] develop *C-Tree* by proposing the concept of *graph closure*, which is a summarized graph that contains a set of graphs. Based on the concept, *C-Tree* builds a hierarchical tree where each leaf is a graph in $\mathcal{G}$ and each internal node represents a graph closure that summarizes the structures of its child nodes. Such an index structure can be built using the hierarchical clustering algorithm [33]. In the query processing of a query graph Q , *C-Tree* traverses the hierarchical tree from root. When visiting a node g in the tree, if g is an internal node, *C-Tree* performs the *pseudograph isomorphism testing* which can check whether there exists a leaf in the subtree under g that contains Q as a subgraph. If g fails in the pseudograph isomorphism testing, *C-Tree* does not need to visit the descendants of g in the tree; otherwise, *C-Tree* continues to check the child nodes of g recursively. On the other hand, if g is a leaf in the tree, *C-Tree* performs the subgraph isomorphism testing to check whether Q is a subgraph of g . However, *C-Tree* has severe performance drawbacks as compared in [37], since (i) the pseudograph isomorphism tests are extremely expensive and (ii) the size of graph closures and the number of nodes in the tree could be significantly large, which explodes the limited size of memory.

Finally, Zou et al. [110] devise the technique of *gCode* which first generates a *vertex signature* for each vertex of each graph G in $\mathcal{G}$. Based on that, it computes a *graph*

signature for G , which are then indexed in a tree structure. Given a query graph Q , *gCode* computes the vertex signatures and graph signature for Q using the same approach, and then probes the tree structure to obtain a set of candidate graphs for verification. As shown in [37], the pruning power of this technique is low in real sparse graphs, leading to a large number of candidate graphs produced by this approach.

Generalized subgraph query processing. Different from the aforementioned problems that adopt traditional subgraph matching, there are some existing studies of graph matching problem by allowing edges to map to paths for graphs [28, 29, 47, 109]. However, their queries are too rigid by fixing the length of all the mapping paths [109], or too relax by allowing node similarity matching [29]. Besides, all these works are tailored to query a single large graph making them unsuitable for querying a large set of small or medium-sized graphs.

On the other hand, various types of graph query processing on a large set of small or medium-sized graphs have been studied in the literature in recent years and we restrict our discussion on the closely related ones, namely subgraph query processing [22, 38, 80, 83, 101, 104, 106], supergraph query processing [20, 21, 81, 105], and similarity graph query processing [79, 102, 103, 107]. All these works proposed some indexing techniques to filter out as many unmatching data graphs as possible. Although many different types of graph indexing techniques have been proposed, none of them is similar to our indexes except FG-index [22], which is similar to FP-index. However, the only similarity lies on the use of frequent patterns to avoid verification and the use of infrequent edges to reduce the candidate set size, while the index structure of FP-index (which builds on B+-trees) is totally different from that of FG-index (which is an unbalanced tree built on the clusters of frequent patterns). Apart from that, both D-index and S-index are entirely different from all existing indexes. In addition, our work is the first to propose indexes for processing generalized subgraph queries.

2.4 Summary

In this chapter, we first review the approaches for frequent subgraph mining in transaction setting, which can be classified into two categories depending on how we traverse the graph lattice, i.e., BFS-based approaches and DFS-based approaches. In the BFS-based approaches, we first compute a set $\mathcal{C}$ of size- $(k + 1)$ candidate graphs from the set $\mathcal{F}_k$ of size- k frequent graphs. Then, we calculate the frequency for each graph in $\mathcal{C}$ and obtain the set $\mathcal{F}_{k+1}$ of size- $(k + 1)$ frequent graphs. Compared with BFS-based approaches, the DFS-based approaches traverse the graph lattice in the DFS manner. Therefore, the memory consumption of the DFS-based approaches is much smaller than that of the BFS-based approaches. Since the DFS-based approaches generate the size- $(k + 1)$

candidate graph from a single size- k frequent graph, whereas the BFS-based approaches use the whole set of size- k frequent graphs, the BFS-based approaches are more powerful in pruning the search space.

Then, we summarize the previous approaches in network motif discovery, which typically consists of two phases: subgraph enumeration and frequency computation. Both phases rely on the AM-index and CL-index to accelerate the process and reduce the overhead in subgraph isomorphism tests. Since the number of random graphs could be sizable and the computation cost on each random graph is as expensive as on real graph, however, these approaches often cannot terminate within a reasonable time. In this thesis, we propose to utilize GPUs for speeding up the processes on random graphs, as the GPU often contains thousands of cores working simultaneously which makes it highly capable in parallelism. Nevertheless, as will be discussed in Section 4.2, the previous CPU-based approaches cannot be directly translated on GPU due to the specific properties of GPU computing model.

Finally, we present a detail survey on the existing algorithms for subgraph query processing in transaction setting. Almost all of these approaches follow the filter-and-verification framework with the aid of index structures, constructed in a pre-processing step. In this framework, given a query graph Q , we first probe the index structures to obtain a set of candidate graphs. The number of candidate graphs should be much smaller than the size of database $\mathcal{G}$. Then, we verify each candidate graph by performing subgraph isomorphism testing with Q . Depending on whether index construction uses frequent subgraph mining (FSG), we classify these approaches into two categories: FSG-based approaches and non-FSG-based approaches. Compared with non-FSG-based approaches, the FSG-based approaches have relatively higher cost in index construction due to the extensive large number of frequent subgraphs that will be generated. However, the FSG-based approaches are more effective than the non-FSG-based approaches in pruning candidate graphs, since frequent subgraphs preserve more structure information than the features used in non-FSG-based approaches.

Chapter 3

Large-Scale Frequent Subgraph Mining

This chapter presents a novel two-step filter-refinement computation workflow which uses MapReduce for frequent subgraph mining, is highly parallelizable and avoids excessive amounts of data communication. Besides, we devise a statistical model for predicting which locally frequent subgraphs are likely to also be globally frequent. The model increases slightly the amount of computation in the filter step, but has as benefit significant gains in the refinement step. Then, we develop a lightweight technique for graph compression that reduces significantly the amount of data transmission between the worker nodes and the file system, and hence decreases further the overall runtime of the mining algorithm. Finally, through an extensive experimental evaluation on several real-world graph datasets, we show that the proposed approach clearly outperforms the existing state-of-the-art.

The remainder of this chapter is organized as follows: Section 3.1 introduces fundamental concepts and definitions. Section 3.2 gives an overview of the proposed approach, whereas Sections 3.3 and 3.4 provide specific details of the filter and refinement steps, respectively. The graph compression scheme to reduce communication cost is presented in Section 3.5. An extensive experimental evaluation is presented in Section 3.6, followed by the summary in Section 3.7.

3.1 Preliminaries

Section 3.1.1 focuses on *mining frequent subgraphs*, whereas Section 3.1.2 provides a brief MapReduce primer.

Figure 3.1: Frequent subgraph computation.

3.1.1 Mining Frequent Subgraphs

Let $\mathcal{G}$ be a set of n graphs where each node and edge is labeled. We denote each graph $G \in \mathcal{G}$ as a quadruple $G = (V, E, L, l)$, where V and E are the set of vertices and edges in G , respectively, and l is a labeling function that maps each vertex and edge in G to a label in a finite alphabet L . For ease of exposition, we assume that G is undirected and connected; however, our results can be easily extended to the case of directed or disconnected graphs.

Given any graph $G' = (V', E', L', l')$, we say G' is a *subgraph* of another graph $G = (V, E, L, l)$, if there exists an injective function $\mu : V' \rightarrow V$ such that $\forall (u, v) \in E'$ it holds that

$$(l(u) = l'(\mu(u))) \wedge (l(v) = l'(\mu(v))) \wedge (l(u, v) = l'(\mu(u), \mu(v))).$$

In other words, the labels for each edge as well as the labels for the edge's endpoints are identical. We use $G' \subseteq G$ to denote that G' is a subgraph of G , and we refer to G as a *super-graph* of G' .

The *frequency* of a graph G' in $\mathcal{G}$, denoted as $f(G')$, is defined as the number of graphs in $\mathcal{G}$ that contain G' as a subgraph. That is,

$$f(G') = |\{G \mid G \in \mathcal{G} \wedge G' \subseteq G\}|.$$

Meanwhile, the *support* of G' in $\mathcal{G}$ is defined as $f(G')/n$, i.e., the fraction of graphs in $\mathcal{G}$ that are super-graphs of G' , where n is the number of graphs in $\mathcal{G}$. We say that G'

is a *frequent subgraph* in $\mathcal{G}$, if the support of G' is not less than a *support threshold* θ ($0 \leq \theta \leq 1$).

Example 3.1: Figure 3.1 shows 13 graphs, $g_1, \dots, g_{10}$, and G_1, G_2, G_3 , where the label of each vertex is in the domain $[0, 2]$, and the label of each edge is in the domain $[1, 3]$. For each g_i ($1 \leq i \leq 10$), there exists a graph G_j where $1 \leq j \leq 3$ such that g_i is a subgraph of G_j . For instance, g_{10} is a subgraph of G_1 , since we can injectively map all the vertices and edges of g_{10} to the vertices and edges of G_1 . Consider a graph set $\mathcal{G} = \{G_1, G_2, G_3\}$, the frequency of g_1 is $f(g_1) = |\{G_1, G_2, G_3\}| = 3$, and, similarly, the frequency of g_{10} is $f(g_{10}) = |\{G_1\}| = 1$. That is, the support of g_1 is $s(g_1) = 1$, and the support of g_{10} is $s(g_{10}) = 1/3$. $\square$

Given θ and $\mathcal{G}$, our objective is to identify all frequent subgraphs in $\mathcal{G}$, as well as the frequency of each frequent subgraph in $\mathcal{G}$. We aim to accomplish this task using a *MapReduce* program on m machines.

3.1.2 MapReduce

A MapReduce program, also referred to as a *workflow*, consists of several *rounds*, each of which contains three *phases*: *map*, *shuffle*, and *reduce*, as follows:

- (i) *Map*. In this phase, each machine reads data from a *distributed file system (DFS)*, and applies a *map function* on the data to convert them into a set of *pairs*. Each pair consists of a *key* and a *value*.
- (ii) *Shuffle*. In this phase, the key-value pairs are aggregated by keys, and the values in the pairs with the same key are grouped as a list. Each key, along with the corresponding list of values, is then sent to one of the machines.
- (iii) *Reduce*. Each machine examines the keys and lists that it receives in the shuffle phase, and then applies a *reduce function* on each list of values. The function transforms each list into a new key-value pair, which is then stored in the DFS and can be utilized by subsequent rounds of the MapReduce program.

In subsequent sections, we focus on the design of the Map and Reduce functions, as the shuffle phase is automatically handled by the MapReduce infrastructure (e.g., Hadoop [8]).

Figure 3.2: Proposed filter-refinement MapReduce workflow for frequent subgraph mining.

3.2 Solution Overview

The proposed filter-and-refinement MapReduce solution is illustrated in Figure 3.2 and consists of three rounds of computation:

Round 1: *Filter*. In the map phase of this round, each machine M_i ($i = 1, \dots, m$) reads a *disjoint* subset $\mathcal{G}_i$ of $\mathcal{G}$ and identifies a set of graphs H_i , such that (i) each graph $G \in H_i$ is the subgraph of at least one graph in $\mathcal{G}_i$, and (ii) G is likely to be a frequent subgraph in $\mathcal{G}$. Then, for each graph $G \in H_i$, M_i outputs a key-value pair where the key is G and the value indicates $f_i(G)$, i.e., the number of graphs in $\mathcal{G}_i$ that are super-graphs of G . Next, in the shuffle phase (not explicitly shown in the diagram), all key-value pairs having G as key are sent to the same machine, say M_j . Finally, in the reduce phase, M_j inspects the list of values with key G , and computes the sum of all individual frequencies. Based on the sum, M_j evaluates whether G is likely to be a frequent subgraph in $\mathcal{G}$. If G cannot be a frequent subgraph, then it is discarded; otherwise, M_j outputs a key-value pair with key G and value equal to the sum of frequencies (as we will discuss in Section 3.3 the value contains some additional information as well).

Round 2: *Sorting*. The set of key-value pairs obtained from the filter round is sorted in ascending order of *graph size* (defined as number of graph edges) in the key. This is accomplished by invoking a single-round MapReduce sorting algorithm such as *TeraSort* in *Hadoop* [8]. The sorted sequence is then stored in the DFS.

Round 3: *Refinement*. In the map phase, each machine M_i reads $\mathcal{G}_i$ and the sorted sequence S from the DFS. Then, for each graph G that appears in S , M_i determines $f_i(G)$, the frequency of G in $\mathcal{G}_i$. Next, M_i outputs a key-value pair $\langle G, f_i(G) \rangle$. Such key-value pairs are re-distributed among the machines in the shuffle phase. Then, for each key G , the reduce phase computes the sum of the values corresponding to G . (Note that this sum equals $f(G)$, the exact frequency of G in $\mathcal{G}$.) If $f(G)/n$ is not less than the

support threshold θ , then a key-value pair $\langle G, f(G) \rangle$ is written to the DFS to indicate that G is a frequent subgraph in $\mathcal{G}$.

Since there is nothing specific to the studied problem within the sorting round, we omit it further from consideration, and focus on the filter and refinement rounds, where the challenges reside. First, in the filter round, given that each machine M_i sees only a subset $\mathcal{G}_i$ of the graphs in $\mathcal{G}$, we focus on how M_i can identify subgraphs that are likely to be frequent in $\mathcal{G}$ (i.e., globally frequent). We refer to such subgraphs as frequent subgraph *candidates*. This must be done in a manner that ensures both correctness and efficiency of the overall solution (Section 3.3). Second, in the refinement round, when each machine M_i computes the frequency of candidate graphs in $\mathcal{G}_i$, we focus on how to lower the cost of subgraph isomorphism tests required to identify super-graphs of the candidates (Section 3.4). Third, as each machine may generate a relatively large set of candidate subgraphs, we study how to reduce the communication overhead in re-distributing and manipulating the key-value pairs for the candidates (Section 3.5).

3.3 Filter Round

The filter round distributes graphs in $\mathcal{G}$ onto the m machines, and asks each machine to report candidates for frequent subgraphs. To ensure completeness of the result, the set of candidates reported should not incur any *false negative*, i.e., every frequent subgraph in $\mathcal{G}$ must be reported as a candidate by at least one machine.

3.3.1 A Preliminary Approach

One simple approach to implement the filter round is as follows:

- (i) In the map phase, divide $\mathcal{G}$ into m disjoint subsets $\mathcal{G}_1, \mathcal{G}_2, \dots, \mathcal{G}_m$, and send $\mathcal{G}_i$ ($i = 1, \dots, m$) to machine M_i . Next, each M_i reports a graph G as a candidate whenever $f_i(G)/n_i \geq \theta$.
- (ii) In the reduce phase, the union of the sets of candidates produced by the m machines is written to the DFS.

In other words, we report a graph as a candidate whenever it is *locally frequent* on some machine. This avoids false negatives because if a graph G' is not reported as a candidate by any machine, then the frequency of G' in $\mathcal{G}$ must satisfy the following inequality:

$$f(G') = \sum_{i=1}^m f_i(G') < \sum_{i=1}^m (\theta \cdot n_i) = \theta \cdot n,$$

i.e., we have $s(G') = f(G')/n < \theta$, and hence, G' cannot be a frequent subgraph.

However, this approach is inefficient as it may lead to a large number of *false positives*, i.e., candidate graphs that are reported even though they are actually infrequent in $\mathcal{G}$. Consider a graph G such that $f(G) < n \cdot \theta$. For G not to be reported, it must hold that on each of the m machines

$$\forall i \in [1, m], f_i(G) < n_i \cdot \theta. \quad (3.1)$$

For Eq. (3.1) to hold, the graphs in $\mathcal{G}$ that are super-graphs of G must be distributed uniformly to the m machines. However, such an even distribution is difficult to obtain, especially when $f(G)$ is close to $n \cdot \theta$, as it requires prior knowledge of the super-graphs of G in $\mathcal{G}$, which is not available before the map phase of the filter round.

To reduce the number of false positives, one can utilize the map phase of the filter round. As a naive strategy, assume that in the map phase each machine M_i outputs the frequency for *all* subgraphs G in $\mathcal{G}_i$ (regardless of whether G is locally frequent or not). Then, in the reduce phase, we can sum up the frequency of G on each machine to obtain its global frequency $f(G)$, based on which we can precisely decide whether G is globally frequent. As such, we can eliminate all false positives in the candidate graphs, but at the cost of computing local frequencies for an excessive number of graphs. Next, we develop a more advanced approach for candidate graph generation that reduces false positives without incurring the prohibitive cost of frequency computation for all subgraphs.

3.3.2 An Improved Approach

The improved approach for the filter round works as follows:

- (i) In the map phase, we distribute each graph $G \in \mathcal{G}$ to a randomly selected machine, i.e., each machine M_i receives a sample set of $\mathcal{G}_i$ with a sampling rate $1/m$.
- (ii) Then, M_i outputs $f_i(G)$ of any graph G that is locally frequent in $\mathcal{G}_i$ (i.e., $f_i(G) \geq n_i \cdot \theta$). In addition, for a *selected set* of graphs that are locally infrequent in $\mathcal{G}_i$, M_i also outputs key-value pairs that record the local frequencies of those graphs.
- (iii) In the reduce phase, for each graph G that is locally frequent on some machine, we derive an upper-bound of $f(G)$ (i.e., the global frequency of G). If the upper-bound is at least $\theta \cdot n$, then we output a key-value pair for G to indicate that it is a candidate frequent subgraph.

Specifically, the upper-bound of $f(G)$ that we use in the reduce phase is $f^\top(G) = \sum_{i=1}^m f_i^\top(G)$, where

$$f_i^\top(G) = \begin{cases} f_i(G), & \text{if } \mathbf{M}_i \text{ reports } f_i(G) \text{ in the} \\ & \text{map phase} \\ \lceil n_i \cdot \theta \rceil - 1, & \text{otherwise} \end{cases} \quad (3.2)$$

That is, whenever $f_i(G)$ is unknown, we use $\lceil n_i \cdot \theta \rceil - 1$ as an optimistic estimation of $f_i(G)$. The rationale of using locally infrequent subgraphs is to lower the upper-bound of each candidate's global frequency for more powerful pruning.

Selection of locally infrequent graphs. For the above approach to be efficient, we need to carefully select the set of locally infrequent graphs output by each machine $\mathbf{M}_i$ in the map phase. If this set is too large in size, performance will decrease. On the other hand, if it is too small, then the reduce phase may leave a large number of false positives in the candidate graphs, since the frequency upper-bound $f^\top(G)$ for a candidate graph G tends to be loose when the local frequencies of G are not reported by a large portion of the machines. This calls for a method that can *selectively* choose locally infrequent graphs from each machine while maintaining the pruning power of $f^\top(G)$.

To obtain a good trade-off, one needs to determine when will a locally infrequent graph be useful for pruning false positives in the candidate graphs. Consider a graph G' that is locally infrequent on machine $\mathbf{M}_i$. Assume that $\mathbf{M}_i$ reports the local frequency of G' (i.e., $f_i(G')$) in the map phase. Then, $f_i(G')$ will help eliminate false positives in the reduce phase only if G' is reported as a candidate graph by at least one machine $\mathbf{M}_j$, in which case $f_i(G')$ will contribute to the calculation of the upper-bound $f^\top(G')$ that decides whether G' should be pruned. In other words, $\mathbf{M}_i$ should report $f_i(G')$ only if G' is locally frequent on some other machine $\mathbf{M}_j$. Nevertheless, as $\mathbf{M}_i$ only sees its local data $\mathcal{G}_i$ in the map phase, it cannot determine whether G' might be locally frequent on other machines. To address this issue, we propose a probabilistic inference method as follows.

First, given that (i) $\mathcal{G}_i$ is a random sample set of $\mathcal{G}$ and (ii) a $f_i(G')/n_i$ fraction of the graphs in $\mathcal{G}_i$ are super-graphs of G' , $\mathbf{M}_i$ estimates that the frequency of G' in $\mathcal{G}$ is roughly $n \cdot f_i(G')/n_i$. Given this estimation, $\mathbf{M}_i$ infers that the local frequency of G' on any other machine $\mathbf{M}_j$ follows a binomial distribution:

$$Pr \{f_j(G') = k\} = \binom{n_j}{k} p^k (1-p)^{n_j-k},$$

where $p = f_i(G')/n_i$. Accordingly, the event that G' is locally frequent on $\mathbf{M}_j$ should occur with the following probability:

$$Pr \{f_j(G') \geq \lceil \theta \cdot n_j \rceil\} = \sum_{k=\lceil \theta \cdot n_j \rceil}^{n_j} \binom{n_j}{k} p^k (1-p)^{n_j-k}.$$

By the union bound, M_i gets the following upper-bound on the probability that G' is locally frequent on at least one machine besides M_i :

$$\begin{aligned}
 & Pr \{ \exists j \neq i, f_j(G') \geq \lceil \theta \cdot n_j \rceil \} \\
 & \leq \sum_{j \neq i} Pr \{ f_j(G') \geq \lceil \theta \cdot n_j \rceil \} \\
 & = \sum_{j \neq i} \sum_{k=\lceil \theta \cdot n_j \rceil}^{n_j} \binom{n_j}{k} p^k (1-p)^{n_j-k}.
 \end{aligned} \tag{3.3}$$

Note that this upper-bound is easy to compute given the cumulative distribution function of the binomial distribution.

Based on the above upper-bound, M_i can then decide whether the local frequency of G' should be output. In particular, if the upper-bound falls below a threshold ρ which is a system parameter (e.g., $< 5\%$), then M_i infers that G' is unlikely to be locally frequent on any other machines, and hence, it would not output any key-value pair pertinent to G' . As such, M_i can avoid reporting a large number of key-value pairs that are useless for the pruning procedure in the reduce phase. In Section 3.6 we discuss a methodology for setting the value of ρ in practice.

Computing graph frequencies. To compute the set of locally frequent subgraphs, one of several techniques [41, 43, 56, 64, 70, 92, 100] that address subgraph mining in a sequential setting can be used. All such techniques follow a branch-and-bound approach. All subgraphs are organized in a lattice, where each node denotes a subgraph. We add a *dummy root* node above the single-edge subgraphs (corresponding to an empty subgraph). Each node in the lattice extends the graph of its parent node(s) by adding a single edge, i.e., a child node is a super-graph of all its parent nodes. The lattice is traversed top-down, and the support of each visited node is computed. If a visited graph is frequent, the corresponding graph and its frequency are output, and the traversal continues with the child nodes. Otherwise, all its child nodes and the descendants are pruned.

Example 3.2: Figure 3.3 shows the lattice formed by the graphs $g_1, \dots, g_{10}$ in Figure 3.1. Each node in the lattice denotes a subgraph. Let $\theta = 0.5$. In the beginning of the mining process, we visit the single-edge subgraphs, i.e., g_1 and g_2 . Then, we compute their supports, as $s(g_1) = 1$ and $s(g_2) = 1/3$. Since $s(g_2) < \theta$, we stop visiting all descendants of g_2 in the lattice. As $s(g_1) > \theta$, we output g_1 , as well as its frequency, and continue the traversal on the branch of g_1 by visiting g_3 and g_4 . Next, we compute $s(g_3) = s(g_4) = 1/3$, and the branches of g_3 and g_4 are pruned. The traversal stops since there are no more nodes to visit. $\square$

For illustration purposes, we use white nodes to denote frequent subgraphs, grey nodes to denote visited nodes which are infrequent, and black nodes to denote unvisited

Figure 3.3: Subgraph lattice.

subgraphs. Existing algorithms return only white nodes (i.e., frequent subgraphs). However, to improve the efficiency of the refinement step, we choose to also return some grey nodes, if their probability to be globally frequent is above threshold ρ . Note that, even though existing techniques do not output grey nodes, they still have to compute their frequencies, as required by the termination condition of the traversal algorithm. Therefore, our decision to output grey subgraphs as well comes at no additional computational cost in the filter step.

Efficient computation of upper-bound probabilities. Each machine M_i may produce a large number of subgraphs, so computing the probability in Eq. (3.3) separately for each subgraph would be costly. Due to the fact that Eq. (3.3) is monotonic in the range $[1, \lceil \theta \cdot n_j \rceil]$, given a user-defined probability upper-bound ρ , we can employ *binary search* to determine an appropriate value for frequency f_i that is equal or close to the frequency of a graph G whose upper-bound probability is ρ . Let $\alpha = \lceil \theta \cdot n_j \rceil$. The binary search first considers the range $[1, \alpha]$ and computes the probability for frequency $f_i = \lceil \alpha/2 \rceil$. If the probability is larger than ρ , we change the range to $[1, \lceil \alpha/2 \rceil]$, and compute the probability at $\lceil \alpha/4 \rceil$; otherwise, the range of $[\lceil \alpha/2 \rceil + 1, \alpha]$ is considered, and we compute the probability at $\lceil (\lceil \alpha/2 \rceil + 1 + \alpha)/2 \rceil$ to determine the next range. The search will stop if the obtained probability value converges close enough to ρ . Let f_p denote the frequency value obtained as above. For a given graph G , if $f_i(G)$ is smaller than f_p , the probability that G is globally infrequent is high, hence we prune G from further computation.

Implementation Issues. In the map phase, each key-value pair that is sent out by a machine M_i has a graph G as the key, and the frequency of G as well as the *machine id* i of M_i as the value. In the key, G is represented in a graph canonical form, which will be compressed as discussed in Section 3.5. The canonical form of each graph G allows us to aggregate the graphs in the shuffle phase so as to obtain the upper-bound of $f(G)$ in the reduce phase. As to the value, the frequency of G in each machine will be calculated

Figure 3.4: Top-Down vs. Bottom-Up strategy.

as a sum in the reduce phase, while the machine ids will be retained as one part of the new value. Note that, the machine id in the value enables us to identify the graphs that are sent out by all machines, i.e., a subset of globally frequent subgraphs. These graphs as well as their frequencies are written to DFS as part of the final result. Furthermore, when the key-value pairs are read by machine M_i in the refinement round, if M_i already appears in the list of machine ids then the frequency of the graph in the key does not need any refinement on behalf of M_i , so redundant computation is avoided.

3.4 Refinement Round

At the start of the refinement round, each machine M_i receives the sorted sequence S of candidate frequent graphs from the DFS. Machine M_i constructs $\mathcal{S}_i$ by removing from S all key-value pairs that contain the machine id of M_i (if a value contains the id of M_i , it means that there is no refinement required in the current partition $\mathcal{G}_i$ of the data). Each candidate graph G in $\mathcal{S}_i$ is locally infrequent in M_i , i.e., $f_i(G) < \lceil \theta \cdot n_i \rceil$, and its frequency needs to be determined.

Given a candidate graph G , we denote as $\mathcal{A}(G)$ the set of super-graphs of G in $\mathcal{G}_i$. To compute the frequency $f_i(G) = |\mathcal{A}(G)|$, a naive approach is to perform a subgraph isomorphism test between G and each graph in $\mathcal{G}_i$. However, due to the computational complexity of subgraph isomorphism testing, such an approach will not scale well. Recall that, inclusion relationships exist among candidate graphs, captured by a lattice structure. A child node G in the lattice is a super-graph of a parent node G' , hence $\mathcal{A}(G) \subseteq \mathcal{A}(G')$. Such dependencies help save redundant computation. We devise two strategies for frequency computation, *Top-Down* and *Bottom-Up*, named after the direction in which the strategy is traversing the lattice.

3.4.1 The Top-Down Approach

In the Top-Down approach, we scan $\mathcal{S}_i$ once in ascending order of graph size (i.e., from top to bottom of the lattice), and compute $\mathcal{A}(G)$ for each candidate graph G as soon as we visit node G . Specifically, if there is no candidate graph G' in $\mathcal{S}_i$ such that G' is a subgraph of G (i.e., G has no parents), we perform isomorphism testing against every graph in $\mathcal{G}_i$ to compute $\mathcal{A}(G)$. Otherwise, we first compute an *upper-bound* of $\mathcal{A}(G)$ by intersecting $\mathcal{A}(G')$ for all $G' \subseteq G$ and $G' \in \mathcal{S}_i$. We denote the upper-bound of $\mathcal{A}(G)$ as $ub(G)$, i.e.,

$$ub(G) = \bigcap_{G' \subseteq G \wedge G' \in \mathcal{S}_i} \mathcal{A}(G').$$

Note that, for any graph $G' \in \mathcal{S}_i$ which is a subgraph of G ,

$$|\mathcal{A}(G)| \leq |ub(G)| \leq |\mathcal{A}(G')| < \lceil \theta \cdot n_i \rceil \ll n_i,$$

if θ is sufficiently small. Besides, since we process the candidate graphs in $\mathcal{S}_i$ in ascending order of graph size, we have all $\mathcal{A}(G')$ before visiting G . Therefore, to compute $\mathcal{A}(G)$, we perform isomorphism testing only against the graphs in $ub(G)$, instead of the entire $\mathcal{G}_i$.

The computational overhead can be reduced further if we consider only the set of *maximum* subgraphs of G . We say that a graph G' in $\mathcal{S}_i$ is a maximum subgraph of G if there does not exist another graph G^+ in $\mathcal{S}_i$ such that G^+ is a subgraph of G and also a super-graph of G' . To explain that, consider that there exists a graph $G^+ \in \mathcal{S}_i$ such that G^+ is a subgraph of G' , then we have $\mathcal{A}(G') \subseteq \mathcal{A}(G^+)$. Therefore, $\mathcal{A}(G^+)$ is not useful to reduce the size of $ub(G)$ if we already visited G' . In the lattice, the parent nodes of G are the maximum subgraphs of G , which are easy to determine.

Example 3.3: Continuing Example 3.2, Figure 3.4(a) shows the graph processing order of the Top-Down strategy with dashed arrows. We begin by processing g_5 , which does not have any parents, so $\mathcal{A}(g_5)$ is computed by performing isomorphism testing against every graph in $\mathcal{G}_i$. The traversal then continues as shown, with $g_6, \dots, g_{10}$. In the case of nodes that have parents, it is possible to reuse some of the earlier computation. For instance, when visiting g_{10} , we compute $ub(g_{10}) = \mathcal{A}(g_7) \cap \mathcal{A}(g_8)$, and examine only the graphs in $ub(g_{10})$ to obtain $\mathcal{A}(g_{10})$. $\square$

3.4.2 The Bottom-Up Approach

The drawback of the Top-Down approach is that it still performs some redundant computations. Since the super-graphs of G are also the super-graphs of G 's subgraphs, the super-graphs of G will be examined for each of G 's subgraphs in the Top-Down approach.

To avoid the redundancy, we devise the Bottom-Up approach, which consists of two scans of $\mathcal{S}_i$.

The first scan processes the candidate graphs in $\mathcal{S}_i$ in ascending order of graph size. Given a candidate graph G , if G does not have subgraphs in $\mathcal{S}_i$, we compute $\mathcal{A}(G)$ by examining every graph in $\mathcal{G}_i$. That is, we obtain a tight upper-bound as $ub(G) = \mathcal{A}(G)$. Otherwise, if G has subgraphs G' in $\mathcal{S}_i$, the upper-bound of $\mathcal{A}(G)$ is then computed by intersecting the upper-bounds of all $\mathcal{A}(G')$, i.e.,

$$ub(G) = \bigcap_{G' \subseteq G \wedge G' \in \mathcal{S}_i} ub(G').$$

However, the isomorphism testing against graphs in $ub(G)$ is not performed right away, instead it is deferred until later on in the algorithm. According to the analysis in Section 3.4.1, $ub(G')$ is obtained before processing G , and $|ub(G)| < \lceil \theta \cdot n_i \rceil$. In addition, we only need to consider the maximum subgraph of G to compute $ub(G)$.

The second scan processes the candidate graphs in $\mathcal{S}_i$ in *descending* order of graph size. Given a candidate graph G , if G has no super-graphs in $\mathcal{S}_i$, we examine every graph in $ub(G)$, which is computed in the first scan, to obtain $\mathcal{A}(G)$. Otherwise, a *lower bound* of $\mathcal{A}(G)$ is computed by merging $\mathcal{A}(G')$ for all G 's super-graphs $G' \in \mathcal{S}_i$. We denote the lower bound of $\mathcal{A}(G)$ as $lb(G)$, i.e.,

$$lb(G) = \bigcup_{G \subseteq G' \wedge G' \in \mathcal{S}_i} \mathcal{A}(G').$$

Since we scan the graphs in descending order of graph size, the frequency of G' has been computed before G . Note that, the graphs in $lb(G)$ are super-graphs of G , therefore, to compute the frequency of G , we only need to examine the graphs in a subset of $ub(G)$, i.e.,

$$ub(G) \setminus lb(G).$$

To avoid re-computing the frequency of G in the second scan in cases where $\mathcal{A}(G)$ has been computed in the first scan, we mark G in the first scan. Before processing a graph in the second scan, we first check its mark to determine whether we need to compute its frequency or not.

Example 3.4: Figure 3.4(b) illustrates the Bottom-Up strategy, which first visits g_5 , and then visits all the other nodes following the dashed arrows. The upper bounds are constructed in this traversal, but no isomorphism tests are performed (except for nodes without parents). When g_{10} is reached, it is determined that $ub(g_{10}) = ub(g_7) \cap ub(g_8)$, and the actual subgraph isomorphism testing is performed for graphs in $ub(g_{10})$. Then, the traversal returns following the dotted arrows, towards the first node g_5 . During the second traversal, when re-visiting some node, only graphs in the difference between the

upper bound and lower bound of the respective node must be included for isomorphism testing. For instance, when re-visiting g_8 , we only examine the graphs in $ub(g_8) \setminus \mathcal{A}(g_{10})$. $\square$

Recall that for each graph G , $ub(G)$ is already sufficiently small if θ is small. Removing the graph in the lower bound from the upper bound further shrinks the total number of graphs for subgraph isomorphism test, especially when $\mathcal{S}_i$ is large. Therefore, a significant amount of computation can be saved with this strategy.

3.5 Reducing Communication Overhead

Frequent subgraph mining with MapReduce produces a large set of key-value pairs which must be transferred via network communication. This may incur significant overhead, since keys are canonical labelings of graphs, which are sizable. We propose a lightweight compression technique that is able to represent the canonical labeling of a graph $G = (V, E, L, l)$ using only $O(|E| \log_w |L|)$ bits, whereas the computational complexity of encoding/decoding is only $O(|E|^2 \log_w |L|)$, where w is the size of a machine word.

3.5.1 Compression of Canonical Labeling

The canonical labeling of a graph corresponds to a unique permutation of its edges or vertices. For example, *CAM* (*Canonical Adjacency Matrix*) [43] computes a permutation of vertices such that the code produced from the adjacency matrix for that permutation is maximal, whereas the DFS code [100] finds a permutation of edges such that the ordering obtained is a maximal code. We adopt the CAM approach, but our technique can be extended to other canonical labeling methods.

For real-world graph datasets, trees are very common, and most of frequent subgraphs are trees [64, 70]. Therefore, our approach first focuses on spanning trees, and then extends to the case of non-tree edges.

Encoding Spanning Trees. Given graph $G = (V, E, L, l)$, denote its spanning tree by $T = (V, E', L, l)$, where $E' \subseteq E$. Let the permutation of vertices be $V = (v_0, v_1, \dots, v_{d-1})$, $d = |V|$, where v_0 is the root of T and each other vertex v_i is incident to one and only one vertex v_j for $j < i$ (i.e., its parent). To store T in an adjacency list, for each vertex v_i , we record its label $l(v_i)$ as well as the edge $\langle v_{e(i)}, l(v_i, v_{e(i)}) \rangle$ that connects v_i to its parent $e(i)$. We use a vector to store the column of vertex labels, denoted as $L_v = (l(v_0), l(v_1), \dots, l(v_{d-1}))$. Similarly, we denote $I_v = (e(1), e(2), \dots, e(d-1))$ as the set of parent nodes for vertices v_i ($i = 1, \dots, d-1$), and $L_e = (l(v_1, v_{e(1)}), l(v_2, v_{e(2)}), \dots, l(v_{d-1}, v_{e(d-1)}))$ as the corresponding vertex-parent edges (note that $e(0)$ is not defined).

Compressing tree T boils down to compressing the vectors L_v , I_v , and L_e . Note that, the elements in those vectors are often from a small domain. For instance, the elements in L_v and L_e are chosen from the set L of distinct labels in the dataset, and the elements of I_v are positive numbers smaller than the number of vertices in T . Furthermore, each vector is also small in size, where $|L_v| = d$ and $|L_e| = |I_v| = d - 1$. Based on these observations, we devise a lightweight compressing technique that can represent each vector as an integer. Let L_e^{max} be the maximum element in L_e . Using the numeration base $b = L_e^{max} + 1$, we represent L_e in an integer

$$\widehat{L_e} = \sum_{i=0}^{d-1} l(v_i, v_{e(i)}) \times b^i. \quad (3.4)$$

I_v^{max} , L_v^{max} , $\widehat{I_v}$ and $\widehat{L_v}$ are similarly defined. A labeling of T is uniquely determined by the values:

$$|V|, I_v^{max}, L_v^{max}, L_e^{max}, \widehat{I_v}, \widehat{L_v}, \widehat{L_e}.$$

To decompress $\widehat{L_e}$, we use the *base conversion algorithm* [53] to reconstruct L_e . Maintaining a dynamic vector $\mathcal{L}$, we divide $\widehat{L_e}$ by b with a remainder r and append r to the end of $\mathcal{L}$. The process repeats until $\widehat{L_e}$ is zero. If $|\mathcal{L}| < |L_e| = d - 1$, we append $d - 1 - |\mathcal{L}|$ zeros to $\mathcal{L}$. L_v and I_v are decoded similarly.

Extension to non-tree edges. Consider graph $G = (V, E, L, l)$ and one of its spanning trees $T = (V, E', L, l)$. A non-tree edge is denoted as a triple $\langle i, j, l(v_i, v_j) \rangle$, where $(v_i, v_j) \in E \setminus E'$ and $l(v_i, v_j)$ is the label on the edge. We order v_i and v_j in the triple such that $i > j$. Hence, the set of non-tree edges of G can be denoted as

$$\tilde{T} = \{ \langle i, j, l(v_i, v_j) \rangle \mid (v_i, v_j) \in E \setminus E' \wedge i > j \}.$$

To compress $\tilde{T}$, we extend the vectors of T to accommodate the non-tree edges. Specifically, we append all vertex labels of each element in $\tilde{T}$ to I_v . Similarly, we append each label in $\tilde{T}$ to L_e . Denote the newly obtained sets by $\widehat{I_v}$ and $\widehat{L_e}$, respectively. We have that $|L_e| = |E|$ and $|I_v| = 2|E| - |V| + 1$. Eq. (3.4) still holds, by replacing I_v and L_e with $\widehat{I_v}$ and $\widehat{L_e}$, respectively. Therefore, the graph G can be represented as

$$\mathcal{C}(G) = \{ |V|, |E|, I_v^{max}, L_v^{max}, L_e^{max}, \widehat{I_v}, \widehat{L_v}, \widehat{L_e} \}. \quad (3.5)$$

In case that the size of $\widehat{I_v}$, $\widehat{L_v}$, or $\widehat{L_e}$ is larger than the size of a machine word, we can use an array of machine words to store each value, where the size of each array is $\lceil \log_w \widehat{I_v} \rceil$, $\lceil \log_w \widehat{L_v} \rceil$, and $\lceil \log_w \widehat{L_e} \rceil$, respectively.

3.5.2 Theoretical Analysis

Correctness analysis. The correctness condition consists of two parts: (i) the compression result of canonical labeling must be unique, and (ii) the decompressing technique must correctly reconstruct the canonical labeling of a graph. The first part ensures that aggregating graphs in the filter phase is correct, whereas the second part guarantees the correctness in the refinement phase.

Lemma 3.1 *Given the canonical labeling of a graph G , $\mathcal{C}(G)$ in Eq. (3.5) is unique.*

Proof: Given the canonical labeling of G , it can be equivalently represented by three vectors I_v , L_v , and L_e . Since $\mathcal{C}(G)$ is the combination of the compression of I_v , L_v , and L_e , we are to prove that the compression of each vector is unique. Consider the vector I_v : we compress I_v as three values $|V|$, $b = I_v^{\max} + 1$, and $\widehat{I}_v$. By contradiction, assume that there exist two different vectors $I_v = (v_0, v_1, \dots, v_{d-1})$ and $I_u = (u_0, u_1, \dots, u_{d-1})$ with the same compressed image, that is, for some $i \in [0, d-1]$, we have $I_v[i] \neq I_u[i]$. According to Eq. (3.4), $\widehat{I}_v \neq \widehat{I}_u$, which contradicts the assumption that I_v and I_u have the same compressed image. Therefore, the canonical labeling of a graph G is unique. ■

Lemma 3.2 *Given $\mathcal{C}(G)$ of a graph G , the decompression of $\mathcal{C}(G)$ is the canonical labeling of G .*

Proof: Given $\mathcal{C}(G)$ in Eq. (3.5) after reconstruction, the sizes of I_v , L_v , and L_e are $|V|$, $2|E| - |V| + 1$, and $|E|$, respectively. Let us prove the case of one vector, the same proof can be applied to the other ones. Consider I_v : by contradiction, assume that the constructed vector I'_v is different from I_v . Since the base and size of I'_v and I_v are the same, there exists an $i \in [0, d-1]$ such that $I'_v[i] \neq I_v[i]$. According to Eq. (3.4), $\widehat{I}'_v \neq \widehat{I}_v$, which contradicts that I'_v is reconstructed from $\widehat{I}_v$. Consequently, the decompression of $\mathcal{C}(G)$ is the canonical labeling of G . ■

Complexity analysis. Assume that each element in I_v , L_v , and L_e is stored in a machine word. Therefore, the total size of these three vectors is

$$(2|E| - |V| + 1 + |V| + |E|) \times w = (3|E| + 1)w.$$

Now, consider the size of the compressed image: according to Eq. (3.4), we have

$$\begin{aligned} \widehat{I}_v &\leq (I_v^{\max} + 1)^{|V|}, \\ \widehat{L}_v &\leq (L_v^{\max} + 1)^{2|E| - |V| + 1}, \\ \text{and } \widehat{L}_e &\leq (L_e^{\max} + 1)^{|E|}. \end{aligned}$$

Let $b = \max\{I_v^{max} + 1, L_v^{max} + 1, L_e^{max} + 1\}$. The labels in L can be mapped to a consecutive range $[0, |L|)$, hence we have $b = O(|L|)$. As such, the total size of the compressed image is at most

$$(|V| + 2|E| - |V| + 1 + |E|) \times \lceil \log_w b \rceil = (3|E| + 1) \lceil \log_w b \rceil = O(|E| \log_w |L|).$$

The compression ratio is at least $w / \lceil \log_w b \rceil$.

As to the computational complexity of compression, let us first consider the computation of L_e . Since the size of $\widehat{L_e}$ is at most $|E| \log_w |L|$, and there are $|E|$ elements in L_e , the complexity of computing $\widehat{L_e}$ is $O(|E|^2 \log_w |L|)$. A similar analysis applies to I_v and L_v , whose upper-bound is also $(|E|^2 \log_w |L|)$. That is, the complexity of compression is $O(|E|^2 \log_w |L|)$. Similarly, the complexity of decompression is also $O(|E|^2 \log_w |L|)$.

3.6 Experimental Evaluation

We evaluate experimentally the proposed algorithm *MRFSM* (MapReduce Frequent Subgraph Mining) against two baselines on several real-world large graph datasets. In Section 3.6.1 we describe the experimental setup, followed by an analysis of how to tune the value of probability threshold ρ in Section 3.6.2. Section 3.6.3 presents the head-to-head comparison with competitor techniques.

3.6.1 Experimental Setup

We deploy a *MRFSM* prototype in Amazon EC2 [2] using up to 121 large instances. Each instance has 2 CPUs and 7.5GB RAM and runs Hadoop (version 0.20.203). One instance is set up as the master node and the others as worker nodes. We used the default configuration of Hadoop, i.e., *dfs.replication* = 3 and *fs.block.size* = 64MB.

Implementation. We use a public C++ implementation of *GASTON* [70] to mine the locally frequent subgraphs in the filter round (we emphasize that our framework can be used in conjunction with any other frequent subgraph mining algorithm, not only *GASTON*). We implement *MRFSM* using *Hadoop Streaming API* [9], which places no restrictions on the programming language.

As competitors, we consider the *Iterative Frequent Subgraph Mining (IFSM)* [40] algorithm, with source code made available by the authors. We also adapt the partitioning-based method from [69] (*PGM*) to run on MapReduce, whose filter round is identical to the preliminary approach in Section 3.3.1. Note that, since MapReduce is a share-nothing architecture, the sequential data partition method in *PGM* is performed on a single machine, and the result is fed to the map phase of the filter round.

Table 3.1: Table of datasets.

dataset	number of graphs	disk size	$ E $	$ V $
<i>Pubchem</i> [11]	46,703,496	41.8 GB	52.32	50.37
<i>Akos</i> [1]	15,720,753	6.6 GB	24.00	22.52
<i>ChemDB</i> [4]	7,100,106	6.7 GB	53.23	50.77
<i>Ambinter</i> [3]	6,551,088	3.4 GB	29.40	27.07
<i>Enamine</i> [5]	1,378,907	0.7 GB	27.86	25.78
<i>NCI</i> [10]	265,242	0.2 GB	41.76	40.47

Table 3.2: Experimental parameter values.

parameter	values
θ	0.025, 0.05, 0.1 , 0.2, 0.4
m	80, 90, 100 , 110, 120
ρ	0, 0.00625, 0.0125, 0.05 , 0.1, 0.2, 0.4, 0.8, 1

Datasets. We use seven real-world datasets listed in Table 3.1. We use as default dataset *Pubchem*, the largest dataset which consists of more than 46 million graphs. The average number of edges and vertices are 52.32 and 50.37, respectively. Note that, none of these datasets (except *NCI*) can be handled by a single machine with up to 12GB RAM. For each dataset, given the set L of all its labels, we assign each label $l \in L$ a value $i \in [0, |L|)$, such that (i) each label l is injectively mapped to a value in the range $[0, |L|)$, and (ii) the more frequent label gets the smaller value.

Methodology. We perform several types of measurements: first, we record the total running time (wall clock) of each algorithm, i.e., the time elapsed from starting the MapReduce program until the program returns the result. Second, we report the running time of the filter and refinement rounds independently. Third, we report the communication cost of transferring data in MapReduce, as well as the number of candidate graphs generated in the filter round. We run each experiment three times, and report the average reading.

By default, we adopt the Bottom-Up approach in the refinement round. The parameter values used are shown in Table 3.2, with default values in bold.

3.6.2 Tuning Probability Upper-Bound ρ

Recall from Section 3.3.2 that the probability ρ has an important role in pruning the search space. We show next how to tune the value of ρ in practice. Figure 3.5 shows performance results when varying ρ from 0 to 1. When $\rho = 0$, each worker outputs all

(a) Total running time.

(b) Number of candidates.

(c) Filter round.

(d) Refinement round.

 Figure 3.5: Vary probability threshold ρ .

the subgraphs that are visited in the filter round, whereas when $\rho = 1$, *MRFSM* only outputs the locally frequent subgraphs.

Figure 3.5(a) shows the total running time of *MRFSM*. When ρ grows from 0 to 0.05, the total running time decreases, as the total number of subgraphs that are generated by each worker decreases, and the overhead of the filter round is lower (Figure 3.5(c)). On the other hand, when ρ increases further towards 1, the total running time increases, because the number of candidates increases (Figure 3.5(b)), and so does the overhead in the refinement round (as shown in Figure 3.5(d)). A clear trade-off can be observed between the costs of the filter and refinement rounds: for smaller ρ , the number of subgraphs that are generated in each worker is larger, but the number of candidates sent to the refinement round is smaller. A good setting for the probability threshold is $\rho = 0.05$, when the overhead is minimized. In the rest of the experiments, we set $\rho = 0.05$.

(a) Total running time.

(b) Communication cost.

(c) Filter round.

(d) Refinement round.

Figure 3.6: Vary support threshold θ .

3.6.3 Comparison with competitor techniques

We compare *MRFSM* against *PGM* and *IFSM*. First, we vary support threshold θ from 0.025 to 0.4. Figures 3.6(a)(b) show that *IFSM* does not terminate within reasonable time (one day) when $\theta \leq 0.05$. Furthermore, *PGM* does not terminate when $\theta = 0.025$, owing to the significant computational overhead in the refinement round (Figure 3.6(d)). Even though the filter time of *MRFSM* is slightly higher than that of *PGM*, due to producing locally infrequent subgraphs, this cost is offset by the significant improvement in the refinement round. *MRFSM* significantly outperforms the other algorithms in terms of total running time. Due to its poor performance, we no longer consider *IFSM* in the rest of the section.

Next, we evaluate the impact of number of available workers, and we vary m from 80 to 120. Figure 3.7(a) demonstrates that *MRFSM* clearly outperforms *PGM* in terms of total running time for all settings. Furthermore, communication cost increases only slightly as m grows.

(a) Total running time.

(b) Communication cost.

(c) Filter round.

(d) Refinement round.

Figure 3.7: Vary number of workers m .

We also evaluate the performance of the graph compression technique proposed in Section 3.5. We denote the *MRFSM* approach that uses compression as *MRFSM-C*, and the one without compression as *MRFSM-NC*. Figure 3.8(a) shows that the total running time is improved by compression. Figure 3.8(b) shows the size of data communicated among workers. *MRFSM-C* has much smaller cost in communication, which demonstrates the effectiveness of our compression technique. As expected, *MRFSM-NC* has slightly higher communication cost than *PGM*, since in the filter round, *MRFSM-NC* also outputs the locally infrequent subgraphs on each machine, while *PGM* only outputs the locally frequent subgraphs. Figure 3.8(c) shows the breakdown of the filter running time into communication time and CPU time. *MRFSM-C* has slightly higher CPU cost than *MRFSM-NC* because of encoding/decoding, but this is offset by the significant gain due to the reduction in communication time. Figure 3.8(d) compares the running time for the refinement round: *MRFSM-C* is again slightly worse than *MRFSM-NC*, owing to the overhead of decompression.

(a) Total running time.

(b) Communication cost.

(c) Filter round.

(d) Refinement round.

Figure 3.8: Impact of compression.

Next, we evaluate the Top-Down and Bottom-Up traversal strategies employed in the refinement round. Figure 3.9(a) presents the running time of Top-Down and Bottom-Up with support threshold θ varying from 0.025 to 0.4. Top-Down is slower than Bottom-Up, since the number of *subgraph isomorphism tests* is larger, as shown in Figure 3.9(b). However, Bottom-Up needs to maintain all candidates in memory while Top-Down releases the memory once the candidates are no longer useful. As shown in Figure 3.9(c), Bottom-Up is more memory-efficient than Top-Down. Therefore, if there is a sufficiently large amount of memory, Bottom-Up would be more suitable than Top-Down.

Finally, we present the total running time results on all considered real-world datasets in Table 3.3. *MRFSM* outperforms competitors by more than two times on all datasets (improved ratio compared to closest competitor is shown in the last column). In some cases, the improvement is close to six-fold.

Figure 3.9: Strategy comparison in refinement round.

3.7 Summary

We propose a two-step filter-and-refinement MapReduce framework for frequent subgraph mining in the transaction graph setting. Our method scales well for large datasets, and significantly outperforms competitor techniques. The gain in performance is achieved through careful selection of frequent subgraph candidates, effective strategies for avoiding redundant computation in the refinement step, and a lightweight compression scheme that reduces significant communication cost with low computational overhead of encoding/decoding.

Chapter 4

Network Motif Discovery

This chapter presents the first study on GPU-based algorithms for network motif discovery. We analyze the deficiency of the existing CPU-based methods, and pinpoint the reasons that they cannot be translated into efficient algorithms on GPUs. Based on our analysis, we propose a novel solution that exploits the strengths of GPUs in terms of parallelism and mitigates their limitations in terms of the computation power per GPU core. Then, we develop three optimization techniques that improve the scalability of our solution, avoid under-utilization of the GPU, and eliminate redundant computation in network motif discovery. Together, those optimizations reduce the computation cost of our solution by 75%, and enable our solution to handle graphs that are ten times larger than those studied in previous work. Finally, we empirically compare our solution against the state-of-the-art CPU-based methods, using the largest datasets ever tested in the literature of network motif discovery. We show that, even with a low-end GPU, our solution runs 10 times faster than the best CPU-based method, and this performance gap would further widen by 10-fold when a high-end GPU is used. Furthermore, our solution is around 20 times more cost-effective than the best CPU-based method, when taking into account the monetary costs of the CPU and GPUs used.

The subsequent of this chapter is organized as follows. We first define the problem and introduce GPU architecture in Section 4.1. Then we discuss the difficulties in translating the CPU-based approaches on GPU in Section 4.2, which paves the way for introducing the GPU-based approach. We present an overview of the solution in Section 4.3, and then illustrate our GPU algorithm in Section 4.4. We develop several optimization techniques in Section 4.5 that significantly improve the performance of the GPU algorithm. Section 4.6 evaluates the performance of our GPU-based solution. Finally, we summarize this chapter in Section 4.7.

Figure 4.1: A real graph G , two subgraphs g_1 and g_2 of G , and two random graphs G_1 and G_2 degree-equivalent to G .

4.1 Preliminaries

This section first defines several basic concepts and formalizes the network motif discovery problem, and then introduces the architecture of Graphics Processing Units (GPUs).

4.1.1 Problem Definition

Let $G = (V, E)$ be a directed, unlabelled graph¹ with a set V of vertices and a set E of edges. For any two vertices u, v in V , we say that v is an *out-neighbor* of u if there is a directed edge from u to v , i.e., $(u, v) \in E$. Conversely, we refer to u as an *in-neighbor* of v . We define the *in-degree* (resp. *out-degree*) of u as the number of in-neighbors (resp. out-neighbors) of u . In addition, we define the *bi-degree* of u as the number of vertices that are both in-neighbors and out-neighbors of u , and we refer to those vertices as the *bi-neighbors* of u .

Let $g = (V_g, E_g)$ be a connected graph. We say that g is a *subgraph* of G (denoted by $g \subseteq G$), if and only if there exists at least one injective function $\zeta : V_g \rightarrow V$, such that (i) for any vertex $v \in V_g$, we have $\zeta(v) \in V$, (ii) for any edge $(u, v) \in E_g$, there is an edge $(\zeta(u), \zeta(v)) \in E$. For each subgraph of G that is isomorphic to g , we refer to it as an *occurrence* of g in G . The *frequency* of g in G , denoted by $f(g, G)$, is the total number of occurrences of g in G . g is a *size- k* subgraph, if it contains exactly k vertices.

Example 4.1: Consider the graphs G , g_1 , and g_2 in Figures 4.1a, 4.1b, 4.1c, respectively. Observe that g_1 is a subgraph of G . To explain, consider an injective function ζ_1 that maps each vertex in g_1 to a vertex in G , such that $\zeta_1(u_1) = v_1$, $\zeta_1(u_2) = v_2$, and $\zeta_1(u_3) = v_3$. Note that for any two vertices u_a and u_b in g_1 , if g_1 contains a directed edge from u_a to u_b , then G has a directed edge from $\zeta_1(u_a)$ to $\zeta_1(u_b)$. Therefore, $g_1 \subseteq G$. Furthermore, it

¹We consider that G is directed and unlabelled, as it is a standard assumption in the literature. Nevertheless, our solution can be easily extended to handle undirected or labelled graphs.

can be verified that there is only one occurrence of g in G . Hence, $f(g_1, G) = 1$, i.e., the frequency of g_1 in G equals 1. Similar, we have $g_2 \subseteq G$, and $f(g_2, G) = 5$. $\square$

We say that a graph $G' = (V', E')$ is *degree-equivalent* to G , if and only if (i) $|V'| = |V|$ and $|E'| = |E|$, and (ii) there exists a bijection $\psi : V \rightarrow V'$, such that for any node $v \in V$, v and $\psi(v)$ have the same in-degree, out-degree, and bi-degree.

Let $\mathcal{G}$ denote the set of all graphs that are degree-equivalent to G . For any subgraph g of G , its *expected frequency* $\bar{f}(g)$ is defined as its average frequency in all graphs in $\mathcal{G}$, i.e.,

$$\bar{f}(g) = \frac{1}{|\mathcal{G}|} \sum_{G' \in \mathcal{G}} f(g, G'). \quad (4.1)$$

Note that the exact value of $\bar{f}(g)$ is difficult to compute due to the enormous size of $\mathcal{G}$. Following the standard practice in the literature [68], we estimate $\bar{f}(g)$ using a sample set of $\mathcal{G}$ with r graphs, denoted as $\mathcal{G}_r$. The estimation thus obtained is

$$\tilde{f}(g) = \frac{1}{r} \sum_{G' \in \mathcal{G}_r} f(g, G'), \quad (4.2)$$

and the corresponding sample standard deviation is

$$\tilde{\sigma}(g) = \sqrt{\frac{1}{r-1} \sum_{G' \in \mathcal{G}_r} \left(f(g, G') - \tilde{f}(g) \right)^2}. \quad (4.3)$$

Definition 4.1 (Motif) Let $\theta > 0$ be a user-defined threshold. Given G , $\mathcal{G}_r$, and θ , a subgraph g of G is a **motif** of G , if and only if $\tilde{\sigma}(g) > 0$ and

$$\frac{f(g, G) - \tilde{f}(g)}{\tilde{\sigma}(g)} \geq \theta. \quad (4.4)$$

In other words, a motif of G is a subgraph of G that appears more frequently in G than in random graphs that are degree-equivalent to G . We illustrate Definition 4.1 with an example.

Example 4.2: Figures 4.1d and 4.1e illustrate two random graphs that are degree-equivalent to the graph G in Figure 4.1a. Let $\mathcal{G}_2 = \{G_1, G_2\}$. For the subgraph g_1 in Figure 4.1b, we have $f(g_1, G_1) = 1$ and $f(g_1, G_2) = 0$. Therefore, the estimated frequency of g_1 (with respect to $\mathcal{G}_2$) is $\tilde{f}(g_1) = 1/2$, and the sample standard deviation is $\tilde{\sigma}(g_1) = \sqrt{2}/2$.

Assume that $\theta = 0.1$. Given that $f(g_1, G) = 1$, we have

$$\frac{f(g_1, G) - \tilde{f}(g_1)}{\tilde{\sigma}(g_1)} = \frac{\sqrt{2}}{2} > 0.1 = \theta.$$

Therefore, g_1 is a motif of G . On the other hand, g_2 is not a motif of G , since $f(g_2, G) = f(g_2, G_1) = f(g_2, G_2) = 5$, which leads to $\tilde{\sigma}(g_2) = 0$. $\square$

Problem Statement. Given G , $r > 0$, $k > 2$, and $\theta > 0$, the problem of network motif discovery asks for all size- k motifs of G , with respect to $\mathcal{G}_r$ (i.e., a sample set of $\mathcal{G}$ with r random graphs).

4.1.2 Graphics Processing Units

GPUs were initially designed for graphical processing, but are now widely used for general-purpose parallel computing, e.g., sorting [84] and data mining [30]. Figure 4.2 shows the general architecture of a GPU. Compared with a CPU (which usually contains only a few cores), a GPU can easily have thousands of computation units. Specifically, a GPU contains several multiprocessors (MPs), each of which has a large number of stream processors (SPs). The SPs in each multiprocessor work in single-instruction multiple-data (SIMD) manner, i.e., they execute the same instructions at the same time on different input data. Each MP has a small but fast memory that is shared by all of its SPs. In addition, all SPs in the GPU share accesses to a large but slower *global memory* of the GPU. Data can be exchanged between GPU's global memory and the main memory via a high speed I/O bus (e.g., PCI-Express), albeit at a relatively slow rate.

For parallel computing on GPUs, we adopt the Nvidia CUDA programming framework. In the following, we introduce several key concepts in CUDA, so as to facilitate our discussions in the subsequent sections.

Kernels. A CUDA program alternates between codes running on the CPU and those on the GPU. The latter are referred to as *kernels*, and they are invoked only by the CPU. Each kernel starts by transferring input data from the main memory to the GPU's global memory, and then processes the data on the GPU; after that, it copies the results from the GPU's global memory back to the main memory, and then terminates.

Thread Hierarchy. The GPU executes each kernel with a user-specified number of threads. The threads are divided into a number of *blocks*, each of which is assigned to one MP (and cannot be re-assigned at runtime). In turn, each MP divides an assigned block of threads into smaller *warps*, and executes each warp of threads concurrently. Note that threads in the GPU cannot communicate with each other directly, but are allowed to retrieve data from, or write data to, arbitrary locations of the GPU's global memory.

Branch Divergences. Due to the SIMD nature of the GPU's SPs, all threads on the GPU cannot execute different programs at a given time. As a consequence, if two threads in a warp have different execution paths, then the GPU would execute those paths sequentially. For example, suppose that a piece of GPU code contains a statement “if A

Figure 4.2: The general architecture of a GPU.

then B , else C ". For this statement, the GPU first asks each thread in a warp to evaluate condition A . Then, if A equals *true* in some threads, the GPU executes B on those threads; Meanwhile, the remaining threads in the warp remain idle, and they execute C only after all other threads finish performing B . Such *branch divergence* is detrimental to the efficiency of GPU programs and should be avoided whenever possible [35].

Memory Coalescing. Suppose that the threads in a warp request to access data in the GPU's global memory, and the set of data requested is stored in consecutive memory addresses. In that case, the MP responsible for the warp would retrieve all data with one memory access and then distribute them to each thread, instead of issuing one access for each thread individually. This is referred to as *memory coalescing*, and it helps reduce memory access overheads. In contrast, if the data to be accessed is stored in k disjoint memory spaces, then k random accesses are required. Therefore, it is important that we carefully arrange data in the GPU's global memory, so that the data required by each warp resides in consecutive locations.

4.2 Difficulties in GPU Translations

Recall that we have reviewed the techniques [51, 54, 61, 74, 76, 93, 95] for network motif discovery in Chapter 2. All of these techniques are CPU-based approaches. However, the CPU-based methods reviewed are difficult to be adopted on GPUs, for three reasons. First, both phases of the existing methods require computing the canonical labelings of numerous subgraphs. The algorithms [43, 65, 100] for computing canonical labelings, however, contain complicated execution paths with a large number of branches. As a consequence, if we are to directly adopt those algorithms on GPUs, the execution of the algorithms would be highly inefficient due to the effects of branch divergences (see Section 4.1.2).

Second, CPU-based methods rely on the CL-Index and AM-Index to check whether a subgraph appears in S_k . If we adopt the same approach on a GPU, then we need to store

the CL-Index and AM-Index in the GPU’s global memory, and ask each GPU thread to probe the indices for subgraph matching. In that case, the GPU threads in each warp are likely to access drastically different memory, which prevents the GPU from applying memory coalescing to reduce memory access costs. Furthermore, when G is large, the CL-Index and AM-Index can become so large that they do not even fit in the global memory of the GPU.

Finally, if we are to ask each GPU thread to examine whether a subgraph appears in S_k , then it is likely that some threads will incur considerably higher overheads than the others, since the subgraphs in random graphs may have much different structures. Therefore, there can be significant imbalance in the GPU threads’ workload. In that case, all GPU threads in the same warp would need to wait for the slowest thread to finish, before they can be terminated to allow new GPU threads to be created. This leads to severe under-utilization of the GPU’s parallel processing power.

4.3 Solution Overview

As with existing CPU-based methods, our solution also consists of a subgraph enumeration phase and a frequency estimation phase. In particular, the subgraph enumeration phase of our solution adopts the CPU-based method in [76], and the frequency estimation phase also utilizes the CPU-based switching algorithm [67] to generate the set $\mathcal{G}_r$ of random graphs that are degree-equivalent to G . To compute the average frequency of each subgraph $g \in S_k$ with respect to $\mathcal{G}_r$, however, we employ a GPU-based algorithm that provides much higher efficiency than existing CPU-based methods. The reason that we focus on optimizing the computation of average subgraph frequency is that it incurs significantly higher overheads than the other components of our solution (due to the large number of random graphs to be processed). In the following, we present an overview of our GPU-based algorithm, assuming that G , S_k , and $\mathcal{G}_r$ are given.

In a nutshell, our algorithm examines each pair of g and G' where $g \in S_k$ and $G' \in \mathcal{G}_r$, and it computes the frequency of g in G' with the *FreqComp* method in Algorithm 4.1. The algorithm first arranges the vertices in g in a certain sequence $\langle u_1, u_2, \dots, u_k \rangle$ (Line 1), such that for any $i \in [2, k]$, $u_1, u_2, \dots, u_i$ induces a connected subgraph of g (denoted as $g(i)$). We refer to $\langle u_1, u_2, \dots, u_k \rangle$ as a *matching order*, and we clarify how it is derived in Section 4.4.2. After that, it identifies all subgraphs of G' that are isomorphic to $g(2)$, and stores them in a set C_2 (Lines 2-3).

The subsequent part of the algorithm runs in $k - 2$ iterations (Lines 4-6), such that each iteration utilizes the GPU to transform C_{i-1} ($i \in [3, k]$) into C_i , i.e., the set of all subgraphs of G' that are isomorphic to $g(i)$. Once C_k is computed, the algorithm terminates and returns $|C_k|$, which equals the frequency of g in G' (Line 7). We clarify the generation of C_i in Section 4.4.2.

Algorithm 4.1: FreqComp**input** : $g \in S_k$ and $G' \in \mathcal{G}_r$ **output:** $f(g, G')$

- 1 [CPU]: Choose a matching order of the vertices in g , denoted as $\langle u_1, u_2, \dots, u_k \rangle$ (see Section 4.4.2);
- 2 Let $g(2)$ be the subgraph of g induced by $\{u_1, u_2\}$;
- 3 [CPU]: Identify the set C_2 of all subgraphs in G' that are isomorphic to g_2 ;
- 4 **for** $i = 3, \dots, k$ **do**
- 5 Let $g(i)$ be the subgraph of g induced by $\{u_1, u_2, \dots, u_i\}$;
- 6 [GPU]: Based on C_{i-1} , compute the set C_i of all subgraphs in G that are isomorphic to $g(i)$ (see Algorithm 4.2);
- 7 **return** $|C_k|$;

Compared with the existing CPU-based methods [51, 54, 61, 74, 76, 93, 95], *FreqComp* does not compute any canonical labeling or construct any index on S_k , which helps avoid the branch divergence and memory coalescing issues that render existing methods inefficient on GPUs. Instead, *FreqComp* adopts an incremental approach that first identifies the subgraphs of G' that can be matched to parts of g (i.e., $g(2), g(3), \dots, g(k-1)$), and then utilizes such “partial occurrences” of g to pinpoint the size- k subgraphs of G' that are isomorphic to g . This incremental approach is not as efficient as the CPU-based indexing methods in deciding whether a *single* subgraph of G' is isomorphic to g , but it is much more amenable to GPU parallelization.

It is noteworthy that *FreqComp* is similar in spirit to existing CPU-based algorithms [36, 80, 90, 100] for *subgraph isomorphism tests*, which incrementally match the vertices in a small graph g to those in a larger graph G' to decide whether g appears in G' . However, *FreqComp* aims to decide the exact number of occurrences of g in G' , whereas the algorithms in [80, 90, 100] only determine whether g has at least one occurrence in G' . Furthermore, as the algorithms in [36, 80, 90, 100] are CPU-based, they involve complex execution paths, which render them unsuitable for GPU adoptions, due to the effect of branch divergences. In contrast, *FreqComp* is devised with careful considerations of GPUs’ characteristics and limitations, which lead to design choices that drastically differ from those in [36, 80, 90, 100], as we demonstrate in Section 4.4.

4.4 GPU-Based Subgraph Matching

This section presents the details of the *FreqComp* algorithm. Section 4.4.1 clarifies how we represent each random graph G' in a GPU’s global memory. Sections 4.4.2 and 4.4.3 elaborates each step of *FreqComp*. Section 4.4.4 proves *FreqComp*’s correctness.

 Figure 4.3: Edge and offset arrays of G_1 in Figure 4.1.

4.4.1 Representation of Graphs

We represent each random graph $G' \in \mathcal{G}_r$ using six arrays in the GPU's global memory, namely, E_{out} , E_{in} , E_{bi} , O_{out} , O_{in} , and O_{bi} . Each element in E_{out} (resp. E_{in}) is an ordered pair of vertices $\langle v_a, v_b \rangle$, such that v_b is an out-neighbor (resp. in-neighbor) of v_a . Meanwhile, each element in E_{bi} is an ordered pair of vertices that are bi-neighbors of each other. All pairs in E_{out} , E_{in} , and E_{bi} are sorted by their first vertices, with ties broken based on the second vertices. As a consequence, the pairs with the same first vertices are stored as a *block* of consecutive elements in E_{out} , E_{in} , and E_{bi} . We refer to E_{out} , E_{in} , and E_{bi} as the *edge arrays*.

On the other hand, O_{out} is an array that maps each vertex in G' to its corresponding block in E_{out} . Specifically, for the i -th vertex v_i in G' , if it has at least one out-neighbor, then the i -th element in O_{out} records the position of the first element in E_{out} where v_i is the first vertex. If v_i has no out-neighbor, however, then the i -th element in O_{out} is identical to the $(i + 1)$ -th element. For convenience, we append an extra element to the end of O_{out} , and set its value to the total number of elements in E_{out} plus one. We refer to O_{out} as the *offset array* for E_{out} . Accordingly, O_{in} and O_{bi} are the offset arrays for E_{in} and E_{bi} , respectively, and are defined in the same manner. As an example, Figure 4.3 illustrates the edge and offset arrays for the random graph G_1 in Figure 4.1.

By the way O_{out} is constructed, if we are to identify the out-degree of the i -th vertex in G' , then we can simply subtract the i -th element in O_{out} from the $(i + 1)$ -th element. The in-degree (resp. bi-degree) of any vertex can be computed from O_{in} (resp. O_{bi}) in the same manner.

4.4.2 Construction of C_i

As mentioned in Section 4.3, the *FreqComp* algorithm first determines a matching order $\langle u_1, u_2, \dots, u_k \rangle$ for the vertices in g , and then iteratively identifies the set C_i ($i \in [2, k]$) of subgraphs in G' that are isomorphic to $g(i)$, i.e., the subgraph of g induced by

$\{u_1, u_2, \dots, u_i\}$. For ease of exposition, we defer the discussion of the matching order to end of Section 4.4.2. In what follows, we first clarify the construction of C_i , assuming that $\langle u_1, u_2, \dots, u_k \rangle$ are given.

Construction of C_2 . First, consider the case when $i = 2$. If u_1 and u_2 are bi-neighbors, then C_2 consists of all 2-cycles in G' ; otherwise, C_2 contains all forward (resp. backward) edges in G' if u_2 is an out-neighbor (resp. in-neighbor) of u_1 . In any of those three cases, C_2 can be constructed with a linear scan of E_{in} , E_{out} , or E_{bi} .

For each graph $c \in C_2$, we record the two vertices of the graph as an ordered pair, where the first and second vertices are mapped to u_1 and u_2 , respectively, in the isomorphism of c and $g(2)$. In case that there exist multiple isomorphisms (i.e., when u_1 and u_2 are bi-neighbors), we store in C_2 one ordered pair of each mapping. In general, for graph $c \in C_i$ and each isomorphism of c and $g(i)$, we store a sequence of i vertices in C_i , such that the j -th ($j \in [1, i]$) vertex in the sequence is the vertex in c mapped to u_j in the isomorphism. (We discuss in Section 4.4.3 how we may reduce the number of sequences in C_i without affecting the correctness of our solution.) For convenience, we refer to each sequence s in C_i as a *size- i candidate*, and we abuse notation by using s to refer to the graph that it represents.

Algorithm 4.2: BuildCi

input : g , G' , and C_{i-1}

output: C_i

- 1 $\{A_o, A_v\} = \text{InitArray}(g, C_{i-1});$ // see Algorithm 4.3
 - 2 $\{I, A'_o\} = \text{GenCand}(A_o, A_v, g, G', C_{i-1});$ // see Algorithm 4.4
 - 3 $C_i = \text{CleanCand}(I, A'_o);$ // see Algorithm 4.5
 - 4 **return** $C_i;$
-

Construction of $C_3, C_4, \dots, C_k$. Next, suppose that we have constructed C_{i-1} ($i \in [3, k]$), based on which we are to compute C_i by launching a kernel on the GPU. Our basic idea is to invoke a large number of parallel GPU threads, such that each thread (i) examines a size- $(i-1)$ candidate $c \in C_{i-1}$ and (ii) tries to transform c into a size- i candidate c^* by adding one vertex in G' into c .

To explain, recall that c is a size- $(i-1)$ subgraph of G' that is isomorphic to $g(i-1)$. Let v_j ($j \in [1, i-1]$) be the vertex in c that is mapped to u_j in $g(i-1)$. To convert c into a graph isomorphic to $g(i)$, a natural approach is to inspect each neighbor v of each v_j to see if v can be mapped to u_i . That is, we check whether the following *vertex validity condition* holds:

- *Vertex Validity Condition:* For all $j \in [1, i-1]$, if u_j is an in-neighbor of u_i in g , then v_j is an in-neighbor of v_i ; furthermore, if u_j is an out-neighbor of u_i in g , then v_j is an out-neighbor of v_i .

If the above condition holds for v , the subgraph of G' induced by $\{v_1, \dots, v_{i-1}, v\}$ must be isomorphic to $g(i)$; accordingly, we can record the sequence $\langle v_1, \dots, v_{i-1}, v \rangle$ as a size- i candidate in C_i .

To implement the above approach on a GPU, a straightforward method is to create one GPU thread for each neighbor v of a vertex in $c \in C_{i-1}$, to check whether the vertex validity condition holds for v . This method, however, requires different threads in the same warp to access the neighbors of different vertices v , which diminishes the chance of memory coalescing since the edges of different vertices are unlikely to reside in consecutive memory addresses (see Figure 4.3). Furthermore, the method also leads to workload unbalance among the threads, because different vertices v may have drastically different numbers of neighbors.

Algorithm 4.3: InitArray

input : g and C_{i-1}
output: A_o and A_v

- 1 create arrays A_{deg} and A_v , both of size $|C_{i-1}|$;
- 2 **for** each $x = 1, 2, \dots, |C_{i-1}|$ **in parallel do**
- 3 let $c_x = \langle v_1, \dots, v_{i-1} \rangle$ be the x -th graph in C_{i-1} ;
- 4 identify the vertex v_α in c_x with the smallest non-zero relevant degree;
- 5 $A_{deg}[x] \leftarrow$ the relevant degree of v_α ;
- 6 $A_v[x] \leftarrow v_\alpha$;
- 7 run a parallel prefix sum on A_{deg} ; let A_o be the resulting array;
- 8 **return** A_v and A_o ;

We address the above deficiencies with a more advanced method as follows. Without loss of generality, assume that u_i is an out-neighbor of a vertex u_α ($\alpha \in [1, i-1]$) in g . Then, for each vertex v in G' that is an out-neighbor of v_α , we create $|E(v)|$ GPU threads, where $E(v)$ is the set of all edges incident to v in G' . (We refer to v_α as the *anchor vertex*.) In particular, the ℓ -th thread examines the ℓ -th edge $e' \in E(v)$, and checks whether the following *edge validity conditions* hold simultaneously:

- (i) e' connects v to some v_j ($j \in [1, i-1]$) in c .
- (ii) If e' is an outgoing edge from v , then u_j is an out-neighbor of u .
- (iii) If e' is an incoming edge to v , then u_j is an in-neighbor of u .

The thread returns *true* if all of the above conditions hold, and *false* otherwise. After all $|E(v)|$ threads terminate, we count the number of threads that return *true*. If this number equals the number of edges incident to u_i , then we confirm that v satisfies the validity condition. In that case, we insert $\langle v_1, \dots, v_{i-1}, v \rangle$ into C_i as a size- i candidate.

Compared with the straightforward method, the advanced method increases the total workload on GPU, since the latter examines all $|E(v)|$ edges of each v , whereas the

former only needs to perform $|E(u_i)|$ binary searches on v 's edge lists. As a trade-off, however, the advanced approach has a much smaller running time for two reasons. First, it ensures a balanced workload for each GPU thread. Second, it facilitates memory coalescing, because (i) the threads in the same warp are likely to handle the neighbors of the same v , and (ii) the edges of v are stored in consecutive addresses.

It remains to discuss how we select the anchor vertex v_α ($\alpha \in [1, i-1]$) to start the exploration of candidate vertices v . For each $j \in [1, i-1]$, we define v_j 's *relevant neighbor set* as:

$$R(v_j) = \begin{cases} v_j\text{'s out-neighbor set,} & \text{if } u_i \text{ is } u_j\text{'s out-neighbor;} \\ v_j\text{'s in-neighbor set,} & \text{if } u_i \text{ is } u_j\text{'s in-neighbor;} \\ v_j\text{'s bi-neighbor set,} & \text{if } u_i \text{ is } u_j\text{'s bi-neighbor;} \\ \emptyset, & \text{otherwise.} \end{cases}$$

We also define $|R(v_j)|$ as the *relevant degree* of v_j . Observe that (i) we can set $v_\alpha = v_j$ only if $R(v_j) \neq \emptyset$, and (ii) if $v_\alpha = v_j$, then the advanced method needs to explore $|R(v_j)|$ of v_j 's relevant neighbors. To minimize the number of vertices that need to be explored, we set v_α to the vertex in c with the smallest *non-zero* relevant degree.

Implementation. Algorithm 4.2 shows the pseudo-code of our method (dubbed *BuildCi*) for constructing C_i from C_{i-1} ($i \in [3, k]$). The algorithm first invokes the *InitArray* function (Algorithm 4.3) to create two arrays A_v and A_o , which are utilized in the subsequent step by parallel GPU threads. In particular, A_v stores $|C_{i-1}|$ vertices, such that the x -th vertex is the anchor vertex in the x -th graph in C_{i-1} . Meanwhile, A_o records $|C_{i-1}|$ *offset* values, such that the x -th offset equals the sum of the relevant degrees of first x vertices in A_v . These offsets are used to indicate the memory locations where the GPU threads in the subsequent step should write their outputs to.

Algorithm 4.4: GenCand

input : A_o, A_v, g, G' , and C_{i-1}
output: I and A'_o

- 1 let θ be the last element of A_o ;
- 2 create arrays A'_{deg} and I , both of the size θ ;
- 3 **for** each $y = 1, 2, \dots, \theta$ **in parallel do**
- 4 identify the integer x such that $A_o[x] \leq y < A_o[x+1]$;
- 5 let c_x be the x -th graph in C_{i-1} ;
- 6 let v_α be the vertex recorded in $A_v[x]$;
- 7 let $z = y - A_o[x]$;
- 8 $v \leftarrow$ the z -th vertex in v_α 's relevant neighbor set;
- 9 $I[y] \leftarrow \langle v_1, \dots, v_{i-1}, v \rangle$;
- 10 $A'_{deg}[y] \leftarrow$ the number of edges incident to v in G' ;
- 11 run a parallel prefix sum on A'_{deg} ; let A'_o be the resulting array;
- 12 **return** I and A'_o ;

Next, *BuildCi* feeds A_v and A_o as inputs to the *GenCand* function (Algorithm 4.4). Let c_x denote the x -th graph in C_{i-1} . *GenCand* examines each $c_x = \langle v_1, \dots, v_{i-1} \rangle$, and retrieves from A_v the anchor vertex v_α in c_x . For each vertex v in the relevant neighbor set of v_α , *GenCand* regards $\langle v_1, \dots, v_{i-1}, v \rangle$ as a potential size- i candidate, and uses a GPU thread to write it into an array I . In addition, *GenCand* creates an array A'_{deg} , where the j -th element equals the number of edges incident to the vertex v associated with the j -th element in I . It then generates an array A'_o of offset values, by computing the prefix sum of A'_{deg} . Finally, it returns I and A'_o .

Finally, *BuildCi* applies the *RefineCand* function (Algorithm 4.5) to refine the potential size- i candidates in I . In particular, for each $\langle v_1, \dots, v_{i-1}, v \rangle$ recorded in I , *RefineCand* creates $|E(v)|$ GPU threads, each of which (i) checks whether an edge of v satisfies all of the edge validity conditions, and (ii) writes the result of the check into an array B . Then, *RefineCand* examines B to identify those vertices v that has $E(u_i)$ edges passing the validity check; for each such v , it inserts $\langle v_1, \dots, v_{i-1}, v \rangle$ into C_i as a size- i candidate. After that, the algorithm returns C_i and terminates.

Matching Order. We now discuss how we decide the matching order for the vertices in g . First, observe that the matching order significantly affects the number of candidates in C_i . For example, consider graphs g_1 and G_1 in Figure 4.1. Suppose that we are to compute $f(g_1, G_1)$, using a matching order $\langle u_1, u_2, u_3 \rangle$ for g_1 . In that case, the number of candidates in C_2 is 8. However, if we change the matching order to $\langle u_1, u_3, u_2 \rangle$, then the size of C_2 becomes 2. In general, we aim to select a matching order that minimizes the sizes of C_i ($i \in [2, k-1]$), so as to reduce computation overheads. That is, we aim to arrange the vertices in g into a sequence $u_1, u_2, \dots, u_k$, such that each subgraph induced by $u_1, u_2, \dots, u_j$ ($j \in [2, k]$) has as fewer occurrences in G' as possible. This problem has been studied in the context of subgraph isomorphism tests, and there exist several CPU-based heuristic solutions [36, 80, 90, 100]. In our solution, we adopt the CPU-based technique in [80] for choosing a matching order for g . We do not consider GPU-based techniques, since the costs of generating matching orders are insignificant when compared with the overheads of the other parts of our solution.

Algorithm 4.5: RefineCand

input : I and A'_o
output: C_i

- 1 let γ be the number of edges incident to u_i in g ;
- 2 create an array B of size $\gamma \cdot |I|$, with all elements set to 0 ;
- 3 let θ' be the value of the last element of A'_o ;
- 4 **for** *each* $z = 1, 2, \dots, \theta'$ **in parallel do**
- 5 identify the integer y such that $A'_o[y] \leq z < A'_o[y + 1]$;
- 6 let $\ell = z - A'_o[y]$;
- 7 let $\langle v_1, \dots, v_{i-1}, v \rangle$ be the y -th element of I ;
- 8 let e' be the ℓ -th edge incident to v in G' ;
- 9 let v' be the node that is connected to v by e ;
- 10 **if** *there exists* $v_j = v'$ ($j \in [1, i - 1]$) **then**
- 11 scan the edges of u ;
- 12 **if** *the* β -th edge e *connects* u *to* u_j **then**
- 13 **if** (e *starts from* v *and* e' *starts from* u) *or* (e *starts from* v_j *and* e' *starts from* u_j) **then**
- 14 $B[y \cdot \gamma + \beta] = 1$;
- 15 create an array B^* of size $|I|$ with all elements set to 0;
- 16 **for** *each* $y = 1, 2, \dots, |I|$ **in parallel do**
- 17 **if** $B[y \times \gamma + \ell]$ *equals* 1 *for each* $\ell \in [1, \gamma]$ **then**
- 18 $B^*[y] = 1$;
- 19 run a parallel prefix sum on B^* ; let A_o^* be the result;
- 20 let θ^* be the last element of A_o^* ;
- 21 create an array C_i of size θ^* ;
- 22 **for** *each* $y = 1, 2, \dots, |I|$ **in parallel do**
- 23 **if** $B^*[y]$ *equals* 1 **then**
- 24 $C_i[A_o^*[y]] \leftarrow I[y]$;
- 25 **return** C_i ;

4.4.3 Avoiding Duplicates

Let g'_i be an occurrence of g_i in G' . As mentioned in Section 4.4.2, if there are multiple isomorphisms of g'_i and g_i , then we record each isomorphism as a vertex sequence in C_i . This could lead to an excessive number of vertex sequences in C_i . For example, if g_i is a clique, then there exist $i!$ isomorphisms of g_i and g'_i , which result in $i!$ vertex sequences in C_i . Previous work [32] addresses this problem with technique that exploits graph automorphism, and we adopt the same technique in our GPU-based solution. To explain, we first introduce the concept of *automorphism groups*.

Figure 4.4: A graph g_3 with several automorphism groups.

Definition 4.2 (Automorphism Groups) *An automorphism group of g is a set A of ordered pairs that satisfy the following conditions:*

- (i) *The two elements in each order pair are vertices in g .*
- (ii) *In each ordered pair (u, u') , the vertex ID of u is smaller than that of u' .*
- (iii) *The set of edges in g remains unchanged even if, for each ordered pair $(u, u') \in A$, we exchange u and u' in all edges incident to u or u' .*

For example, consider the graph g_3 in Figure 4.4, assuming that the ID of each node u_i ($i \in [1, 6]$) equals i . $A_1 = \{(u_1, u_2)\}$ is an automorphism group of g_3 because (i) there are only two edges in g_3 that are incident to u_1 or u_2 , namely, $\langle u_1, u_3 \rangle$ and $\langle u_2, u_3 \rangle$, and (ii) even if we exchange u_1 and u_2 into those two edges, we still have $\langle u_2, u_3 \rangle$ and $\langle u_1, u_3 \rangle$, i.e., the set of edges in g_3 remain unchanged. It can be verified that g_3 has another two automorphism groups $A_2 = \{(u_1, u_5), (u_2, u_6), (u_3, u_4)\}$ and $A_3 = \{(u_5, u_6)\}$.

Based on g 's automorphism groups, we construct a *symmetry constraint set (SCS)* Q for g , which contains exactly one ordered pair from each automorphism group. For example, for the graph g_3 in Figure 4.4, $\{(u_1, u_2), (u_3, u_4), (u_5, u_6)\}$ is an SCS, since, as mentioned, (i) g_3 has three automorphism groups A_1 , A_2 , and A_3 , and (ii) $(u_1, u_2) \in A_1$, $(u_3, u_4) \in A_2$, and $(u_5, u_6) \in A_3$.

Given an SCS Q for g , we impose the following *symmetry constraint* on each size- i candidate $c = \langle v_1, v_2, \dots, v_i \rangle$ in C_i :

- *Symmetry Constraint:* For any ordered pair $(u_x, u_y) \in Q$ with $1 \leq x < y \leq i$, the vertex v_x in c has a smaller ID than the vertex v_y does.

It is proved in [32] that even if there are multiple isomorphisms of g and a subgraph of G' , only one of them satisfies the symmetry constraint given an SCS. Therefore, imposing the symmetry constraint eliminates duplicates² in C_k , and ensures that the frequency of g in G' can be correctly computed from C_k .

In our solution, we compute an SCS Q for g using the CPU-based algorithm in [32], and we impose the symmetry constraint on C_i during its generation in our GPU-based Algorithm 4.4. In particular, after Line 9 of Algorithm 4.4 constructs a potential size- i candidate $c = \langle v_1, v_2, \dots, v_i \rangle$, we test whether c satisfies the symmetry constraint by

²It also reduces the number of duplicates in C_i ($i \in [2, k-1]$) but does not necessarily eliminate them, since an SCS is computed based on the automorphism groups of g instead of $g(i)$.

inspecting all ordered pairs in Q that contain u_i . If c fails the test, then we set $A'_{deg}[y] = 0$ in Line 10; this ensures that c will be subsequently eliminated by the *RefineCand* function.

4.4.4 Correctness

The following lemma shows the the correctness of our solution.

Lemma 4.1 *For any g and $G' \in \mathcal{G}_r$, Algorithm 4.1 correctly computes the frequency of g in G' .*

Proof: [Sketch] To prove the lemma, we show that the set C_k constructed by Algorithm 4.2 contains exactly one vertex sequence for each occurrence of g in G' . First, due to the symmetry constraint approach [32] in Section 4.4.3, C_k contains at most one vertex sequence for each occurrence of g in G' . Second, by an induction on i , we can prove that there is at least one vertex sequence in C_i for each occurrence of g_i in G' , since (i) each occurrence of g_i in G' can be obtained by extending an occurrence of g_{i-1} in G' by one vertex, and (ii) Algorithm 4.2 considers all such extensions when constructing C_i from C_{i-1} . ■

4.5 Optimizations

This section presents several crucial techniques for optimizing the performance of our GPU-based method. Section 4.5.1 discusses the handling of large intermediate results in the frequency estimation. Section 4.5.2 introduces a method for processing multiple random graphs simultaneously. Section 4.5.3 analyzes how we can avoid redundant computation by carefully arranging the order in which the subgraphs are processed.

4.5.1 Handling Large Candidate Sets

Our GPU-based solution requires generating a few intermediate results, e.g., the size- i candidate sets C_i and the temporary arrays (e.g., A_{deg} , A_o , I , B) utilized in Algorithms 4.3, 4.4, and 4.5. When G_r is sizable, those intermediate results could be too large to fit in the global memory of the GPU. To address this issue, one straightforward approach is to use the machine's main memory (and harddisk, if necessary) as a secondary storage for the GPU. In particular, if the intermediate results in the conversion from C_i to C_{i+1} ($i \in [2, k-1]$) exceed the size of the GPU memory, then we may store C_i in the main memory, and divide it into several subsets $C_i^{(1)}, C_i^{(2)}, \dots, C_i^{(\beta)}$, such that each subset is small enough to be processed by the GPU. After that, we transfer the subsets to the GPU one by one, and ask the GPU to (i) convert each subset $C_i^{(j)}$ into a *partial* set $C_{i+1}^{(j)}$

of size- $(i + 1)$ candidates and (ii) send each $C_{i+1}^{(j)}$ back to the main memory. Once all $C_{i+1}^{(j)}$ are produced, we take their union to obtain the size- $(i + 1)$ candidate set C_{i+1} .

The above approach, however, is inefficient as it requires numerous rounds of data transfers between the main memory and the GPU memory, which are only connected via a (relatively slow) I/O bus. To address this problem, we propose a divide-and-conquer approach that processes all data in the GPU’s global memory, without utilizing the main memory as a secondary storage. To explain, assume that the GPU memory is sufficient to construct $C_2, C_3, \dots, C_i$, but not C_{i+1} . That is, the GPU would first run out of memory when transforming C_i to C_{i+1} . We first clarify how our approach works when $i = k - 2$, and then extend our discussion to the general case.

Given C_{k-2} , we first invoke Algorithm 4.3 to obtain two arrays A_v and A_o . Recall that the j -th element of A_o equals the number of potential size- $(k - 1)$ candidates that we need to generate from the z -th graphs in C_{k-2} where $z \leq j$. Therefore, based on A_o , we can calculate the amount of memory required in processing each graph in C_{k-2} . Given this information, we divide C_{k-2} into subsets, such that each subset $C_{k-2}^{(j)}$ can be converted into a set $C_{k-1}^{(j)}$ of size- $(k - 1)$ candidates using a fraction λ of the vacant memory on the GPU. (We will discuss the setting of λ shortly.) Then, we process each $C_{k-2}^{(j)}$ in turn. Whenever a size- $(k - 1)$ candidate subset $C_{k-1}^{(j)}$ is generated, however, we do not transfer it to the main memory of the machine; instead, we use the remaining $1 - \lambda$ fraction of the vacant GPU memory to convert $C_{k-1}^{(j)}$ into a size- k candidate subset $C_k^{(j)}$. In case that this conversion requires more memory than available, we further divide $C_{k-1}^{(j)}$ into subsets and process each subset in turn, in the same manner as the processing of C_{k-2} . Once a subset of size- k candidates are produced, we record the size of the subset, and then delete the subset from the GPU memory to make room for the processing of other subsets of $C_{k-1}^{(j)}$. In summary, we partition the vacant memory of the GPU into two parts, and use them to pipeline the generation of size- $(k - 1)$ and size- k candidates.

In general, if we have sufficient GPU memory to construct $C_2, \dots, C_i$ but not C_{i+1} , we start pipelining right after C_i is generated. Specifically, we divide the vacant GPU memory into $k - i$ parts, and assign the j -th part for the conversion from C_{i+j-1} to C_{i+j} . We heuristically set the size of the j -th part to be λ fraction of the GPU memory that is vacant after the first $j - 1$ parts are assigned, except that the last part utilizes all remaining GPU memory. To choose an appropriate value for λ , we model the total number of candidate subsets produced in the pipelining process (i.e., the total number of “splits” required on $C_i, \dots, C_{k-1}$) as a function of λ , and we derive the λ that minimizes the function. The rationale is as follows: each candidate subset needs to be processed with a few GPU kernels, each of which takes a certain amount of time to start up; therefore, if the total number of candidate subsets is large, then the total start-up overhead of the GPU kernels would be significant, which leads to inferior efficiency.

Let M be the amount of vacant GPU memory right after C_i is constructed. Observe that, in the conversion from C_i to C_{i+1} , the total size of the intermediate results is $O(|C_i|)$. Given that we assign λM GPU memory for the conversion from C_i to C_{i+1} , the number of subsets of C_i generated is roughly proportional to $\frac{|C_i|}{\lambda M}$. By the same reasoning, the number of subsets of C_j ($j > i$) produced is approximately proportional to

$$\begin{cases} \frac{|C_j|}{\lambda \cdot (1 - \lambda)^{j-i} \cdot M}, & \text{if } j \in [i + 1, k - 2]; \\ \frac{|C_{k-1}|}{(1 - \lambda)^{k-i-1} \cdot M}, & \text{if } j = k - 1. \end{cases}$$

Observe that $|C_{j+1}| \leq d \cdot |C_j|$, where d is the maximum vertex degree in G . We consider that $|C_{j+1}| = d \cdot |C_j|$, in which case the total number of subsets produced in the pipelining process is roughly proportional to:

$$\left(\sum_{j=i}^{k-2} \frac{d^{j-i} \cdot |C_i|}{\lambda \cdot (1 - \lambda)^{j-i} \cdot M} \right) + \frac{d^{k-i-1} \cdot |C_i|}{(1 - \lambda)^{k-i-1} \cdot M}. \quad (4.5)$$

It can be verified that Equation 4.5 is minimized when $\lambda = \frac{1}{k-i}$. Therefore, we set $\lambda = \frac{1}{k-i}$ in our solution.

4.5.2 Handling Multiple Graphs

Our previous discussions have focused on computing subgraph frequencies in one random graph $G' \in \mathcal{G}_r$. When G' contains relatively small numbers of vertices and edges, the frequency computation processes on G' may not engage all GPU cores, which leads to under-utilization of the GPU. We address this issue as follows. First, we divide the random graphs in $\mathcal{G}_r$ into several groups, each of which contains μ graphs, where μ is a tunable parameter. After that, for each group R of random graphs, we regard it as a graph G^* consisting of $|R|$ disjoint components, each of which is a graph in R . Then, we invoke our GPU-based frequency estimation method on G^* , with additional bookkeeping to (i) record the subgraph frequencies in each random graph separately, and (ii) ignore any subgraph of G^* that contains vertices from different graphs in R . In other words, we process the random graphs in each group in a batch manner, and thus, we avoid under-utilizing the GPU.

One crucial question remains: How do we decide the number μ of random graphs in each group? A naive approach is to set $\mu = |\mathcal{G}_r|$, i.e., we process all random graphs in $\mathcal{G}_r$ in one batch. This, however, severely exacerbates the GPU memory issue discussed in Section 4.5.1, and leads to inferior efficiency. To tackle the problem, we choose μ using a heuristic method, as explained in the following. First, observe that for any subgraph

g in G and any random graph $G' \in \mathcal{G}_r$, the size-2 candidate set of G' (i.e., C_2) has a size at most $\mu \cdot m$, where m is the number of edges in G' . In addition, each size-2 candidate in C_2 leads to at most d possible size-3 candidates in Algorithm 4.4, where d is the maximum vertex degree in G . Given $\mu \cdot m$ and d , we can derive an upperbound τ of the amount of GPU memory required in the conversion from C_2 to C_3 , and we set μ to the maximum integer such that the upperbound τ no more than $\frac{1}{k-2}$ fraction of the vacant GPU memory. In other words, we ensure that the conversion from C_2 to C_3 can be performed without invoking the divide-and-conquer method in Section 4.5.1, which helps avoid overloading the GPU. Although such a μ thus obtained does not guarantee that the GPU won't be overloaded in the computation of $C_4, C_5, \dots, C_k$, we find that it leads to satisfactory performance in our experiments.

4.5.3 Matching Tree

Let g and g' be two size- k subgraphs of G (i.e., $g, g' \in S_k$) that differ in only one node. Further assume that the matching orders of g and g' share a common prefix of length $k-1$, e.g., $g_1 = \langle u_1, \dots, u_{k-1}, u_k \rangle$ and $g_2 = \langle u_1, \dots, u_{k-1}, u'_k \rangle$. Then, when we compute the frequency of g in a random graph $G' \in \mathcal{G}_r$ (i.e., $f(g, G')$), the size- $(k-1)$ candidate set would be the same as in the computation of the frequency of $f(g', G')$. In other words, the computation of $f(g, G')$ overlaps significantly with that of $f(g', G')$.

Generally, if two graphs in S_k share a common prefix in their matching order, then the frequency estimation processes for the two graphs share a common component. A natural question is: How can we avoid redundant computation in the processing of such “similar” graphs? We answer this question with a method that carefully arranges the order in which we process the graphs in S_k .

Specifically, we first compute the matching order for every graph in S_k . After that, we organize all matching orders into a prefix tree, referred as the *matching tree*. Then, we perform a depth-first search (DFS) on the matching tree, and we process the graphs in S_k in the order in which they are encountered during the DFS. We refer to such a sequence of graphs induced by the DFS as the *DFS order*.

For example, Figure 4.5 illustrates 5 graphs in an S_k with $k=4$, as well as a matching tree for S_k . The DFS order corresponding to the matching tree is $g_1, g_2, g_3, g_4, g_5, \dots$. Given this DFS order, we avoid redundant computation in processing g_i ($i \in [1, 5]$) as follows. First, given any random graph $G' \in \mathcal{G}_r$, we compute g_1 's size-2 candidate set C_2 and size-3 candidate set C_3 . Based on C_3 , we derive the occurrences of g_1 in G' , as well as those of g_2 and g_3 , i.e., we avoid recomputing C_3 for g_2 and g_3 . This is feasible since the matching orders of g_1, g_2 , and g_3 share a common prefix $\langle u_1, u_2, u_3 \rangle$. After that, we reuse C_2 to derive the size-3 candidate set C'_3 for g_4 (as g_1 and g_4 have a common prefix of length 2), and then utilize C'_3 to compute $f(g_4, G')$ and $f(g_5, G')$ in one batch.

Figure 4.5: Illustration of a matching tree.

In general, for any size- i ($i \geq 2$) candidate set C_i that corresponds to a node p_i in the matching tree, we compute C_i only once and reuse it for deriving the occurrences of any graph in S_k that corresponds to a leaf node in the subtree under p_i . As such, we avoid all redundant computation of candidate sets. The downside of this approach is that it requires retaining C_i until all graphs in the subtree of p_i are processed, but this issue can be easily addressed with the divide-and-conquer method in Section 4.5.1. That is, we divide C_i into several subsets and process each subset in turn. For each subset $C_i^{(j)}$, we derive the occurrences of all graphs depending on $C_i^{(j)}$, and transfer the relevant results to the main memory of the machine. After that, we remove $C_i^{(j)}$ from the GPU's memory and use the freed space to process the next subset $C_i^{(j+1)}$.

4.6 Experimental Evaluation

4.6.1 Experimental Settings

We implement our GPU-based algorithm for network motif discovery (dubbed *NemoGPU*) in C++ under Nvidia CUDA 5.5, and compare it against four state-of-the-art CPU-based algorithms: *Kavosh* [51], *QateXelero* [54], *NetMode* [61], and *DistributedNM* [76]. We adopt the C++ implementations of *Kavosh*, *QateXelero*, and *NetMode* made available by their respective inventors, and we implement *DistributedNM* in C++ with multi-core optimizations. All of our experiments are conducted on two machines with identical hardware and software configurations. In particular, each machine runs CentOS 5.0, and has 32GB main memory, an Intel Xeon E5645 CPU, a low-end Nvidia Quadro 2000 (Q2000) GPU, as well as a high-end Nvidia Tesla K20 GPU. Table 4.1 shows the specifications of the CPU and GPUs. We run *NetMode*, *DistributedNM*, and the CPU part of *NemoGPU* with 6 threads (i.e., one thread per CPU core), but *Kavosh* and *QateXelero* with only

(a) Total running time.

(b) Running time for the first phase.

(c) Running time for the second phase.

Figure 4.6: Computation time on all datasets.

Table 4.1: Specifications of E5645, Q2000, and K20C.

Name	# of cores	Core freq.	GPU memory	Price (USD) ³
E5645	6	2400MHz	N/A	513.39
Q2000	192	625MHz	1GB	282.75
K20	2496	706MHz	5GB	2695.00

one thread as their implementations do not support parallelism. For the GPU part of *NemoGPU*, we run it on Q2000 and K20 separately.

We use seven biological networks in our experiments, as shown in Table 4.2. In particular, *Yeast* (YE) is the transcription network of yeasts; *H.sapiens* (HS) captures the protein-protein interaction (PPI) in the MINT dataset; *YeastPPI* (YP), *M.musculus* (MM), and *D.melanogaster* (DM) are the PPI networks of the budding yeast, fly genes, and mouse genes, respectively; *A.thaliana* (AT) describes the shared domains in Arabidopsis proteins; *C.elegans* (CE) represents the co-expression of worm genes. YE and YP are obtained from [12], while the other datasets are available from [6]. Our datasets are relatively small compared with those (with millions of nodes and edges) used in the literature of graph databases [36, 45], but we note that (i) biological networks are typically small, and (ii) identifying network motifs from our data is highly challenging due

to the large number of random graphs that we need to process, and the huge number of subgraph isomorphism tests required. Furthermore, to our knowledge, YP is the largest dataset used in the previous work on network motif discovery [32, 51, 54, 61, 71, 74, 95]. In other words, the datasets that we use are up to one order of magnitude larger than those used in previous work, in terms of the numbers of nodes and edges.

Table 4.2: Table of datasets.

Name	$ V $	$ E $	AVG deg.	MAX out-deg.	MAX in-deg.
YE	688	1,079	3.14	71	13
HS	1,509	5,598	7.42	71	45
YP	2,361	6,646	5.63	64	47
MM	4,293	7,987	3.72	91	111
DM	6,303	18,224	5.78	88	122
AT	9,216	50,669	11.00	58	89
CE	17,179	124,599	14.51	67	107

Following previous work [32, 61, 74, 76], we vary k (i.e., the size of the network motifs to be discovered) from 4 to 8, and set the default value of r (i.e., the number of random graphs) to 1000. However, if an algorithm’s running time exceeds 24 hours in an experiment, then we reduce r to 100 for that particular algorithm in the experiment, and measure its computation time t_1 (resp. t_2) for the first (resp. second) phase of network motif discovery; after that, we estimate the running time of the algorithm for $r = 1000$ as $t_1 + 10 \cdot t_2$. That said, if the algorithm does not terminate within 24 hours even when $r = 100$, then we omit it from the experiment. We repeat each experiment 3 times and report the average computation cost of each method.

4.6.2 Comparisons with CPU-Based Techniques

In the first set of experiments, we evaluate the computation time of all algorithms on all datasets, setting $k = 6$. Figure 4.6 illustrates the results. As shown in Figure 4.6a, our *NemoGPU* algorithm, when running with the high-end K20 GPU, outperforms all CPU-based methods by two orders of magnitude in terms of computation efficiency, regardless of the dataset used. When running with the low-end Q2000 GPU, *NemoGPU* is approximately ten times slower than with K20, as Q2000 has a much smaller GPU memory and considerably fewer GPU cores. However, even with Q2000, *NemoGPU* is still significantly more efficient than all CPU-based solutions. Among the CPU-based methods, *NetMode* and *DistributedNM* yield similar performance, with the latter slightly outperforming the former in most cases. Meanwhile, *Kavosh* and *QuateXelero* incur noticeably larger overheads than *NetMode* and *DistributedNM* do.

(a) Improvements over DistributedNM.

(b) Performance-price ratio.

Figure 4.7: *NemoGPU* (with Q2000 and K20) vs. *DistributedNM*.

Figure 4.6b and 4.6c illustrate the running time of each algorithm’s first and second phases, respectively. Observe that, for all algorithms, the computation overhead of the first phase is negligible compared with that of the second phase, which justifies our choice of optimizing only the second phase in our GPU-based solution. The first phase of *NemoGPU* incurs exactly the same overhead as *DistributedNM* does, since they adopt the same CPU-based algorithm for the first phase, as mentioned in Section 4.3.

To more clearly illustrate the superiority of our GPU-based solution, we compute the *improvement ratio* of *NemoGPU* over *DistributedNM* (i.e., the most efficient CPU-based method), defined as the running time of the latter divided by that of the former. Figure 4.7a shows the improvement ratio of *NemoGPU* when $k = 6$, on all datasets except CE (as *DistributedNM* fails to terminate on CE). On the other hand, we also take into account the price differences among the CPU and GPUs, and compute the *performance-price ratio* of *NemoGPU*, defined as

$$\text{Improvement ratio of } NemoGPU \times \frac{\text{Price of the E5625 CPU}}{\text{Price of the GPU}}.$$

Figure 4.7b plots the performance-price ratio of *NemoGPU* with Q2000 and K20. The ratio for K20 (resp. Q2000) is up to 33 (resp. 27), and is above 18 (resp. 15) in all cases. This indicates that both K20 and Q2000 yield much higher “performance per dollar” than the E5625 CPU does. In other words, if one is to improve the efficiency of network motif discovery, it is much more economical to invest in GPUs instead of CPUs.

Next, we evaluate the effects of k on the efficiency of network motif discovery. Figure 4.8 shows the computation time of each algorithm as a function of k , using datasets YE, HS, YP, and MM. We omit DM, AT, and CE from this experiment, as all CPU-based methods incur prohibitive overheads on those datasets. In addition, we omit *NetMode* when $k > 6$, since it is exclusively designed for the cases when $k \leq 6$. As shown in

Figure 4.8: Computation time vs. k .

Figure 4.8, our GPU-based solutions still outperform all CPU-based methods by large margins, regardless of the value of k . In particular, when running with K20, *NemoGPU* is more than 250 times faster than *DistributedNM* for $k \geq 7$.

4.6.3 Effects of Optimizations

Finally, we evaluate the effects of the three optimization techniques proposed in Section 4.5: the divide-and-conquer (DC) method for handling large candidate sets, the graph merging (GM) technique for processing multiple random graphs simultaneously, and the matching tree (MT) approach for avoiding redundant computation. We consider three “crippled” versions of *NemoGPU* on K20: one with all three optimization disabled (denoted as *NA*), one with only DC enabled (denoted as *DC*), and one with only DC and GM enabled (denoted as *DC-GM*). For each crippled version of *NemoGPU*, we define its

Figure 4.9: Effects of optimization vs. dataset.

relative overhead on a dataset D as its running time on D divided by the running time of *NemoGPU* with all three optimizations enabled.

Figure 4.9 shows the relative overheads of *DC-GM*, *DC*, and *NA* on each dataset. *DC-GM*'s relative overhead is around 2 in all cases, which indicates that the MT optimization reduces the running time of *NemoGPU* by half. Meanwhile, the relative overhead of *DC* is roughly two times that of *DC-GM*, implying that the GM optimization improves the efficiency of *NemoGPU* by a factor of 2. On the other hand, *NA*'s relative overhead is slightly lower than that of *DC* on the two smallest datasets, *YE* and *HS*. The reason is that the pipelining approach employed by *DC* incurred additional costs in terms of the running time of *NemoGPU*. However, *NA* fails to handle any of the four larger graphs due to its excessive demand on the GPU's global memory. This shows that, although *DC* entails additional overheads, it is crucial for the scalability of *NemoGPU*. In summary, the three optimizations in Section 4.5 improve the efficiency of *NemoGPU* by four-fold, and help scale *NemoGPU* to large graphs whose candidate sets do not fit in the GPU memory.

4.7 Summary

In this chapter, we study the problem of network motif discovery, and propose the first GPU-based solution to the problem. Our solution is drastically different from the existing CPU-based methods, due to the design choices that we make to exploit the strengths of GPUs in terms of parallelism and mitigate their limitations in terms of the computation power per GPU core. In addition, we develop three optimization techniques that considerably improve the scalability and efficiency of our solution. With extensive experiments on a variety of biological networks, we show that our solution is up to two orders of magnitude faster than the best CPU-based approach, and is around 20 times more cost-effective than the latter, when taking into account the monetary costs of the CPU and GPUs used.

Chapter 5

Generalized Subgraph Query Processing

This chapter proposes the problem of generalized subgraph query processing, which is useful in applications where subgraph queries are too restrictive to apply while similarity queries may return low quality answers due to large edit distance arisen from abundant irrelevant information. To address the problem, we devise a fast algorithm for generalized subgraph matching, which is a significantly more complicated matching problem than subgraph isomorphism. Besides, we develop three indexes for the efficient processing of generalized subgraph queries, namely, a *distance-based* index, a *frequent-pattern-based* index, and a *star-structure-based* index. We discuss in details the strengths and limitations of the indexes. Finally, we verify the efficiency of our matching algorithm (for candidate verification) and our indexes (for filtering) using both real and synthetic datasets.

The rest of this chapter is organized as follows. Section 5.1 gives the notations and formally defines the problem. Section 5.2 presents the generalized subgraph matching algorithm. Section 5.3 discusses in details the three indexes. Section 5.4 reports the experimental results. Section 5.5 summarizes this chapter.

5.1 Problem Statement

Let $\mathcal{G}$ be a database that contains a set of simple and labeled graphs. We denote each graph $g \in \mathcal{G}$ as a triplet $g = (V_g, E_g, L_g, l_g)$, where V_g and E_g are the sets of vertices and edges in g , respectively, and l_g is a labelling function that maps each vertex in g to a label in a finite alphabet L_g . For ease of exposition, we assume that all edges in g are undirected; our results can be easily extended for directed graphs.

For any vertices u and v in a graph $g \in \mathcal{G}$, we define the *distance* between u and v , denoted as $dist_g(u, v)$, as the number edges in the shortest path between u and v . For

instance, in the graph g_3 in Figure 1.1, we have $\text{dist}_{g_3}(v_1, v_3) = 2$, since the shortest path between v_1 and v_3 contains two edges (v_1, v_2) and (v_2, v_3) .

We aim to support *generalized subgraph queries* on $\mathcal{G}$. In particular, a *generalized subgraph* q is a simple, undirected, and labelled graph where each edge carries a positive integer weight. We denote q as a quadruple (V_q, E_q, l_q, t) , where V and E are the sets of vertices and edges in q , respectively, l_q is the labelling function for q , and t is a function that maps each edge in q to its weight. We say that a graph $g \in \mathcal{G}$ *matches* q , if there exists an injective function f from V_q to V_g , such that for any edge (u, v) in q , (i) the labels of u and $f(u)$ are the same, (ii) the labels of v and $f(v)$ are the same, and (iii) the distance between $f(u)$ and $f(v)$ in g is no more than the weight of (u, v) .

For example, in Figure 1.1, the graph g_3 matches the generalized subgraph q_1 . To explain this, let us consider an injective function f that maps u_1 to v_3 , u_2 to v_1 , u_3 to v_6 , and u_4 to v_5 . For the edge (u_1, u_2) in q_1 , we have $f(u_1) = v_3$ and $f(u_2) = v_1$, and the distance dist_{g_3} between v_3 and v_1 in g_3 equals 2, which is no more than the weight associated with (u_1, u_2) . The cases for the other edges in q_1 can be verified in a similar manner.

Given a generalized subgraph q , a generalized subgraph query on $\mathcal{G}$ returns the graphs in $\mathcal{G}$ that match q . For convenience, we refer to q as the *query graph*, and the graphs in $\mathcal{G}$ as the *data graphs*. In addition, we say that a data graph g *contains* q (denoted by $q \subseteq g$), if g matches q .

5.2 Generalized Subgraph Matching Algorithm

To enable generalized subgraph queries on $\mathcal{G}$, we need to first address a crucial problem: *How do we decide whether a data graph $g \in \mathcal{G}$ matches the query graph q ?* We refer to this problem as the *generalized subgraph matching* problem. It is not hard to see that this problem is NP-hard; in particular, when the weights of all edges in q equal 1, testing whether a data graph g matches q is equivalent to the *subgraph isomorphism* problem, which has been shown to be NP-complete [25].

Given that generalized subgraph matching is theoretically intractable, we resort to heuristics and propose a solution that provides practical efficiency. The core of our solution is a cost-based matching approach that significantly extends and improves the existing heuristic algorithms [80, 90] for the subgraph isomorphism problem. In what follows, we will first introduce the existing methods for subgraph isomorphism (in Section 5.2.1), and then present the details of our solution (in Section 5.2.2).

5.2.1 Existing Algorithms for Subgraph Isomorphism

The classic solution for the subgraph isomorphism is Ullmann’s algorithm [90], which matches the vertices in the query graph q to the vertices in the data graph g in an iterative manner. Specifically, in each iteration, the algorithm selects an unmatched vertex u in q , maps it to an unmatched vertex in g with the same label, and then checks whether the mapping is *feasible*, i.e., whether any two matched vertices in q that induce an edge in q are mapped to two vertices in g that induce an edge in g . If the mapping is feasible, the algorithm will enter the next iteration to match the remaining vertices in q . Otherwise, the algorithm will try matching u to another unmatched vertex in g . If there is no vertex that u can be matched to, the algorithm backtracks to the last matched vertex u' in q , re-maps u' to an unmatched vertex in g , and then re-starts the current iteration.

For example, in Figure 1.1, given the query graph q_2 and the data graph g_3 , Ullmann’s algorithm may first map u_5 to v_3 , and then map u_6 to v_5 . In that case, u_5 and u_6 induce an edge in q_2 , while v_3 and v_5 also induce an edge in g_3 , i.e., the matching is feasible. Assume that, in the next iteration, the algorithm maps u_7 to v_4 . Then, u_6 and u_7 induce an edge in q_2 , but the vertices that they are mapped to (i.e., v_5 and v_4) do not induce any edge in g_3 . As a consequence, the mapping is infeasible, and hence, the algorithm would proceed to re-map u_7 to another unmatched vertex in g_3 .

Intuitively, the efficiency of Ullmann’s algorithm depends on the order in which the vertices in q are matched. For instance, assume that q contains only two vertices u_1 and u_2 , such that u_1 has the same label with only one vertex v_1 in the data graph g , whereas u_2 has the same label with almost all vertices in g . If we invoke Ullmann’s algorithm and map u_1 to v_1 in the first iteration, then in the remaining iterations, we only need to examine whether u_2 can be mapped to a vertex adjacent to v_1 . In contrast, if the first iteration maps u_2 (instead of u_1) to some vertex v in g , then in the remaining iterations, we not only need to try mapping u_1 to the neighbors of v , but also need to consider other possible mappings that match u_2 to other vertices in g , i.e., the search space of the algorithm becomes significantly larger.

Despite the importance of vertex mapping order, it is not taken into account in Ullmann’s algorithm. This motivates a more advanced method called *QuickSI* [80], which improves over Ullmann’s algorithm by heuristically choosing a mapping order that is likely to reduce computation cost. Specifically, QuickSI decides the vertex mapping order based on two sets of statistics pre-computed from the graph database $\mathcal{G}$. First, for any vertex u that can possibly appear in a query graph, QuickSI pre-computes its *frequency* in $\mathcal{G}$, i.e., the average number of vertices in each data graph (in $\mathcal{G}$) that have the same label with u . Second, for any edge e that may appear in a query graph, QuickSI also pre-computes its frequency in $\mathcal{G}$, i.e., the average number of edges in each data

graph that have endpoints with labels matching those of the endpoints of e . With these statistics, for any given query graph q , QuickSI first generates a spanning tree of q , such that vertices and edges closer to the root of tree tend to have lower frequencies in $\mathcal{G}$. After that, QuickSI generates an ordering of the vertices in q following a traversal of the spanning tree that recursively visits the branch with the least frequent edge. The resulting vertex order is then used whenever QuickSI compares a data graph g with q . Intuitively, this vertex order improves efficiency, as it tends to ensure that the search space of the matching algorithm would be reduced significantly after each iteration.

5.2.2 A Cost-based Approach for Generalized Subgraph Isomorphism

Both Ullmann’s algorithm and QuickSI can be extended for generalized subgraph isomorphism, with a modified feasibility check in each iteration. Specifically, each time after we map a vertex u in the query graph q to a vertex v in the data graph g , we would decide whether the mapping is feasible by examining every edge e in q that is induced by u and any vertex u' in q that has been matched. Let v' be the vertex in g that u' is mapped to. If for each e , the distance $dist_g(v, v')$ between v and v' is no more than the weight $w(e)$ of e , then the mapping is feasible, and we would proceed to the next iteration. Otherwise, we would re-map u to other unmatched vertex in g ; if there does not exist any feasible mapping for u , we would backtrack to the last matched vertex in q and re-map it (as with the case of subgraph isomorphism).

The aforementioned extensions of Ullmann’s algorithm and QuickSI, however, leave much room for improvements. In particular, Ullmann’s algorithm does not exploit the order of vertex mapping for efficiency; QuickSI heuristically tunes the vertex mapping order, but its tuning method is rather ad hoc and is without a formal model that justifies mapping a vertex ahead of any other. To remedy this, we propose a novel algorithm for generalized subgraph isomorphism that incorporates a cost model for selecting a preferable order of vertex mapping. In the following, we will first present the rationale behind our method, and then provide the details about our cost model and algorithm.

Assume that the query graph q and the data graph g contain m and n vertices, respectively. Totally, there exist $P(n, m) = n!/(n - m)!$ different ways to map the vertices in q to distinct vertices in g , and these $P(n, m)$ possible matchings constitute the search space for the generalized subgraph isomorphism algorithm. ($P(n, m)$ denotes the number of m -permutations of n .) To efficiently decide whether g matches q , it is essential that the algorithm should traverse the search space in a judicious order that enables it to pinpoint a solution (if any) as quickly as possible. This motivates us to match vertices in q in an order based on how likely they can reduce the search space that we need to explore. Note that we use the same node mapping order for all data graphs

(as in QuickSI), so as to avoid the overhead of re-computing the node order for each data graph.

Specifically, to pick the first vertex in q to be matched, we would inspect each edge e in q , and examine the *frequency of e* (denoted as $c(e)$) in the data graphs in $\mathcal{G}$. The frequency of (u', u^*) in q is defined as the average number of vertex pairs (v', v^*) in each data graph in $\mathcal{G}$, such that (i) the labels of u' and v' are the same, (ii) the labels of u^* and v^* are the same, and (iii) the distance between v' and v^* is no more than the weight of (u', u^*) . (To facilitate this step of the algorithm, we pre-compute the frequency of any edge that may appear in the query graph by scanning the graphs in $\mathcal{G}$ in a pre-processing step.)

For each e , we intuitively estimate that it can be matched to $c(e)$ vertex pairs in the data graph. Given this estimation, if a vertex u is an endpoint of e and we choose to match u first, then the search space size induced by mapping u can be estimated as $c(e) \cdot P(\bar{n} - 1, m - 1)$, where $\bar{n}$ denotes the average number of vertices in the data graphs. The rationale here is that u is expected to be mapped to around $c(e)$ vertices in a data graph, and the other unmatched $m - 1$ vertices in q are expected to be matched to around $\bar{n} - 1$ vertices in g in $P(\bar{n} - 1, m - 1)$ different ways; therefore, the number of possible matchings that remain to be explored can be estimated as $c(e) \cdot P(\bar{n} - 1, m - 1)$. Accordingly, we pick a vertex u incident to the edge e with the smallest $c(e)$, and set u as the first vertex to be matched. The term $P(\bar{n} - 1, m - 1)$ is ignored since its value is the same for all vertices in q . (This helps us avoid the pathological case when $\bar{n} < m$, in which case $P(\bar{n} - 1, m - 1)$ is undefined.) Given that the edge e with the smallest $c(e)$ has two endpoints, we choose the endpoint u with the smaller frequency $c(u)$.

The order of the remaining vertices is decided in a similar manner. Assume that we have picked a set S of k vertices and we are about to choose the next vertex to be matched. Let u' be any vertex that has not been selected. If u' is not connected to any vertex in S by an edge in q , then we estimate the search space size induced by mapping u' as

$$\min_{\text{any edge } e \text{ adjacent to } u'} c(e) \cdot N(S) \cdot P(\bar{n} - k - 1, m - k - 1), \quad (5.1)$$

where $N(S)$ denotes the number of ways to match the first k vertices, and $P(\bar{n} - k - 1, m - k - 1)$ is the number of ways to match the remaining $m - k - 1$ vertices except u' . As will be shown shortly, we do not need to compute the values of $N(S)$ and $P(\bar{n} - k - 1, m - k - 1)$.

On the other hand, if u' has some edges that are incident to the vertices in S , then our estimation of the search space size would take those edges into account. Let E be the set of edges in q that connect u' to the vertices in S . For each e in E that connects u' to a vertex u^* , we examine the frequency of u^* (denoted as $c(u^*)$) in $\mathcal{G}$, as well as the frequency of e (denoted as $c(e)$). Given $c(u^*)$ and $c(e)$, we intuitively estimate that the

vertex u^* is connected to around $c(e)/c(u^*)$ vertices that have the same label with u' . Therefore, the search space size induced by mapping u' is estimated as

$$c(e)/c(u^*) \cdot N(S) \cdot P(\bar{n} - k - 1, m - k - 1), \quad (5.2)$$

where $N(S)$ and $P(\bar{n} - k - 1, m - k - 1)$ are as explained in Equation 5.1. We refer to $c(e)/c(u^*)$ as the *matching rate of u' implied by e* , and we denote it as $r(u', e)$.

Observe that each edge $e \in E$ may imply a different matching rate of u' , leading to different estimations of the search space size. We combine all estimations by taking the smallest one, i.e., the size of the search space is estimated as

$$\min_{e \in E} r(u', e) \cdot N(S) \cdot P(\bar{n} - k - 1, m - k - 1). \quad (5.3)$$

For convenience, we let $r(u') = \min_{e \in E} r(u', e)$ if u' is connected to the vertices in S by at least one edge in the query graph, otherwise we let $r(u')$ be the minimum frequency of an edge in q that is adjacent to u' . Given Equations 5.1 and 5.3, we choose the next vertex u' to be matched as the one that minimizes the estimated search space size, i.e.,

$$u' = \arg \min_u \{r(u)\}. \quad (5.4)$$

Note that Equation 5.4 does not involve the terms $N(S)$ and $P(\bar{n} - k - 1, m - k - 1)$ (which appear in both Equations 5.1 and 5.3). This is because their values are the same for all possible u' , and hence, they have no effect on the selection of u' .

In summary, our algorithm optimizes the vertex matching order by a qualitative prediction of how each vertex may help reduce the search space size. As will be shown in Section 5.4, our experimental results demonstrate the superiority of our algorithm over both Ullmann's algorithm and QuickSI on both standard and generalized subgraph isomorphism tests.

5.3 Indexing Techniques

Although in Section 5.2 we proposed a reasonably fast algorithm for generalized subgraph matching, it is still impractical to answer a query by sequentially scanning the input database and matching the query graph with each data graph, especially if the database is large. We apply the *filter-and-verification* strategy to reduce the matching cost, that is, we first filter out as many unmatching data graphs as possible and then verify the remaining candidate data graphs by matching them with the query graph one by one. To do this, it is important to design an effective indexing technique to filter out the unmatching data graphs.

In this section, we propose three indexing techniques: **D-Index**, **FP-Index** and **S-Index**. First, in Section 5.3.1 we present D-Index, which can be easily constructed but its pruning power is relatively weak. Then, we propose FP-Index in Section 5.3.2, which has an expensive construction cost but is partially verification-free. Lastly, in Section 5.3.3 we propose S-Index, which explores the star structures to achieve effective pruning as well as a low construction cost.

5.3.1 Distance Index

We first present D-index, which is constructed based on the distance among pairs of vertices in each data graph. Given a data graph $g = (V_g, E_g, L_g, l_g) \in \mathcal{G}$, we obtain the *distance set* ($\mathcal{DS}$) of all triplets of every two vertices consisting of their ordered labels and the correspond distance in g as follows.

$$\mathcal{DS}(g) = \{(l_g(u), l_g(v), dist_g(u, v)) : u, v \in V_g, l_g(u) \leq l_g(v)\}$$

A distance triplet $(l_1, l_2, d) \in \mathcal{DS}(g)$ is subsumed by another distance triplet $(l_1, l_2, d') \in \mathcal{DS}(g)$ if $d > d'$. We say that a subset $\mathcal{DS}_{min}(g) \subseteq \mathcal{DS}(g)$ is minimal if each distance triplet in $\mathcal{DS}_{min}(g)$ is not subsumed by any other distance triplet, that is, for each $(l_1, l_2, d) \in \mathcal{DS}_{min}(g)$, there does not exist $(l_1, l_2, d') \in \mathcal{DS}_{min}(g)$ such that $d' < d$.

Example 5.1: Assume that vertex labels are ordered lexicographically, i.e., $0 < N < H < C$. Consider the data graph g_3 in Figure 1.1, the distance set of g_3 is $\mathcal{DS}(g_3) = \{(C, C, 1), (C, C, 2), (C, C, 3), (H, C, 1), (H, C, 2), (H, C, 3), (N, C, 1), (N, C, 2), (O, C, 1), (O, C, 2), (O, C, 3), (O, C, 4), (N, H, 2), (O, H, 3), (O, H, 4), (O, N, 1), (O, N, 2), (O, O, 1)\}$, and $\mathcal{DS}_{min}(g_3) = \{(C, C, 1), (H, C, 1), (N, C, 1), (O, C, 1), (N, H, 2), (O, H, 3), (O, N, 1), (O, O, 1)\}$. Note that $|\mathcal{DS}(g_3)| = 18$ while $|\mathcal{DS}_{min}(g_3)| = 8$. $\square$

The minimal set of distinct distance triplets in the database is then given by

$$\mathcal{DS} = \cup_{g \in \mathcal{G}} \mathcal{DS}_{min}(g).$$

For each distance triplet $(l_1, l_2, d) \in \mathcal{DS}$, the set of data graphs that contain (l_1, l_2, d) is given by

$$\mathcal{A}(l_1, l_2, d) = \{g : (l_1, l_2, d) \in \mathcal{DS}_{min}(g)\}.$$

The *Distance Index* (*D-index*) is constructed on $\mathcal{DS}$ and $\mathcal{A}(l_1, l_2, d)$ for each $(l_1, l_2, d) \in \mathcal{DS}$, which is to be detailed as follows.

Algorithm 5.1: BUILD-DINDEX($\mathcal{G}$)

input : the graph database, $\mathcal{G}$
output: the D-index, LPI

```

1 for  $g \in \mathcal{G}$  do
2   Compute  $\mathcal{DS}_{min}(g)$  ;
3   for  $(l_1, l_2, d) \in \mathcal{DS}_{min}(g)$  do
4      $LPI \leftarrow (l_1, l_2)$  ;
5      $LPI(l_1, l_2).DV \leftarrow d$  ;
6      $LPI(l_1, l_2).DV(d) \leftarrow g$  ;
7 return  $LPI$ 

```

5.3.1.1 Index Construction

The structure of D-index consists of the following parts:

- A B+-tree index, called the *Label Pair Index* (LPI), stores all pairs of labels of the triplets in $\mathcal{DS}$.
- A sorted list of distance values for each label pair $(l_1, l_2) \in LPI$, denoted by $LPI(l_1, l_2).DV$, that is, $LPI(l_1, l_2).DV = \{d : (l_1, l_2, d) \in \mathcal{DS}\}$.
- Each distance value $d \in LPI(l_1, l_2).DV$ for any $(l_1, l_2) \in LPI$ is associated with a set of data graphs $\mathcal{A}(l_1, l_2, d)$, we further denote it as $LPI(l_1, l_2).DV(d) = \mathcal{A}(l_1, l_2, d)$.

The algorithm for D-index construction, BUILD-DINDEX, is shown in Algorithm 5.1. For each data graph $g \in \mathcal{G}$, we first compute its minimal distance set $\mathcal{DS}_{min}(g)$ (Line 2). Then, for each distance triplet $(l_1, l_2, d) \in \mathcal{DS}_{min}(g)$, we assign (l_1, l_2) to LPI and put the distance value d to the sorted list $LPI(l_1, l_2).DV$ (Line 4-5). Finally, g is included in $LPI(l_1, l_2).DV(d)$ (Line 6).

5.3.1.2 Query Processing

Given a query graph $q = (V_q, E_q, l_q, t)$, we first obtain its minimal distance set $\mathcal{DS}_{min}(q)$ by all pairs shortest path algorithm. For each distance triplet $(l_1, l_2, d) \in \mathcal{DS}_{min}(q)$, the correspond candidate set $\mathcal{C}(l_1, l_2, d)$ can be obtained by merging all the graph sets associated with $LPI(l_1, l_2).DV(k)$ for $1 \leq k \leq d$. The final candidate set $\mathcal{C}(q)$ for verification is the intersection of all the candidate sets of each distance triplet $(l_1, l_2, d) \in \mathcal{DS}_{min}(q)$, that is

$$\mathcal{C}(q) = \bigcap_{(l_1, l_2, d) \in \mathcal{DS}_{min}(q)} \mathcal{C}(l_1, l_2, d).$$

Algorithm 5.2: QUERY-DINDEX($q, LPI, \mathcal{G}$)

input : the query graph, $q = (V_q, E_q, l_q, t)$
 D-Index, LPI
 the graph database, $\mathcal{G}$
output: the candidate set of q , $\mathcal{C}(q)$

```

1  $\mathcal{C}(q) \leftarrow \mathcal{G}$  ;
2 for  $(l_1, l_2, d) \in \mathcal{DS}_{min}(q)$  do
3    $\mathcal{C}(l_1, l_2, d) \leftarrow \emptyset$  ;
4   for  $k \in LPI(l_1, l_2).DV$  and  $k \leq d$  do
5      $\mathcal{C}(l_1, l_2, d) \leftarrow \mathcal{C}(l_1, l_2, d) \cup LPI(l_1, l_2).DV(k)$  ;
6    $\mathcal{C}(q) \leftarrow \mathcal{C}(q) \cap \mathcal{C}(l_1, l_2, d)$  ;
7 return  $\mathcal{C}(q)$ 
    
```

As shown in Algorithm 5.2, processing the query graph q by D-index obtains the candidate set for each distance triplet $(l_1, l_2, d) \in \mathcal{DS}_{min}(q)$ (Lines 3-5), and then intersects them to output the candidate set of q (Line 6).

Lemma 5.1 *Given a query graph $q = (V_q, E_q, l_q, t)$, its answer set $\mathcal{A}(q)$ is a subset of BUILD-DINDEX($q, LPI, \mathcal{G}$).*

Proof: Consider a data graph $g = (V_g, E_g, L_g, l_g) \in \mathcal{G}$ that matches q . For each edge $(v, u) \in E_q$, we can map it to a path $P = \langle f(v), \dots, f(u) \rangle$ such that $|P| \leq t(v, u)$. Without the loss of generality, we assume that $l_g(f(v)) \leq l_g(f(u))$. Consider the distance triplet $(l_g(f(v)), l_g(f(u)), d) \in \mathcal{DS}_{min}(g)$, we have $d \leq |P|$, that is $d \leq t(v, u)$, which completes the proof. ■

5.3.1.3 Complexity Analysis

Assume that α is the average number of vertices and β is the average number of edges for the graphs in $\mathcal{G}$, the space complexity of D-index is $O(\alpha^2|\mathcal{G}|)$. Since the construction time for each $\mathcal{DS}_{min}(g)$ is $O(\alpha\beta)$ by starting a BFS from each vertex in g , the time complexity of constructing D-index is $O(\alpha\beta|\mathcal{G}|)$.

The generation of minimal distance set of query graph q can be done in $O(|V_q|^3)$ time. For each distance triplet in $\mathcal{DS}_{min}(q)$, the response time of D-index is $O(\log(md))$ where m is the number of distinct labels in $\mathcal{G}$ and d is the largest distance. Thus, the index response time for the graph pattern is $O(|V_q|^3 + k^2 \log(md))$, where k is the number of distinct labels in q . Note that, in practice, both $|V_q|$ and k are very small.

5.3.2 Frequent Pattern Index

The shortcoming of D-index is that it loses the structural information of data graphs. As a result, the filtering is not effective enough, leading to a lot of unmatched candidate graphs. Although there are graph indexing approaches that retain structural information of data graphs [22, 38, 83, 101, 106], they cannot be directly applied to answer generalized subgraph queries.

In order to employ structural information for answering generalized subgraph queries, we propose the concept of *frequent generalized subgraph (FGG)* patterns and apply FGGs to design a structural index called *Frequent Pattern Index (FP-index)*. The challenges, however, are 1) how to efficiently mine the FGGs, 2) how to apply and index FGGs for filtering. We address the two challenges as follows.

The first challenge can be addressed by the pattern-growth approach [14]. The difference is that in an FGG, edges are weighted. To obtain weighted edges for FGGs, we grow the frequent patterns from weighted edges. We initialize the set of weighted edges by taking the set of distinct distance triplets $\mathcal{E} = \cup_{g \in \mathcal{G}} \mathcal{DS}(g)$ introduced in Section 5.3.1, where each distance triplet $(l_1, l_2, d) \in \mathcal{E}$ is considered as an edge (l_1, l_2) with weight d , while $|\mathcal{C}(l_1, l_2, d)|$ is the frequency of the edge.

A subgraph pattern f is frequent if its frequency is greater than a pre-defined threshold σ . Since the number of FGGs can be too large and indexing a large number of FGGs will increase the index size and hence the search time, we apply a maximum pattern size threshold, γ , and a maximum edge weight threshold, ρ , to obtain only FGGs with size at most γ and any edge weight at most ρ . In our experiments, we set these thresholds as the best possible values such that the FGGs can fit in the machine memory.

5.3.2.1 Index Construction

The FP-index consists of two parts: *frequent pattern graph index (FPG-index)* and *edge index (E-index)*. FPG-index stores the FGGs in a B+-tree, with the key as an FGG and the data value as the set of data graphs containing the FGG. To answer queries that may contain infrequent edges, we also construct E-index that builds a B+-tree on the set of infrequent edges and frequent large-weight edges whose weights are larger than ρ , with the key as an edge and the data value as the set of data graphs containing the edge.

5.3.2.2 Query Processing

Given a query graph q , we process the query with FP-index as follows:

- Case 1: q is an FGG indexed by FPG-index. In this case, we obtain q 's answer set from the index directly without verification.

- Case 2: q is not an FGG indexed by FPG-index. If q contains infrequent edges, then the candidate set of q is relatively small (at most $\sigma|\mathcal{G}|$), which can be retrieved from E-index. Otherwise, we reduce the weight of some edges in q to obtain an FGG q' , then we have a partial answer, $\mathcal{A}(q')$, of q , without verification. And then, we obtain the rest of the answer, i.e., $\mathcal{A}(q) \setminus \mathcal{A}(q')$, as follows. We decompose q into several FGGs in FPG-index and frequent large-weight edges in E-index, as $f_1, f_2, \dots, f_k$, and compute the candidate set by intersecting the answer sets of these frequent patterns: $\cap_{1 \leq i \leq k} \mathcal{A}(f_i)$.

5.3.2.3 Complexity Analysis

Similar to other structural graph indexes such as FG-index [22], the construction cost of FP-index is dominated by the cost of mining FGGs and the index size is dominated by the overall size of FGGs. Likewise, the query processing complexity also heavily depends on the number of FGGs indexed as well as the value of $\sigma|\mathcal{G}|$. However, the complexity of mining FGGs, as well as the size and number of FGGs, may vary significantly from database to database and we are not aware of any formal analysis for these factors in the literature.

5.3.3 Star Index

Since the number of FGGs can be large, FP-index can only be used to process queries of small size efficiently. Moreover, mining FGGs may also be too expensive. Thus, we propose another index, called *Star Index* (*S-index*), which uses only star structures (instead of subgraph structures) to reduce both the index construction and storage overhead, while still capturing much of the structural information for effective filtering in query processing.

For a vertex v in a data graph $g = (V_g, E_g, L_g, l_g) \in \mathcal{G}$, we define the *star structure* of v as $s_g(v) = (V_g, E_g^v, l_g, w)$, where (1) v is the center of the star structure; (2) E_g^v consists of the edges from v to other vertices in V_g , that is $E_g^v = \{v\} \times (V_g \setminus \{v\})$; (3) w is a function that assigns the distance $dist_g(v, u)$ to each edge $(v, u) \in E_g^v$, i.e., $w(v, u) = dist_g(v, u)$. For example, Figure 5.1 shows the star structures of v_1 and v_3 of the graph g_3 in Figure 1.1.

We group the weights of the edges in a star structure by the label of the non-center end vertex. Let $L = \{l_g(u) : u \in V_g \setminus \{v\}\}$. We obtain a *multiset* of weights (called *weight multiset*) for each label as follows

$$\mathcal{W}_g(v, l) = \{w(u, v) : l_g(u) = l, l \in L, \text{ and } u \in V_g \setminus \{v\}\}.$$

 Figure 5.1: Two star structures: $s_{g_3}(v_1)$ and $s_{g_3}(v_3)$.

 Table 5.1: Weight multisets of g_3 .

Vertex v	$l_{g_3}(v)$	N	C	O	H
v_1	H	2	1,2,3	3,4	
v_2	C	1	1,2	2,3	1
v_3	N		1,1,2	1,2	2
v_4	O	1	2,2,3	1	3
v_5	C	1	2,3	1,2	3
v_6	O	2	1,3,4	1	4
v_7	C	2	1,3	3,4	2

The weight values in the weight multiset are sorted in ascending order. Table 5.1 lists all the weight multisets for each label and each star structure of g_3 . For example, for the star structure $s_{g_3}(v_1)$, $\mathcal{W}_{g_3}(v_1, \text{C}) = \{1, 2, 3\}$, $\mathcal{W}_{g_3}(v_1, \text{O}) = \{3, 4\}$, and $\mathcal{W}_{g_3}(v_1, \text{N}) = \{2\}$.

Given two weight multisets $\mathcal{W}_1 = \{w_1, \dots, w_k\}$ and $\mathcal{W}_2 = \{w'_1, \dots, w'_t\}$, where $k \leq t$. We define the merge of $\mathcal{W}_1$ and $\mathcal{W}_2$ as follows.

$$\mathcal{W}_1 \cap \mathcal{W}_2 = \{\min(w_1, w'_1), \dots, \min(w_k, w'_k), w'_{k+1}, \dots, w'_t\}.$$

For some vertices in a graph, they may share the same label. So, we further compress the weight multisets as follows.

$$\mathcal{W}_g(l_1, l_2) = \cap_{l_g(v)=l_1, v \in V_g} \mathcal{W}_g(v, l_2).$$

Table 5.2 lists the compressed weight multisets. For example, v_2 and v_5 share the same label C, so $\mathcal{W}_{g_3}(v_2, \text{H}) = \{1\}$ and $\mathcal{W}_{g_3}(v_5, \text{H}) = \{3\}$ are merged into one $\mathcal{W}_{g_3}(\text{C}, \text{H}) = \{\min(1, 3)\} = \{1\}$.

The set of distinct compressed weight multisets of each label pair (l_1, l_2) in the database $\mathcal{G}$ can be obtained as follows.

$$\mathcal{W}(l_1, l_2) = \cup_{g \in \mathcal{G}} \mathcal{W}_g(l_1, l_2).$$

Table 5.2: Compressed weight multisets of g_3 .

Label	N	C	O	H
N		1,1,2	1,2	2
C	1	1,2	1,2	1
O	1	1,2,3	1	3
H	2	1,2,3	3,4	

Algorithm 5.3: BUILD-SINDEX($\mathcal{G}$)

input : the graph database, $\mathcal{G}$

output: the S-index, SI

```

1 for  $g \in \mathcal{G}$  do
2   Let  $L(g) = \{(l_g(u), l_g(v)) : u, v \in V_g \text{ and } u \neq v\}$  ;
3   for  $(l_1, l_2) \in L(g)$  do
4     Compute  $w = \mathcal{W}_g(l_1, l_2)$  ;
5     Add  $w$  to the record in  $SI$  with key  $(l_1, l_2)$  ;
6     Add  $g$  to the record in  $SI(l_1, l_2)$  with key  $w$  ;
7 return  $SI$ 
    
```

For each compressed weight multiset $w \in \mathcal{W}(l_1, l_2)$, the set of data graphs whose corresponding compressed weight multiset is w is defined as follows.

$$\mathcal{A}(w) = \{g : \mathcal{W}_g(l_1, l_2) = w \text{ and } g \in \mathcal{G}\}.$$

5.3.3.1 Index Structure

S-index is constructed based on $\mathcal{W}(l_1, l_2)$ for each distinct label pair (l_1, l_2) and $\mathcal{A}(w)$ for each $w \in \mathcal{W}(l_1, l_2)$. The index structure of S-index consists of the following parts:

- A B+-tree on the set of distinct label pairs with each pair (l_1, l_2) as the key and $\mathcal{W}(l_1, l_2)$ as the data value. We name this B+-tree as SI .
- A nested B+-tree on the set of distinct compressed weight multisets $\mathcal{W}(l_1, l_2)$ for each pair (l_1, l_2) , where each compressed weight multiset $w \in \mathcal{W}(l_1, l_2)$ is the key and $\mathcal{A}(w)$ is the data value. We name this nested B+-tree as $SI(l_1, l_2)$.

Algorithm 5.3 outlines the construction of S-index. For each data graph $g \in \mathcal{G}$, for each distinct label pair (l_1, l_2) in g , we obtain the compressed weight multiset $\mathcal{W}_g(l_1, l_2)$, and store it in the record with the key (l_1, l_2) in SI . Then, we access the record with the key $\mathcal{W}_g(l_1, l_2)$ in the nested B+-tree $SI(l_1, l_2)$ and add g to the corresponding record.

Algorithm 5.4: QUERY-SINDEX($q, SI, \mathcal{G}$)

input : the query graph, $q = (V_q, E_q, l_q, t)$
 S-Index, SI
 the graph database $\mathcal{G}$
output: the candidate set of q , $\mathcal{C}(q)$

```

1  $\mathcal{C}(q) \leftarrow \mathcal{G}$  ;
2 Let  $L(q) = \{(l_g(u), l_g(v)) : u, v \in V_g \text{ and } u \neq v\}$  ;
3 for  $(l_1, l_2) \in L(q)$  do
4   Compute  $\mathcal{W}_q(l_1, l_2)$  ;
5    $\mathcal{C}(l_1, l_2) \leftarrow \emptyset$  ;
6   for  $w \in SI(l_1, l_2)$  and  $w \leq \mathcal{W}_q(l_1, l_2)$  do
7      $\mathcal{C}(l_1, l_2) \leftarrow \mathcal{C}(l_1, l_2) \cup \mathcal{A}(w)$  ;
8    $\mathcal{C}(q) \leftarrow \mathcal{C}(q) \cap \mathcal{C}(l_1, l_2)$  ;
9 return  $\mathcal{C}(q)$ 
    
```

5.3.3.2 Query Processing

We now discuss query processing by S-index. Given two compressed weight multisets $\mathcal{W}_1 = \{w_1, \dots, w_k\}$ and $\mathcal{W}_2 = \{w'_1, \dots, w'_t\}$, we say that $\mathcal{W}_1 \leq \mathcal{W}_2$ if and only if 1) $k \geq t$, and 2) $w_r \leq w'_r$ for $1 \leq r \leq t$.

Lemma 5.2 *Let $s_g(u)$ and $s_q(v)$ be two star structures, such that u and v are vertices in graphs $g = (V_g, E_g, l_g)$ and $q = (V_q, E_q, l_q, t)$, respectively. If $s_g(u)$ matches $s_q(v)$, then we have $\mathcal{W}_g(l_1, l_2) \leq \mathcal{W}_q(l_1, l_2)$ for all distinct label pairs (l_1, l_2) of q .*

Proof: Consider a label pair (l_1, l_2) of q , l_1 is the label of the center vertex of a star structure of q and l_2 is the label of a non-center vertex. Since g matches q , the number of vertices in V_g with label l_2 is no smaller than the number of vertices in V_q with the same label. Therefore, we have $|\mathcal{W}_g(l_1, l_2)| \geq |\mathcal{W}_q(l_1, l_2)|$. Moreover, for each vertex $v' \in V_q$ of label l_2 , there exists a vertex $u' \in V_g$ such that u' maps to v' and $\text{dist}_g(u, u') \leq \text{dist}_q(v, v')$. Thus, the proof is complete. $\blacksquare$

According to Lemma 5.2, we process a query by S-index as shown in Algorithm 5.4. For each distinct label pair (l_1, l_2) of q , we merge all the candidate sets associated with (l_1, l_2) that are smaller than $\mathcal{W}_q(l_1, l_2)$, which gives $\mathcal{C}(l_1, l_2)$. The candidate set of q is then obtained by intersecting $\mathcal{C}(l_1, l_2)$ for all distinct pairs (l_1, l_2) of q .

5.3.3.3 Complexity Analysis

Let m be the average number of distinct labels in a data graph. Thus, the number of distinct label pairs is $O(m^2)$, and the number of compressed weight multisets in the

database $\mathcal{G}$ is $O(m^2|\mathcal{G}|)$. Assume that the average number of vertices in a data graph is α , then the average size of a compressed weight multiset is α/m . Thus, the space complexity of S-index is $O(\alpha m|\mathcal{G}|)$.

Assume that the number of distinct labels in a query graph q is t . The index response time is the summation of the time for searching the compressed weight multisets of q , thus the running time complexity is $O(t^2 \log(m^2|\mathcal{G}|))$. Note that, in real life queries, the number of distinct labels is often small.

5.4 Experimental Evaluation

This section experimentally evaluates our indices and algorithms for generalized subgraph matching. Section 5.4.1 describes the experimental settings. Section 5.4.2 evaluates our algorithms for the generalized subgraph isomorphism problem, and Section 5.4.3 tunes the parameters for the proposed FP-Index. After that, Sections 5.4.4 and 5.4.5 demonstrate the efficiency of our indexing methods on real and synthetic datasets, respectively.

5.4.1 Experimental Settings

Datasets. We use two benchmark datasets commonly adopted in the literature [37, 101]. Both datasets contain graphs that represent chemical molecules. The first one is the AIDS Antiviral Screen Dataset [101], which consists of 10,000 graphs and 51 distinct vertex labels. Each molecule has 25.4 vertices and 27.4 edges on average. The second dataset is referred to as PubChem [101], and it contains 100,000 graphs, each of which has 24.0 vertices and 25.8 edges on average. The number of distinct vertex labels in PubChem is 81. We use PubChem. m K to denote a sample set of PubChem with m thousands of graphs. In addition, we use synthetic datasets produced from GraphGen [7], a publicly available synthetic graph generator.

Query sets. For the AIDS dataset, we adopt the query sets from [101], but we ignore the label on each edge and add a weight on the edge uniformly at random with the average weight varying from 1 to 1.5 (since we target at query graphs where the edges are unlabelled and weighted). For the other datasets, we generate the query sets by first extracting generalized subgraphs from the graphs in the datasets, such that the number of data graphs matching each extracted generalized subgraph is at most 10% of the total number of data graphs. In other words, we avoid generating generalized subgraph matching queries that would return excessive numbers of results, which can be easily computed using the frequent subgraph based indices. (Note that, queries with considerably small answers, e.g., empty answer sets, can be efficiently handled by either infrequent edge indices or the pruning techniques in the filtering phase.)

All of our experiments are conducted on a machine with a Intel Xeon 2.4GHz CPU with 48GB RAM.

Figure 5.2: Efficiency of generalized subgraph isomorphism algorithms on AIDS dataset.

5.4.2 Generalized Subgraph Isomorphism

Our first set of experiments compares three algorithms for generalized subgraph isomorphism: our cost-based approach (denoted as CBA), as well as the extensions of Ullmann's algorithm and QuickSI. Figure 5.2 illustrates the average running time required by each algorithm to match each query graph in query set Q_i to all data graphs in the AIDS dataset. In particular, each query set Q_i contains 1000 query graphs, and each query graph in Q_i contains i vertices. Figure 5.2a shows the results when the edges in the query graph have average weight 1.5. Observe that CBA considerably outperforms the extension of QuickSI, which in turn is superior than the extension of Ullman's algorithm. Figure 5.2b shows the results when the average edge weight in the query graph equals 1, i.e., when the generalized subgraph isomorphism problem degenerates to the standard subgraph isomorphism problem. Even in this degenerated case, CBA still consistently outperforms QuickSI and Ullmann's algorithm. This demonstrates the superiority of our cost-based method for optimizing vertex matching order. Note that, the running time of both QuickSI and CBA increase when the query set varies from Q_4 to Q_8 , since (i) Q_4 and Q_8 have a large number of query answers and (ii) the increase of the size of query graphs results in higher cost in subgraph matching; and their running time decreases when varying from Q_8 to Q_{24} , due to that (i) the number of answers to Q_i for $i \in (8, 24]$ become less and less and (ii) the subgraph matching algorithm terminates early thanks to the pruning techniques applied. We have conducted a similar set of experiments on the PubChem datasets, and we found that the results are qualitative similar; we omit those results for the interests of space.

(a) $\sigma = 0.05$, $\gamma = 4$. (b) $\sigma = 0.05$, $\gamma = 4$. (c) $\rho = 3$, $\gamma = 4$. (d) $\rho = 3$, $\gamma = 4$.

Figure 5.3: Space and pre-computation costs of the FP-index on PubChem dataset.

5.4.3 Tuning the FP-index

The second set of our experiments evaluates the space and pre-computation costs of the FP-index (presented in Section 5.3.2) on a set of 40 thousands data graphs sampled from the PubChem dataset. Figure 5.3a illustrates the number of Frequent Generalized subGraphs (FGG) that need to be stored in the FP-index, varying the maximum edge weight ρ in the graph patterns from 1 to 4, with the frequency threshold set to $\sigma = 0.05$ and the maximum number of vertices in the FGGs set to $\gamma = 4$. Note that the number of FGGs increases exponentially with maximum edge weight ρ . Figure 5.3b shows the time required to mine the FGGs, which also exhibits an exponential growth with the increase of ρ . These results indicate that maximum edge weight adopted in the construction in the FP-index has to be carefully selected and has to be reasonably small.

Figures 5.3c and 5.3d illustrate the number of FGGs and pre-computation time required by FP-index, respectively, varying the frequency threshold σ from 0.01 to 0.05, with the maximum edge weight set to $\rho = 3$ and the maximum number of vertices in the

Figure 5.4: Index performance v.s. dataset size on PubChem dataset.

FGGs set to $\gamma = 4$. Observe that both the number of FGGs and the pre-computation time decrease exponentially when the frequency threshold σ increases. Hence, we may use a large σ to reduce the space and pre-computation cost of the FP-index. One may be tempted to set σ to be even larger than 0.05, which, however, may significantly reduce the effectiveness of FP-index, as an excessively large σ would make it difficult for FP-index to answer a query without invoking the verification process.

Based on the results in Figure 5.3, we set $\rho = 3$, $\sigma = 0.05$, and $\gamma = 4$ for the FP-index in all following experiments.

5.4.4 Performance of Indices

Our next set of experiments compares the performance of the three proposed indices (i.e., D-index, S-index, and FP-index) in terms of query processing performance, space overhead, and construction time. For these experiments, we use sample sets of PubChem

Figure 5.5: Index performance v.s. parameters of the query graphs on PubChem.40K dataset.

dataset with sizes varying from 10K to 100K. We do not use the AIDS dataset as it contains only a small number of data graphs.

Figure 5.4a illustrates the average query processing time of each index for a query set with 1000 graphs, such that on average each graph has 5 vertices and 7 edges, and the average edge weight equals 2.5. Both the S-index and the FP-index significantly outperform the D-index, and the FP-index is slightly better than the S-index. This is consistent with the results in Figure 5.4b, which shows the average size of the candidate set induced by each index during query processing. As shown in Figures 5.4c and 5.4d, however, the space and construction overheads of FP-index are significantly higher than those of the S-index, which in turn are higher than those of the D-index.

Figure 5.5a shows the average query time of each index for query sets V_iE_j on the dataset with 40K data graphs, such that each query set V_iE_j contains 1000 query graphs, each of which has i vertices and j edges, and the average weight of the edges equals 2.5. The FP-index achieves the smallest query time when the numbers of vertices and edges in the query graphs are small, but it is outperformed by the S-index on large query graphs. This is because of the large overheads incurred in the decomposition of large query graphs in the query processing of FP-index. In addition, the D-index is consistently slower than both the FP-index and the S-index.

Figure 5.5b illustrates the average query time of each index for query set V_5E_7 , with the average edge weight varying from 1 to 5. The FP-index performs the best when the average edge weight is no more than 3, which is the maximum edge weight handled in its preprocessing step. When the average edge weight is larger than 3, however, the performance of the FP-index degrades, and the S-index becomes the most efficient one.

Figure 5.6: Index performance v.s. graph density on synthetic dataset.

5.4.5 Performance on Synthetic Dataset

The experiments use synthetic datasets to evaluate the performance of our indices with respect to a parameter that has not been investigated in the previous experiments, i.e., the densities of the data graphs. In particular, the density of a data graph with n vertices and m edges equals $m/\binom{n}{2}$. We generate synthetic graphs with densities varying from 0.3 to 0.7, and we use them to construct datasets, such that each dataset contains 10K data graphs, each of which has 30 edges and a fixed density. The query graphs for each dataset is constructed in a manner similar to previous experiments, such that each query graph on average has 5 vertices, 7 edges, with an average edge weight 2.5.

Figure 5.6 illustrates the performance of each index as a function of the data graph density. As with our previous experiments, the FP-index achieves the best query performance, but it incurs the highest space and pre-computation overheads. The D-index requires the smallest space and preprocessing time, but its query time is the largest. The S-index consistently lands on the middle ground between the FP-index and the D-index.

Summary. Our experiments show that the FP-index offers superior query performance at the cost of space and pre-computation time. Therefore, it is suitable for the applications where (i) efficient query processing is crucial, and (ii) space and pre-computation overheads are not a major concern. In contrast, the D-index entails relatively high query cost, but it incurs minimal space and preprocessing overhead. This renders it preferable in the scenarios with stringent requirements on space consumption or pre-computation time. Finally, the S-index’s space and pre-computation costs are only slightly higher than that of the D-index, but its query efficiency is almost comparable to that of the FP-index. Hence, it offers user a choice to strike a good balance between query processing and space (preprocessing) overheads.

5.5 Summary

We study a new type of graph queries, generalized subgraph queries. We propose a succinct and effective cost model to minimize the cost of generalized subgraph isomorphism. We also develop three indexes that can effectively filter out unmatching data graphs, which significantly reduces the total query response time. We evaluate our algorithms with experiments on both real datasets and synthetic datasets. The results show that our matching algorithm is efficient in candidate verification as it considerably outperforms the direct extension of existing graph matching algorithms, while our indexes are also effective in filtering. Thus, the results verify that our method is efficient in query processing (in both filtering and candidate verification). Although some of the indexes have weaknesses, we show how the weaknesses are addressed by another index; in particular, our results show that S-index achieves both a low index construction cost and a short query response time.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we study three problems on subgraph mining and query processing, each of which finds enormous real-world applications.

First, we study the problem of frequent subgraph mining in a large set of graphs. Previous approaches cannot handle large datasets in an efficient way due to a significant number of subgraphs that will be generated. To address this problem, we propose a two-step filter-and-refinement MapReduce framework, which is highly suitable to massive parallelization. In order to reduce the overhead in communication cost, we devise a statistical model for carefully selecting frequent subgraph candidates, as well as a lightweight technique for graph compression that significantly reduces the amount of data transmission between the worker nodes. In addition, we propose an efficient strategy for reusing computation in the expensive process of isomorphism testing for graphs that share edges. Through extensive experiments on several large real-world datasets, we demonstrate that the proposed MapReduce algorithm for frequent subgraph mining scales well for large datasets and significantly outperforms previous approaches.

Then, we investigate the problem of network motif discovery, and propose a fast GPU-based solution to the problem. To the best of our knowledge, we are the first to devise efficient algorithms for network motif discovery on GPU. Compared with existing CPU-based approaches, our solution is drastically different due to the design choices that we make to exploit the strengths of GPUs in terms of parallelism, and mitigate their limitations in terms of the computation power per GPU core. Besides, we develop three optimization techniques that improve the scalability of our solution, effectively avoid under-utilization of the GPU, and eliminate redundant computation in network motif discovery. These techniques considerably improve the scalability and efficiency of our solution. With extensive evaluations on a variety of biological networks, we show that our solution is much more efficient than the CPU-based solutions by several orders

of magnitude, and is more cost-effective than the latter, when taking into account the monetary costs of the CPU and GPUs used.

Finally, we study a new type of graph queries, namely generalized subgraph queries. To minimize the cost of generalized subgraph isomorphism testing, we propose a succinct and effective cost model for careful selection of vertex matching order. With the filter-and-verification framework, we develop three indexes that can effectively filter out unmatching data graphs, which significantly reduces the total query response time. These indexes have different strengths that make them applicable to different scenarios and also complementary to each other. We evaluate our algorithms with experiments on both real datasets and synthetic datasets. The results verify the efficiency of our matching algorithm and the effectiveness of our indexes.

6.2 Future Work

We now list some potential problems for future work, as follows.

6.2.1 Frequent subgraph mining in single-large-graph scenario

In contrast to the frequent subgraph mining problem in transaction setting, the one in single-large-graph scenario would be drastically different.

First, the definition of subgraph frequency leaves a huge discussion room, as illustrated in the work [78], depending on whether we allow edge/vertex overlaps between occurrences that are isomorphic to the subgraph. If edge/vertex overlaps are allowed, the downward closure property (see Section 2.1) no longer holds which renders most existing algorithms in the transaction setting unable to solve this problem.

Second, given a large graph G as the database and a small graph g , to calculate g 's frequency in G , we need to compute g 's occurrences in G . However, there might exist too many duplicates due to graph isomorphism and graph symmetry. Besides, the number of g 's occurrences would be extensively large, which make them impossible to compute efficiently in memory.

Third, when deploying the algorithms in single-large-graph scenario in distributed parallel computing environment, it would be necessary to partition the graph among machines. However, partitioning a large graph is difficult since (i) it might easily lead to workload unbalance among machines, and (ii) it might incur significant communication overhead between machines due to data transmission.

6.2.2 Dense subgraph discovery on GPU

Given a graph, we say that it is dense if its *density* is above a pre-defined threshold. The problem of dense subgraph discovery is to find all subgraphs of a given large graph G such that each subgraph is dense. This is different from network motif discovery, which finds all subgraphs in G whose statistical values are above a user-defined threshold. Dense subgraph discovery has tremendous applications, such as social network analysis and community detection. However, designing efficient algorithms for dense subgraph discovery is highly challenging, due to the extensive number of subgraphs and the large size of input graph. And it is even more challenging when the algorithms are to run efficiently on GPUs, but it is a great chance for us to improve the algorithms in a cost-efficient way since the GPU is highly capable in parallelism.

To design efficient algorithms for dense subgraph discovery on the GPU, several critical issues should be addressed. First, since a dense subgraph (e.g., clique) might be enumerated several times from different parts of the graph, the redundant computation should be avoided in an efficient way. Second, as the number of dense subgraphs would be substantially large, whereas the size of GPU's memory are often small, the scalability solutions are required to handle the intermediate results that cannot fit in GPU's memory. Finally, since the running time of the GPU program depends on the slowest GPU threads, it is critical to achieve workload balance and memory coalescing, and avoid branch divergences, as discussed in Section 4.1.2.

6.2.3 Subgraph query processing in single-large-graph scenario

The problem of subgraph query processing in single-large-graph scenario is significantly different from the one in transaction setting, since (i) the former is often to ask for the set of occurrences of the query graph in a large graph, and (ii) the filter-and-verification framework is no longer effective in this scenario. Most existing work [36, 90] are the *backtracking* approaches that traverse the search space in the DFS manner. However, these approaches are difficult to be parallelized. Some other work [63, 82, 87] adopt the join approaches that compute the occurrences of the query graph Q by joining the occurrences of Q 's subgraphs. Due to the uneven distribution of subgraphs and the large number of edges that are cut in the query graph decomposition, these approaches often suffer from issues in workload unbalance and significant overhead in communication between machines. As such, there still exists much room for improvement. Besides, it is interesting to investigate the problem of generalized subgraph matching in the single-large-graph scenario, which finds numerous applications but is highly challenging.

Author's Publications

Papers published

- (i) Andy Diwen Zhu, **Wenqing Lin**, Sibor Wang, and Xiaokui Xiao. “Reachability Queries on Large Dynamic Graphs: A Total Order Approach”. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD)*, pages 1323-1334, 2014.
- (ii) **Wenqing Lin**, Xiaokui Xiao, and Gabriel Ghinita. “Large-Scale Frequent Subgraph Mining in MapReduce”. In *Proceedings of the 30th IEEE International Conference on Data Engineering (ICDE)*, pages 844-855, 2014.
- (iii) Yufei Tao, **Wenqing Lin**, and Xiaokui Xiao. “Minimal MapReduce algorithms”. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD)*, pages 529-540, 2013.
- (iv) Lingkun Wu, **Wenqing Lin**, Xiaokui Xiao, and Yabo Xu. “LSII: An indexing structure for exact real-time search on microblogs”. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*, pages 482-493, 2013.
- (v) Andy Diwen Zhu, Xiaokui Xiao, Sibor Wang, and **Wenqing Lin**. “Efficient Single-Source Shortest Path and Distance Queries on Large Graphs”. In *Proceedings of ACM SIGKDD Conference on Knowledge, Discovery, and Data mining (KDD)*, pages 998-1006, 2013.
- (vi) **Wenqing Lin**, Xiaokui Xiao, James Cheng, and Sourav S. Bhowmick. “Efficient algorithms for generalized subgraph query processing”. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM)*, pages 325-334, 2012.

Papers to appear

- (i) **Wenqing Lin**, Xiaokui Xiao, Xing Xie, and Xiao-Li Li. “Network Motif Discovery: A GPU Approach”. In *Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE)*, to appear, 2015.

References

- [1] Akos. <http://www.akosgmbh.de>.
- [2] Amazon ec2. <http://aws.amazon.com/ec2>.
- [3] Ambinter. <http://www.ambinter.com>.
- [4] Chemdb. <http://cdb.ics.uci.edu>.
- [5] Enamine. <http://www.enamine.net>.
- [6] Genemania. <http://genemania.org>.
- [7] graphgen. <http://www.cse.ust.hk/graphgen/>.
- [8] Hadoop. <http://hadoop.apache.org>.
- [9] Hadoop streaming. <http://hadoop.apache.org/docs/stable/streaming.html>.
- [10] Nci. <http://cactus.nci.nih.gov/download/nci>.
- [11] Pubchem. <http://pubchem.ncbi.nlm.nih.gov>.
- [12] Quatexelero. <http://lbb.ut.ac.ir/Download/LBBsoft/QuateXelero/>.
- [13] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *ICDE*, pages 62–73, 2013.
- [14] C. C. Aggarwal and H. Wang, editors. *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*. Springer, 2010.
- [15] I. Albert and R. Albert. Conserved network motifs allow protein-protein interaction prediction. *Bioinformatics*, 20(18):3346–3352, 2004.

- [16] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp. Biomolecular network motif counting and discovery by color coding. In *ISMB*, pages 241–249, 2008.
- [17] I. Bordino, D. Donato, A. Gionis, and S. Leonardi. Mining large networks with subgraph counting. In *ICDM*, pages 737–742, 2008.
- [18] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *ICDM*, pages 51–58, 2002.
- [19] G. Buehrer, S. Parthasarathy, and Y.-K. Chen. Adaptive parallel graph mining for cmp architectures. In *ICDM*, pages 97–106, 2006.
- [20] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu. Towards graph containment search and indexing. In *VLDB*, pages 926–937, 2007.
- [21] J. Cheng, Y. Ke, A. W.-C. Fu, and J. X. Yu. Fast graph query processing with a low-cost index. *VLDB J.*, 20(4):521–539, 2011.
- [22] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD Conference*, pages 857–872, 2007.
- [23] D. W.-L. Cheung, J. Han, V. T. Y. Ng, A. W.-C. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *PDIS*, pages 31–42, 1996.
- [24] D. J. Cook, L. B. Holder, G. Galal, and R. Maglothin. Approaches to parallel graph-based knowledge discovery. *J. Parallel Distrib. Comput.*, 61(3):427–446, 2001.
- [25] S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158, 1971.
- [26] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.
- [27] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
- [28] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, pages 39–50, 2011.

- [29] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. *PVLDB*, 3(1):1161–1172, 2010.
- [30] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo. Frequent itemset mining on graphics processors. In *DaMoN*, pages 34–42, 2009.
- [31] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- [32] J. A. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *RECOMB*, pages 92–106, 2007.
- [33] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.
- [34] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD Conference*, pages 1–12, 2000.
- [35] T. D. Han and T. S. Abdelrahman. Reducing branch divergence in gpu programs. In *GPGPU*, page 3, 2011.
- [36] W.-S. Han, J. Lee, and J.-H. Lee. Turbo_{iso}: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD Conference*, pages 337–348, 2013.
- [37] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. ighrph: A framework for comparisons of disk-based graph indexing techniques. *PVLDB*, 3(1):449–459, 2010.
- [38] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, page 38, 2006.
- [39] H. He and A. K. Singh. Graphrank: Statistical modeling and mining of significant subgraphs in the feature space. In *ICDM*, pages 885–890, 2006.
- [40] S. Hill, B. Srichandan, and R. Sunderraman. An iterative mapreduce approach to frequent subgraph mining in biological datasets. In *BCB*, pages 661–666, 2012.
- [41] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM*, pages 549–552, 2003.

- [42] J. Huan, W. Wang, J. Prins, and J. Yang. Spin: mining maximal frequent subgraphs from graph databases. In *KDD*, pages 581–586, 2004.
- [43] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, pages 13–23, 2000.
- [44] S. Itzkovitz, R. Levitt, N. Kashtan, R. Milo, M. Itzkovitz, and U. Alon. Coarse-graining and self-dissimilarity of complex networks. *Phys. Rev. E*, 71:016127, 2005.
- [45] C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. *Knowledge Eng. Review*, 28(1):75–105, 2013.
- [46] X. Jiang, H. Xiong, C. Wang, and A.-H. Tan. Mining globally distributed frequent subgraphs in a single labeled graph. *Data Knowl. Eng.*, 68(10):1034–1058, 2009.
- [47] W. Jin and J. Yang. A flexible graph pattern matching framework via indexing. In *SSDBM*, pages 293–311, 2011.
- [48] C. L. S. John W. Moore and P. C. Jurs. *Chemistry: The Molecular Science*, volume 2. Brooks Cole, 2007.
- [49] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. *TKDD*, 5(2):8, 2011.
- [50] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009.
- [51] Z. R. M. Kashani, H. Ahrabian, E. Elahi, A. Nowzari-Dalini, E. S. Ansari, S. Asadi, S. Mohammadi, F. Schreiber, and A. Masoudi-Nejad. Kavosh: a new algorithm for finding network motifs. *BMC Bioinformatics*, 10:318, 2009.
- [52] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics*, 20(11):1746–1758, 2004.
- [53] H. Kettani. On the conversion between number systems. In *MSV/AMCS*, pages 317–320, 2004.
- [54] S. Khakabimamaghani, I. Sharafuddin, N. Dichter, I. Koch, and A. Masoudi-Nejad. Quatexelero: An accelerated exact network motif detection algorithm. *PLoS ONE*, 8(7):e68073, 07 2013.

- [55] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.
- [56] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM*, pages 313–320, 2001.
- [57] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. In *SDM*, 2004.
- [58] M. Kuramochi and G. Karypis. Grew-a scalable frequent subgraph discovery algorithm. In *ICDM*, pages 439–442, 2004.
- [59] E. Li and L. Liu. Optimization of frequent itemset mining on multiple-core processor. In *VLDB*, pages 1275–1285, 2007.
- [60] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang. Pfp: parallel fp-growth for query recommendation. In *RecSys*, pages 107–114, 2008.
- [61] X. Li, D. S. Stones, H. Wang, H. Deng, X. Liu, and G. Wang. Netmode: Network motif detection without nauty. *PLoS ONE*, 7(12):e50093, 12 2012.
- [62] J.-L. Lin and M. H. Dunham. Mining association rules: Anti-skew algorithms. In *ICDE*, pages 486–493, 1998.
- [63] S. Ma, Y. Cao, J. Huai, and T. Wo. Distributed graph pattern matching. In *WWW*, pages 949–958, 2012.
- [64] A. Maunz, C. Helma, and S. Kramer. Large-scale graph mining using backbone refinement classes. In *KDD*, pages 617–626, 2009.
- [65] B. D. McKay and A. Piperno. Practical graph isomorphism, ii. *J. Symb. Comput.*, 60:94–112, 2014.
- [66] I. Miliaraki, K. Berberich, R. Gemulla, and S. Zoupanos. Mind the gap: large-scale frequent sequence mining. In *SIGMOD Conference*, pages 797–808, 2013.
- [67] R. Milo, N. Kashtan, S. Itzkovitz, M. E. J. Newman, and U. Alon. On the uniform generation of random graphs with prescribed degree sequences. *Arxiv preprint cond-mat/0312028*, 2003.

- [68] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, October 2002.
- [69] S. N. Nguyen, M. E. Orlowska, and X. Li. Graph mining based on a data partitioning approach. In *ADC*, pages 31–37, 2008.
- [70] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *KDD*, pages 647–652, 2004.
- [71] S. Omid, F. Schreiber, and A. Masoudi-Nejad. MODA: an efficient algorithm for network motif discovery in biological networks. *Genes & genetic systems*, 84(5):385–395, 2009.
- [72] S. Ranu and A. K. Singh. Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In *ICDE*, pages 844–855, 2009.
- [73] S. Reinhardt and G. Karypis. A multi-level parallel implementation of a program for finding frequent patterns in a large sparse graph. In *IPDPS*, pages 1–8, 2007.
- [74] P. M. P. Ribeiro and F. M. A. Silva. g-tries: an efficient data structure for discovering network motifs. In *SAC*, pages 1559–1566, 2010.
- [75] P. M. P. Ribeiro, F. M. A. Silva, and M. Kaiser. Strategies for network motifs discovery. In *eScience*, pages 80–87, 2009.
- [76] P. M. P. Ribeiro, F. M. A. Silva, and L. M. B. Lopes. Parallel discovery of network motifs. *J. Parallel Distrib. Comput.*, 72(2):144–154, 2012.
- [77] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB*, pages 432–444, 1995.
- [78] F. Schreiber and H. Schwöbbermyer. Frequency concepts and pattern detection for the analysis of motifs in networks. pages 89–104, 2005.
- [79] H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang. Connected substructure similarity search. In *SIGMOD Conference*, pages 903–914, 2010.
- [80] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

- [81] H. Shang, K. Zhu, X. Lin, Y. Zhang, and R. Ichise. Similarity search on supergraph containment. In *ICDE*, pages 637–648, 2010.
- [82] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In *SIGMOD Conference*, pages 625–636, 2014.
- [83] D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52, 2002.
- [84] E. Sintorn and U. Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, 2008.
- [85] R. Sole and S. Valverde. Are network motifs the sprandrels of cellular complexity? *Trends Ecol. Evol.*, 21(8):419–422, 2006.
- [86] O. Sporns and R. Ktter. Motifs in brain networks. *PLoS Biol*, 2(11):e369, 10 2004.
- [87] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.
- [88] Y. Tao, W. Lin, and X. Xiao. Minimal mapreduce algorithms. In *SIGMOD Conference*, pages 529–540, 2013.
- [89] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *KDD*, pages 837–846, 2009.
- [90] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.
- [91] J. H. Van Drie, D. Weininger, and Y. C. Martin. Aladdin: An integrated tool for computer-assisted molecular design and pharmacophore recognition from geometric, steric, and substructure searching of three-dimensional molecular structures. *Journal of Computer-Aided Molecular Design*, 3:225–251, 1989.
- [92] C. Wang, W. Wang, J. Pei, Y. Zhu, and B. Shi. Scalable mining of large disk-based graph databases. In *KDD*, pages 316–325, 2004.
- [93] T. Wang, J. W. Touchman, W. Zhang, E. B. Suh, and G. Xue. A parallel algorithm for extracting transcription regulatory network motifs. In *BIBE*, pages 193–200, 2005.

- [94] S. Wernicke. Efficient detection of network motifs. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 3(4):347–359, 2006.
- [95] S. Wernicke and F. Rasche. Fanmod: a tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, 2006.
- [96] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *ICDE*, pages 976–985, 2007.
- [97] B. Wu and Y. Bai. An efficient distributed subgraph mining algorithm in extreme large graphs. In *AICI (1)*, pages 107–115, 2010.
- [98] J. Xiang, C. Guo, and A. Aboulnaga. Scalable maximum clique computation using mapreduce. In *ICDE*, pages 74–85, 2013.
- [99] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by leap search. In *SIGMOD Conference*, pages 433–444, 2008.
- [100] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.
- [101] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD Conference*, pages 335–346, 2004.
- [102] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD Conference*, pages 766–777, 2005.
- [103] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou. Comparing stars: On approximating graph edit distance. *PVLDB*, 2(1):25–36, 2009.
- [104] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. In *ICDE*, pages 966–975, 2007.
- [105] S. Zhang, J. Li, H. Gao, and Z. Zou. A novel approach for efficient supergraph query processing on graph databases. In *EDBT*, pages 204–215, 2009.
- [106] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta $\geq$ graph. In *VLDB*, pages 938–949, 2007.
- [107] X. Zhao, C. Xiao, X. Lin, and W. Wang. Efficient graph similarity joins with edit distance constraints. In *ICDE*, pages 834–845, 2012.

- [108] Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. S. A. Kumar, and M. V. Marathe. Sahad: Subgraph analysis in massive networks using hadoop. In *IPDPS*, pages 390–401, 2012.
- [109] L. Zou, L. Chen, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.
- [110] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *EDBT*, pages 181–192, 2008.