Collège Doctoral Languedoc-Roussillon

THESIS

To obtain the degree of
Doctor of Philosophy (Ph.D.)

Awarded by **University of Montpellier**

Prepared at the Graduate School of
INFORMATION, STRUCTURES AND SYSTEMS (**I2S**),
LIRMM Research Unit, **ADVANSE** Team.

Speciality: **COMPUTER SCIENCE**

Defended by **Mr. Vijay INGALALLI**
vijay@lirmm.fr

Querying and Mining Multigraphs

Defended on 27/02/2017 in front of a jury composed of:

`Information is liberating.   Knowledge is power.`

Abstract

With the ever increasing growth of data and information, extracting the right knowledge has become a real challenge. Further, the advanced applications demand for the analysis of complex and interrelated data which cannot be adequately described using a propositional representation. The graph representation is of great interest for the knowledge extraction community, since graphs are versatile data structures and are one of the most general forms of data representation. Among several classes of graphs, *multigraphs* have been captivating the attention in the recent times, thanks to their inherent property of succinctly representing the entities by allowing the rich and complex relations among them.

The focus of this thesis is streamlined into two themes of knowledge extraction; one being *knowledge retrieval*, where the focus is on the subgraph query matching aspects in multigraphs, and the other being *knowledge discovery*, where the focus is on the problem of frequent pattern mining in multigraphs.

This thesis makes three main contributions in the field of query matching and data mining. The first contribution deals with querying subgraphs in multigraphs that yields isomorphic matches, and this problem finds potential applications in the domains of remote sensing, social networks, bioinformatics, and chemical informatics. The second contribution, which focusses on knowledge graphs, deals with querying subgraphs in RDF multigraphs that yields homomorphic matches. In both the contributions, efficient indexing structures are introduced that capture the multiedge information. The proposed query matching processes have been carefully optimized for improved time performance and the proposed heuristics assure robust performance. The third contribution is in the field of data mining, where an efficient frequent pattern mining algorithm for multigraphs is proposed. We observe that multigraphs pose challenges while exploring the search space, and hence novel optimization techniques and heuristic search methods are introduced to swiftly traverse the search space.

For each proposed approach, extensive experimental analysis is performed by comparing with the existing state-of-the-art approaches in order to validate the performance and correctness of the proposed approaches. In the end, a case study analysis is performed on a remote sensing dataset, where the dataset is modelled as a multigraph, and the mining and query matching processes are employed to discover some useful knowledge.

Résumé

Avec des volumes de données et d'informations de plus en plus importants, des données de plus en plus complexes et fortement inter-reliées, l'extraction de connaissances reste un véritable défi. Les graphes offrent actuellement un support de représentation efficace pour représenter ces données. Parmi les approches existantes, les multi-graphes ont montré que leur pouvoir d'expression était particulièrement adapté pour manipuler des données complexes possédant de nombreux types de relations entre elles.

Cette thèse aborde deux aspects principaux liés aux multigraphes : la recherche de sous graphes et la fouille de sous graphes fréquents dans des multigraphes.

Elle propose trois propositions dans le domaines du requêtage et de la fouille de données. La première contribution s'inscrit dans la recherche de sous graphes et concerne l'isomorphisme de sous graphes dans des multigraphes. Cette approche peut, par exemple, être appliquée dans de nombreux domaines d'applications comme l'analyse d'images satellites ou de réseaux sociaux. Dans la seconde, nous nous intéressons aux graphes de connaissances et abordons la problématique de l'homorphisme de graphes dans des multigraphes RDF. Dans les deux contributions, nous proposons de nouvelles techniques d'indexations pour représenter efficacement les informations contenues dans les multigraphes. La recherche des sous graphes tire avantage de ces nouveaux index et différentes heuristiques et optimisations sont également proposées pour garantir de bonnes performances lors de l'exécution des requêtes. La seconde contribution s'inscrit dans le domaine de la fouille de données et nous proposons un algorithme efficace pour extraire les multigraphes fréquents. Etant donné l'espace de recherche à considérer, la recherche de motifs fréquents dans des graphes est un problème difficile en fouille de données. Pour parcourir efficacement l'espace de recherche encore plus volumineux pour les multigraphes, nous proposons de nouvelles techniques et méthodes pour le traverser efficacement notamment en éliminant des candidats où détectant à l'avance les motifs non fréquents.

Pour chacune de ces propositions de nombreuses expérimentations sont réalisées pour valider à la fois leurs performances et exactitudes en les comparant avec les approches existantes. Finalement, nous proposons une étude de cas sur des jeux de données issues d'images satellites modélisées sous la forme de multigraphe et montrons que l'application de nos propositions permet de mettre en évidence de nouvelles connaissances utiles.

Acknowledgements

First and foremost I would like to thank my supervisor Prof. Pascal Poncelet and co-supervisor Dr. Dino Ienco for providing me an opportunity to accomplish this doctoral research with them. It is their immense support and belief in me that has made this thesis possible.

Working on my PhD has been a research odyssey with several crests and troughs, hoping to see the light at the end of the tunnel. "We are almost seeing the light, Vijay", Dino used to say, whenever we were about to finish a particular task. Whenever I got stuck in the details of a problem and lost orientation, Pascal provided me with the right direction unveiling the otherwise concealed general problem. The board room discussions with Pascal and Dino yielded timely insights, whenever I was mired in the details of a problem. Both Pascal and Dino have been proactive in supporting my work throughout the PhD.

I am grateful to have been part of the vibrant *Advanse* team in LIRMM, where I have met some wonderful people. The valuable suggestions I received from my colleagues has shaped me in many ways. The discussions and debates that I have had with them will hardly ever fade away as much as the humorous moments that we shared.

I am also thankful to the many people from LIRMM whom I befriended during my PhD; I really spent many memorable moments with them that I will cherish for a long time. I am also thankful to the people and colleagues from MTD lab, where I worked sometimes.

I would like to thank all the people responsible for the administrative aspects of my PhD at LIRMM, MTD and University of Montpellier. In particular, I am thankful to Mr. Nicolas Serrurier and Ms. Guylaine Martinoty who have helped immensely for all the bureaucracy that I had to confront. I also express my gratitude for the NUMEV[1] project, which has allowed me to pursue this research with the much needed support in the form of generous funding, and facilitating me to attend various conferences as well as several summer/winter schools.

I also feel privileged to have made many friends outside the lab, and am thankful to them for being with me through thick and thin. Many of my friends who live in different places of the world have been so supportive of me all this time, and I thank them for believing in me. I am delighted to be still in contact with them.

[1]Labex NUMEV (NUMEV, ANR-10-LABX-20).

Last but not least, I am extremely indebted to my family who, although are living far away in India, have held me close all this time. My father, in particular, has been very excited about my PhD studies. Being a professor of Philosophy, his keen interest in the field of computer science paved the way for many interesting discussions over the course of my PhD. I owe it to my father, for any philosophical tone in the thesis. My mother, on the other hand, is a wonderful human. I never felt being far away from her for she always embraced me through all the situations that I had to confront here. Our frequent talks were warm enough to keep me going. My brother and his wife have been a lovely couple, to whom I owe a lot for keeping me sane throughout my PhD. Many a time we discussed about work and many other times, we discussed about life. I feel elated to have a brother, who is such a dear friend that I can share everything with him.

I am also thankful to all the people who have been part of this journey, directly or indirectly, at various times.

Vijay Ingalalli
Montpellier, February 2017

Contents

Abstract **i**

Acknowledgements **iv**

List of Figures **ix**

List of Tables **xii**

Context **1**

1 Introduction **3**
- 1.1 Principles of Knowledge Extraction 3
  - 1.1.1 Knowledge Retrieval 4
  - 1.1.2 Knowledge Discovery 5
- 1.2 Managing and Mining Graph Data 7
- 1.3 Contributions 10
- 1.4 Organization of Thesis 11

2 Foundations **15**
- 2.1 Graphs, Multigraphs 15
- 2.2 Graph Matching 17
- 2.3 Graph Databases, Indexes 19
- 2.4 On Querying Graphs 20
- 2.5 Graph Data Mining 21
  - 2.5.1 Basics of FSM 22
  - 2.5.2 FSM Approaches 23

I Query Matching in Multigraphs **27**

Overview Part I **28**

3 Subgraph Query Matching in Multigraphs **31**
3.1 Introduction 31
3.2 Related Work 32
3.3 Problem Definition 34
3.4 An Overview of SuMGra 36
3.5 Indexing 37
3.5.1 Offline Index Construction 38
3.6 Subgraph Query Processing 41
3.6.1 Query Vertex Ordering 42
3.6.2 Select Candidates for Initial Query Vertex 43
3.6.3 Subgraph Searching 43
3.7 Experimental Evaluation 47
3.7.1 Description of Datasets 47
3.7.2 Performance of SuMGra 50
3.8 Summary 57

4 Querying RDF Data **59**
4.1 Introduction 59
4.2 Related Work 61
4.3 Background and Preliminaries 62
4.3.1 RDF Data 62
4.3.2 SPARQL Query 64
4.3.3 SPARQL Querying by Adopting Multigraph Homomorphism . 67
4.4 AMBER: A SPARQL Querying Engine 68
4.5 Index Construction 69
4.5.1 Attribute Index 69
4.5.2 Vertex Signature Index 70
4.5.3 Vertex Neighbourhood Index 72
4.6 Query Matching Procedure 73
4.6.1 Vertex Level Processing 75
4.6.2 Processing Satellite Vertices 76
4.6.3 Arbitrary Query Processing 78
4.7 Experimental Evaluation 81
4.7.1 Experimental Setup 82
4.7.2 Workload Generation 82
4.7.3 Comparison with RDF Engines 83
4.8 Summary 88

Conclusions Part I **90**

II Mining Multigraphs 93

Overview Part II 94

5 Frequent Pattern Mining in Multigraphs 97
5.1 Introduction 97
5.2 Related Work 98
5.3 Preliminaries and problem definition 99
5.4 MuGraM: A frequent pattern mining algorithm 101
5.4.1 Multi-edge representation and pattern enumeration 101
5.4.2 Overview of MuGraM 102
5.4.3 Search space spanned by DFS traversal 103
5.4.4 Discovering frequent patterns 104
5.4.5 Pattern extension 105
5.4.6 Support computation for multigraphs 107
5.5 Experimental Evaluation 114
5.5.1 Quantitative analysis: Time performance evaluation 114
5.5.2 Qualitative analysis 118
5.6 Summary 121

Conclusions Part II 122

Case Study 124

6 Knowledge Extraction for Remote Sensing Data 127

Summary 131

7 Summary and Future Directions 133
Thesis Summary 133
Future Works 135

Bibliography 139

List of Figures

1.1 Process of knowledge discovery in databases (KDD) 6
1.2 An instance of a multigraph 9

3.1 A sample example of subgraph embeddings 35
3.2 Equivalent representation of multigraph 36
3.3 Neighbourhood Index for a data vertex 42
3.4 Characteristics of datasets 47
3.5 Time for querying *DBLP* dataset 52
3.6 Time for querying *BIOGRID* dataset 53
3.7 Time for querying *FLICKR* dataset 54
3.8 Time for querying *YOUTUBE* dataset 55
3.9 Time for querying *DBPEDIA* dataset 55
3.10 Time for querying *DBLP* dataset by varying synopses fields 56
3.11 Time for querying *Synthetic* dataset 57

4.1 RDF data in both n-triple and multigraph reprentation 63
4.2 SPARQL query in both n-triple and multigraph reprentation 66
4.3 The AMbER Framework 68
4.4 Neighbourhood index structure 72
4.5 Query decomposition 74
4.6 A star structure in a multigraph 76
4.7 Star-shaped query evaluation for *DBPEDIA* dataset 84
4.8 Complex-shaped query evaluation for *DBPEDIA* dataset 85
4.9 Star-shaped query evaluation for *YAGO* dataset 86
4.10 Complex-shaped query evaluation for *YAGO* dataset 86
4.11 Star-shaped query evaluation for *LUBM100* dataset 87
4.12 Complex-shaped query evaluation for *LUBM100* dataset 87

5.1 Multigraph patterns of a data multigraph 98
5.2 Embeddings of a pattern 100
5.3 Search space for a set of frequent seeds 103
5.4 Various possible extensions of a pattern and automorphic grouping . 106
5.5 Time performance of *Amazon* and *DBLP* datasets 116
5.6 Time performance of *Citeseer* and *Microsoft* datasets 117

5.7 Time performance Vs. # of outputted patterns for *DBLP* dataset . . 118
5.8 Few interesting patterns for *AUCS* dataset 119
5.9 Few interesting patterns for *ATN* dataset 119
5.10 Few interesting patterns for *MRM* dataset 120

6.1 A satellite image of *Basse Plaine de l'Aude* (BPA) 128
6.2 Two frequent patterns for the BPA dataset 129
6.3 Embeddings of the frequent patterns in the original BPA image . . . 130
6.4 Zoomed original image to study the validity of the extracted knowledge 130

List of Tables

1.1 A comparison of Data, Information and Knowledge Retrieval 5

3.1 Vertex signatures for the data graph 38
3.2 Synopses for the data vertices 39
3.3 Statistics of datasets 48
3.4 Execution time and memory usage for index construction 51

4.1 Dictionary look-up table 67
4.2 Vertex signatures and the corresponding synopses 70
4.3 Statistics of RDF Datasets 82
4.4 Index construction time 83

5.1 Properties of the graph datasets 115

Context

Where is the Life we have lost in living?
Where is the Wisdom we have lost in knowledge?
Where is the Knowledge we have lost in information?
— *T.S. Eliot, The Rock, 1934.*

Chapter 1

Introduction

In this chapter we briefly introduce the concepts of knowledge retrieval and discovery. We then discuss about managing graph data along with the mining operations and the importance of exploring multigraphs - a generic class of graphs. Finally, we discuss the contributions and organization of the thesis.[1]

1.1 Principles of Knowledge Extraction

Knowledge Extraction deals with discovering knowledge from either structured (relational databases), semi-structured (graphs, trees, XML) or unstructured (text, documents, images) sources of data. The discovered knowledge needs to be in a machine-readable and machine-interpretable format and must represent knowledge in a manner that unambiguously defines its meaning, and facilitates inferencing [Unbehauen et al., 2012].

At the outset, it is essential to understand the subtleties in the meaning of data, information and knowledge, as described in [Bellinger et al., 2004]. *Data* is a set of values of certain variables that could be both qualitative or quantitative. Data that is collected from various domains is often raw, and unprocessed, and might not have meaning in itself. *Information* is the data that is processed to be useful, and has been given meaning by a way of relational connection, and can provide answers to questions. A trivial example is relational database or a graph database. *Knowledge*

[1]This work has been funded by Labex NUMEV (NUMEV, ANR-10-LABX-20)

is an understanding of the information, which is acquired through experience or discovery, such that its intent can be put to use.

Since the notion of knowledge extraction is very generic, this thesis focuses on two widely studied thematics of the process of knowledge extraction: (i) *knowledge retrieval*, a field of retrieving a specific knowledge from the data, where the data often might be organized as a database; (ii) *knowledge discovery*, a field where knowledge is discovered from the data, hitherto unknown.

1.1.1 Knowledge Retrieval

Knowledge Retrieval (KR) systems are advanced systems capable of retrieving knowledge, rather than mere information - as observed in the information retrieval (IR) systems. In IR systems, once information is retrieved, one has to go through the laborious work of finding meaning or knowledge from the retrieved information. The goal of knowledge retrieval systems is to reduce the burden of those processes by improved search and representation. KR systems focus at the knowledge level, and thus, we need to examine how to extract, represent, and use the knowledge in data and information [Bellinger et al., 2004]. While information retrieval systems organize the data and documents by indexing, knowledge retrieval systems organize information by indicating connections between elements in those documents. Further, knowledge retrieval systems provide knowledge to users in a structured way, since they focus on semantics and better organization of information. Such KR systems will be used by advanced and expert users to tackle the challenging problem of knowledge seeking [Yao et al., 2007].

Growth and evolution of the Web has opened a plethora of opportunities, making knowledge retrieval systems a necessity for supporting the future generations of the Web. Over the time, other domains such as - remote sensing, bioinformatics, chemistry, have also become domains of interest for knowledge retrieval systems. Many research works have been proposed [Kamel and Quintana, 1990, Martin and Eklund, 2000, Yao, 2002], that cover diverse aspects and provide us insights into further development of knowledge retrieval.

Since the task of retrieving knowledge not only involves gathering the available information but also enriching it with other information to gain knowledge, the process is rather challenging. In Table 1.1, the salient features of knowledge retrieval are compared with the data and information retrieval. As we observe, from retrieval model perspective, knowledge retrieval systems focus on the semantics and knowledge organization. Further, knowledge retrieval systems organize knowledge by allowing connections among knowledge structures, whereas, information retrieval systems organize the data by indexing. Knowledge retrieval is also based on partial match (approximate matching) and best match (exact matching).

$\longleftrightarrow$	Data Retrieval	Information Retrieval	Knowledge Retrieval
Match	Boolean match	partial match, best match	partial match, best match
Inference	deductive inference	inductive inference	deductive/inductive inference, associative reasoning
Model	deterministic model	statistical and probabilistic model	semantic model + inference model
Query	artificial language	natural language	knowledge structure + natural language
Organization	table, index	table, index	knowledge unit and knowledge structure
Representation	number, rule	natural language, markup language	concept graph, semantic network, ontology
Storage	database	document collections	knowledge base
Retrieved Results	data set	sections or documents	a set of knowledge unit

Table 1.1: A Comparison of Data, Information, and Knowledge Retrieval[2]

In this thesis, we focus on the querying aspects of knowledge retrieval, and in particular, *exact matching* of the queries in the case of semi-structured data of graphs. In Section 1.2, we discuss about an interesting and generic class of graphs called *multigraphs*, which are the focus of this thesis.

1.1.2 Knowledge Discovery

The traditional approach for turning low-level data into high-level knowledge relies on thorough manual analysis performed by specialised data analysts. Modern techniques allow an analyst to easily use the huge amounts of data available to test formulated hypothesis. However, to discover novel, interesting knowledge from the available data, the number of possible questions to be formulated and evaluated might simply exceed the capability of a human analyst. The process of analysing millions of data records is a tedious task and demands for techniques to automate the process of evaluating the data in a formal and yet efficient way.

This kind of data overload is the key motivation behind the process of knowledge discovery in databases. *Knowledge Discovery in Databases* (KDD) is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data [Fayyad et al., 1996].

According to the definition, KDD is a process and therefore comprises many steps. This multi-step process involving data preparation, data selection, data clean-

[2]Adapted from [Yao et al., 2007]

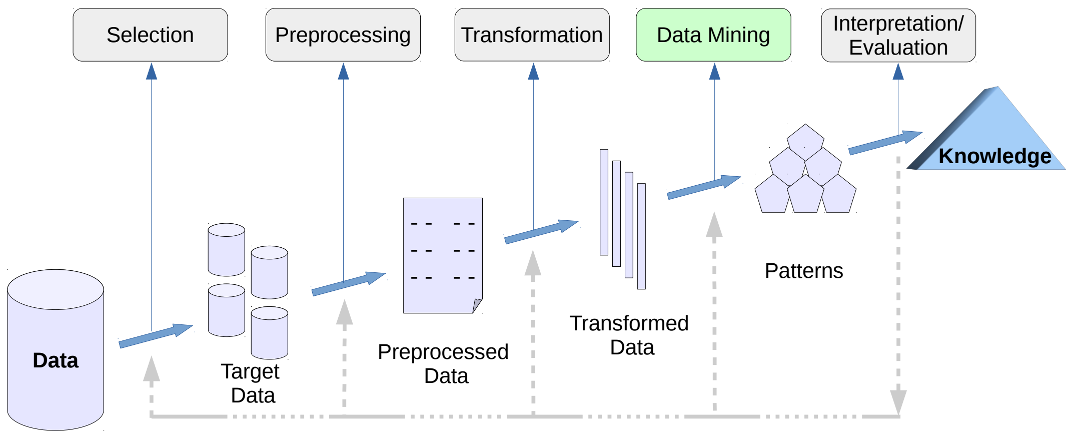


Figure 1.1: Process of knowledge discovery in databases (KDD)[3]

ing, data mining, incorporation of appropriate prior knowledge, and proper interpretation is generally understood as interactive and iterative, as depicted in Figure 1.1. The process is supposed to be non-trivial, involving some form of search or inference to discover patterns.

While all the steps in the process of KDD are of paramount importance to achieve the ultimate goal of knowledge discovery, we focus on the step of *data mining*, which can be considered as the core stage of KDD process.

Data Mining

Data mining is a crucial step in the knowledge discovery process. It applies algorithms that extract patterns from preprocessed data that are subsequently interpreted to gain insight and knowledge about the data, and ideally about the underlying process as well. In [Fayyad et al., 1996], data mining is described as a process, consisting of the application of data analysis and pattern discovery algorithms that yield a particular set of patterns over the data. More specifically, a data mining algorithm comprises three components: (i) model representation, (ii) model evaluation, and (iii) search. We will now elaborate on the three components and specify the representations and methods studied in this thesis respectively.

Model Representation. Model representation, in general, is concerned with the language employed to describe patterns. If the representation chosen is too limited, no amount of training time or further training examples will lead to an accurate model for the data. However, more powerful representations bear the danger of overfitting the training data, leading to limited prediction accuracy on unseen data. The adequate choice of a representation language has to be considered carefully, as

[3]Adapted from [Fayyad et al., 1996]

it is important that the analyst fully understands the assumptions inherent to the representation method employed.

This thesis is mainly concerned with graph structured data employed to model various domains of real world data. While graphs are a very powerful concept, they come at a cost, owing to the computational complexity of operations on graphs such as subgraph isomorphism, which is essential for pattern mining and is NP-complete [Garey and Johnson, 1979].

Model Evaluation. In general terms, a model evaluation criterion is a quantitative statement of how well a specific model meets the goals of the KDD process. Predictive models, for instance, are often judged by the empirical prediction accuracy on some test data set. Descriptive models can be evaluated along the dimensions of predictive accuracy, novelty, utility, and understandability of the fitted model. In this thesis, in the context of pattern discovery, the frequency of a pattern is a fundamental measure. Combined with a threshold, this measure yields a criterion, known as minimum frequency, often used to extract rare and thus potentially unknown – but hopefully genuine – patterns from data.

Pattern Search. Once the representation language and evaluation criteria are fixed, the data mining problem resorts to a pure optimisation or constraint satisfaction task: find (all) the pattern(s) from the selected representation which optimise (or satisfy) the evaluation criteria on the given data set. The motivation to study the problem of frequent pattern mining emanates from the challenge of extracting association rules from retail data, known as market basket analysis, which are derived from frequent patterns. In the course of this thesis, we employ Depth First Search (DFS) strategies, combined with several optimization techniques, in order to search the patterns.

In the following section, we discuss on graph data structures, that are the focus our knowledge extraction process.

1.2 Managing and Mining Graph Data

The concepts of knowledge retrieval and knowledge discovery readily find their relevance for graph data, in the form of graph data management and graph mining. Informally, a *graph* can be conceived as a data structure, where a set of nodes are interconnected by a set of relations. Mining and management of graph data has become more important than ever, owing to a wide range of applications in the domains of remote sensing, social networks, computational biology, chemical data analysis, and communication networking. Traditional data management approaches such as indexing, querying and data mining algorithms such as clustering, classification, frequent pattern mining have now been extended to the graph scenario.

Managing Graph Data: Since graphs form a complex and expressive data type, we need methods for representing graphs in databases, manipulating and querying them. In this thesis, we explore the design of indexing structures for graph data, and employ efficient structures to retrieve the relevant information. When it comes to querying graph databases, a plenty of specialized querying procedures exist such as, query matching, keyword search and reachability queries, that can be exploited depending on the type of knowledge one is interested in retrieving [Aggarwal et al., 2010].

In this work, we focus on the *query matching* aspect in graph databases, which itself is achieved by introducing efficient indexing and retrieval structures.

Mining Graph Data: In the recent years, the need for mining structured data has increased and as we have already learnt, graphs have been deemed to be a promising structured data. Much like the other data structures - text data, sequential data, the mining problems have been designed for graphs as well. Many graph data mining techniques exist, which include pattern mining, clustering and classification [Yan and Han, 2003, Kudo et al., 2004, Rattigan et al., 2007, Yan et al., 2008]. However, owing to the structural nature of the data, these mining techniques are met with challenges that are not faced in the domains of other data structures. This indeed comes at a cost, owing to the greater expressive power of structured data like graphs.

In this work, we streamline our focus on discovering graph patterns. Of the various graph patterns, frequent patterns are the basic types of patterns that can be discovered in graph databases [Han et al., 2011]. These frequent patterns are further useful in discriminating different groups of graphs, classifying and clustering graphs, building graph indices, and facilitating similarity search in graph databases.

Scope of the Thesis: Mining and managing graph data has grown to become a multi-faceted domain, since the challenges offered by the data from various fields result in different kinds of graphs. For example, chemical data graphs are relatively small but only a few labels on different nodes may be repeated many times in a single molecule (graph) [Aggarwal et al., 2010]. In many large scale domains [Kumar et al., 2000, Raghavan and Garcia-Molina, 2003] such as the Web, computer networks, and social networks, the distinct node labels (e.g., users, URLs) are huge, which leads to difficulty in characterizing such graphs succinctly. The massive size of computer network graphs is a considerable challenge for mining algorithms. With the rise of multiple platforms of social networks, the graphs may be dynamic and time evolving, where the structure of the graph may change rapidly over time, and the temporal aspects could be worth considering.

Owing to extremely diverse graph domains, several algorithms have been designed in order to manage and mine such graph data. For example, the algorithms which are designed for the web or social networks need to be constructed for graphs with very large size, but with distinct node labels. On the other hand, the algo-

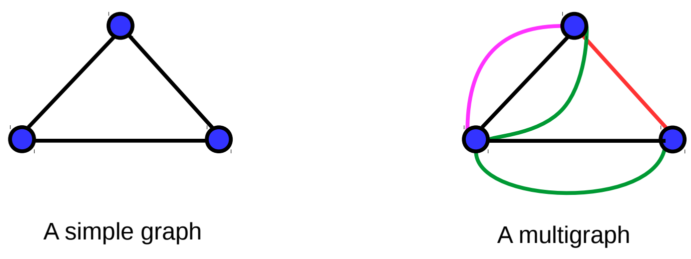


Figure 1.2: An instance of a simple graph and a multigraph

rithms which are designed for chemical data need to take into account repetitions in node labels. Similarly many graphs may have additional information associated with nodes and edges. Such variations make different applications much more challenging. However, in the past decade, several approaches have been proposed that are able to manage and mine graph data that encompass several graph domains [Kuramochi and Karypis, 2004, Wu et al., 2013, Berlingerio et al., 2013, Hsieh et al., 2014] and there still remains a lot of potential work to be achieved in this direction.

In this thesis, our objective is to propose efficient querying and mining approaches that are applicable to various graph domains alike. In this direction, we focus on a generic class of graphs called *multigraphs*, which has been gaining prominence lately [Godehardt, 2013, Boden et al., 2012]. Multigraphs are a rich class of graph data structures, that allow different types of relations between a pair of nodes [Bonchi et al., 2014]. In Figure 1.2, an instance of a multigraph is depicted, where multiple relations between a pair of nodes can exist; in contrast, a simple graph has exactly one relation. Many real world datasets can be modelled as a network with a set of nodes interconnected with each other with multiple relations. Various domains are abound with multigraphs: social networks spanning over the same set of people, but with different life aspects (e.g., social relationships such as Facebook, Twitter, LinkedIn, etc.); protein-protein interaction multigraphs created by considering the pairs of proteins that have direct interaction/physical association or they are co-localised [Zhang, 2009]; gene multigraphs, where genes are connected by considering the different pathway interactions belonging to different pathways; RDF knowledge graph where the same subject/object node pair is connected by different predicates [Libkin et al., 2013].

Since multigraphs, unlike simple graphs, allow more than one relation between a pair of nodes, we can represent real world data more succinctly, which in turn helps in performing various operations in an effective manner. For example, a recent work in the field of bioinformatics ([Li and Li, 2012]) creates multigraphs by merging heterogeneous genomic and phenotype data, in order to identify the disease genes. Many such applications can be catalysed if the community of graph data management can devote for a systematic development of multigraph querying and mining.

To sum up, in this thesis, we focus on the querying aspects of knowledge retrieval and frequent pattern mining aspects of knowledge discovery, in the context of multigraph data. In the following section, we briefly discuss about our contributions in this direction.

1.3 Contributions

The overall objective of this thesis is to efficiently retrieve and discover knowledge from multigraphs. In the recent past, modelling the real world data as graph data structures has yielded more insights as to discover and retrieve knowledge that is rich with structures. Further, the graph data is getting structurally much more richer over the time, thereby prompting to model such graph data in a succinct manner, which is possible with multigraph modelling. Thus, in this thesis, we specifically work on multigraph datasets.

In this thesis, we answer the following research questions:

Q_1 Why is it essential to introduce a novel subgraph query matching approach for multigraphs? And what are the challenges?

Q_2 Is it possible to make multigraph based querying more time efficient and robust, when compared with relational database approaches?

Q_3 Is it necessary to introduce a novel multigraph mining approach? If so, what is the computational feasibility of such mining approaches?

At the outset we assure that the contributions of this thesis are optimistic in answering the above mentioned research questions.

The contributions of this thesis about question Q_1 w.r.t. the importance and challenges of subgraph querying in multigraphs are the following:

- We propose an algorithm SuMGra (***Su**bgraph Matching for **M**ulti**Gra**phs*) that performs subgraph query matching for multigraphs and outperforms the existing approaches. The existing approaches either can not be trivially applied for multigraphs or only a few approaches can be extended for multigraphs.
- To exploit the multigraph properties, we propose indexing structures that leverage the multiedge information for speeding up the query matching process.
- A backtracking procedure is introduced to work on multigraphs, that employs several pruning strategies and optimization techniques.

The contributions of this thesis towards question Q_2 are the following:

- We propose a model to represent the RDF data and SPARQL queries as multi-graphs, where SPARQL[4] is a W3C standard language to query RDF data.
- We propose an RDF querying engine called AMbER (***A****ttributed* ***M****ultigraph* ***B****ased* ***E****ngine for* ***R****DF querying*) that works on RDF multigraph and takes multigraph format of the SPARQL query, to retrieve the SPARQL solutions. The multigraph query engine AMbER is a graph based query engine, that competes very successfully w.r.t. the conventional relational approaches.
- At the core of the proposed AMbER lie a few graph theoretical ideas that are dependent on the structural properties of the queries themselves, which we exploit to offer competitive performance.

The contribution of this thesis towards question Q_3 on multigraph mining are the following:

- We propose an efficient multigraph mining algorithm MuGraM (*Frequent* ***M****ulti****G****raph* ***M****iner*) that discovers frequent multigraph patterns. The existing approaches either do not discover multigraph patterns or the set of discovered frequent patterns are often incomplete.
- An efficient way of computing the frequency measure of a pattern is proposed. Several observations are made that help us to quickly decide if a pattern is frequent or not.
- Several optimization and pruning strategies are discussed that significantly reduce the search space exploration to discover frequent patterns.

At the end of the thesis, we dedicate a chapter (Chapter 6) to perform a case study analysis of the application of our thesis contributions for remote sensing data. This analysis discovers interesting and useful patterns and helps us to analyse the significance of such patterns.

1.4 Organization of Thesis

In Chapter 2, we discuss about the foundations of the thesis in the context of multigraphs, and querying and mining multigraphs. The rest of the thesis is structured upon the earlier published work. Unless otherwise stated, the work reported

[4]https://www.w3.org/TR/rdf-sparql-query/

throughout this thesis, was done primarily by the author in co-ordination with his supervisors.

The contribution of this thesis can be broadly organised into two themes, which are presented as two different parts. Part I is dedicated to the querying aspects in multigraphs, where we focus on subgraph query matching. The task of *query matching* is itself a sub-field of graph data management which in turn is coherent with the theme of *knowledge retrieval*. Part II is dedicated to the mining aspects in multigraphs, where we focus on *frequent pattern mining*. Frequent pattern mining adheres to the theme of *knowledge discovery*.

Part I of the thesis focuses on the subgraph query matching problem in multigraphs.

Part I

Chapter 3 proposes a subgraph query matching algorithm for multigraphs. This chapter introduces novel index structures that leverage the multigraph properties and enable optimized exploration of search space to discover query matches. Then a query matching process is proposed, which follows the backtracking principle to explore search space. Several experiments are conducted to validate the time performance of proposed SuMGra when compared with the existing approaches. This chapter is composed of the following research contributions that are already published.

- Vijay Ingalalli, Dino Ienco, and Pascal Poncelet. *SuMGra: Querying Multigraphs via Efficient Indexing.* In Proceedings of 27th International Conference on Database and Expert Systems Applications (DEXA 2016), Porto, Portugal, September 2016, pp. 387-401.

- Vijay Ingalalli, Dino Ienco, and Pascal Poncelet. *Leveraging efficient indexing schema to support multigraph query answering.* In: Ingénierie des Systèmes d'Information (ISI), Volume 21, Issue 3, 2016, pp. 53-74.

- Vijay Ingalalli, Dino Ienco, and Pascal Poncelet. *On Querying Large Graphs with Multiple Relationships.* In La conférence sur la Gestion de Données - Principes, Technologies et Applications (BDA 2015), 12 pages, October 2015, Île de Porquerolles, France.

Chapter 4 makes contributions in the field of querying RDF data. Models to represent the RDF data and the SPARQL queries as multigraphs are introduced. On the RDF multigraph, indexing structures are built, which takes inspiration from the previous chapter. However, there are several distinctions for index structures owing to the vertex labelled and directed multigraphs, which is a realistic modelling of an RDF data. A multigraph modelling of a SPARQL query is used to to perform homomorphic matching, which yields SPARQL solutions. Several optimizing procedures are introduced that help in speedy retrieval of homomorphic solutions. Rigorous experiments are conducted to validate both the time performance and robustness of the proposed RDF querying engine AMbER, with the state-of-the-art approaches. This chapter is composed of the following published work.

- Vijay Ingalalli, Dino Ienco, Pascal Poncelet, and Serena Villata. *Querying RDF Data Using A Multigraph-based Approach.* In Proceedings of 19th International Conference on Extending Database Technology (EDBT 2016), Bordeaux, France, March 2016, pp. 245-256.

Part II of the thesis focuses on the frequent pattern mining problem in multigraphs.

Part II

Chapter 5 explores the field of frequent subgraph mining (FSM) in multigraphs. Limitations of the existing FSM approaches are elucidated, and the contributions for FSM in multigraphs are illustrated. Several competing approaches and their methodologies are briefly discussed. Optimized exploration of complex search space, which is borne out of the multigraph property of the graphs are explored. Several novel ideas to compute support measure of a pattern are introduced; further, several pruning strategies to speed up the computational performance of the proposed multigraph mining algorithm MuGraM are also explored. Both quantitative (time performance) analysis and qualitative (interesting pattern discovery) analysis on several real world datasets are performed, that portrays the competitiveness of the proposed MuGraM. This chapter is composed of the following work.

- Vijay Ingalalli, Dino Ienco, and Pascal Poncelet. *Mining Frequent Subgraph Patterns in Multigraphs.* Submitted.

Chapter 6, makes a case study analysis of the entire thesis work, both in terms of knowledge retrieval and knowledge discovery. The case study is for the application of remote sensing dataset, where the remote sensing images are modelled as multigraphs by following the image segmentation procedure. In the beginning, the proposed multigraph mining algorithm MuGraM is applied to discover a set of frequent patterns, for a user defined frequency value. Then for a few selected patterns, the query matching algorithm SuMGra is invoked to trace the location of patterns in the image to draw some interesting conclusions.

Chapter 7 summarises the overall contribution of the thesis and lays ideas for the potential future extension of the proposed works in this thesis.

Chapter 2

Foundations

In this chapter we introduce the basics of graphs and multigraphs, and then establish the theoretical concepts used for query matching. We then make a quick tour on graph databases and indexes, and finally introduce the basic principles of graph data mining in the context of the thesis.

2.1 Graphs, Multigraphs

A *graph* is a mathematical structure employed to model pairwise relations between objects, where the objects are termed as *vertices* and the relations are called *edges* [West et al., 2001]. Thus, a graph structure amounts to a set of objects where some pairs of objects are in some sense "related". Simple graphs are one of the versatile forms of graphs that have labels on both vertices and edges, formally defined as follows.

Definition 2.1 (Graph). *Given a vertex label alphabet Σ_V and an edge label alphabet Σ_E, we define a directed/undirected labelled graph g as an ordered four-tuple $g = (V, E, L_V, L_E)$, where:*

- *V denotes a finite set of nodes*
- *$E \subseteq V \times V$ denotes a set of edges*
- *$L_V : V \rightarrow \Sigma_V$ denotes a node labelling function*
- *$L_E : E \rightarrow \Sigma_E$ denotes an edge labelling function*

In *undirected* graphs, E is a set of unordered pair of vertices called edges; in *directed* graphs, E is a set of ordered pair of vertices called directed edges. Simple graphs usually do not have loops. The set V can be regarded as a set of n vertex identifiers and is often represented as $V = \{v_1, v_2, \ldots, v_n\}$; the set of edges E represents the structure of the graph. That is, a vertex $v_i \in V$ is connected to a node $v_j \in V$ by an edge $e = (v_i, v_j)$ if $(v_i, v_j) \in E$. The labelling functions L_V and L_E can be used to integrate information about vertices and edges into graphs by assigning attributes from L_V and L_E to nodes and edges, respectively.

Definition 2.2 (Subgraph)**.** *A graph $g^1 = (V^1, E^1, L_V^1, L_E^1)$ is a subgraph of $g^2 = (V^2, E^2, L_V^2, L_E^2)$ if $V^1 \subseteq V^2$ and $E^1 \subseteq E^2$ and $\forall v \in V^1$, $L_V^1(v) = L_V^2(v)$ and $\forall e \in E^1$, $L_E^1(e) = L_E^2(e)$.*

The notion of subgraph is important for subgraph matching and subgraph mining approaches. Further, a graph g^2 is called a *supergraph* of g^1 if g^1 is a subgraph of g^2.

We now introduce a generic class of graphs called multigraphs, that allow multiple edges between a pair of vertices, formally defined below.

Definition 2.3 (Multigraph)**.** *Given a set of edge types T, a vertex label alphabet Σ_V and an edge label alphabet Σ_E, we define a multigraph G as an ordered five-tuple $G = (V, E, \mu, L_V, L_E)$, where:*

- *V denotes a finite set of nodes*
- *$E \subseteq V \times V$ denotes a set of edges*
- *$\mu : E \to 2^T$ denotes a multiedge mapping function*
- *$L_V : V \to \Sigma_V$ denotes a node labelling function*
- *$L_E : E \to \Sigma_E$ denotes an edge labelling function*

Since a multigraph can have multiple edges between a pair of vertices, we denote a set of edge types T, that distinctly represents these multiple edges; further, all possible multiedges for a given edge set T are represented as a power set 2^T, whose elements are a multiset of edge types. Thus, μ is a multiedge mapping function that assigns a multiset of edge types from the power set 2^T to an edge $(v_i, v_j) \in E$. The labelling functions L_V and L_E can be used to integrate information about vertices and edges into multigraphs.

The multigraph that we define here is in its most generic form. This multigraph can take a set of labels on both vertices as well as on multiedges. However, in reality, the simpler variants of multigraphs are needed; for example, multigraphs without edge labels or vertex labels, directed or undirected. Thus, depending on the kind of dataset we deal with, in the following thesis (Chapter 3, 4, 5) we introduce the variants of Definition 2.3. A graph G^1 is a *sub-multigraph* of G^2 if all the conditions in Definition 2.2 are met.

2.2 Graph Matching

In *graph matching*, the objective is to determine whether or not the labels on vertexes/ edges and structure, or part of the structure, of two graphs are identical. A graph matching can be achieved in both exact and approximate ways [Bunke, 2000]. In *exact graph matching*, an isomorphic mapping exists between the structures that have to be matched. However, in *approximate graph matching*, either the isomorphism does not exist or there is no need to find an isomorphic matching, and hence there are no matching vertices of a graph with the vertices of the other graph, but a best possible matching has to be discovered.

In this thesis, we focus only on the *exact graph matching* problem. Previous works show that it is easy to determine equality of patterns in case of feature vectors or strings, whereas the same computation is much more complex for graphs [Cook and Holder, 2006, Chakrabarti and Faloutsos, 2012]. Because the nodes and edges of a graph cannot be ordered in general, unlike the components of a feature vector or the symbols of a string, the problem of graph equality, is computationally very demanding. As already discussed in the previous chapter, several notions of exact graph matching exist: (i) Graph isomorphism (ii) Graph homomorphism.

Definition 2.4 (Graph Isomorphism)**.** *If we consider a graph $g^1 = (V^1, E^1, L_V^1, L_E^1)$ and a graph $g^2 = (V^2, E^2, L_V^2, L_E^2)$, then a graph isomorphism between g^1 and g^2 is a bijective function $\psi : V^1 \to V^2$ that satisfies the following conditions:*

- $L_V^1(u) = L_V^2(\psi(u)) \; \forall u \in V^1$
- $\forall \; (u, v) \in E^1, \; \exists \; (\psi(u), \psi(v)) \in E^2 : L_E^1((u, v)) = L_E^2((\psi(u), \psi(v)))$
- $\forall \; (u, v) \in E^2, \; \exists \; (\psi^{-1}(u), \psi^{-1}(v)) \in E^1 : L_E^1((u, v)) = L_E^2((\psi^{-1}(u), \psi^{-1}(v)))$

Given two graphs g^1 and g^2, the problem of graph isomorphism is to check if there exists an isomorphism between the two graphs; as per Definition 2.4, isomorphic graphs are identical in terms of labels and structure. The definition of graph isomorphism is trivial to extend for multigraphs, since a multiedge between a pair of vertices can be mapped to a labelled edge, which is conceptually a simple graph as introduced in Definition 2.1.

The problem of graph isomorphism occupies an important position in the world of complexity analysis, since it is one of the few problems that is not known to be in either P or NP-complete [Fortin, 1996]. Although recently there has been a significant milestone in proposing a new quasi-polynomial time algorithm [Babai et al., 2016], many practical approaches exist that solve the problem in near polynomial time for most of the real world graphs structures [McKay et al., 1981, McKay and Piperno, 2014]. In this thesis, we confront the problem of graph isomorphism in Chapter 5, while performing the task of graph mining.

Closely related to graph isomorphism is the problem of detecting the presence of a smaller graph in a larger graph. If graph isomorphism is regarded as a formal notion of graph equality, subgraph isomorphism can be seen as subgraph equality [Cook and Holder, 2006].

Definition 2.5 (Subgraph Isomorphism)**.** *A graph $g^1 = (V^1, E^1, L^1_V, L^1_E)$ is subgraph-isomorphic to a graph $g^2 = (V^2, E^2, L^2_V, L^2_E)$ if and only if there exists a subgraph s of g^2 that is isomorphic to g^1.*

An injective function $\psi : V^1 \to V^2$ is called a subgraph isomorphism or a subgraph embedding from g^1 to g^2 if ψ is a graph isomorphism between g^1 and s, that satisfies the following conditions:

- $(u, v) \in E^1 \implies (\psi(u), \psi(v)) \in E^2$
- $L^1_V(u) = L^2_V(\psi(u))$
- $L^1_E((u, v)) = L^2_E((\psi(u), \psi(v)))$

The problem of subgraph isomorphism plays a key role when dealing with exact subgraph matching problems. However, the fact that the problem of subgraph isomorphism is known to be NP-complete [Garey and Johnson, 1979] has inspired many works [Ullmann, 1976, Cordella et al., 2004, Kim et al., 2011, Lee et al., 2012, Han et al., 2013, Ren and Wang, 2015] to propose subgraph matching approaches that employ efficient search heuristics. The notion of subgraph isomorphism for multigraphs can be defined in the same spirit of Definition 2.5; depending on the variant of multigraph that we employ in the thesis, we introduce the definitions in the corresponding chapter (Chapter 3, 5).

Another graph matching concept is graph homomorphism. If isomorphism on two graphs was a structure preserving mapping, the homomorphism mapping preserves connectedness between two graphs.

Definition 2.6 (Graph Homomorphism)**.** *A graph homomorphism from a graph $g^1 = (V^1, E^1, L^1_V, L^1_E)$ to graph $g^2 = (V^2, E^2, L^2_V, L^2_E)$ is a mapping $\phi : V^1 \to V^2$ so that:*

- $L^1_V(u) = L^2_V(\phi(u))\ \forall u \in V^1$
- $\forall\ (u, v) \in E^1,\ \exists\ (\phi(u), \phi(v)) \in E^2 : L^1_E((u, v)) = L^2_E((\phi(u), \phi(v)))$

It is to be noted that if the graph homomorphism $\phi : g^1 \to g^2$ is a bijection whose inverse function is also a homomorphism, then ϕ is a graph isomorphism.

Definition 2.7 (Subgraph Homomorphism)**.** *A graph $g^1 = (V^1, E^1, L^1_V, L^1_E)$ is subgraph-homomorphic to a graph $g^2 = (V^2, E^2, L^2_V, L^2_E)$ if and only if there exists a subgraph s of g^2 that is homomorphic to g^1.*

The notion of subgraph homomorphism for multigraphs will be introduced in Chapter 4, where we also consider directed multigraphs. The idea of employing graph homomorphism for graph matching is an active area [Baget, 2005, Fan et al., 2010] in order to address several real world applications.

2.3 Graph Databases, Indexes

In this section we discuss about the graph representation of real world data, and efficient ways to retrieve information from such graph representations. In this thesis, we consider graphs as data structures used for structured representation of the real world data. Thus, depending on the application and the nature of dataset itself, graph data structures can be broadly categorised into the following two domains.

Transactional graph database. A transactional graph database is represented as a finite set $D = \{g^1, g^2, \dots, g^n\}$ of n graph structures, where each graph g_i represents a transaction in the scope of a dataset. For example, a chemical compound dataset [Yan and Han, 2002], can be conceived as a transactional graph database, where the structure of each chemical compound is represented as a graph g. Transactional graphs have been explored by the graph management community for more than a decade, and numerous works have been proposed both for querying [Shasha et al., 2002, Cheng et al., 2007, Shang et al., 2008] and mining [Inokuchi et al., 2000, Kuramochi and Karypis, 2001, Yan and Han, 2002] transactional graphs.

Single large graph database. A single large graph database is composed of one large connected graph, where vertices and edges represent the entities and the relations among them for the entire dataset. For example, a social network or the World Wide Web can be conceived as single large graphs [Boden et al., 2012, Zou et al., 2014b]. Instances of datasets that have to be represented as single large graphs have been proliferating in the past few years, thereby propelling the graph management community to manage such graph databases. With the emergence of single large graphs in the recent past, several works have been proposed for both querying [Kim et al., 2011, Zhao and Han, 2010, Han et al., 2013] and mining [Kuramochi and Karypis, 2005, Thomas et al., 2010, Elseidy et al., 2014] in single large graphs.

In this thesis, we solely focus on single large graph databases, where a graph is allowed to handle multiedges, and hence a single large multigraph database.

Graph Indexes

In the context of databases, an *index* is a data structure that improves the speed of data retrieval operations on a database at the cost of storage space, which is required to maintain the index data structure [Yan et al., 2004]. The notion of *index* is thus applicable to graph databases as well. The challenge associated with graph indexes is to identify the intrinsic characteristics of the graph data and represent it in a data structure for the efficient retrieval of information from the data. Further, the index structures have to be stable w.r.t database updates, so as to achieve incremental index maintenance.

Several indexing approaches have been proposed for managing graph data, in particular for querying graph data. In [Yan et al., 2004], frequent substructures are used to propose index structures; in [Yan et al., 2005a] the authors propose an indexing model based on discriminative frequent structures; in [Wang et al., 2012], a two-level inverted index is proposed to speed up graph similarity search. In this thesis, we propose indexing structures (Chapter 3, 4) by summarising the characteristics of the multiedges, and exploring the multigraph properties.

2.4 On Querying Graphs

Graph querying refers to the task of querying subgraph patterns in a given graph database. In this thesis, we focus on the *subgraph query matching* problem, where we are interested in finding the *embeddings* of a subgraph query in a graph database. In Chapter 3 and 4, we propose efficient approaches for two variants of subgraph query matching, namely isomorphic match and homomorphic match, respectively. During subgraph query matching, one discovers the embeddings of a subgraph query by traversing the search space. The search space is typically traversed using the backtracking search.

Backtracking Search

A backtracking search is a recursive function where at each stage, the match that we have found so far is extended until the entire query is matched. Consider an instance where we are interested in determining if a query q is subgraph isomorphic to a graph database g. Now let us assume that we have already determined that a partial query subgraph q^1 is isomorphic to a subgraph g^1 of the database graph g. We then try to add a vertex to the partial query subgraph g^1 such that the new subgraph, say q^2, is isomorphic to another subgraph g^2 of database graph g. At some point we may hit a dead end, since there might be no vertices that can be added to extend the isomorphic subgraphs. We then backtrack to previous smaller matching subgraphs, and try to extend with a different vertex choice. The process ends by either finding a complete match of the subgraph query q in the database graph g and returning true, or by exhausting all possibilities and returning false.

Algorithm 2.1: BACKTRACKING(q, g, mq, mg)

```
1 if $mq.size \neq q.size$ then
2     $u$ := CHOOSENEXTVERTEX($q, mq$)
3     $M$ := FINDALLMATCHES($u, q, g, mg$)
4     for each matched vertex $m \in M$ do
5         EXPAND: $mq = mq \cup u$; $mg = mg \cup m$
6         BACKTRACKING($q$, $g$, $mq$, $mg$)        /* Make a recursive call */
7         CONTRACT: $mq = mq \setminus u$
8 else
9     OUTPUT: $mg$                                  /* Matched embedding */
```

A typical backtracking algorithm (an exhaustive approach) to check if a query q is subgraph isomorphic to g is depicted in the BACKTRACKING procedure (Algorithm 2.1). The procedure takes a query subgraph q, a data graph g, a partially matched query mq and a partially matched graph mg as inputs, and outputs the completely matched graph mg, if a match is found. Initially, a query vertex $u \in q$ is chosen and the corresponding set M of matched vertices are discovered. Then for each matched vertex $m \in M$, the partially matched query mq is extended with the vertex u and the partially matched graph mg is extended with the vertex m. Then the backtracking approach BACKTRACKING is called recursively until the entire query q is matched. If the entire query q is matched, then corresponding matched graph mg is outputted; else the backtracking terminates after the exhaustive search.

The problem with the exhaustive backtracking search is that, for a database with n vertices, there are $n!$ possible vertex mappings, and hence we need to prune the search space. In Chapter 3 and 4, we propose several search space pruning techniques that makes the backtracking approach very efficient.

2.5 Graph Data Mining

Part II of the thesis is dedicated to graph data mining. The objective of graph data mining is to discover interesting patterns/substructures in graphs. One of the fundamental fields of graph data mining is *frequent pattern mining* that searches for recurring relationships in a given graph data set. Finding such frequent patterns plays an essential role in mining associations, correlations, and many other interesting relationships among data [Aggarwal and Han, 2014]. The notion of mining frequent patterns can be traced back to mining frequent itemsets where association rules are explored [Agrawal et al., 1993, Agrawal et al., 1994]. For example, a set of items, such as milk and bread, that appear frequently together in a transaction data set is a frequent itemset. The concept of frequent pattern has later on been extended for mining frequent subsequences [Zaki, 2001] and substructures [Kenji et al., 2004]. A substructure is relatively complex when compared to itemsets or subsequences; a substructure could a tree [Han et al., 2004] or a graph [Yan and Han, 2002].

This work is dedicated to graph substructures and we are interested in *frequent subgraph mining* (FSM) in the context of multigraphs.

2.5.1 Basics of FSM

Since a lot of contributions have been made in the field of FSM for transactional database settings, we first introduce the concept of frequent subgraph in transactional graph data. In literature, the frequency of a graph pattern is referred to as *support*.

Definition 2.8 (Support)**.** *We define the support of a subgraph pattern p as the percentage (or number) of graphs in D where p is a subgraph.*

Thus, given a labelled graph dataset $D = \{g_1, g_2, \dots, g_n\}$, a subgraph pattern p is frequent only if the support is no less than a minimum support threshold δ.

For single large graphs, it is difficult to find an appropriate support definition since multiple embeddings of a subgraph may have overlaps [Kuramochi and Karypis, 2005]. If arbitrary overlaps between non-identical embeddings are allowed, the resulting support does not satisfy the anti-monotonicity property (Definition 2.9), which is essential for most frequent pattern mining algorithms. Therefore, many works have proposed ideas on various support measures that are anti-monotonic [Kuramochi and Karypis, 2005, Fiedler and Borgelt, 2007, Bringmann and Nijssen, 2008].

Definition 2.9 (Anti-Monotone)**.** *A constraint f is called anti-monotone iff, for a given graph g, $f(g) \implies f(g')$, for every subgraph g' of g.*

In [Kuramochi and Karypis, 2005], the notion of *simple overlap* is introduced, which is formally defined as follows.

Definition 2.10 (Simple Overlap)**.** *Given a pattern $p = (V, E)$, a simple overlap of occurrences θ_1 and θ_2 of the pattern p exists if $\theta_1(E) \cap \theta_2(E) \neq \emptyset$.*

When all possible occurrences $\{\theta_1, \theta_2, \dots\}$ of a pattern p in a graph g are computed, an overlap graph is constructed where each occurrence θ_i corresponds to a node and there is an edge between the nodes of θ_i and θ_j if they overlap.

In [Fiedler and Borgelt, 2007], the authors suggest a definition that relies on the non-existence of equivalent ancestor embeddings in order to guarantee that the resulting support is anti-monotone. The basic idea of this measure is that some of the simple overlaps (Definition 2.10) can be disregarded without harming the anti-monotonicity of the support measure. The *harmful overlap* is defined below.

Definition 2.11 (Harmful Overlap)**.** *Given a pattern $p = (V, E)$, a harmful overlap of occurrences θ and θ' of pattern p exists if $\exists\ v \in V : \theta(v), \theta'(v) \in \theta(V) \cap \theta'(V)$.*

In both simple overlap and harmful overlap based support measures, the support of p is defined as the size of the maximum independent set (MIS) of the overlap-graph. In [Fiedler and Borgelt, 2007], the MIS-support measure has been proved to be anti-monotone.

In [Bringmann and Nijssen, 2008], the authors examined the above two measures and identified the expensive operation of solving the MIS problem, and proposed a new computationally efficient support measure called *minimum node image support* (MNI), defined below.

Definition 2.12 (Minimum Node Image Support)**.** *Given a pattern p with a set of vertices V and a graph g, let $\psi = \{\psi_1, \psi_2, \ldots, \psi_k\}$ be a set of k isomorphic embeddings of p in g. Then the minimum node image (MNI) support Δ is defined as $\Delta(p) = \min_{u \in V} |\{\psi_i(u) : i = 1 \to k\}|$.*

The MNI measure is based on the number of unique nodes in the graph g to which a node of the pattern p is mapped. By taking the vertex in p which is mapped to the least number of unique vertices in g, the anti-monotonicity of Δ can be guaranteed. For the definition of support, several computational benefits could be identified: (i) instead of $O(n^2)$ potential overlaps, where n is the possibly exponential number of occurrences, the method only needs to maintain a set of vertices for every node in the pattern, which can be done in $O(n)$; (ii) the method does not need to deal with an NP complete MIS problem; and (iii) it is not necessary to compute all embeddings, since it is sufficient to determine for every pair of $u \in V(p)$ and $v \in V(g)$ if there is one occurrence in which $\psi(u) = v$.

Since MNI support measure is computationally less expensive, and many recent works have used this measure [Elseidy et al., 2014], in this thesis we use the MNI measure by extending Definition 2.12 for multigraphs, to be discussed in Chapter 5. Further, in this thesis, support and frequency are used almost interchangeably.

2.5.2 FSM Approaches

Two basic approaches exist for the problem of FSM: the *apriori-based* approach and the *pattern-growth* approach.

Apriori-based approach. The apriori-based approach is a breadth-first search (BFS) approach, where frequent subgraphs are searched starting with small sized subgraphs. At each iteration, the size of newly discovered frequent subgraphs is increased by one. These new subgraphs are generated by joining two similar but slightly different frequent subgraphs that were discovered already. The task of joining, also called candidate generation step, is the most expensive step in apriori-based approaches, since there are many ways to join two subgraphs. The frequency of the newly formed graphs is then checked. Several apriori-based algorithms for FSM exist [Inokuchi et al., 2000, Kuramochi and Karypis, 2001].

Algorithm 2.2: FSM-Apriori

```
1  INPUT: A graph database $g$, minimum support threshold $\delta$
2  OUTPUT: A set of frequent patterns $\mathscr{P}$
3  COLLECT: Frequent vertices of $g$ in $\mathscr{P}^k$, where $k = 1$
4  CALL: GRAPHAPRIORI($g$, $\delta$, $\mathscr{P}^k$)

5  PROCEDURE: GRAPHAPRIORI($g$, $\delta$, $\mathscr{P}^k$)
6  while $\mathscr{P}^k \neq \emptyset$ do
7      INITIALIZE: $\mathscr{P}^{k+1} \leftarrow \emptyset$
8      for each frequent $p_i \in \mathscr{P}^k$ do
9          for each frequent $p_j \in \mathscr{P}^k$ do
10             for each size (k+1) pattern $p$ formed by the merge of $p_i$ and $p_j$ do
11                 if $p$ is frequent in $g$ and $p \notin \mathscr{P}^{k+1}$ then
12                     INSERT: $p$ into $\mathscr{P}^{k+1}$

13 if $\mathscr{P}^{k+1} \neq \emptyset$ then
14     GRAPHAPRIORI($g$, $\delta$, $\mathscr{P}^{k+1}$)
15 return
```

The framework of apriori-based approach is outlined in Algorithm 2.2. The approach takes a graph g and a minimum support threshold δ as input, and outputs a set of all frequent patterns $\mathscr{P}$. In the beginning, all the frequent single elements (vertices) are fetched and the procedure GraphApriori is invoked. The GraphApriori procedure, returns all the frequent patterns of level $k + 1$. The patterns of size $k + 1$ are generated by joining two size k patterns - $p_i \in \mathscr{P}^k$ and $p_j \in \mathscr{P}^k$; then for each size $k + 1$ patterns, if we discover that p is frequent, we add it to $\mathscr{P}^{k+1}$, and the process is repeated until all the patterns are verified. If there are any new frequent patters of size $k + 1$, GraphApriori is invoked to explore patterns of next size; else, the algorithm returns.

Pattern-growth approach. The pattern-growth approach is more flexible regarding its search method, as it can use breadth-first search as well as depth-first search (DFS), the latter of which consumes less memory. Recall that, the apriori-based approach has to use the breadth-first search (BFS) strategy because of its level-wise candidate generation. In order to determine whether a size-$(k+1)$ subgraph is frequent, it must check all of its corresponding size-k subgraphs to obtain an upper bound of its frequency. Thus, before mining any size-$(k+1)$ subgraph, the apriori-like approach usually has to complete the mining of size-k subgraphs. Therefore, BFS is necessary in the apriori-based approach.

The pattern-growth approach extends a frequent subgraph directly by adding a new edge in every possible position, and hence does not perform expensive join operations. A potential problem with the edge extension is that the same graph can be discovered multiple times. Thus, many approaches have proposed efficient ways of doing away with the repeated discovery of the same pattern. For example, gSpan algorithm helps avoiding the discovery of duplicates by introducing a right-most extension technique, where the only extensions take place on the right-most path [Yan and Han, 2002]. The other prominent pattern-growth approaches include: FREQT in [Kenji et al., 2004], FFSM in [Huan et al., 2003], SPIN in [Huan et al., 2004], vSiGram in [Kuramochi and Karypis, 2005] and GraMi in [Elseidy et al., 2014].

Algorithm 2.3: FSM-PatternGrowth

```
1  INPUT: A graph database $g$, minimum support threshold $\delta$
2  OUTPUT: A set of frequent patterns $\mathscr{P}$
3  INITIALIZE: $\mathscr{P} \leftarrow \emptyset$
4  FETCH: $p$ a frequent size 1 pattern
5  CALL: PATTERNGROWTH($p$, $g$, $\delta$, $\mathscr{P}$)

6  PROCEDURE: PATTERNGROWTH($p$, $g$, $\delta$, $\mathscr{P}$)
7  if $p \in \mathscr{P}$ then
8      return
9  else
10     INSERT: $p$ in $\mathscr{P}$
11 FIND: all the edges $e$ such that $p$ can be extended to $p \circ e$
12 for each frequent $p \circ e$ do
13     PATTERNGROWTH($p \circ e$, $g$, $\delta$, $\mathscr{P}$)
14 return
```

The framework of pattern-growth approach is outlined in Algorithm 2.3. The approach takes a graph g and a minimum support threshold δ as input, and outputs a set of all frequent patterns $\mathscr{P}$. In the beginning, a frequent size-1 pattern p is fetched, which is a seed pattern for the further pattern extension. Whenever the PatternGrowth procedure - a recursion procedure - is invoked, the pattern p of size k is checked if it has already been discovered to be frequent; if not, then, we extend p by all possible edges, and check if they are frequent. Thus, the PatternGrowth recursion procedure extends a pattern p, only if it is frequent; else it continues extending other patterns until no pattern can be extended further, and the algorithm terminates.

In Part II of this thesis, we adopt the *pattern growth approach*, since this approach is more suited for single large graphs, as attested by [Kuramochi and Karypis, 2005]. Further, we contribute to the field of FSM by proposing a novel pattern-growth approach that can seamlessly handle multigraphs.

Part I

Query Matching in Multigraphs

Overview Part I

This part of the thesis focuses on the *query matching* aspects of the field of graph data management. To be more specific, in this part we are concerned about the problem of *subgraph query matching.* The problem demands that given a graph database and a subgraph query, one has to find all the instances of the subgraph query in the graph database; in other words, one needs to find all the possible *matches* for the subgraph that may exist in the graph database. However, this subgraph matching itself can be done in several ways. Thus, the two chapters in this part are dedicated to the two different ways of subgraph query matching that we perform. When we delve into each of these two works, they might appear to be different from each other, although the underlying theme of query matching enfolds them together.

In Chapter 3, we address the subgraph matching problem by discovering *isomorphic* matches. *Graph isomorphism* is a structure preserving map between two graphs, where inverse mapping must be strictly maintained. Two graphs are isomorphic to each other, if there exists a bijective mapping between the vertices, edges and the labels on them; thus two isomorphic graphs share a common structure. Thus, in this work, the core problem is to discover and enumerate all subgraphs of the graph database that are isomorphic to the query subgraph.

Querying subgraphs for isomorphic matches has a plenty of applications in retrieving interesting knowledge from social networks, protein-protein interaction networks. For example, by employing isomorphic matches, in a social network, we can discover the various instances of a query that exist in different geographical location; or in a protein-protein interaction network, a biologist would find all the protein structures that physically interact with each other and are partially located in a particular area.

In Chapter 4, we address the subgraph matching problem by discovering *homomorphic* matches. Homomorphism is also a structure preserving map between two structures, much like isomorphism; however, the inverse mapping is not a strict necessity. Two graphs are homomorphic if they are isomorphic to each other, without the injective constraint on the vertices.

Querying homomorphic matches has potential applications in the domains of knowledge graphs. In particular, Resource Description Framework (RDF) data is readily represented as multigraphs, where the subject/object node pair is connected by different predicates [Libkin et al., 2013]. Thus, the RDF knowledge graph can be queried by enumerating homomorphic matches. Employing homomorphic matching on RDF knowledge graphs is another way of handling SPARQL querying on RDF data, where SPARQL[1] is a standard language for querying RDF data.

[1] http://www.w3.org/TR/sparql11-overview/

Chapter

3

Subgraph Query Matching in Multigraphs

In this chapter, we introduce the problem of subgraph query matching in single large multigraphs, and propose a novel algorithm SuMGra *to retrieve all the matched embeddings.* SuMGra *is composed of novel indexing schema for multiedges, which will help to efficiently retrieve the vertices of the multigraph that match the query vertices. Then we perform query matching by following backtracking procedure, to output the entire set of embeddings for the given query. We then perform extensive experiments to highlight the time efficiency as well as the scalability of the proposed approach.*

3.1 Introduction

Subgraph query matching is one of the major challenges faced in the field of graph data management [Lee et al., 2012], where the challenge is to enumerate all the embeddings of a query subgraph in a graph database. The underlying complexity of subgraph query matching problem is due to the decision problem of subgraph isomorphism, which is NP-complete [Garey and Johnson, 1979]. Due to the inherent complexity of the problem, several approaches have been proposed to efficiently traverse the search space by employing a good query matching order and intelligent pruning rules, and thus, different families of subgraph matching algorithms exist.

A plenty of approaches exist that address the subgraph query problem; however, either they consider only simple graphs [Han et al., 2013, Lee et al., 2012] or graphs with some additional information associated with vertices (attributes) [Yang et al., 2011]. Since our focus is on multigraphs, and multigraphs are more generic than the aforementioned graphs, we are addressing a more generic problem of subgraph query matching.

In this chapter, we introduce a novel approach called SuMGra (***Su**bgraph Matching for **M**ulti**Gra**phs*) that addresses the challenge of finding the embeddings of a subgraph query in a multigraph. SuMGra involves two main phases: (i) an offline phase where two separate indexes are constructed that are later used during subgraph query matching procedure; (ii) an online phase, where a subgraph search strategy exploits the indexing schema previously built to enumerate all the available embeddings of the query in the multigraph. The indexing schema exploits the rich structure supplied by the multigraph, and it utilizes the information associated with the edge types, in order to facilitate the retrieval of data vertices.

3.2 Related Work

As our work addresses the exact subgraph query processing, we will explore the related works that appear in the same hue and we present them under the theme that is paramount in defining them.

Feature based indexing approaches follow the filtering and verification framework. During filtering, some graph patterns are chosen as indexing features to minimize the number of candidate graphs. Then the verification step checks for the subgraph isomorphism using the selected candidates. GraphGrep [Shasha et al., 2002] considers the length of the path within a threshold, as the indexing feature. Owing to the weak pruning power of GraphGrep, the concept of 'discriminative ratio' to select the set of features was introduced in gIndex [Cheng et al., 2007]. Tree+Δ [Zhao et al., 2007] uses discriminative subtrees as indexing features that are more efficient than frequent subgraphs. In another approach called FG-Index [Cheng et al., 2007], both frequent subgraphs and edges are used as indexing features. An alternative approach of swift-index [Shang et al., 2008] has been proposed that uses tree features that maintains a prefix-tree structure.

Backtracking algorithms find embeddings by growing the partial solutions. In the beginning, they obtain a potential set of candidate vertices for every vertex in the query graph. Then a recursive subroutine called SubgraphSearch is invoked to find all the possible embeddings of the query graph in the data graph. Ullmann [Ullmann, 1976] proposed the first algorithm under this framework. During SubgraphSearch, Ullmann adopts a very simple pruning rule (condition on the degree of the vertex) and follows the input order of the query vertices to choose the next vertex. On the other hand, VF2 [Cordella et al., 2004] chooses the next vertex that is connected to the already matched data vertex. It also employs very efficient pruning rules that reduces the search space to find the embeddings. QuickSI [Shang et al., 2008] builds a minimum spanning tree to find the next query vertex, by assigning weights to the edges of the query graph, depending on the frequency of occurrence of query vertex in the data graph. GADDI [Zhang et al., 2009] proposes a novel graph indexing method called neighbouring discriminating substructure (NDS), and claims to have high pruning capabilities. Unlike many indexing approaches mentioned before that only index subgraph structures, which might result in huge amount of index substructures, NDS technique allows the indexing to grow in proportion with the number of neighbouring vertices in the database. GraphQL [He and Singh, 2008] and sPath [Zhao and Han, 2010] follow neighbourhood signature based pruning (in a much similar way) to choose the initial set of candidates (the aforementioned approaches simply choose vertices with matching labels), even before calling the SubgraphSearch. GraphQL additionally employs the pruning technique called pseudo subgraph isomorphism that recursively checks if adjacent subtree of a query vertex is subgraph isomorphic to the corresponding feasible data vertex. [Lin and Bei, 2014] exploits neighbourhood tree based approach to index the large graphs. This work introduces the concept of Neighbourhood Trees (NTree), that records the neighbourhood relationships of each vertex in the large graph to filter the non-potential vertices.

All these approaches are able to manage graph with a single label on the vertex while no discussions and no experiments are supplied to deal with graph containing edge information.

Although index based approaches focus on transactional database graphs, some backtracking algorithms address the large single graphs. Also, in [Lee et al., 2012] we see that all the backtracking algorithms have been employed to test their performance for both database and single graphs. A much recent work TurboISO [Han et al., 2013], not quite falling into any of the above themes, proposes a novel concept of *candidate region exploration* to address matching order problem during subgraph isomorphism search, and a novel query processing strategy called *combine and permute* that avoids useless enumerations between query and data vertices. To address the issue of selecting query vertices for *combination*, they propose a novel concept of neighbourhood equivalent class (NEC) defined for a query graph, which is a set of vertices that are equivalent to any vertex chosen.

A very recent work [Ren and Wang, 2015] extends the ideas proposed in [Han et al., 2013] defining equivalent classes at query and database level exploiting vertex relationships. Once the data vertices are grouped into equivalence classes, an hypergraph is built and the query search is performed on the hypergraph structure instead of the original data graph. The hypergraph can be seen as a summary (or index structure) to speed up the retrieval step.

Both [Han et al., 2013] and [Ren and Wang, 2015] exploit equivalent classes in order to speed up the graph isomorphism task. While the first approach exploits only vertex relationships on they query graph, the second and most recent work exploits vertex relationships on both query and data graph.

Although the contribution of backtracking algorithms and TurboISO have been significant, they have not been explicitly developed for multigraphs and hence a novel approach is required. Unfortunately both approaches do not discuss possible extension to graphs containing edge information. Adapting these two methods to multigraph is not straightforward since, the different types of relationships between vertices can exponentially increase the number of equivalent classes (for both query and data graph) thereby drastically reducing the efficiency of the strategies.

A recent work [Bonnici et al., 2013] proposes an approach called RI that employs light pruning rules in order to avoid visiting useless candidates. The goal of this algorithm is to maintain a balance between the size of the generated search space and the time needed to visit it. This approach is a unique approach that is able to directly manage graph with multiple edges between vertices. And since this is the only approach that directly manages multigraphs, we chose it as a competitor; our experiments validate that SuMGra outperforms RI.

3.3 Problem Definition

In this chapter, we address the problem of subgraph query matching in single large multigraphs with undirected edges and unlabelled vertices. We now introduce the variant of the multigraph definition (as introduced in Definition 2.3), restricted to unlabelled, undirected multigraphs.

Definition 3.1. Unlabelled, Undirected Multigraph. *An unlabelled undirected multigraph G is a tuple of four elements (V, E, L_E, T) where V is the set of vertices and T is the set of edge types, $E \subseteq V \times V$ is the set of undirected edges and $L_E : V \times V \to 2^T$ is a labelling function that assigns the subset of edge types to each edge it belongs to.*

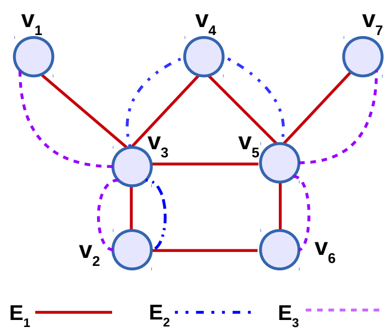
In order to address the problem of subgraph query matching, one has to solve the problem of subgraph isomorphism. Since the notion of sub-multigraph isomorphism will facilitate to introduce the problem of subgraph query matching in multigraphs, we formally define the problem of sub-multigraph isomorphism. The following definition is a variant of the subgraph isomorphism (as introduced in Definition 2.5), relevant to the unlabelled undirected multigraphs.

Definition 3.2. Subgraph isomorphism for multigraphs. *Given a query multigraph $Q = (V^q, E^q, L^q_E, T^q)$ and a data multigraph $G = (V, E, L_E, T)$, the subgraph isomorphism from Q to G is an injective function $\psi : V^q \to V$ such that:*

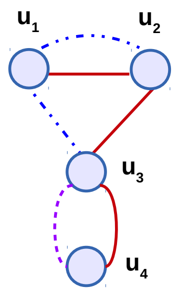
$$\forall (u_m, u_n) \in E^q,\ \exists\ (\psi(u_m), \psi(u_n)) \in E \text{ and } L^q_E(u_m, u_n) \subseteq L_E(\psi(u_m), \psi(u_n)).$$

Having formally introduced the notion of multigraphs and the notion of subgraph isomorphism, we now define the problem of subgraph query matching in multigraphs.

Problem 3.1. Sub-multigraph query matching. *Given a query multigraph Q and a data multigraph G, the sub-multigraph query matching problem is to enumerate all the embeddings of Q in G, so that each embedding is isomorphic to the query Q.*



(a) A data multigraph G



(b) A query multigraph Q

Figure 3.1: A sample example to portray the embeddings of a query in graph

The intuition of the problem can be conveyed with the following example. Consider the Figure 3.1, where the data multigraph G is depicted in Figure 3.1a, with a set of edge types $\{E_1, E_2, E_3\}$, and unlabelled vertices, whose identifiers are denoted by v_i. Now consider the sub-multigraph query Q, as depicted in Figure 3.1b, again with unlabelled vertices, whose identifiers are denoted by u_i. Then the Problem 3.1 demands us to enumerate all possible embeddings of Q in G, which are enumerated as:

$$M_1 := \{[u_1, v_4], [u_2, v_5], [u_3, v_3], [u_4, v_1]\}$$

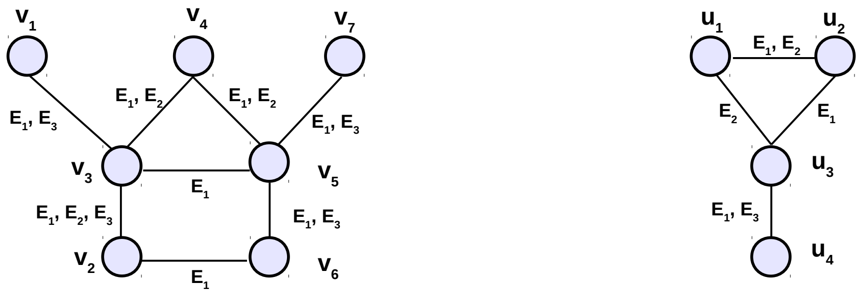
$$M_2 := \{[u_1, v_4], [u_2, v_3], [u_3, v_5], [u_4, v_6]\}$$

In the above embeddings, a pair of elements represent that a query vertex u_i is matched with a data vertex v_i in such a way that, the isomorphic mapping is retained. Thus both M_1 and M_2 are isomorphic to the query multigraph Q.

In Figure 3.2, we propose an equivalent representation of the data multigraph as well as the query subgraph that are depicted in Figure 3.1. This equivalent representation will be used throughout the chapter for illustration purposes.

3.4 An Overview of SuMGra

In this section, we sketch the generic framework of the proposed algorithm SuMGra. The entire approach can be divided into two parts: (i) an indexing schema for the multigraph G that exploits edge types and the vertex neighbourhood structure (Section 3.5) (ii) a subgraph search algorithm, that integrates recent advances in the graph data management field, to enumerate the embeddings of the subgraph (Section 3.6).



(a) A data multigraph (b) A query multigraph

Figure 3.2: Representation of data multigraph, and the query multigraph

The overall idea of SuMGra is depicted in Algorithm 3.1. Initially, we order the set of query vertices U using a heuristic proposed in Section 3.6.1. With an ordered set of query vertices U^o, we use the indexing schema to find a list of possible candidate matches only for the initial query vertex u_{init} by calling SelectCand (Line 5), as described in Section 3.6.2. Then, for each possible candidate of the initial query vertex, we call the recursive subroutine SubgraphSearch, that performs the subgraph isomorphism test.

The SubgraphSearch procedure (Section 3.6.3), finds the embeddings starting with the possible matches for the initial query vertex u_{init} (Lines 7-11). Since u_{init} has $|C_{u_{init}}|$ possible matches, SubgraphSearch iterates through $|C_{u_{init}}|$ solution trees in a depth first manner until an embedding is found. That is, SubgraphSearch is recursively called to find the matchings that correspond to all ordered query vertices U^o. The partial embedding is stored in $M = [M_q, M_g]$ - a pair that contains the already matched query vertices M_q and the already matched data vertices M_g. Once the partial embedding grows to become a complete embedding, the repository of embeddings R is updated.

Algorithm 3.1: SuMGra

```
INPUT: subgraph $Q$, graph $G$, indexes $\mathcal{S}$, $\mathcal{N}$
OUTPUT: $R$: all the embeddings of $Q$ in $G$
$U^o$ = ORDERQUERYVERTICES($Q$, $G$)
$u_{init} = U^o_1$                              /* Fetch initial query vertex */
$C_{u_{init}}$ = SELECTCAND($u_{init}$, $\mathcal{S}$)
$R = \emptyset$                                  /* Embeddings of $Q$ in $G$ */
for each $v_{init} \in C_{u_{init}}$ do
    $M_q = u_{init}$;                            /* Matched initial query vertex */
    $M_g = v_{init}$;                            /* Matched possible data vertex */
    $M = [M_q, M_g]$                             /* Partial matching of $Q$ in $G$ */
    UPDATE: $R$ := SUBGRAPHSSEARCH($R, M, \mathcal{N}, Q, G, U^o$)
return $R$
```

3.5 Indexing

In this section, we propose the indexing structures that are built on the graph G that are used during the subgraph querying procedure. The primary goal of indexing is to make the query processing time efficient.

Indexing includes (i) an offline *construction phase* to obtain useful features f from the graph to build indexes (ii) an online *querying phase*, where indexes are exploited to enumerate the possible candidate vertices for the corresponding query vertices.

For a lucid understanding of our indexing schema, we introduce a few definitions.

Definition 3.3. Vertex signature. *For a vertex v, the vertex signature $\sigma(v)$ is multiset containing all the multiedges that are incident on v, where any multiedge between v and a neighbouring vertex v' is represented by a set that corresponds to edge types. Formally,*

$$\sigma(v) = \bigcup_{v' \in N(v)} L_E(v, v')$$

where $N(v)$ is the set of neighbourhood vertices of v, and $\cup$ is the union operator for multiset.

For instance, in Figure 3.2a, $\sigma(v_6) = \{\{E_1, E_3\}, \{E_1\}\}$. The vertex signature is an intermediary representation that is exploited by our indexing schema. All the vertex signatures of the vertices of the graph in Figure 3.2 are depicted in Table 3.1.

v_i	$\sigma(v)$
v_1	$\{\{E_1, E_3\}\}$
v_2	$\{\{E_2, E_3, E_1\}, \{E_1\}\}$
v_3	$\{\{E_2, E_3, E_1\}, \{E_1, E_3\}, \{E_1, E_2\}, \{E_1\}\}$
v_4	$\{\{E_1, E_2\}, \{E_1, E_2\}\}$
v_5	$\{\{E_1, E_3\}, \{E_1, E_3\}, \{E_1, E_2\}, \{E_1\}\}$
v_6	$\{\{E_1, E_3\}, \{E_1\}\}$
v_7	$\{\{E_1, E_3\}\}$

Table 3.1: Vertex signatures for the graph in Figure 3.2a

3.5.1 Offline Index Construction

The goal of constructing indexing structures is to find the *possible candidate set* for the set of query vertices u, thereby reducing the search space for the SUBGRAPH-SEARCH procedure, making SUMGRA time efficient.

Definition 3.4. Candidate set. *For a query vertex u, the candidate set $C(u)$ is defined as $C(u) = \{v \in G | \sigma(u) \dot{\subseteq} \sigma(v)\}$, where $\dot{\subseteq}$ is a subset operation on a multiset of a set $\sigma(\cdot)$; i.e., $\sigma(u) \dot{\subseteq} \sigma(v)$ iff each set $S_u \in \sigma(u)$ has a unique superset $S_v \in \sigma(v)$.*

In this light, we propose two indexing structures that are built offline: (i) given the vertex signature of all the vertices of graph g, we construct a single vertex signature index $\mathcal{S}$ by exploring a set of features f of the signature $\sigma(v)$ (ii) we build a vertex neighbourhood index $\mathcal{N}$ for every vertex in the graph G.

The index $\mathcal{S}$ is used to choose the possible candidates for the initial query vertex during the SELECTCAND procedure, and the index $\mathcal{N}$ is used to choose the possible candidates for the rest of the query vertices during SUBGRAPHSEARCH procedure.

Vertex Signature Index $\mathcal{S}$

This index is constructed to enumerate the possible candidate set only for the initial query vertex. Since we cannot exploit any structural information for the initial query vertex, $\mathcal{S}$ captures the edge type information from the data vertices, so that the non suitable candidates can be pruned away.

We construct the index $\mathcal{S}$ by organizing the information supplied by the vertex signature of the graph; i.e., observing the vertex signature of data vertices, we intend to extract some interesting features. Referring to Table 3.1, the vertex signature of v_6, $\sigma(v_6) = \{\{E_1, E_3\}, \{E_1\}\}$ has two sets of edge types in it and hence v_6 is eligible to be matched with query vertices that have at most two sets of items

Data vertex	Synopses					
v	f_1	f_2	f_3	f_4	f_5	f_6
v_1	1	2	2	1	3	2
v_2	2	3	4	1	3	3
v_3	4	3	8	1	3	3
v_4	2	2	4	1	2	2
v_5	4	3	7	1	3	2
v_6	2	2	3	1	3	2
v_7	1	2	2	1	3	2

Table 3.2: Synopses for all the data vertices in Figure 3.2a

in their signature. Also, $\sigma(v_2) = \{\{E_2, E_3, E_1\}, \{E_1\}\}$ has the edge type set of maximum size 3 and hence a query vertex must have the edge type set of size at most 3. More such features (e.g., the number of unique edge types, the total number of occurrences of edge types, etc.) can be proposed to filter out irrelevant candidate vertices. In particular, for each vertex v, we propose to extract a set of characteristics summarizing useful features of the neighbourhood of a vertex. Those features constitute a *synopses* representation (surrogate) of the original vertex signature.

In this light, we propose six $|f| = 6$ useful features that will be illustrated with the help of the vertex signature $\sigma(v_3) = \{\{E_2, E_3, E_1\}, \{E_1, E_3\}, \{E_1, E_2\}, \{E_1\}\}$:

f_1 Cardinality of vertex signature, ($f_1(v_3) = 4$)

f_2 The number of unique edge types in the vertex signature, ($f_2(v_3) = 3$)

f_3 The number of all occurrences of the edge types (repetition allowed), ($f_3(v_3) = 8$)

f_4 Minimum index value of the edge type alphabet (position of the sequenced alphabet), ($f_4(v_3) = 1$)

f_5 Maximum index value of the edge type alphabet (position of the sequenced alphabet), ($f_5(v_3) = 3$)

f_6 Maximum cardinality of the vertex sub-signature, ($f_6(v_3) = 3$)

In Table 3.2 we list the synopses for each data vertex shown in Figure 3.2, for a clear understanding.

By exploiting the aforementioned features, we build the synopses to represent the vertices in an efficient manner that will help us to select the eligible candidates during query processing. To support and explain the choice of synopses features, we conduct few experiments, which will be discussed in detail in Section 3.7.2.

With the synopses representation of every data vertex, we want to represent the synopses with an efficient data structure. Since each vertex is represented by a synopses of several fields, a data structure that helps in efficiently performing range search for multiple elements would be an ideal choice. For this reason, we build an $|f|$-dimensional R-tree, whose vertices are the synopses having $|f|$ fields, where a synopses is nothing but the surrogate representation of the vertices of the multigraph.

The general idea of using an R-tree structure is as follows: A synopses $F = \{f_1, \ldots, f_{|f|}\}$ of a data vertex spans an axes-parallel rectangle in an f-dimensional space, where the maximum co-ordinates of the rectangle are the values of the synopses fields $(f_1, \ldots, f_{|f|})$, and the minimum co-ordinates are the origin of the rectangle (filled with zero values). For example, a data vertex represented by the synopses with two features $F_v = (2, 3)$ spans a rectangle in a 2-dimensional space in the interval range $([0, 2], [0, 3])$. Now if we consider synopses of two query vertices, $F_{u_1} = (1, 3)$ and $F_{u_2} = (1, 4)$, we observe that the rectangle spanned by F_{u_1} is wholly contained in the rectangle spanned by F_v but F_{u_2} is not wholly contained in F_v. Formally, the possible candidates for vertex u can be written as $\mathcal{P}(u) = \{v | \forall_{i \in [1,\ldots,f]} F_{u(i)} \leq F_{v(i)}\}$, where the constraints are met for all the $|f|$-dimensions. Since we apply the same inequality constraint to all the fields, we need to pre-process few synopses fields; e.g., the field f_4 contains the minimum value of the index, and hence we negate f_4 so that the rectangular containment problem still holds good. Thus, we keep on inserting the synopses representations of each data vertex v into the R-tree and build the index $\mathcal{S}$, where each synopses is treated as an $|f|$-dimensional node of the R-tree.

Vertex Neighbourhood Index $\mathcal{N}$

The aim of this indexing structure is to find the possible candidates for the rest of the query vertices.

Since the previous indexing schema enables us to select the possible candidate set $C(u)$ for the initial query vertex, we propose an index structure to obtain the possible candidate set for the subsequent query vertices. The index $\mathcal{N}$ will help us to find the possible candidate set for a query vertex u during the SubgraphSearch procedure by maintaining the structural connectivity with the previously matched candidate vertices, thereby retaining the structural property of the subgraph s for the possible embeddings in the graph.

The index $\mathcal{N}$ comprises of neighbourhood trees built for each of the data vertex v. To understand the index structure, let us consider the data vertex v_3 from

Figure 3.2, shown separately in Figure 3.3a. For this vertex v_3, we collect all the neighbourhood information (vertices and multiedges), and represent this information by a tree structure. Thus, the tree representation of a vertex v contains the neighbourhood vertices and their corresponding multiedges, as shown in Figure 3.3b, where the nodes of the tree structure are represented by the edge types.

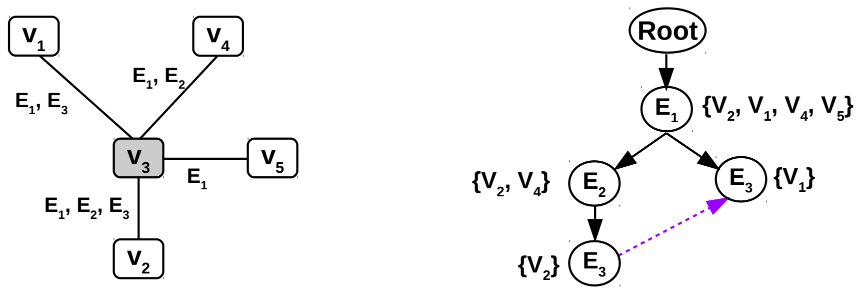
In order to construct an efficient tree structure, we take inspiration from [Terrovitis et al., 2006] to propose the structure - Ordered Trie with Inverted List (OTIL). Consider a data vertex v_i, with a set of n neighbourhood vertices $N(v_i)$. Now, for every pair $(v_i, N^j(v_i))$, where $j \in \{1, \ldots, n\}$, there exists a multiedge (set of edge types) $\{E_1, \ldots, E_d\}$, which is inserted into the OTIL structure. Each multiedge is ordered (with the increasing edge types), before inserting into OTIL structure, and the order is universally maintained for both query and data vertices. Further, for every edge type E_i that is inserted into the OTIL, we maintain an *inverted list* that contains all the neighbourhood vertices $N(v_i)$, that have the edge type E_i incident on them. For example, as shown in Figure 3.3b, the edge E_2 will contain the list $\{v_2, v_4\}$, since E_2 forms an edge between v_3 and both v_2 and v_4.

To construct the OTIL index as shown in Figure 3.3b, we insert each ordered multiedge that is incident on v at the root of the trie structure. To make index querying more time efficient, the OTIL nodes with identical edge type (e.g., E_3) are internally connected and thus form a linked list of data vertices. For example, if we want to query the index in Figure 3.3b with a vertex having edges $\{E_1, E_3\}$, we do not need to traverse the entire OTIL. Instead, we perform a pre-ordered search, and as soon as we find the first set of matches, which is $\{V_2\}$, we will be redirected to the OTIL node, where we can fetch the matched vertices much faster (in this case $\{V_1\}$), thereby outputting the set of matches as $\{V_2, V_1\}$.

To sum up, both the vertex signature index $\mathcal{S}$ and vertex neighbourhood index $\mathcal{N}$ are constructed offline and hence we can afford to invest time on building the index structure. Once constructed, both these indexes will be used for subgraph query processing for any type of subgraph over the indexed multigraph data.

3.6 Subgraph Query Processing

In order to find the embeddings of a subgraph, we not only need to find the valid candidates for each query vertex, but also retain the structure of the subgraph to be matched.



(a) Neighbourhood structure of v_3 (b) OTIL structure for v_3

Figure 3.3: Building Neighbourhood Index for data vertex v_3

3.6.1 Query Vertex Ordering

Before performing query processing, we order the set of query vertices U into an ordered set of query vertices U^o. It is argued that an effective ordering of the query vertices improves the efficiency of subgraph querying [Lee et al., 2012]. In order to achieve this, we propose a heuristic that employs two scoring functions.

The first scoring function relies on the number of multiedges of a query vertex. For each query vertex u_i, the number of multiedges incident on it is assigned as a score; i.e., $r_1(u_i) = \sum_{j=1}^{m} |\sigma(u_i^j)|$, where u_i has m multiedges, $|\sigma(u_i^j)|$ captures the number of edge types in the j^{th} multiedge. Query vertices are ordered in descending order considering the scoring function r_1, and thus $u_{init} = \text{argmax}(r_1(u_i))$. For example, in Figure 3.1b, vertex u_3 has the maximum number of edges incident on it, which is 4, and hence is chosen as an initial vertex. We use the ranking function r_1 to determine the initial vertex in the ordering; for subsequent vertices, we use the following ranking function r_2.

The second scoring function depends on the structure of the subgraph. We maintain an ordered set of query vertices U^o and keep adding the next *eligible* query vertex. In the beginning, only the initial query vertex u_{init} is in U^o. The set of next eligible query vertices U^o_{nbr} are the vertices that are in the 1-neighbourhood of U^o. For each of the next eligible query vertex $u_n \in U^o_{nbr}$, we assign a score depending on a second scoring function defined as $r_2(u_n) = |\{U^o \cap adj(u_n)\}|$. It considers the number of the adjacent vertices of u_n that are present in the already ordered query vertices U^o. The ranking function r_2 is used to order the vertices that are in the 1-neighbourhood of the already ordered partial set of vertices U^o.

Then, among the set of next eligible query vertices U^o_{nbr} for the already ordered U^o, we give first priority to function r_2 and the second priority to function r_1. Thus, in case of any tie ups, w.r.t. r_2, the score of r_1 will be considered. When both r_1 and r_2 leave us in a tie up situation, we resolve by random selection.

3.6.2 Select Candidates for Initial Query Vertex

For the initial query vertex u_{init}, we exploit the index structure $\mathcal{S}$ to retrieve the set of possible candidate data vertices, thereby pruning the unwanted candidates for the reduction of search space.

Theorem 3.1. *Querying the vertex signature index $\mathcal{S}$ constructed with synopses, guarantees to output at least the entire set of valid candidate vertices.*

Proof. Consider the field f_1 in the synopses that represents the cardinality of the vertex signature. Let $\sigma(u)$ be the signature of the query vertex u and $\{\sigma(v_1), \ldots, \sigma(v_n)\}$ be the set of signatures on the data vertices. By using f_1 we need to show that $C(u)$ has at least all the valid candidates. Since we are looking for a superset of query vertex signature, and we are checking the condition $f_1(u) \leq f_1(v_i)$, where $v_i \in \{v_1, \ldots, v_n\}$, v_i is pruned if it does not match the inequality criterion, since it can never be an eligible candidate. We can extend this analogy to all the synopses fields, since they all can be applied disjunctively. □

During the SELECTCAND procedure (Algorithm 3.1, Line 4), we retrieve at least all the valid candidate vertices from the data graph by exploiting the vertex signature index $\mathcal{S}$. However, since querying $\mathcal{S}$ would not prune away all the unwanted vertices (w.r.t. edge type equality) for u_{init}, the corresponding partial embeddings would be discarded during the SUBGRAPHSEARCH procedure. For instance, to find candidate vertices for $u_{init} = u_3$, we build the synopses for u_3 and find the matchable vertices in g using the index $\mathcal{S}$. As we can recall, synopses representation of each data vertex spans a rectangle in the d-dimensional space. Thus, it remains to check, if the rectangle spanned by u_3 is contained in any of rectangles spanned by the synopses of the data vertices, with the help of R-tree built on data vertices, results in the set $\{v_3, v_5\}$.

Once we obtain the candidate vertices for u_{init}, we order the candidate data vertices in the decreasing order of the synopses fields, with decreasing priorities from f_1 to f_6. Thus, if $v_1, \ldots, v_c$ compose the ordered set of candidate vertices, the rectangles spanned by the synopses $F(v_1)$, will be of maximum size and that of $F(v_c)$ will be of minimum size.

3.6.3 Subgraph Searching

The SUBGRAPHSEARCH recursive procedure is described in Algorithm 3.2. Once an initial query vertex u_{init} and its possible data vertex $v_{init} \in C_{u_{init}}$, that could be a potential match, is chosen from the set of select candidates, we have the partial solution pair $M = [M_q, M_g]$ of the subgraph query pattern we want to grow. If v_{init}

Algorithm 3.2: SubgraphSearch($R, M, \mathcal{N}, Q, G, U^o$)

```
1  Fetch $u_{nxt} \in U^o$                       /* Fetch query vertex to be matched */
2  $M_C$ = FindJoinable($M_q, M_g, \mathcal{N}, u_{nxt}$)  /* Matchable candidate vertices */
3  if $|M_C| \neq \emptyset$ then
4      for each $v_{nxt} \in M_C$ do
5          $M_q.push(u_{nxt})$;
6          $M_g.push(v_{nxt})$;
7          $M = [M_q, M_g]$                      /* Partial matching grows */
8          SubgraphSearch($R, M, \mathcal{N}, Q, G, U^o$)
9          if ($|M| == |U^o|$) then
10             $R = R \cup M$                     /* Embedding found */
11         $M_q.pop$;                              /* Remove $u_{nxt}$ */
12         $M_g.pop$;                              /* Remove $v_{nxt}$ */
13 return $R$
```

is a right match for u_{init}, and we succeed in finding the subsequent valid matches for U^o, we will obtain an embedding; else, the recursion would revert back and move on to next possible data vertex to look for the embeddings.

In the beginning of SubgraphSearch procedure, we fetch the next query vertex u_{nxt} from the set of ordered query vertices U^o, that is to be matched (Line 1). Then FindJoinable procedure finds all the valid data vertices that can be matched with the next query vertex u_{nxt} (Line 2). The main task of subgraph matching is done by the FindJoinable procedure, depicted in Algorithm 3.3. Once all the valid matches for u_{nxt} are obtained, we update the solution pair $M = [M_q, M_g]$ (Line 5-7). Then we recursively call SubgraphSearch procedure until all the vertices in U^o have been matched (Line 8). If we succeed in finding matches for the entire set of query vertices U^o, then we update the repository of embeddings (Line 9-10); else, we keep on looking for matches recursively in the search space, until there are no possible candidates to be matched for u_{nxt} (Line 3).

The FindJoinable procedure guarantees the structural connectivity of the embeddings that are outputted. Referring to Figure 3.1, let us assume that the already matched query vertices $M_q = \{u_2, u_3\}$ and the corresponding matched data vertices $M_g = \{v_3, v_5\}$, and the next query vertex to be matched $u_{nxt} = u_1$. Initially, in the FindJoinable procedure, for the next query vertex u_{nxt}, we collect all the neighbourhood vertices that have been already matched, and store them in A_q; formally, $A_q := M_q \cap adj(u_{nxt})$ and also collect the corresponding matched data vertices A_g (Line 1-2). For instance, for the next query vertex u_1, $A_q = \{u_2, u_3\}$ and correspondingly, $A_g = \{v_3, v_5\}$.

Algorithm 3.3: FindJoinable($M_q, M_g, \mathcal{N}, u_{nxt}$)

```
1 $A_q := M_q \cap adj(u_{nxt})$                    /* Matched query neighbours */
2 $A_g := \{v | v \in M_g\}$          /* Corresponding matched data neighbours */
3 Intialize: $M_C^{temp} = 0, M_C = 0$
4 $M_C^{temp} = \cap_{i=1}^{|A_q|}$ NeighIndexQuery($\mathcal{N}, A_g^i, (A_q^i, u_{nxt})$)
5 for each $v_c \in M_C^{temp}$ do
6     if $\sigma(v_c) \supseteq \sigma(u_{nxt})$ then
7         add $v_c$ to $M_C$                      /* A valid matchable vertex */
8 return $M_C$
```

Theorem 3.2. *The algorithm* FindJoinable *guarantees to retain the structure of the embeddings.*

Proof. Consider a query s of size $|u|$. For $n = 1$, let us assume the first matching M_d^1 corresponds to the initial query vertex M_q^1. Now, A_q and A_d contain all the adjacent vertices of the previously matched vertices M_q^1 and M_d^1 respectively, thus maintaining the connectivity with the partially matched solution M. Hence for $n > 1$, by induction, the structure of entire embedding (that corresponds to the subgraph) is retained. □

Now we exploit the neighbourhood index $\mathcal{N}$ in order to find the valid matches for the next query vertex u_{nxt}. With the help of vertex $\mathcal{N}$, we find the possible candidate vertices M_C^{temp} for each of the matched query neighbours A_q^i and the corresponding matched data neighbour A_g^i.

To perform querying on the index structure $\mathcal{N}$, we fetch the multiedge that connects the next matchable query vertex u_{nxt} and the i^{th} previously matched query vertex A_q^i. We now take the multiedge (A_q^i, u_{nxt}) and query the index structure $\mathcal{N}$ of the correspondingly matched data vertex A_g^i (Line 4). For instance, with $A_q^i = u_2$, and $u_{nxt} = u_1$ we have a multiedge $\{E_1, E_2\}$. As we can recall, each data vertex v_j has its neighbourhood index structure $\mathcal{N}(v_j)$, represented by an OTIL structure. The elements that are added to OTIL are nothing but the multiedges that are incident on the vertex v_j, and hence the nodes in the tree are nothing but the edge types. Further, each of these edge types (nodes) maintain a list of neighbourhood (adjacent) data vertices of v_j that contain the particular edge type as depicted in Figure 3.3b. Now, when we look up for the multiedge (A_q^i, u_{nxt}), which is nothing but a set of edge types, in the OTIL structure $\mathcal{N}(A_g^i)$, two possibilities exist. (1) The multiedge (A_q^i, u_{nxt}) has no matches in $\mathcal{N}(A_g^i)$ and hence, there are no matchable data vertices for the next query vertex u_{nxt}. (2) The multiedge (A_q^i, u_{nxt}) has matches in $\mathcal{N}(A_g^i)$ and hence, NeighIndexQuery returns a set of possible candidate vertices M_C^{temp}. The set of vertices M_C^{temp}, present in the OTIL structure as a linked list, are the possible data vertices since, these are the neighbourhood vertices of the already matched data vertex A_g^i, and hence the structure is maintained. For instance, multiedge $\{E_1, E_2\}$ has a set of matched vertices $\{v_2, v_4\}$ as we can observe in Figure 3.3a.

Further, we check if the next possible data vertices are maintaining the structural connectivity with all the matched data neighbours A_g, that correspond to matched query vertices A_q, and hence we collect only those possible candidate vertices M_C^{temp}, that are common to all the matched data neighbours with the help of intersection operation $\cap$. Thus we repeat the process for all the matched query vertices A_q and the corresponding matched data vertices A_g to ensure structural connectivity (Line 4). For instance, with $A_q^1 = u_2$ and corresponding $A_g^1 = v_3$, we have ${M_C}^{temp^1} = \{v_2, v_4\}$; with $A_q^2 = u_3$ and corresponding $A_g^2 = v_5$, we have ${M_C}^{temp^2} = \{v_4\}$, since the multiedge between (A_q^i, u_{nxt}) is $\{E_2\}$. Thus, the common vertex v_4 is the one that maintains the structural connectivity, and hence belongs to the set of matchable candidate vertices $M_C^{temp} = v_4$.

The set of matchable candidates M_C^{temp} are the valid candidates for u_{nxt} both in terms of edge type matching and the structural connectivity with the already matched partial solution. However, at this point, we propose a strategy that predicts whether the further growth of the partial matching is possible, w.r.t. to the neighbourhood of already matched data vertices, thereby pruning the search space. We can do this by checking the condition whether the vertex signature $\sigma(u_{nxt})$ is contained in the vertex signature of $v \in M_C^{temp}$ (Line 11-13). This is possible since, the vertex signature σ contains the multiedge information about the unmatched query vertices that are in the neighbourhood of already matched data vertices. For instance, v_4 can be qualified as M_C since $\sigma(v_4) \supseteq \sigma(u_1)$. That is, considering the fact that we have found a match for u_1, which is v_4, and that the next possible query vertex is u_4, the superset containment check will assure us the connectivity (in terms of edge types) with the next possible query vertex u_4. Suppose a possible candidate data vertex fails this superset containment test, it means that, the data vertex will be discarded by FindJoinable procedure in the next iteration, and we are avoiding this useless step in advance, thereby making the search more time efficient.

In order to efficiently address the superset containment problem between the vertex signatures $\sigma(v_c)$ and $\sigma(u_{nxt})$, we model this task as a maximum matching problem on a bipartite graph [Hopcroft and Karp, 1973]. Basically, we build a bipartite graph whose vertices are the sub-signatures of $\sigma(v_c)$ and $\sigma(u_{nxt})$; and an edge exists between a pair of vertices only if the corresponding sub-signatures do not belong to the same signature, and the i^{th} sub-signature of v_c is a superset of j^{th} sub-signature of u_{nxt}. This construction ensures to obtain at the end a bipartite graph. Once the bipartite graph is built we run a maximum matching algorithm to find a maximum match between the two signatures. If the size of the maximum match found is equal to the size of $\sigma(u_{nxt})$, the superset operation returns true otherwise

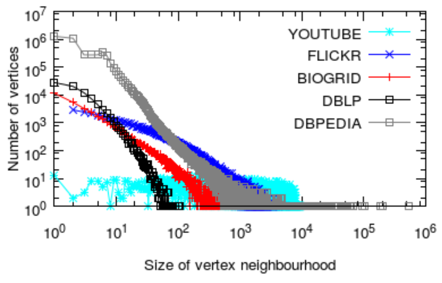
$\sigma(u_{nxt})$ is not contained in the signature $\sigma(v_c)$. To solve the maximum matching problem on the bipartite graph, we employ the *Hopcroft-Karp* [Hopcroft and Karp, 1973] algorithm.

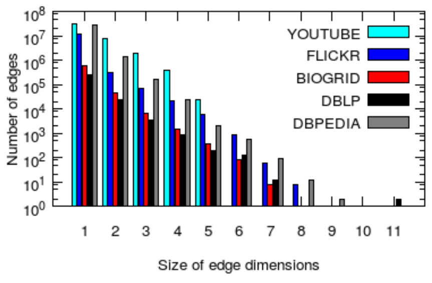
3.7 Experimental Evaluation

In this section, we evaluate the performance of SuMGra on real and synthetic multigraphs and compare it with a state of the art method that is able to manage edge labels. We consider five real world multigraphs that have very different characteristics in terms of size (vertices, edges, edge types) and density. All the experiments were run on a server, with 64-bit Intel 6 processors @ 2.60GHz, and 250GB RAM, running on a Linux OS - Ubuntu. Our methods have been implemented using C++.

3.7.1 Description of Datasets

To validate the correctness, efficiency and versatility of SuMGra, we consider five real world datasets that span over biological and social network data. Further, to test the scalability of our approach, we consider a synthetic data set. All the multigraphs considered in this work are undirected and they do not contain any attribute on the vertices. Table 3.3 offers a quick description of all the characteristics of the benchmarks.





(a) # vertices and vertex neighbourhood size (b) Number of edge types for all the datasets

Figure 3.4: Characteristics of multigraph datasets

Real Datasets

For our analysis, we consider five real world data sets: *DBLP* data set is built by following the procedure adopted in [Boden et al., 2012]. In this graph, the vertices correspond to different authors and the edge types represent the 50 conferences in Computer Science having the highest number of publications. Two authors are connected over an edge type if they co-authored at least one paper together in that conference.

BIOGRID dataset [Bonchi et al., 2014] is a protein-protein interactions network, where vertices represent proteins and the edges represent interactions between the proteins. The data set has seven distinct edge types which correspond to the 7 different types of interactions between a pair of proteins.

FLICKR[1] dataset has been crawled from Flickr, which is an image and video hosting website, web services suite, and an online community. In this data set, the users are represented by vertices, and the blogger's friends are represented using edges (since edge network is the friendship network among the bloggers). In addition, the data set represents the friendship network based on the group memberships (195 in number), which we represent as edge types. Thus multiple edges exist between two users if they have common multiple memberships.

YOUTUBE dataset [Tang et al., 2012] treats users as the vertices and the various connections among them as multiedges. The edge information includes the contacts, mutual-contact, co-subscription network, co-subscribed network: two users are connected if they are both subscribed by the same user and favorite network (two users are connected if they share favourite videos).

DBPEDIA[2] is a knowledge base built by the Semantic Web Community to structure information coming from Wikipedia. The RDF format employed to store such knowledge base can naturally be modeled as a multigraph where vertices are subjects and objects of the RDF triplets and edges represent the predicate between them. Since our framework manages undirected multigraphs, we do not consider edge direction between vertices.

Dataset	Vertices	Edges	Dim	Density	A_{deg}	A_{dim}
DBLP	83 901	141 471	50	4.0e-5	1.7	1.126
BIOGRID	38 936	310 664	7	4.1e-4	8.0	1.103
FLICKR	80 513	5 899 882	195	1.8e-3	73.3	1.046
YOUTUBE	15 088	19 923 067	5	1.8e-1	1320	1.321
DBPEDIA	4 495 642	14 721 395	676	1.4e-6	3.2	1.063
SYNTH	500 000	25 000 000	20	2.0e-4	50	1.15

Table 3.3: Statistics of datasets

To support the analysis of the results, for all the real graphs, we provide the vertex neighbourhood distribution as depicted in Figure 3.4, where the distribution

[1]http://socialcomputing.asu.edu/pages/datasets
[2]http://dbpedia.org/

of the number of vertices with the increasing size of vertex neighbourhood is plotted on a logarithmic scale.

Referring to Figure 3.4 and Table 3.3, we make few observations on the data sets. The *YOUTUBE* data set has a flat spectrum of vertex distribution due to its high density of 1.8e-1, and is mostly concentrated in the region of larger neighbourhood size, given its high average degree $A_{deg} = 1320$. *FLICKR*, *BIOGRID*, *DBLP* and *DBPEDIA* datasets are less dense and hence exhibit a more common power law distribution. Also, as the A_{deg} values reduce from *FLICKR* to *BIOGRID* to *DBPEDIA* and finally to *DBLP*, the distribution shifts towards the smaller neighbourhood size. The sparsest multigraph we consider is *DBPEDIA* that has a density of 1.4e-6 while it exhibits a very high number of edge types and is the biggest real multigraph, in terms of vertices, with more than 4M vertices.

Synthetic Dataset

As previously done in [He and Singh, 2008], we generate a synthetic graph employing the Erdos Renyi (ER) random model, which is a classical random graph generator model. The synthetic graph contains 500 000 vertices. For each vertex, we set the average degree to 50. The resulting synthetic multigraph has 25 million multiedges. We name this multigraph as *SYNTH*.

Description of Query Subgraphs

To test the behavior of our approach, we generate *random* queries and *clique* queries at random, as done by standard subgraph querying methods [He and Singh, 2008, Shang et al., 2008]. The size of the generated queries for random queries vary from 3 to 11 in steps of 2, while for clique queries, we vary the size from 3 to 9. The size of a subgraph is the number of vertices in the subgraph. Since it is hard to inject cliques into the synthetic graphs [Lin and Bei, 2014], we generate only random queries. For the *DBPEDIA* dataset, we are not able to generate enough multigraph clique queries due its high sparsity.

All the generated queries contain one (or more) edge with at least two edge types. In order to generate queries that can have at least one embedding, we sample them from the corresponding multigraph.

For each dataset and query size we obtain 1 000 samples. Following the methodology previously proposed for random query matching algorithms [Han et al., 2013, Lin and Bei, 2014], we report the average time values considering the first 1 000 embeddings for each query. It should be noted that the queries returning no answers were not counted in the statistics (the same statistical strategy has been used by

[Zhao and Han, 2010, He and Singh, 2008, Lin and Bei, 2014]).

Baseline Approaches

We compare the performance of SuMGra w.r.t. the *RI* approach recently proposed in [Bonnici et al., 2013]. The *RI* method is a subgraph isomorphism algorithm that employs light pruning rules in order to avoid visiting useless candidates. The goal of this algorithm is to maintain a balance between the size of the generated search space and the time needed to visit it. It is composed of two main steps, the first one is devoted to find a static order of the query vertices using a set of three heuristics that consider the structure of the subgraph. The second step is the subgraph search procedure that makes use of pruning rules to traverse the search space and find embeddings that match the query. The implementation is obtained from the original authors.

In order to evaluate the effectiveness of our indexing schema we introduce a variant of our proposal, which we call SuMGra-No-SC. This approach constitutes a baseline w.r.t. our proposal. Practically, it does not consider constructing the vertex signature index $\mathcal{S}$, and hence does not select any candidates for the initial query vertex u_{init}. Thus, it initializes the candidate set of the initial vertex $C(u_{init})$ with the whole set of data vertices. This baseline can help us to have a more clear picture about the impact of the $\mathcal{S}$ index over the performance of our submultigraph isomorphism algorithm.

3.7.2 Performance of SuMGra

In Section 3.5, we gave emphasis on constructing the vertex signature index $\mathcal{S}$ to store vertex signatures with the help of synopses representation, and the neighbourhood vertex signature $\mathcal{N}$ to organize vertex neighbourhood by exploiting the set of edge types. We recall that SuMGra constructs both $\mathcal{S}$ and $\mathcal{N}$ offline. While index $\mathcal{S}$ is explored during the query processing, to retrieve valid candidates for the initial query vertex u_{init}, the index $\mathcal{N}$ is used to retrieve neighbourhood vertices in the subgraph search routine.

Table 3.4 reports the index construction time of SuMGra for each of the employed dataset.

All the benchmarks show reasonable time performance and it is strictly related to the size and density of the considered multigraph. As we can observe, construction of the index $\mathcal{N}$ takes more time when compared to the construction of $\mathcal{S}$ for all the datasets except *DBLP*. The behaviour is evident for the bigger datasets like *FLICKR*, *YOUTUBE*, *DBPEDIA* and *SYNTH*, owing to either huge number of edges, or vertices or both. Considering *DBLP* and *BIOGRID*, we can note that the difference in time construction is strictly related to the size in terms of vertices and edges of the two benchmarks. *DBLP* has a large number of vertices when compared to *BIOGRID*, which influences the construction time of the $\mathcal{S}$ index while the construction time of the $\mathcal{N}$ index reflects the difference in terms of edge size between the two data sets.

Among all the datasets, *DBPEDIA* is the most expensive dataset to construct both $\mathcal{S}$ and $\mathcal{N}$, since it has huge number of vertices and relatively more edges.

In Table 3.4, we also give an overall picture of the memory consumption of our proposed algorithm. We capture the memory usage during the runtime when we build our indexing structures. As we can observe, the cost of storing the index structures increases with increasing density of graphs, as well as with the increasing number of vertices and edges. Among all data sets, *YOUTUBE* is the most expensive in terms of space consumption.

To conclude, we highlight that the offline step is fast enough since, in the worst case, for *DBPEDIA*, we need a bit more than two minutes to index 4 million vertices and 14 million edges, with a reasonable memory consumption.

Data set	Index $\mathcal{S}$	Index $\mathcal{N}$	Index $\mathcal{S} + \mathcal{N}$
	Time (seconds)	Time (seconds)	Size (Mega bytes)
DBLP	1.15	0.37	161
BIOGRID	0.45	0.50	266
FLICKR	1.55	8.89	448
YOUTUBE	1.55	41.81	862
DBPEDIA	64.51	66.59	552
SYNTH	9.15	38.70	438

Table 3.4: Execution time and memory usage for offline index construction

Query Processing Time

Figures 3.5-3.11 summarise the time performance of SuMGra. All the times we report are in milliseconds; the Y-axis (logarithmic in scale) represents the query matching time, which includes query processing time, query ordering time, time required to select the candidate vertices for the initial query vertex and the subgraph matching time; the X-axis represents the increasing query sizes. Except for *DB-PEDIA* and *SYNTH* datasets (due to unavailability of clique queries), we produce plots for both random subgraph and clique queries.

We also analyse the time performance of SuMGra by varying the number of

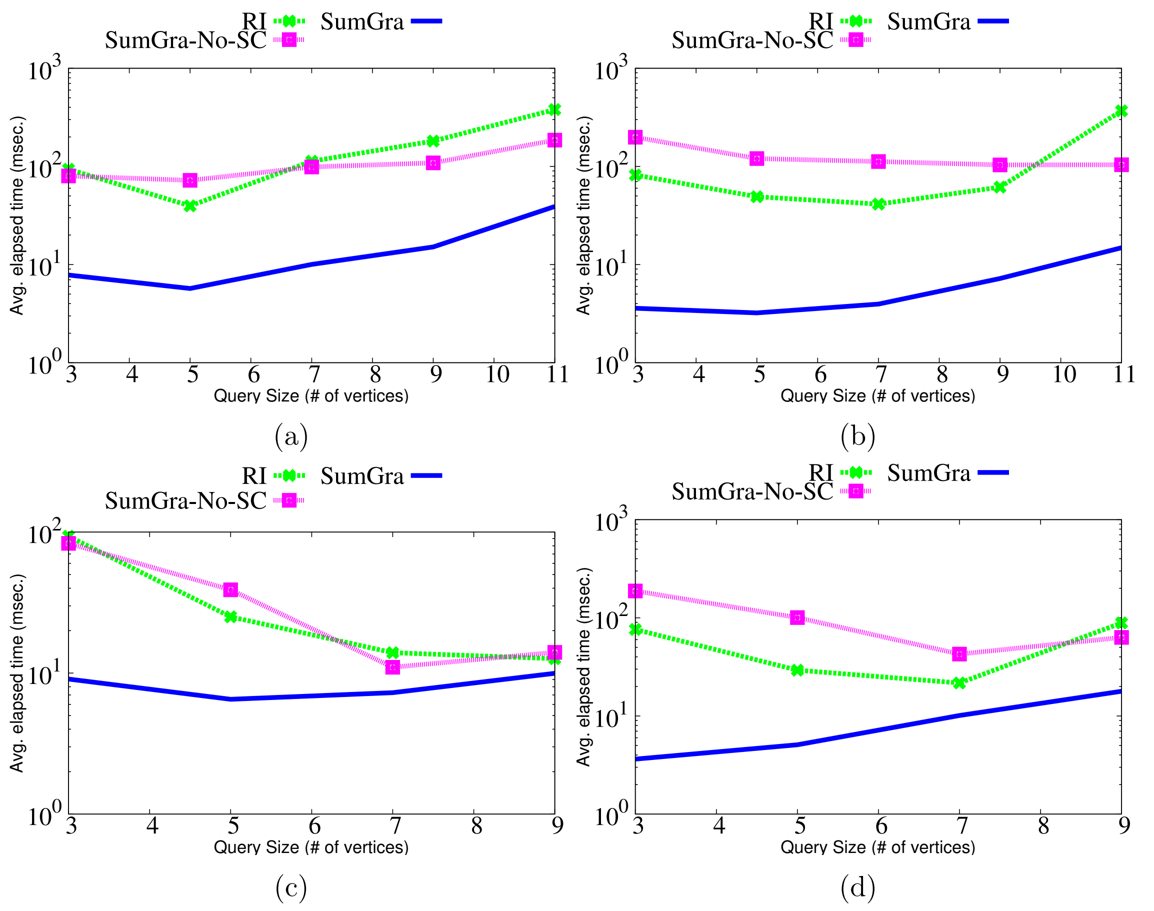


Figure 3.5: Query Time on *DBLP* for (a) Random subgraphs with d=2 (b) Random subgraphs with d=4 (c) Cliques with d=2 (d) Cliques with d=4

edge types in the subgraph. We perform experiments for query multigraphs with two different edge types: $d = 2$ and $d = 4$: a query with $d = 2$ has at least one edge that exists in at least 2 edge types. The same analogy applies to queries with $d = 4$. We use both setting to generate random subgraph and clique queries.

For *DBLP* dataset, we observe in Figure 3.5 that SumGra performs the best in all the situations, it outperforms the other approaches by a huge margin thanks to the rigorous pruning of candidate vertices for initial query vertex. However, SumGra-No-SC approach and RI give a tough competition to each other. Since *DBLP* is a relatively small and yet sparse dataset, the only indexing $\mathcal{N}$ used by SumGra-No-SC seems to cause a little bit of overhead even when compared to RI.

Figure 3.6 for *BIOGRID* and Figure 3.7 for *FLICKR* show similar behaviour for both random subgraph and clique queries. For these 2 datasets, both SumGra and SumGra-No-SC outperform RI. For many query instances, especially for *FLICKR*, SumGra-No-SC obtains better performance than RI while SumGra

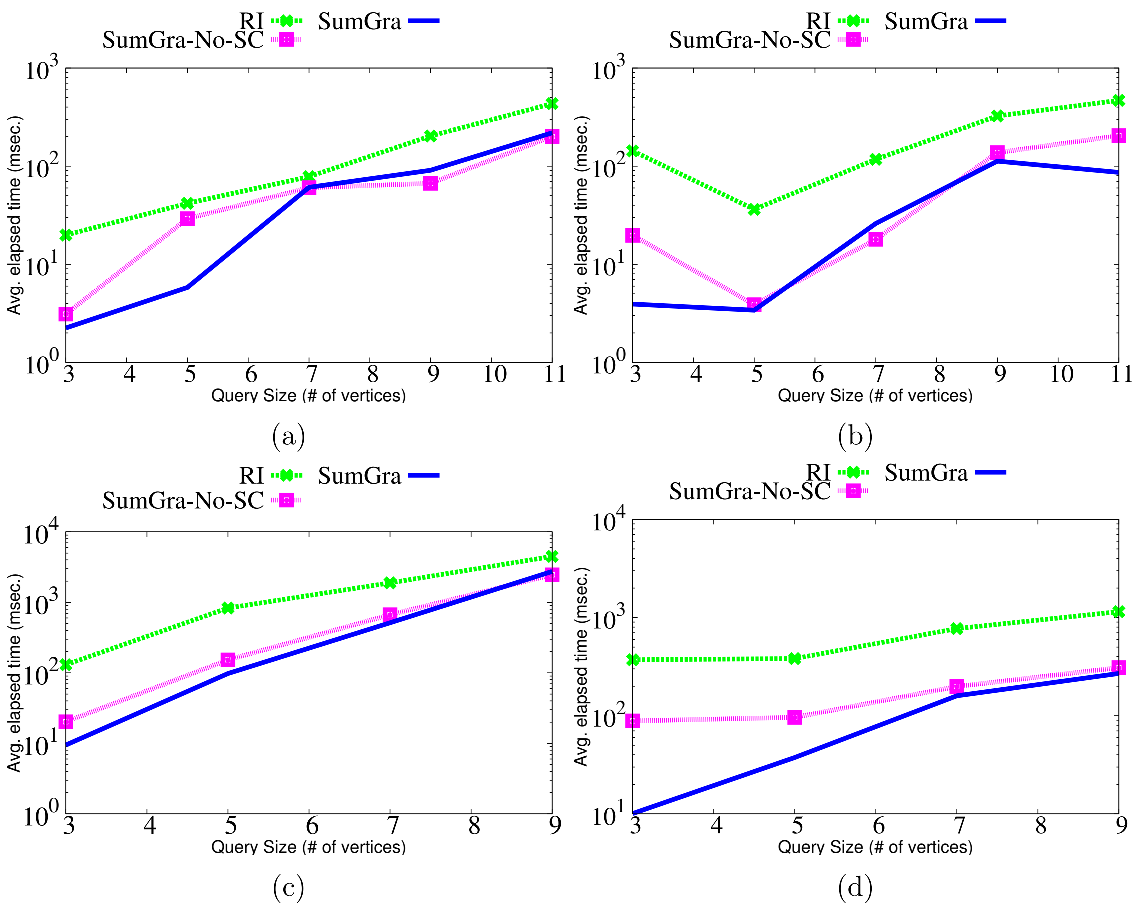


Figure 3.6: Query Time on *BIOGRID* for (a) Random subgraphs with d=2 (b) Random subgraphs with d=4 (c) Cliques with d=2 (d) Cliques with d=4

still outperforms both competitors.

For *YOUTUBE* dataset (Figure 3.8), again SuMGra is the clear winner. However, in this case, *RI* is better than SuMGra-NO-SC, for random queries, although SuMGra-NO-SC is better than *RI* for cliques. This could be the case because, cliques exploit the neighbourhood structure to the maximum extent and thanks to the vertex neighbourhood indexing scheme $\mathcal{N}$, they both can outperform *RI*. Since random subgraph queries do not exploit much of the neighbourhood information, and due to the very high density of the data graph, SuMGra-NO-SC has a poor performance.

Moving to *DBPEDIA* dataset in Figure 3.9, we observe a significant deviation between *RI* and SuMGra, with SuMGra winning by a huge margin. Even for *SYNTH* dataset (Figure 3.11), SuMGra and SuMGra-No-Sc, outperform *RI*.

To conclude, we note that SuMGra outperforms the considered base line approaches, for a variety of different real datasets as well as synthetic dataset. Its

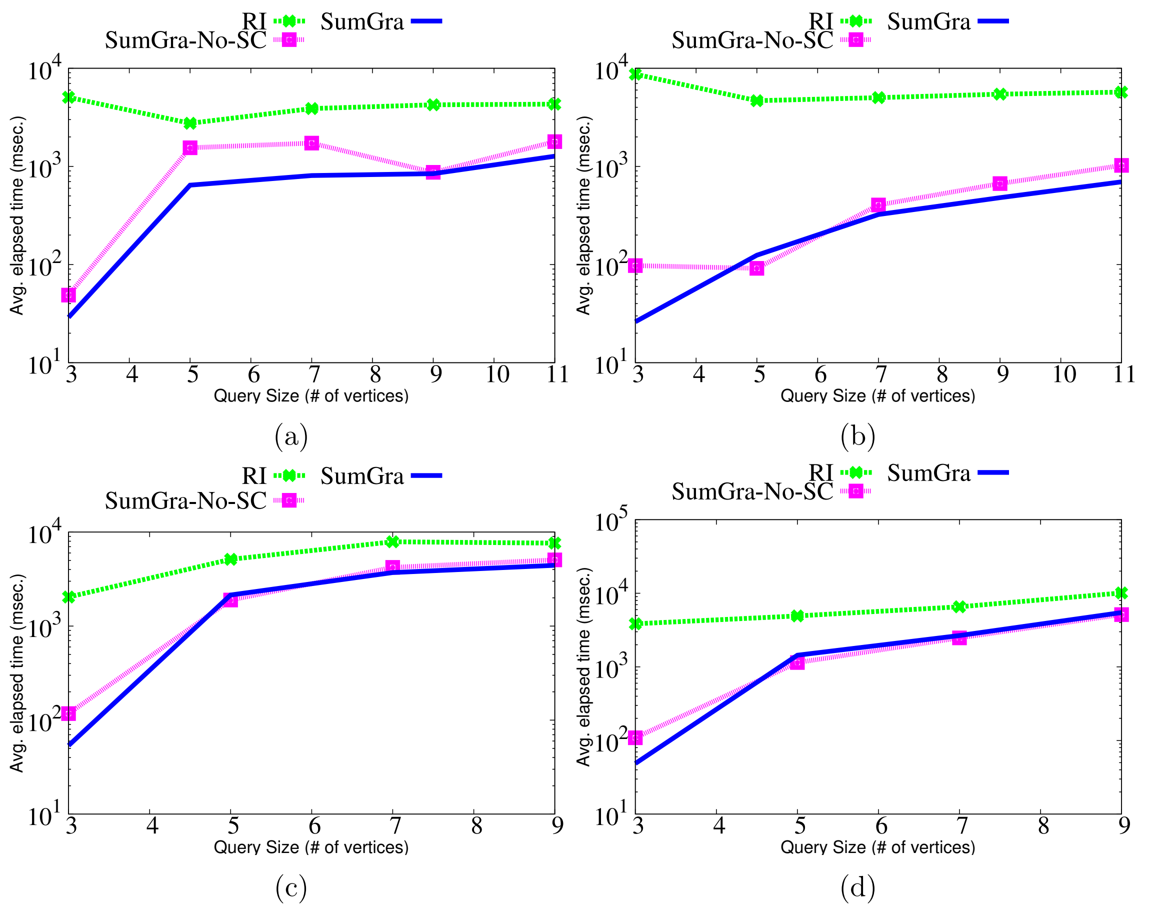
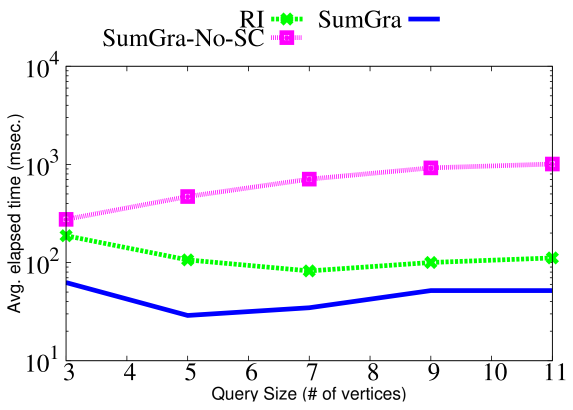


Figure 3.7: Query Time on *FLICKR* for (a) Random subgraphs with d=2 (b) Random subgraphs with d=4 (c) Cliques with d=2 (d) Cliques with d=4

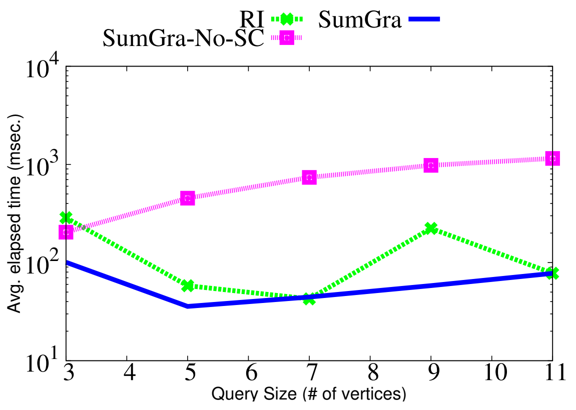
performance is reported as best for small datasets - *DBLP* and *BIOGRID*, for multigraphs having many edge types - *FLICKR*, *DBPEDIA*, high density - *YOUTUBE*, high sparsity - *DBPEDIA* and synthetic dataset *SYNTH*. Thus, we highlight that SuMGra is robust in terms of time performance considering both subgraph and clique queries, with varying edge types.

Assessing the Set of Synopses Features

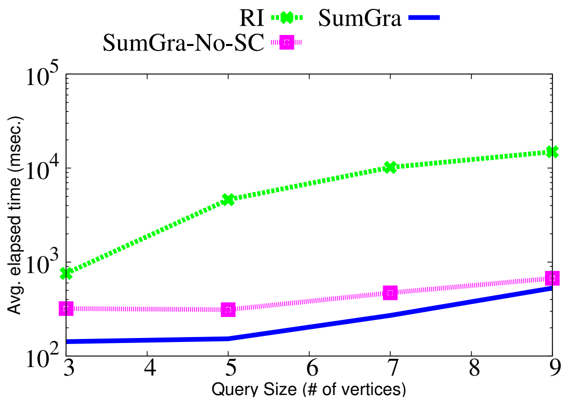
In this section we assess the quality of the features composing the synopses representation for our indexing schema. To this end, we vary the features we consider to build the synopses representation to understand if some of the features can be redundant and/or do not improve the final performance. Since visualizing the combination of the whole set of features will be hard, we limit this experiment to a subset of combinations. Hence, we choose to vary the size of the feature set from



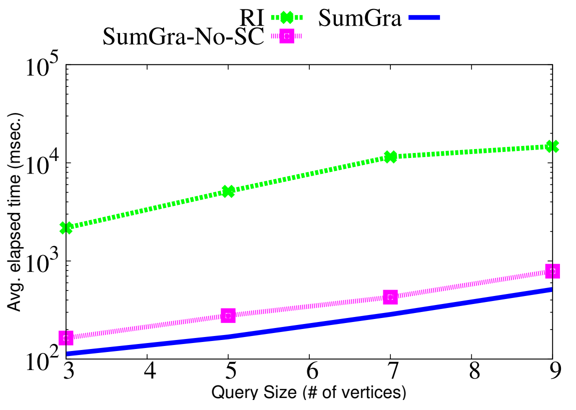
(a)



(b)

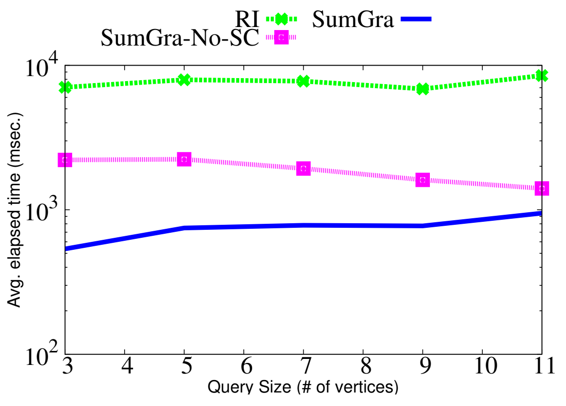


(c)



(d)

Figure 3.8: Query Time on *YOUTUBE* for (a) Random subgraphs with d=2 (b) Random subgraphs with d=4 (c) Cliques with d=2 (d) Cliques with d=4



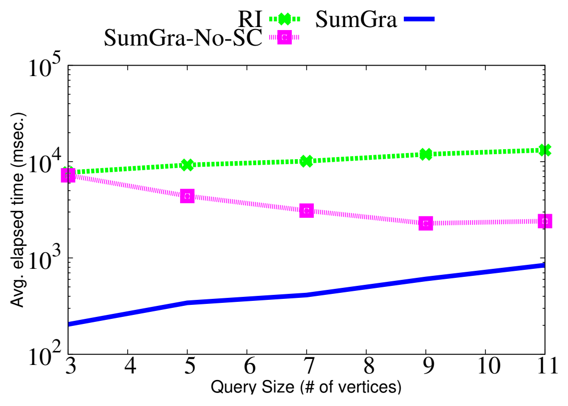


Figure 3.9: Query Time on *DBPEDIA* for: (a) d=2 (b) d=4

one to six, by considering the order defined in Section 3.5.1. Using all the six features results in the proposed approach SUMGRA. We denote the different configuration with the number of features it contains; for instance $|f| = 3|$ means that it considers only three features to build synopses and in particular it employs the feature set $\{f_1, f_2, f_3\}$. We also compare these six tests with the SuMGra-No-SC approach, where no synopses is used and hence no candidates are selected for the initial query vertex. For the sake of representation, we report only plots for two datasets: *DBLP* for subpgraph queries with $d = 4$ and *YOUTUBE* with subgraph queries with $d = 2$. We select those datasets as they represent cases in which our indexing schema reach the best gain in time efficiency.

Results are reported in Figure 3.10. We can note that, considering the entire set of features drastically improves the time performance, when compared to a subset of these six features. This behaviour can be highlighted for the subgraphs of almost all size. This experiment provides evidence about the usefulness of considering the entire feature set to build synopsis. The different features are not redundant and they are all helpful in pruning the useless data vertices.

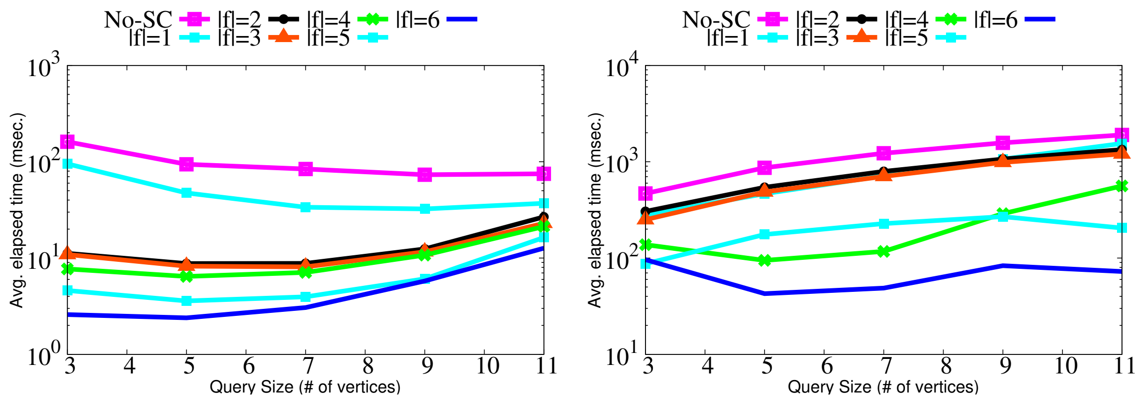


Figure 3.10: Query time with varying synopses fields for: (a) *DBLP* dataset with d=4 (b) *YOUTUBE* dataset with d=2

Scalability of SuMGra

To verify the scalability of SuMGra, we conduct experiments on the synthetic graph previously introduced. *SYNTH* has 500 000 vertices, 25 000 000 edges that span over 20 edge types. In Figure 3.11, we report the time required to output the first 1 000 embeddings of subgraphs with d=2 and d=4, for random subgraphs. Although we can already observe the best time performance of SuMGra among the dense and huge real datasets - *YOUTUBE*, *DBPEDIA*, we conduct test on synthetic datasets as well, by injecting huge number of multiedges. Again, we can observe that, also on this huge graph, SuMGra is able to find the embeddings much faster than its competitors, thus underlining the scalability and the robustness of the proposed approach.

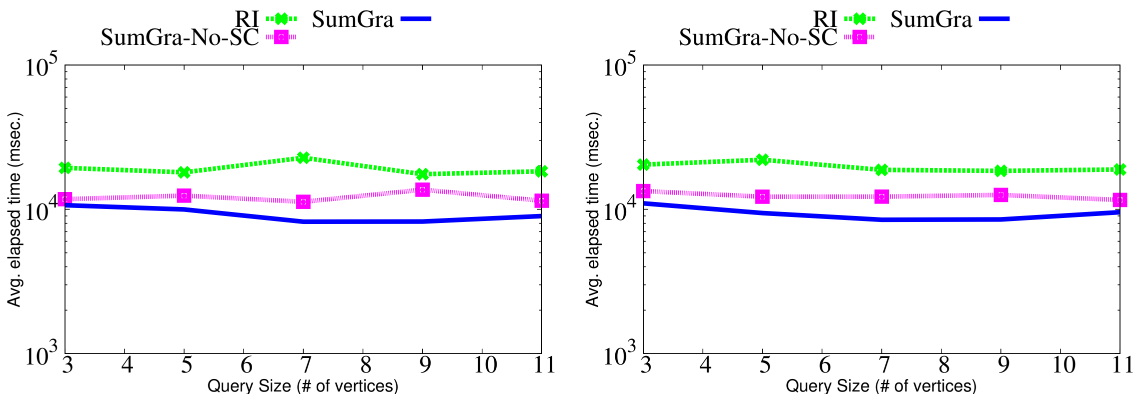


Figure 3.11: Query Time on *Synthetic* for: (a) d=2 (b) d=4

3.8 Summary

In this chapter, the primary goal has been to propose an efficient subgraph query matching algorithm for undirected multigraphs. In this direction, we proposed an efficient approach called SuMGra. Since multigraphs are rich with multiedge information, we proposed several indexing schema that leverage the rich structure available in the multigraph. One of the proposed indexing structures - *vertex signature index* - considers several graph theoretical properties and are collectively represented as synopses, which is further organized as an R-Tree structure. Another index structure called *vertex neighbourhood index* stores the neighbourhood multiedge information of each vertex in the data multigraph.

These index structures are then exploited by a subgraph search procedure that works on multigraphs. The subgraph search procedure follows the backtracking schema, by carefully allowing an optimized order on the query vertices. The *vertex signature index* plays a vital role in pruning the search space to look for subgraph query matches. The experimental section highlights the efficiency, versatility and scalability of our approach over different real datasets. The comparison with a state of the art approach holds a convincing argument that it is indeed necessary to propose efficient approaches to manage multigraphs. This work has a considerable follow up work, where one can extend SuMGra for labelled multigraphs. Extending the work to labelled vertices is pretty straightforward. We can simply maintain a prefix tree of vertex labels, where a label maintains an inverted list of vertices to which the label belongs to. Such labelled multigraphs, for instance, find prominent applications in the field of e-commerce networks where a multigraph generated by the product ratings in an e-commerce network can have a user-product pair (vertices) that can share several relations (multiedges) and each relation itself can be labelled. One instantiation could be a user is connected to a product with a relation of *user rating* (an edge type), and the user rating itself would range from values 1 to 5. Another relation of timestamps, when a user bought the product, can also have several labels on the relation of time stamp. In such scenarios, it is necessary to generalize the existing SuMGra approach.

CHAPTER 4

Querying RDF Data

In this chapter, we address the problem of subgraph query matching in the context of RDF data. We introduce a framework to model RDF data as RDF graph and propose a novel RDF querying engine - AMbER, *that is specifically designed to optimize the computation of matching solutions for complex queries. We conduct extensive experiments to compare our proposed* AMbER *with the existing state-of-the-art RDF querying systems.*

4.1 Introduction

In the previous chapter we discussed about querying the isomorphic matches in multigraphs of various domains. As already discussed in the introduction (Chapter 1), apart from isomorphic matches, a variety of query matching operations are possible; one such operations is retrieving homomorphic matches. Homomorphic matches can be used to query a vast amount of data that has been emerging lately, called RDF data.

Resource Description Framework (RDF) is a standard for the conceptual description of knowledge. The RDF data is cherished and exploited by various domains such as life sciences, Semantic Web, social network, etc. Further, its integration at Web-scale compels RDF management engines to deal with complex queries in terms of both size and structure. Popular examples are provided by Google, that exploits the so called *knowledge graph* to enhance its search results with semantic information gathered from a wide variety of sources, or by Facebook, that implements the so called *entity graph* to empower its search engine and provide further information extracted, for instance by Wikipedia. Another example is supplied by recent question-answering systems [Cabrio et al., 2012, Zou et al., 2014a] that automatically translate natural language questions in SPARQL queries and successively retrieve answers by considering the available information in the different Linked Open Data sources. In all these examples, complex queries (in terms of size and structure) are generated to ensure the retrieval of all the required information. Since the use of large knowledge bases that are commonly stored as RDF triplets is becoming a common way to ameliorate a wide range of applications, efficient querying of RDF data sources using SPARQL[1] (query language conceived to query RDF data) is becoming crucial for modern information retrieval systems.

All these different scenarios pose new challenges to the RDF query engines for two vital reasons: firstly, the automatically generated queries cannot be bounded by their structural complexity and size (e.g., the DBPEDIA SPARQL Benchmark [Morsey et al., 2011] contains some queries having more than 50 triplets [Aluç et al., 2014]); secondly, the queries generated by retrieval systems (or by any other applications) need to be efficiently answered in a reasonable amount of time. Modern RDF data management, such as *x-RDF-3X* [Neumann and Weikum, 2010] and *Virtuoso* [Erling, 2012], are designed to address the scalability of SPARQL queries but they still have problems to answer big and structurally complex SPARQL queries [Aluç et al., 2014].

In the Semantic Web context, question-answering systems (e.g., [Cabrio et al., 2012, Zou et al., 2014a]) over Linked Open Data (LOD) experience the same issues, given ∼31 billion triples contained in this collection of data sets [Aluç et al., 2014] and the complexity of the SPARQL queries that need to involve several properties.

In this chapter we address the following research questions.

Q_1 Can we efficiently model the RDF data as a multigraph and perform SPARQL querying?

Q_2 How can we make a graph database (multigraph model of RDF data) approach outperform the existing relational database approaches?

Q_3 What graph theoretical concepts have to be incorporated in order to perform SPARQL equivalent querying?

To answer Q_1, in this chapter, we introduce AMbER (***A****ttributed* ***M****ultigraph* ***B****ased* ***E****ngine for* ***R****DF querying*), which is a graph-based RDF querying engine. RDF data is represented as a multigraph where subjects/objects constitute vertices and multiple edges (predicates) can appear between the same pair of vertices.

[1]http://www.w3.org/TR/sparql11-overview/

The second question Q_2 is partially answered by the approach presented in the previous chapter where we have proved that an offline stage, that involves constructing indexes is followed by query matching process. In the case of RDF data, we apply the same philosophy but consider the specificity of the RDF data. Thus, the second question Q_2 is answered rather in two stages: (i) an offline stage where the RDF data is transformed into multigraph with dedicated indexes curated to leverage the muligraph properties; (ii) an online stage where we exploit the topological properties of a query to optimize the performance of query matching. To answer query Q_3, we also model SPARQL queries as multigraphs, and the query answering task can be reduced to the problem of enumerating homomorphic matches of a query multigraph in an RDF multigraph. Other graph theoretical concepts like neighbourhood structure of vertices, vertex degrees are also explored.

4.2 Related Work

The proliferation of semantic web technologies has influenced the popularity of RDF as a standard to represent and share knowledge bases. In order to efficiently answer SPARQL queries, many stores and API inspired by relational model were proposed [Erling, 2012, Broekstra et al., 2002, Neumann and Weikum, 2010, Carroll et al., 2004]. x-RDF-3X [Neumann and Weikum, 2010], inspired by modern RDBMS, represent RDF triples as a big three-attribute table. The RDF query processing is boosted using an exhaustive indexing schema coupled with statistics over the data. Also Virtuoso [Erling, 2012] heavily exploits RDBMS mechanism in order to answer SPARQL queries. Virtuoso is a column-store based systems that employs sorted multi-column column-wise compressed projections. Also these systems build table indexing using standard B-trees. Jena [Carroll et al., 2004] supplies API for manipulating RDF graphs. Jena exploits multiple-property tables that permit multiple views of graphs and vertices which can be used simultaneously.

Recently, the database community has started to investigate RDF stores based on graph data management techniques [Das et al., 2014, Zou et al., 2014b, Shang et al., 2008]. The work in [Das et al., 2014] addresses the problem of supporting property graphs as RDF, since majority of the graph databases are based on property graph model. The authors introduce a property graph to RDF transformation scheme and propose three models to address the challenge of representing the key/-value properties of property graph edges in RDF. gStore [Zou et al., 2014b] applies graph pattern matching techniques using filter-and-refinement strategy to answer SPARQL queries. It employs an indexing schema, named VS*-tree, to concisely represent the RDF graph. Once the index is built, it is used to find promising subgraphs that match the query. Finally, exact subgraphs are enumerated in the refinement step. TurboHom++ [Shang et al., 2008] is an adaptation of a state of the art subgraph isomorphism algorithm (TurboISO [Han et al., 2013]) to the problem of

SPARQL queries. Exploiting the standard graph isomorphism problem, the authors relax the injectivity constraint to handle the graph homomorphism, which is the RDF pattern matching semantics.

Unlike our approach, TurboHom++ does not index the RDF graph, while gStore concisely represents RDF data through VS*-tree. Another difference between AMbER and the other graph stores is that our approach explicitly manages the multigraph induced by the SPARQL queries while no clear discussion is supplied for the other tools.

4.3 Background and Preliminaries

In this section we provide basic definitions on the interplay between RDF and its multigraph representation. Later, we explain how the task of answering SPARQL queries can be reduced to multigraph homomorphism problem.

4.3.1 RDF Data

As per the W3C[2] standards, RDF data is represented as a set of triples $<S, P, O>$, as shown in Figure 4.1a, where each triple $<s, p, o>$ consists of three components: a *subject*, a *predicate* and an *object*. Further, each component of the RDF triple can be of any two forms; an *IRI* (Internationalized Resource Identifier) or a literal. For brevity, an *IRI* is usually written along with a prefix (e.g., `<http://dbpedia.org/resource/isPartOf>` is written as 'x:isPartOf'), whereas a literal is always written with double quotes (e.g., "90000"). While a subject s and a predicate p are always an *IRI*, an object o is either an *IRI* or a literal.

RDF data can also be represented as a directed graph where, given a triple $<s, p, o>$, the subject s and the object o can be treated as vertices and the predicate p forms a directed edge from s to o, as depicted in Figure 4.1b. Further, to underline the difference between an *IRI* and a literal, we use standard rectangles and arc for the former while we use beveled corner and edge (no arrows) for the latter.

Data Multigraph Representation

Motivated by the graph representation of RDF data (Figure 4.1b), we take a step further by transforming it to a data multigraph G, as shown in Figure 4.1c.

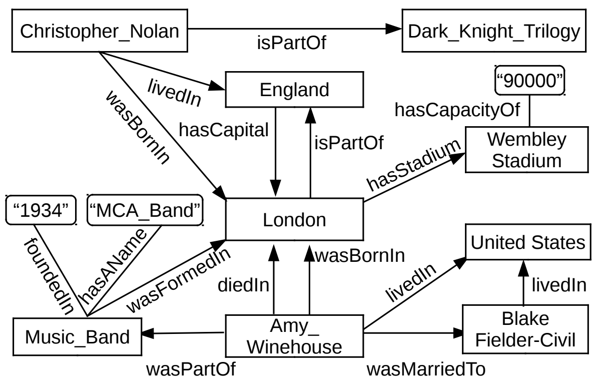
Let us consider an RDF triple $<s, p, o>$ from the RDF tripleset $<S, P, O>$.

[2]`http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`

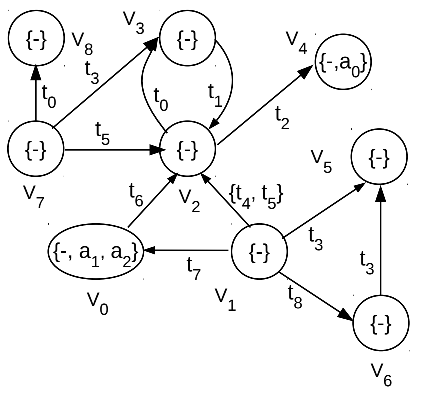
Prefixes: x= http://dbpedia.org/resource/ ; y=http://dbpedia.org/ontology/

Subject	***Predicate***	***Object***
x:London	y:isPartOf	x:England
x:England	y:hasCapital	x:London
x:Christophar_Nolan	y:wasBornIn	x:London
x:Christophar_Nolan	y:LivedIn	x:England
x:Christophar_Nolan	y:isPartOf	x:Dark_Knight_Trilogy
x:London	y:hasStadium	x:WembleyStadium
x:WembleyStadium	y:hasCapacityOf	“90000”
x:Amy_Winehouse	y:wasBornIn	x:London
x:Amy_Winehouse	y:diedIn	x:London
x:Amy_Winehouse	y:wasPartOf	x:Music_Band
x:Music_Band	y:hasName	“MCA_Band”
x:Music_Band	y:FoundedIn	“1994”
x:Music_Band	y:wasFormedIn	X:London
x:Amy_Winehouse	y:livedIn	x:United States
x:Amy_Winehouse	y:wasMarriedTo	x:Blake Fielder-Civil
x:Blake Fielder-Civil	y:livedIn	x:United States

(a) RDF tripleset



(b) Graph representation of RDF data



(c) Equivalent multigraph G

Figure 4.1: (a) RDF data in n-triple format; (b) graph representation; (c) and the attributed multigraph G

Now to transform the RDF tripleset into data multigraph G, we set four protocols: (i) we always treat the subject s as a vertex; (ii) a predicate p is always treated as an edge; (iii) we treat the object o as a vertex only if it is an *IRI* (e.g., vertex v_2 corresponds to object 'x:London'); (iv) when the object is a literal, we combine the object o and the corresponding predicate p to form a tuple $<p, o>$ and assign it as an attribute to the subject s (e.g., <'y:hasCapacityOf', "90000"> is assigned to vertex v_4). Every vertex is assigned a null value {-} in the attribute set. However, to realize this in the realms of graph management techniques, we maintain three different dictionaries, whose elements are a pair of 'key' and 'value', and a mapping function that links them. The three dictionaries depicted in Table 4.1 are: a vertex dictionary (Table 4.1a), an edge-type dictionary (Table 4.1b) and an attribute dictionary (Table 4.1c). In all the three dictionaries, an RDF entity represented by a 'key' is mapped to a corresponding 'value', which can be a vertex/edge/attribute identifier. Thus by using the mapping functions - $\mathcal{M}_v$, $\mathcal{M}_e$, and $\mathcal{M}_a$ for vertex, edge-type and attribute mapping respectively, we obtain a directed, vertex attributed data multigraph G (Figure 4.1c). We now introduce the variant of the multigraph definition (as introduced in Definition 2.3), restricted to vertex labelled, directed multigraphs.

Definition 4.1. Directed, Vertex Attributed Multigraph. *A directed, vertex attributed multigraph G is defined as a 4-tuple (V, E, L_V, L_E) where V is a set of vertices, $E \subseteq V \times V$ is a set of directed edges with $(v, v') \neq (v', v)$, L_V is a labelling function that assigns a subset of vertex attributes A to the set of vertices V, and L_E is a labelling function that assigns a subset of edge-types T to the edge set E.*

To summarise, an RDF tripleset is transformed into a data multigraph G, whose elements are obtained by using the mapping functions as already discussed. Thus, the set of vertices $V = \{v_0, \ldots, v_m\}$ is the set of mapped subject/object *IRI*, and the labelling function L_V assigns a set of vertex attributes $A = \{\text{-}, a_0, \ldots, a_n\}$ (mapped tuple of predicate and object-literal) to the vertex set V. The set of directed edges E is a set of pair of vertices (v, v') that are linked by a predicate, and the labelling function L_E assigns the set of edge types $T = \{t_0, \ldots, t_p\}$ (mapped predicates) to these set of edges. The edge set E maintains the topological structure of the RDF data. Further, mapping of object-literals and the corresponding predicates as a set of vertex attributes, results in a compact representation of the multigraph. For example (in Figure 4.1c), all the object-literals and the corresponding predicates are reduced to a set of vertex attributes.

4.3.2 SPARQL Query

A SPARQL query usually contains a set of triple patterns, much like RDF triples, except that any of the subject, predicate and object may be a variable, whose bindings are to be found in the RDF data[3]. In the current work, we address the SPARQL queries with 'SELECT/WHERE' clause of the SPARQL language[4], that constitutes the most important operation of any RDF query engines. The `SELECT` clause identifies the variables to appear in the query results while the WHERE clause provides triple patterns to match against the RDF data. It is out of the scope of this work to consider operators like `FILTER`, `UNION` and `GROUP BY` or manage RDF update. Such operations, as of now, are left for future extensions.

In the current work, we address the SPARQL queries that have subject/object as unknown variables, and the predicate as a known entity, where the predicate is always instantiated as an *IRI*, as seen in Figure 4.2a..

Query Multigraph Representation

In any valid SPARQL query (as in Figure 4.2a), every triplet has at least one unknown variable $?X$, whose bindings are to be found in the RDF data. It should now be easy to observe that a SPARQL query can be represented in the form of a graph as in Figure 4.2b, which in turn is transformed into query multigraph Q (as in Figure 4.2c).

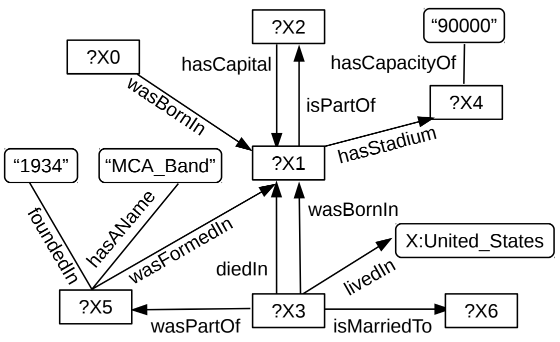
In the query multigraph representation, each unknown variable $?X_i$ is mapped to a vertex u_i that forms the vertex set U component of the query multigraph Q (e.g., $?X_6$ is mapped to u_6). Since a predicate is always instantiated as an *IRI*, we use the edge-type dictionary in Table 4.1b, to map the predicate to an edge-type identifier $t_i \in T$ (e.g., 'isMarriedTo' is mapped as t_8). When an object o_i is a literal, we use the attribute dictionary (Table 4.1c), to find the attribute identifier a_i for the predicate-object tuple $<p_i, o_i>$ (e.g., $\{a_0\}$ forms the attribute for vertex u_4). Further, when a subject or an object is an *IRI*, which is a not a variable, we use the vertex dictionary (Table 4.1a), to map it to an *IRI*-vertex u_i^{iri} (e.g., 'x:United-States' is mapped to u_0^{iri}) and maintain a set of IRI vertices R. Since this vertex is not a variable and hence not a real vertex of the query, we portray it differently by a shaded square shaped vertex. When a query vertex u_i does not have any vertex attributes associated with it (e.g., u_0, u_1, u_2, u_3, u_6), a null attribute {-} is assigned to it. On the other hand, an *IRI*-vertex $u_i^{iri} \in R$ does not have any attributes. Thus, a SPARQL query is transformed into a query multigraph Q.

In this work, we always use the notation V for the set of vertices of G, and U for the set of vertices of Q. Consequently, a data vertex $v \in V$, and a query vertex $u \in U$. Also, an incoming edge to a vertex is positive (default), and an outgoing

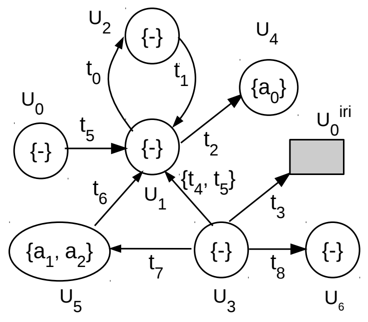
[3] http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/
[4] http://www.w3.org/TR/sparql11-overview/

```
SELECT ?X0 ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 WHERE {
    ?X0    y:livedIn         ?X1 .
    ?X1    y:isPartOf        ?X2 .
    ?X2    y:hasCapital      ?X1 .
    ?X1    y:hasStadium      ?X4 .
    ?X3    y:wasBornIn       ?X1 .
    ?X3    y:diedIn          ?X1 .
    ?X3    y:isMarriedTo     ?X6 .
    ?X3    y:wasPartOf       ?X5 .
    ?X5    y:wasFormedIn     ?X1 .
    ?X4    y:hasCapacity     “90000” .
    ?X5    y:hasName         “MCA_Band” .
    ?X5    y:foundedIn       “1934” .
    ?X3    y:livedIn         x:United States . }
```

(a) SPARQL Query



(b) Graph representation of SPARQL



(c) Equivalent Multigraph Q

Figure 4.2: (a) SPARQL query representation; (b) graph representation (c) attributed multigraph Q

s/o	$\mathcal{M}_v(s/o)$
x:Music_Band	v_0
x:Amy_Winehouse	v_1
x:London	v_2
x:England	v_3
x:WembleyStadium	v_4
x:United States	v_5
x:Blake Fielder-Civil	v_6
x:Christopher-Nolan	v_7
x:Dark-Knight-Trilogy	v_8

(a) Vertex Dictionary

p	$\mathcal{M}_e(p)$
y:isPartOf	t_0
y:hasCapital	t_1
y:hasStadium	t_2
y:livedIn	t_3
y:diedIn	t_4
y:wasBornIn	t_5
y:wasFormedIn	t_6
y:wasPartOf	t_7
y:wasMarriedTo	t_8

(b) Edge-type Dictionary

$<p, o>$	$\mathcal{M}_a(<p, o>)$
<y:hasCapacityOf, 90000">	a_0
<y:wasFoundedIn, "1994">	a_1
<y:hasName, "MCA_Band">	a_2

(c) Attribute Dictionary

Table 4.1: Dictionary look-up tables for vertices, edge-types and vertex attributes

edge from a vertex is labelled negative (‘-’).

4.3.3 SPARQL Querying by Adopting Multigraph Homomorphism

As we recall, the problem of SPARQL querying is addressed by finding the solutions to the unknown variables $?X$, that can be bound with the RDF data entities, so that the relations (predicates) provided in the SPARQL query are respected. In this work, to harness the transformed data multigraph G and the query multigraph Q, we reduce the problem of SPARQL querying to a sub-multigraph homomorphism problem. The RDF data is transformed into data multigraph G and the SPARQL query is transformed into query multigraph Q. Let us recall that finding SPARQL answers in the RDF data is equivalent to finding all the sub-multigraphs of Q in G that are homomorphic. We now formally introduce homomorphism for a vertex attributed, directed multigraph, which is a variant of the generic definition of homomorphism as introduced in Definition 2.7.

Definition 4.2. Sub-multigraph Homomorphism. *Given a query multigraph $Q = (U, E^Q, L_U, L_E^Q)$ and a data multigraph $G = (V, E, L_V, L_E)$, the sub-multigraph homomorphism from Q to G is a surjective function $\psi : U \rightarrow V$ such that:*

1. $\forall u \in U, L_U(u) \subseteq L_V(\psi(u))$
2. $\forall (u_m, u_n) \in E^Q, \exists\ (\psi(u_m), \psi(u_n)) \in E$, where (u_m, u_n) is a directed edge, and $L_E^Q(u_m, u_n) \subseteq L_E(\psi(u_m), \psi(u_n))$.

Thus, by finding all the sub-multigraphs in G that are homomorphic to Q, we enumerate all possible homomorphic embeddings of Q in G. These embeddings contain the solution for each of the query vertex that is an unknown variable. Thus, by using the inverse mapping function $\mathcal{M}_v^{-1}(v_i)$ (introduced in table 4.1), we find the bindings for the SPARQL query. The decision problem of subgraph homomorphism is NP-complete. This standard subgraph homomorphism problem can be seen as a particular case of sub-multigraph homomorphism, where both the labelling functions L_E and L_E^Q always return the same subset of edge-types for all the edges in both Q and G. Thus the problem of sub-multigraph homomorphism is at least as hard as subgraph homomorphism. Further, the subgraph homomorphism problem is a generic scenario of subgraph isomorphism problem where, the injectivity constraints are slackened [Shang et al., 2008].

4.4 AMBER: A SPARQL Querying Engine

We now present an overview of AMBER (Attributed Mulitgraph Based Engine for RDF querying) that contains two different stages: (i) an offline stage during which, RDF data is transformed into multigraph G and then a set of index structures $\mathcal{I}$ is constructed that captures the necessary information contained in G; (ii) an online stage during which, a SPARQL query is transformed into a multigraph Q, and then by exploiting the subgraph matching techniques along with the already built index structures $\mathcal{I}$, the homomorphic matches of Q in G are obtained. The AMBER framework is depicted in Figure 4.3.

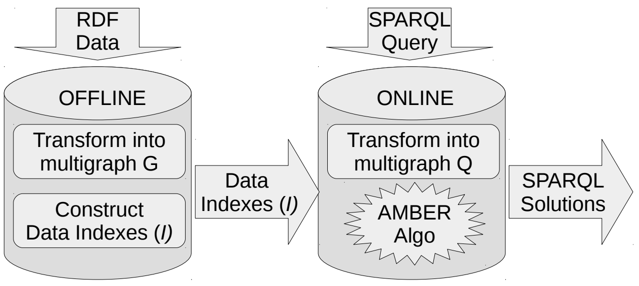

RDF Data
OFFLINE
Transform into multigraph G
Construct Data Indexes (I)
Data Indexes (I)
SPARQL Query
ONLINE
Transform into multigraph Q
AMBER Algo
SPARQL Solutions

Figure 4.3: The AMBER Framework

Given a multigraph representation Q of a SPARQL query, AMBER decomposes the query vertices U into a set of core vertices U_c and satellite vertices U_s. Intuitively, a vertex $u \in U$ is a core vertex, if the degree of the vertex is more than one; on the other hand, a vertex u with degree one is a satellite vertex. For example, in Figure 4.2c, $U_c = \{u_1, u_3, u_5\}$ and $U_s = \{u_0, u_2, u_4, u_6\}$. Once decomposed, we run the sub-multigraph matching procedure on the query structure spanned only by the core vertices. However, during the procedure, we also process the satellite vertices (if available) that are connected to a core vertex that is being processed. For example, while processing the core vertex u_1, we also process the set of satellite vertices $\{u_0, u_2, u_4\}$ connected to it; whereas, the core vertex u_5 has no satellite vertices to be processed. In this way, as the matching proceeds, the entire structure of the query mulitgraph Q is processed to find the homomorphic embeddings in G. The set of indexing structures $\mathcal{I}$ are extensively used during the process of sub-multigraph matching. The homomorphic embeddings are finally translated back to the RDF entities using the inverse mapping function $\mathcal{M}_v^{-1}$ as discussed in Section 4.3.

4.5 Index Construction

In Section 3.5 of the previous chapter, we proposed indexing structures for unlabelled, undirected multigraphs. In this section, we propose indexing structures that can manage labelled and directed multigraphs; although some of the proposed indexing structures in this section are in the same spirit of previous chapter, we do elaborate them, since it enables us to understand the subtleties. The primary goal of indexing is to make the SPARQL querying time efficient.

Given a data multigraph G, we build the following three different indices: (i) an inverted list $\mathcal{A}$ for storing the set of data vertex for each attribute in $a_i \in A$ (ii) a trie index structure $\mathcal{S}$ to store features of all the data vertices V (iii) a set of trie index structures $\mathcal{N}$ to store the neighbourhood information of each data vertex $v \in V$. For brevity of representation, we ensemble all the three index structures into $\mathcal{I} := \{\mathcal{A}, \mathcal{S}, \mathcal{N}\}$.

During the query matching procedure (the online step), we access these indexing structures to obtain the candidate solutions for a query vertex u. Formally, for a query vertex u, the *candidate solutions* are a set of data vertices $C_u = \{v | v \in V\}$ obtained by accessing $\mathcal{A}$ or $\mathcal{S}$ or $\mathcal{N}$, denoted as $C_u^{\mathcal{A}}$, $C_u^{\mathcal{S}}$ and $C_u^{\mathcal{N}}$ respectively.

4.5.1 Attribute Index

The set of vertex attributes is given by $A = \{a_0, \ldots, a_n\}$ (Section 4.3), where a data vertex $v \in V$ might have a subset of A assigned to it. We now build the vertex attribute index $\mathcal{A}$ by creating an inverted list where a particular attribute a_i has the list of all the data vertices in which it appears.

Data vertex v	Signature σ_v	Synopses f_1^+	f_2^+	f_3^+	f_4^+	f_1^-	f_2^-	f_3^-	f_4^-
v_0	$\{\{-t_6\}, \{t_7\}\}$	1	1	-7	7	1	1	-6	6
v_1	$\{\{-t_3\}, \{-t_7\}, \{-t_8\}, \{-t_4, -t_5\}\}$	0	0	0	0	2	5	-3	8
v_2	$\{\{-t_0\}, \{t_1\}, \{-t_2\}, \{t_5\}, \{t_6\}, \{t_4, t_5\}\}$	2	4	-1	6	1	2	0	2
v_3	$\{\{t_0\}, \{t_3\}, \{-t_1\}\}$	1	2	0	3	1	1	-1	1
v_4	$\{\{t_2\}\}$	1	1	-2	2	0	0	0	0
v_5	$\{\{t_3\}, \{t_3\}\}$	1	1	-3	3	0	0	0	0
v_6	$\{\{t_8\}, \{-t_3\}\}$	1	1	-8	8	1	1	-3	3
v_7	$\{\{-t_0\}, \{-t_3\}, \{-t_5\}\}$	0	0	0	0	1	3	0	5
v_8	$\{\{t_0\}\}$	1	1	0	0	0	0	0	0

Table 4.2: Vertex signatures and the corresponding synopses for the vertices in the data multigraph G (Figure 4.1c)

Given a query vertex u with a set of vertex attributes $u.A \subseteq A$, for each attribute $a_i \in u.A$, we access the index structure $\mathcal{A}$ to fetch a set of data vertices that have a_i. Then we find a common set of data vertices that have the entire attribute set $u.A$. For example, considering the query vertex u_5 (Figure 4.2c), it has an attribute set $\{a_1, a_2\}$. The candidate solutions for u_5 are obtained by finding all the common data vertices, in $\mathcal{A}$, between a_1 and a_2, resulting in $C_{u_5}^{\mathcal{A}} = \{v_0\}$.

4.5.2 Vertex Signature Index

The index $\mathcal{S}$ captures the edge type information from the data vertices. For a lucid understanding of this indexing schema we formally introduce the notion of vertex signature in directed multigraphs, that is defined for a vertex $v \in V$, which encapsulates the edge information associated with it.

Definition 4.3. Vertex signature for directed multigraphs. *For a vertex $v \in V$, the vertex signature σ_v is a multiset containing all the directed multi-edges that are incident on v, where a multi-edge between v and a neighbouring vertex v' is represented by a set that corresponds to the edge types. Formally, $\sigma_v = \bigcup_{v' \in N(v)} L_E(v, v')$ where $N(v)$ is the set of neighbourhood vertices of v, and $\cup$ is the union operator for multiset.*

The index $\mathcal{S}$ is constructed by tailoring the information supplied by the vertex signature of each vertex in G. To extract some interesting features, let us observe the vertex signature σ_{v_2} as supplied in Table 4.2. To begin with, we can represent the vertex signature σ_{v_2} separately for the incoming and outgoing multi-edges as $\sigma_{v_2}^+ = \{\{t_1\}, \{t_5\}, \{t_6\}, \{t_4, t_5\}\}$ and $\sigma_{v_2}^- = \{\{-t_0\}\{-t_2\}\}$ respectively. Now we observe that $\sigma_{v_2}^+$ has four distinct multi-edges and $\sigma_{v_2}^-$ has two distinct multi-edges. Now, let us think that we want to find the candidate solutions for a query vertex u. The data vertex v_2 can be a match for u only if the signature of u has at most four incoming ('+') edges and at most two outgoing ('-') edges; else v_2 can not be a match for u. Thus, more such features (e.g., maximum cardinality of a set in the vertex signature) can be proposed to filter out irrelevant candidate vertices. Thus, for each vertex v, we propose to extract a set of features by exploiting the corresponding vertex signature. These features constitute a *synopses*, which is a surrogate representation that approximately captures the vertex signature information.

The synopsis of a vertex v contains a set of features F, whose values are computed from the vertex signature σ_v. In this background, we propose four distinct features: f_1 - the maximum cardinality of a set in the vertex signature; f_2 - the number of unique dimensions in the vertex signature; f_3 - the minimum index value of the edge type; f_4 - the maximum index value of the edge type. For f_3 and f_4, the index values of edge type are nothing but the position of the sequenced alphabet. These four basic features are replicated separately for outgoing (negative) and incoming (positive) edges, as seen in Table 4.2. Thus for the vertex v_2, we obtain $f_1^+ = 2$, $f_2^+ = 4$, $f_3^+ = -1$ and $f_4^+ = 7$ for the incoming edge set and $f_1^- = 1$, $f_2^- = 2$, $f_3^- = 0$ and $f_4^- = 2$ for the outgoing edge set. Synopses for the entire vertex set V for the data multigraph G are depicted in Table 4.2.

It is to be noted that the number of synopses fields in this chapter are 4 (8, after considering separately for directed graphs); however, in the previous chapter, we had 6 synopses fields. This is because, in this chapter, we are focusing on the homomorphic matches, and in the previous chapter, our focus was on isomorphic matches. And we have learnt that homomorphism is less constrained when compared to isomorphism. Thus, for a query, the number of homomorphic matches are always at least as much as the isomorphic matches.

Further, as we have seen in the previous chapter that R-tree was a very efficient structure; here also we use the R-tree data structure to store the entire synopses. This R-tree constitutes the vertex signature index $\mathcal{S}$. A synopsis with $|F|$ fields forms a leaf in the R-tree. When a set of possible candidate solutions are to be obtained for a query vertex u, we create a vertex signature σ_u in order to compute the synopsis, and then obtain the possible solutions from the R-tree structure.

Formally, the candidate solutions for a vertex u can be written as $C_u^{\mathcal{S}} = \{v | \forall_{i \in [1,\ldots,|F|]} f_i^{\pm}(u) \leq f_i^{\pm}(v)\}$, where the constraints are met for all the $|F|$-dimensions. Since we apply the same inequality constraint to all the fields, we negate the fields that refer to the minimal index value of the edge type (f_3^+ and f_3^-) so that the rectangular containment problem still holds good. Further to respect the rectangular containment, we populate the synopses fields with '0' values, in case, the signature does not have either positive or negative edges in it, as observed in Table 4.2 for v_1, v_3, v_4, v_5 and v_7.

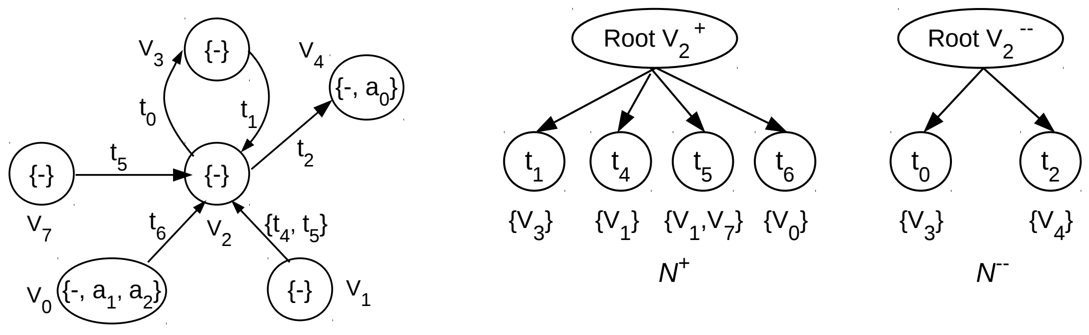
For example, if we want to compute the possible candidates for a query vertex u_0 in Figure 4.2c, whose signature is $\sigma_{u_0} = \{-t_5\}$, we compute the synopsis which is [0 0 0 0 1 1 5 5]. Now we look for all those vertices that subsume this synopsis in the R-tree, whose elements are depicted in Table 4.2, which gives us the candidate solutions $C^{\mathcal{S}}_{u_0} = \{v_1, v_7\}$, thus pruning the rest of the vertices.

The $\mathcal{S}$ index helps to prune the vertices that do not respect the edge type constraints. This is crucial since this pruning is performed for the initial query vertex, and hence many candidates are cast away, thereby avoiding unnecessary recursion during the matching procedure. For example, for the initial query vertex u_0, whose candidate solutions are $\{v_1, v_7\}$, the recursion branch is run only on these two starting vertices instead of the entire vertex set V.

4.5.3 Vertex Neighbourhood Index

The *vertex neighbourhood index* $\mathcal{N}$ captures the topological structure of the data multigraph G. The index $\mathcal{N}$ comprises of 1-neighbourhood trees built for each data vertex $v \in V$. Since G is a directed multigraph, and each vertex $v \in V$ can have both the incoming and outgoing edges, we construct two separate index structures $\mathcal{N}^+$ and $\mathcal{N}^-$ for incoming and outgoing edges respectively, that constitute the structure $\mathcal{N}$.

To understand the index structure, let us consider the data vertex v_2 from Figure 4.1c, shown separately in Figure 4.4a. For this vertex v_2, we collect all the neighbourhood information (vertices and multi-edges), and represent this information by a tree structure, built separately for incoming ('+') and outgoing ('-') edges. Thus, the tree representation of a vertex v contains the neighbourhood vertices and the corresponding multi-edges, as shown in Figure 4.4b, where the vertices of the tree structure are represented by the edge types.



(a) Neighbourhood structure of v_2 (b) OTIL structure for v_2

Figure 4.4: Building Neighbourhood Index for data vertex v_2

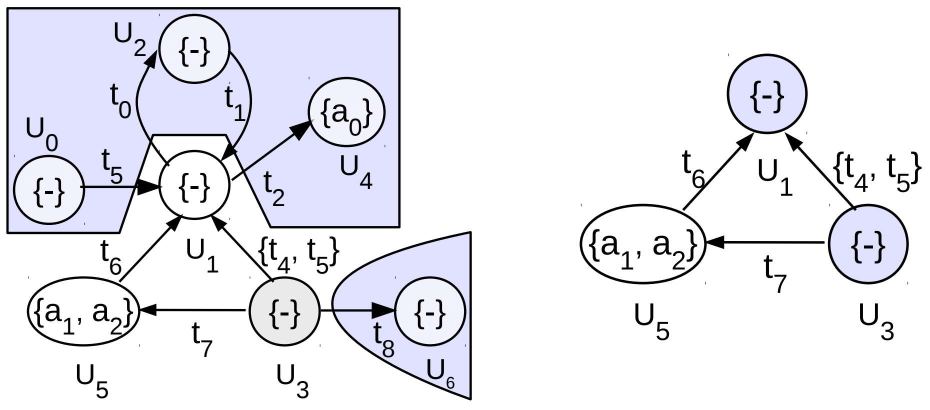
In order to construct an efficient tree structure, we again consider the OTIL (Ordered Trie with Inverted List) structure, proposed in the previous chapter. To construct the OTIL index as shown in Figure 4.4b, we insert each ordered multi-edge that is incident on v at the root of the trie. Consider a data vertex v_i, with a set of n neighbourhood vertices $N(v_i)$. Now, for every pair of incoming edge $(v_i, N^j(v_i))$, where $j \in \{1, \dots, n\}$, there exists a multi-edge $\{t_i, \dots, t_j\}$, which is inserted into the OTIL structure $\mathcal{N}^+$. Similarly for every pair of outgoing edge $(N^j(v_i), v_i)$, there exists a multi-edge $\{t_m, \dots, t_n\}$, which is inserted into the OTIL structure $\mathcal{N}^-$ maintaining two OTIL structures that constitute $\mathcal{N}$. Each multi-edge is ordered (w.r.t. increasing edge type indexes), before inserting into the respective OTIL structure, and the order is universally maintained for all data vertices. Further, for every edge type t_i, we maintain a *list* that contains all the neighbourhood vertices $N^+(v_i)/N^-(v_i)$, that have the edge type t_i incident on them.

To understand the utility of $\mathcal{N}$, let us consider an illustrative example. Considering the query multigraph Q in Figure 4.2c, let us assume that we want to find the matches for the query vertices u_1 and u_0 in order. Thus, for the initial vertex u_1, let us say, we have found the set of candidate solutions which is $\{v_2\}$. Now, to find the candidate solutions for the next query vertex u_0, it is important to maintain the structure spanned by the query vertices, and this is where the indexing structure $\mathcal{N}$ is accessed. Thus to retain the structure of the query multigraph (in this case, the structure between u_1 and u_0), we have to find the data vertices that are in the neighbourhood of already matched vertex v_2 (a match for vertex u_1), that has the same structure (edge types) between u_1 and u_0 in the query graph. Thus to fetch all the data vertices that have the edge type t_5, which is directed towards v_2 and hence '+', we access the neighbourhood index trie $\mathcal{N}^+$ for vertex v_2, as shown in Figure 4.4. This gives us a set of candidate solutions $C^{\mathcal{N}}_{u_0} = \{v_1, v_7\}$. It is easy to observe that, by maintaining two separate indexing structures $\mathcal{N}^+$ and $\mathcal{N}^-$, for both incoming and outgoing edges, we can reduce the time to fetch the candidate solutions.

Thus, in a generic scenario, given an already matched data vertex v, the edge direction '+' or '-', and the set of edge types $T' \subseteq T$, the index $\mathcal{N}$ will find a set of neighbourhood data vertices $\{v' | (v', v) \in E \wedge T' \subseteq L_E(v', v)\}$ if the edge direction is '+' (incoming), while $\mathcal{N}$ returns $\{v' | (v, v') \in E \wedge T' \subseteq L_E(v, v')\}$ if the edge direction is '-' (outgoing).

4.6 Query Matching Procedure

In order to follow the working of the proposed query matching procedure, we formalize the notion of *core* and *satellite* vertices. Given a query graph Q, we decompose the set of query vertices U into a set of *core* vertices U_c and a set of



(a) Query graph Q (with satellite vertices) (b) Query graph spanned by core vertices

Figure 4.5: Decomposing the query multigraph into *core* and *satellite* vertices

satellite vertices U_s. Formally, when the degree of the query graph $\Delta(Q) > 1$, $U_c = \{u | u \in U \wedge deg(u) > 1\}$; however, when $\Delta(Q) = 1$, i.e, when the query graph is either a vertex or a multiedge, we choose one query vertex at random as a *core* vertex, and hence $|U_c| = 1$. The remaining vertices are classified as satellite vertices, whose degree is always 1. Formally, $U_s = \{U \setminus U_c\}$, where for every $u \in U_s$, $deg(u) = 1$. The decomposition for the query multigraph Q is depicted in Figure 4.5, where the satellite vertices are separated (vertices under the shaded region in Figure 4.5a), in order to obtain the query graph that is spanned only by the core vertices (Figure 4.5b).

The proposed AMbER-Algo (Algorithm 4.3) performs recursive sub-multigraph matching procedure only on the query structure spanned by U_c as seen in Figure 4.5b. Since the entire set of satellite vertices U_s is connected to the query structure spanned by the core vertices, AMbER-Algo processes the satellite vertices while performing sub-multigraph matching on the set of core vertices. Thus during the recursion, if the current *core* vertex has *satellite* vertices connected to it, the algorithm retrieves directly a list of possible matching for such *satellite* vertices and it includes them in the current partial solution. Each time the algorithm executes a recursion branch with a solution, the solution not only contains a data vertex match v_c for each query vertex belonging to U_c, but also a set of matched data vertices V_s for each query vertex belonging to U_s. Each time a solution is found, we can generate not only one, but a set of embeddings through the Cartesian product of the matched elements in the solution.

Since finding SPARQL solutions is equivalent to finding homomorphic embeddings of the query multigraph, the homomorphic matching allows different query vertices to be matched with the same data vertices. Recall that there is no injectivity constraint in sub-multigraph homomorphism as opposed to sub-multigraph isomorphism [Shang et al., 2008]. Thus during the recursive matching procedure, we do not have to check if the potential data vertex has already been matched with previously matched query vertices. This is an advantage when we are processing satellite vertices: we can find matches for each satellite vertex independently without the necessity to check for a repeated data vertex.

Before getting into the details of the AMbER-Algo, we first explain how a set of candidate solutions is obtained when there is information associated only with the vertices. Then we explain how a set of candidate solutions is obtained when we encounter the satellite vertices.

4.6.1 Vertex Level Processing

To understand the generic query processing, it is necessary to understand the matching process at vertex level. Whenever a query vertex $u \in U$ is being processed, we need to check if u has a set of attributes A associated with it or any *IRI*s are connected to it (recall Section 4.3.2).

Algorithm 4.1: ProcessVertex$(u, Q, \mathcal{A}, \mathcal{N})$

```
1 if $u.A \neq \emptyset$ then
2     $C_u^A = $ QueryAttIndex$(\mathcal{A}, u.A)$
3 if $u.R \neq \emptyset$ then
4     $C_u^I = \bigcap_{u_i^{iri} \in u.R}($ QueryNeighIndex$(\mathcal{N}, L_E^Q(u, u_i^{iri}), u_i^{iri})$ )
5 $CandAtt_u = C_u^A \overline{\cap} C_u^I$                    /* Find common candidates */
6 return $CandAtt_u$
```

To process an arbitrary query vertex, we propose a procedure ProcessVertex, depicted in Algorithm 4.1. This algorithm is invoked only when a vertex u has at least, either a set of vertex attributes or any *IRI* associated with it. The ProcessVertex procedure returns a set of data vertices $CandAtt_u$, which are matchable with u; in case $CandAtt_u$ is empty, then the query vertex u has no matches in V.

As seen in Lines 1-2, when a query vertex u has a set of vertex attributes i.e., $u.A \neq \emptyset$, we obtain the candidate solutions C_u^A by invoking QueryAttIndex procedure, that accesses the index $\mathcal{A}$ as explained in Section 4.5.1. For example, the query vertex u_5 with vertex attributes $\{a_1, a_2\}$, can only be matched with the data vertex v_0; thus $C_{u_5}^A = \{v_0\}$.

When a query vertex u has *IRI*s associated with it, i.e., $u.R \neq \emptyset$ (Lines 3-4), we find the candidate solutions C_u^I by invoking the QueryNeighIndex procedure. As we recall from Section 4.3.2, a vertex u is connected to an *IRI* vertex u_i^{iri} through

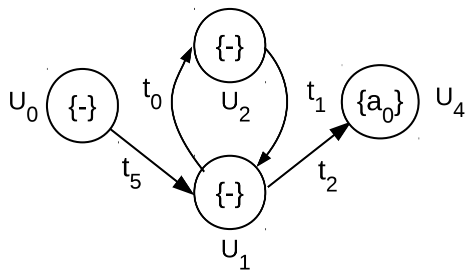


Figure 4.6: A star structure in the query multigraph Q

a multi-edge $L_E^Q(u, u_i^{iri})$. An *IRI* vertex u_i^{iri} always has only one data vertex v, that can match. Thus, the candidate solutions C_u^I are obtained by invoking the QueryNeighIndex procedure, that fetches all the neighbourhood vertices of v that respect the multi-edge $L_E^Q(u, u_i^{iri})$. The procedure is invoked until all the IRI vertices $u.R$ are processed (Line 4). Considering the example in Figure 4.2c, u_3 is connected to an *IRI*-vertex u_0^{iri}, which has a unique data vertex match v_5, through the multi-edge $\{-t_3\}$. Using the neighbourhood index $\mathcal{N}$, we look for the neighbourhood vertices of v_5, that have the multi-edge $\{-t_3\}$, which gives us the candidate solutions $C_{u_3}^I = \{v_1\}$.

Finally in Line 5, the merge operator $\overline{\cap}$ returns a set of common candidates $CandAtt_u$, only if $u.A \neq \emptyset$ and $u.R \neq \emptyset$. Otherwise, C_u^A or C_u^I are returned as $CandAtt_u$.

4.6.2 Processing Satellite Vertices

In this section, we provide insights on processing a set of satellite vertices $U_{sat} \subseteq U_s$ that are connected to a core vertex $u_c \in U_c$. This scenario results in a structure that appears frequently in SPARQL queries called star structure [Gubichev and Neumann, 2014, Huang et al., 2011].

A typical star structure depicted in Figure 4.6, has a core vertex $u_c = u_1$, and a set of satellite vertices $U_{sat} = \{u_0, u_2, u_4\}$ connected to the core vertex. For each candidate solution of the core vertex u_1, we process u_0, u_2, u_4 independently of each other, since there is no structural connectivity (edges) among them, although they are only structurally connected to the core vertex u_1.

Theorem 4.1. *For a given star structure in a query graph, each satellite vertex can be independently processed if a candidate solution is provided for the core vertex u_c.*

Proof. Consider a core vertex u_c that is connected to a set of satellite vertices $U_{sat} = \{u_0, \ldots, u_s\}$, through a set of edge-types $T' = \{t_0, \ldots, t_s\}$. Let us assume v_c is a candidate solution for the core vertex u_c, and we want to find candidate solutions for $u_i \in U_{sat}$ and $u_j \in U_{sat}$, where $i \neq j$. Now, the candidate solutions for u_i and u_j can be obtained by fetching the neighbourhoods of already matched vertex v_c that respect the edge-type $t_i \in T'$ and $t_j \in T'$ respectively. Since two satellite vertices u_i and u_j are never connected to each other, the candidate solutions of u_i are independent of that of u_j. This analogy applies to all the satellite vertices. $\square$

Algorithm 4.2: MatchSatVertices($\mathcal{A}, \mathcal{N}, Q, U_{sat}, v_c$)

```
1 Set: $M_{sat} = \emptyset$, where $M_{sat} = \{[u_s, V_s]\}_{s=1}^{|U_{sat}|}$
2 for all $u_s \in U_{sat}$ do
3     $Cand_{u_s}$ = QueryNeighIndex($\mathcal{N}, L_E^Q(u_c, u_s), v_c$)
4     $Cand_{u_s} = Cand_{u_s} \cap$ ProcessVertex($u_s, Q, \mathcal{A}, \mathcal{N}$)
5     if $Cand_{u_s} \neq \emptyset$ then
6         $M_{sat} = M_{sat} \cup (u_s, Cand_{u_s})$          /* Satellite solutions */
7     else
8         return $M_{sat} := 0$          /* No solutions possible */
9 return $M_{sat}$          /* Matches for satellite vertices */
```

Given a core vertex u_c, we initially find a set of candidate solutions $Cand_{u_c}$, by using the index $\mathcal{S}$. Then, for each candidate solution $v_c \in Cand_{u_c}$, the set of solutions for all the satellite vertices U_{sat} that are connected to u_c are returned by the MatchSatVertices procedure, described in Algorithm 4.2. The set of solution tuple M_{sat} defined in Line 1, stores the candidate solutions for the entire set of satellite vertices U_{sat}. Formally, $M_{sat} = \{[u_s, V_s]\}_{s=1}^{|U_{sat}|}$, where $u_s \in U_{sat}$ and V_s is a set of candidate solutions for u_s. In order to obtain candidate solutions for u_s, we query the neighbourhood index $\mathcal{N}$ (Line 3); the QueryNeighIndex function obtains all the neighbourhood vertices of already matched v_c, that also considers the multi-edge in the query multigraph $L_E^Q(u_c, u_s)$. As every query vertex $u_s \in U_{sat}$ is processed, the solution set M_{sat} that contains candidate solutions grows until all the satellite vertices have been processed (Lines 2-8).

In Line 4, the set of candidate solutions $Cand_{u_s}$ are refined by invoking Algorithm 4.1 (VertexProcessing). After the refinement, if there are finite candidate solutions, we update the solution M_{sat}; else, we terminate the procedure as there can be no matches for a given matched vertex v_c. The MatchSatVertices procedure performs two tasks: firstly, it checks if the candidate vertex $v_c \in Cand_{u_s}$ is a valid matchable vertex and secondly, it obtains the solutions for all the satellite vertices.

4.6.3 Arbitrary Query Processing

Algorithm 4.3 shows the generic procedure we develop to process arbitrary queries.

Recall that for an arbitrary query Q, we define two different types of vertexes: a set of core vertices U_c and a set of satellite vertices U_s. The QUERYDECOMPOSE procedure in Line 1 of Algorithm 4.3, performs this decomposition by splitting the query vertices U into U_c and U_s, as observed in Figure 4.5.

To process arbitrary query multigraphs, we perform recursive sub-mulitgraph matching procedure on the set of core vertices $U_c \subseteq U$; during the recursion, satellite vertexes connected to a specific core vertex are processed too. Since the recursion is performed on the set of core vertices, we propose a few heuristics for ordering the query vertices.

Ordering of the query vertices forms one of the vital steps for subgraph matching algorithms [Shang et al., 2008]. In any subgraph matching algorithm, the embeddings of a query subgraph are obtained by exploring the solution space spanned by the data graph. But since the solution space itself can grow exponentially in size, we are compelled to use intelligent strategies to traverse the solution space. In order to achieve this, we propose a heuristic procedure VERTEXORDERING (Line 2, Algorithm 4.3) that employs two ranking functions.

The first ranking function r_1 relies on the number of satellite vertices connected to the core vertex, and the query vertices are ordered with the decreasing rank value. Formally, $r_1(u) = |U_{sat}|$, where $U_{sat} = \{u_s | u_s \in U_s \wedge (u, u_s) \in E(Q)\}$. A vertex with more satellite vertices connected to it, is rich in structure and hence it would probably yield fewer candidate solutions to be processed under recursion. Thus, in Figure 4.5, u_1 is chosen as an initial vertex. The second ranking function r_2 relies on the number of incident edges on a query vertex. Formally, $r_2(u) = \sum_{j=1}^{m} |\sigma(u)^j|$, where u has m multiedges and $|\sigma(u)^j|$ captures the number of edge types in the j^{th} multiedge. Again, U_c^{ord} contains the ordered vertices with the decreasing rank value r_2. Further, when there are no satellite vertices in the query Q, this ranking function gets the priority. Despite the usage of any ranking function, the query vertices in U_c^{ord}, when accessed in sequence, should be structurally connected to the previous set of vertices. If two vertices tie up with the same rank, the rank with lesser priority determines which vertex wins. Thus, for the example in Figure 4.5, the set of ordered core vertices is $U_c^{ord} = \{u_1, u_3, u_5\}$.

The first vertex in the set U_c^{ord} is chosen as the initial vertex u_{init} (Line 3), and subsequent query vertices are chosen in sequence. The candidate solutions for the initial query vertex $CandInit$ are returned by QUERYSYNINDEX procedure (Line 4), that are constrained by the structural properties (neighbourhood structure) of u_{init}. By querying the index $\mathcal{S}$ for initial query vertex u_{init}, we obtain the candidate solutions $CandInit \in V$ that match the structure (multiedge types) associated with

Algorithm 4.3: AMbER-Algo $(\mathcal{I}, Q)$

```
1  QueryDecompose: Split $U$ into $U_c$ and $U_s$
2  $U_c^{ord}$ = VertexOrdering($Q, U_c$)
3  $u_{init} = u | u \in U_c^{ord}$
4  $CandInit$ = QuerySynIndex($u_{init}$, $\mathcal{S}$)
5  $CandInit = CandInit \cap$ ProcessVertex($u_{init}, Q, \mathcal{A}, \mathcal{N}$)
6  Fetch: $U_{init}^{sat} = \{u | u \in U_s \wedge (u_{init}, u) \in E(Q)\}$
7  Set: $Emb = \emptyset$
8  for $v_{init} \in CandInit$ do
9      Set: $M = \emptyset, M_s = \emptyset, M_c = \emptyset$
10     if $U_{init}^{sat} \neq \emptyset$ then
11         $M_{sat}$ = MatchSatVertices($\mathcal{A}, \mathcal{N}, Q, U_{init}^{sat}, v_{init}$)
12         if $M_{sat} \neq \emptyset$ then
13             for $[u_s, V_s] \in M_{sat}$ do
14                 Update: $M_s = M_s \cup [u_s, V_s]$
15             Update: $M_c = M_c \cup [u_{init}, v_{init}]$
16             $Emb = Emb \cup$ HomomorphicMatch($M, \mathcal{I}, Q, U_c^{ord}$)
17     else
18         Update: $M_c = M_c \cup (u_{init}, v_{init})$
19         $Emb = Emb \cup$ HomomorphicMatch($M, \mathcal{I}, Q, U_c^{ord}$)
20 return $Emb$          /* Homomorphic embeddings of query multigraph */
```

u_{init}. Although some candidates in *CandInit* may be invalid, all valid candidates are present in *CandInit*. Further, ProcessVertex procedure is invoked to obtain the candidate solutions according to vertex attributes and *IRI* information, and then only the common candidates are retained.

Before getting into the algorithmic details, we explain how the solutions are handled and how we process each query vertex. We define M as a set of tuples, whose i^{th} tuple is represented as $M_i = [m_c, M_s]$, where m_c is a solution pair for a core vertex, and M_s is a set of solution pairs for the set of satellite vertices that are connected to the core vertex. Formally $m_c = (u_c, v_c)$, where u_c is the core vertex and v_c is the corresponding matched vertex; M_s is a set of solution pairs, whose j^{th} element is a solution pair (u_s, V_s), where u_s is a satellite vertex and V_s is a set of matched vertices. In addition, we maintain a set M_c whose elements are the solution pairs for all the core vertices. Thus during each recursion branch, the size of M grows until it reaches the query size $|U|$; once $|M| = |U|$, homomorphic matches are obtained.

For all the candidate solutions of initial vertex *CandInit*, we perform recursion to obtain homomorphic embeddings (lines 8-19). Before getting into recursion, for each initial match $v_{init} \in CandInit$, if it has satellite vertices connected to it, we invoke the MATCHSATVERTICES procedure (Lines 10-11). This step not only finds solution matches for satellite vertices, if there are, but also checks if v_{init} is a valid candidate vertex. If the returned solution set M_{sat} is empty, then v_{init} is not a valid candidate and hence we continue with the next $v_{init} \in CandInit$; else, we update the set of solution pairs M_s for satellite vertices and the solution pair M_c for the core vertex (Lines 12-15) and invoke HOMOMORPHICMATCH procedure (Lines 17). On the other hand, if there are no satellite vertices connected to u_{init}, we update the core vertex solution set M_c and invoke HOMOMORPHICMATCH procedure (Lines 18-19).

Algorithm 4.4: HOMOMORPHICMATCH($M, \mathcal{I}, Q, U_c^{ord}$)

```
1  if $|M| = |U|$ then
2      return GENEMB($M$)
3  $Emb = \emptyset$
4  FETCH: $u_{nxt} = u | u \in U_c^{ord}$
5  $N_q = \{u_c | u_c \in M_c\} \cap adj(u_{nxt})$
6  $N_g = \{v_c | v_c \in M_c \wedge (u_c, v_c) \in M_c\}$, where $u_c \in N_q$
7  $Cand_{u_{nxt}} = \bigcap_{n=1}^{|N_q|}$(QueryNeighIndex($\mathcal{N}, L_E^Q(u_n, u_{nxt}), v_n$))
8  $Cand_{u_{nxt}} = Cand_{u_{nxt}} \cap$ PROCESSVERTEX($u_{nxt}, Q, \mathcal{A}, \mathcal{N}$)
9  for each $v_{nxt} \in Cand_{u_{nxt}}$ do
10     FETCH: $U_{nxt}^{sat} = \{u | u \in V_s \wedge (u_{nxt}, u) \in E(Q)\}$
11     if $U_{nxt}^{sat} \neq \emptyset$ then
12         $M_{sat}$ = MATCHSATVERTICES($\mathcal{A}, \mathcal{N}, Q, U_{nxt}^{sat}, v_{nxt}$)
13         if $M_{sat} \neq \emptyset$ then
14             for every $[u^s, V^s] \in M_{sat}$ do
15                 UPDATE: $M_s = M_s \cup [u^s, V^s]$
16             UPDATE: $M_c = M_c \cup (u_{nxt}, v_{nxt})$
17             $Emb = Emb \cup$ HOMOMORPHICMATCH($M, \mathcal{I}, Q, U_c^{ord}$)
18     else
19         UPDATE: $M_c = M_c \cup (u_{nxt}, v_{nxt})$
20         $Emb = Emb \cup$ HOMOMORPHICMATCH($M, \mathcal{I}, Q, U_c^{ord}$)
21 return $Emb$
```

In the HOMOMORPHICMATCH procedure (Algorithm 4.4), we fetch the next query vertex from the set of ordered core vertices U_c^{ord} (Line 4). Then we collect the neighbourhood vertices of already matched core query vertices and the corresponding matched data vertices (Lines 5-6). As we recall, the set M_c maintains the solution pair $m_c = (u_c, v_c)$ of each matched core query vertex. The set N_q collects the already matched core vertices $u_c \in M_c$ that are also in the neighbourhood of u_{nxt}, whose matches have to be found. Further, N_g contains the corresponding matched query vertices $v_c \in M_c$. As the recursion proceeds, we find those matchable data vertices of u_{nxt} that are in the neighbourhood of all the matched vertices $v \in N_g$, so that the query structure is maintained. In Line 7, for each $u_n \in N_q$ and the corresponding $v_n \in N_g$, we query the neighbourhood index $\mathcal{N}$, to obtain the candidate solutions $Cand_{u_{nxt}}$, that are in the neighbourhood of already matched data vertex v_n and have the multiedge $L_E^Q(u_n, u_{nxt})$, obtained from the query multigraph Q. Finally (line 7), the set of candidates solutions that are common for every $u_n \in N_q$ are retained in $Cand_{u_{nxt}}$.

Further, the candidate solutions are refined with the help of ProcessVertex procedure (Line 8). Now, for each of the valid candidate solution $v_{nxt} \in Cand_{u_{nxt}}$, we recursively call the HomomorphicMatch procedure. When the next query vertex u_{nxt} has no satellite vertices attached to it, we update the core vertex solution set M_c and call the recursion procedure (Lines 19-20). But when u_{nxt} has satellite vertices attached to it, we obtain the candidate matches for all the satellite vertices by invoking the MatchSatVertices procedure (Lines 11-12); if there are matches, we update both the satellite vertex solution M_s and the core vertex solution M_c, and invoke the recursion procedure (Line 17).

Once all the query vertices have been matched for the current recursion step, the solution set M contains the solutions for both core and satellite vertices. Thus when all the query vertices have been matched, we invoke the GenEmb function (Line 2) which returns the set of embeddings, that are updated in *Emb*. The GenEmb function treats the solution vertex v_c of each core vertex as a singleton and performs Cartesian product among all the core vertex singletons and satellite vertex sets. Formally, $Emb_{part} = \{v_c^1\} \times \cdots \times \{v_c^{|U_c|}\} \times V_s^1 \times \cdots \times V_c^{|U_s|}$. Thus, the partial set of embeddings Emb_{part} is added to the final result *Emb*.

4.7 Experimental Evaluation

In this section, we perform extensive experiments on the three real RDF datasets. We evaluate the time performance and the robustness of AMbER w.r.t. state-of-the-art competitors by varying the size, and the structure of the SPARQL queries. Experiments are carried out on a 64-bit Intel Core i7-4900MQ @ 2.80GHz, with 32GB memory, running Linux OS - Ubuntu 14.04 LTS. AMbER is implemented in C++.

4.7.1 Experimental Setup

We compare AMBER with the four standard RDF engines: *Virtuoso-7.1* [Erling, 2012], *x-RDF-3X* [Neumann and Weikum, 2010], *Apache Jena* [Carroll et al., 2004] and *gStore* [Zou et al., 2014b]. For all the competitors we employ the source code available on the web site or obtained by the authors. Another recent work *Turbo*$_{HOM}$*++* [Shang et al., 2008] has been excluded since it is not publicly available.

For the experimental analysis we use three RDF datasets - *DBPEDIA*, *YAGO* and *LUBM*. *DBPEDIA* constitutes the most important knowledge base for the Semantic Web community. Most of the data available in this dataset comes from the Wikipedia Infobox. *YAGO* is a real world dataset built from factual information coming from *Wikipedia* and *WordNet* semantic network. *LUBM* provides a standard RDF benchmark to test the overall behaviour of engines. Using the data generator we create *LUBM100* where the number represents the scaling factor.

Dataset	# Triples	# Vertices	# Edges	# Edge types
DBPEDIA	33 071 359	4 983 349	14 992 982	676
YAGO	35 543 536	3 160 832	10 683 425	44
LUBM100	13 824 437	2 179 780	8 952 366	13

Table 4.3: Statistics of RDF Datasets

The data characteristics are summarized in Table 4.3. We observe that the datasets have different characteristics in terms of number of vertices, number of edges, and number of distinct predicates. For instance, *DBPEDIA* has more diversity in terms of predicates (∼700) while *LUBM100* contains only 13 different predicates.

The time required to build the multigraph database as well as to construct the indexes are reported in Table 4.4. We can note that the database building time and the corresponding size are proportional to the number of triples. Regarding the indexing structures, we can underline that both building time and size are proportional to the number of edges. For instance, *DBPEDIA* has the biggest number of edges (∼15M) and, consequently, AMBER employs more time and space to build and store its data structure.

4.7.2 Workload Generation

In order to test the scalability and the robustness of the different RDF engines, we generate the query workloads considering a similar setting as in [Gubichev and

Dataset	Database		Index $\mathcal{I}$	
	Building Time	Size	Building Time	Size
DBPEDIA	307	1300	45.18	1573
YAGO	379	2400	29.1	1322
LUBM100	67	497	18.4	1057

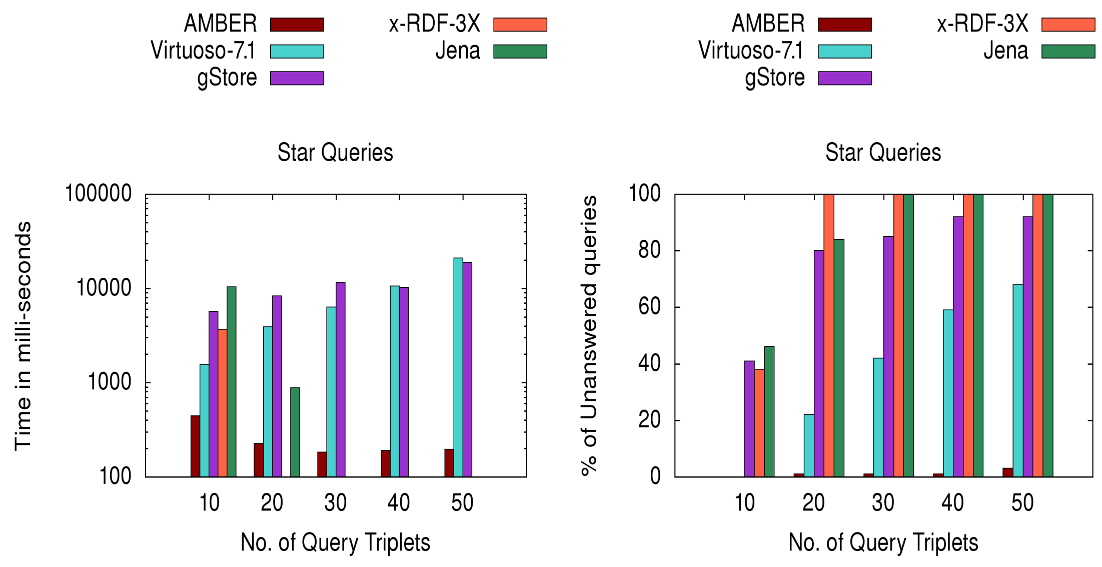
Table 4.4: Execution time (in seconds) and memory usage (in Mbytes) for offline index construction

Neumann, 2014, Aluç et al., 2014, Han et al., 2013]. We generate the query workload from the respective RDF datasets, which are available as RDF tripleset. In specific, we generate two types of query sets: a star-shaped and a complex-shaped query set; further, both query sets are generated for varying sizes (say k) ranging from 10 to 50 triplets, in steps of 10.

To generate star-shaped or complex-shaped queries of size k, we pick an initial-entity at random from the RDF data. Now to generate star queries, we check if the initial-entity is present in at least k triples in the entire benchmark, to verify if the initial-entity has k neighbours. If so, we choose those k triples at random; thus the initial entity forms the central vertex of the star structure and the rest of the entities form the remaining star structure, connected by the respective predicates. To generate complex-shaped queries of size k, we navigate in the neighbourhood of the initial-entity through the predicate links until we reach size k. In both query types, we inject some object literals as well as constant *IRI*s; rest of the *IRI*s (subjects or objects) are treated as variables. However, this strategy could choose some very unselective queries [Gubichev and Neumann, 2014]. In order to address this issue, we set a maximum time constraint of 60 seconds for each query. If the query is not answered in time, it is not considered for the final average (similar procedure is usually employed for graph query matching [Han et al., 2013] and RDF workload evaluation [Aluç et al., 2014]). We report the average query time and, also, the percentage of unanswered queries (considering the given time constraint) to study the robustness of the approaches.

4.7.3 Comparison with RDF Engines

In this section we discuss the results obtained by the different RDF engines. For each combination of query type and benchmark we report two plots by varying the query size: the average time and the corresponding percentage of unanswered queries for the given time constraint. We remind that the average time per approach is computed only on the set of queries that were answered.



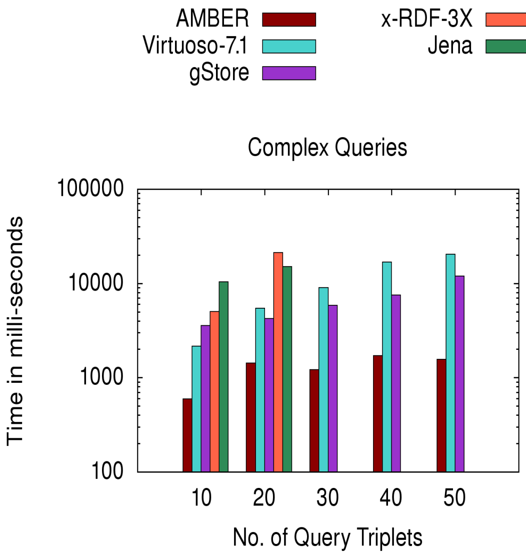
(a) Time performance (b) % Unanswered queries

Figure 4.7: Evaluation of (a) time performance and (b) robustness, for *Star-Shaped* queries on *DBPEDIA*.

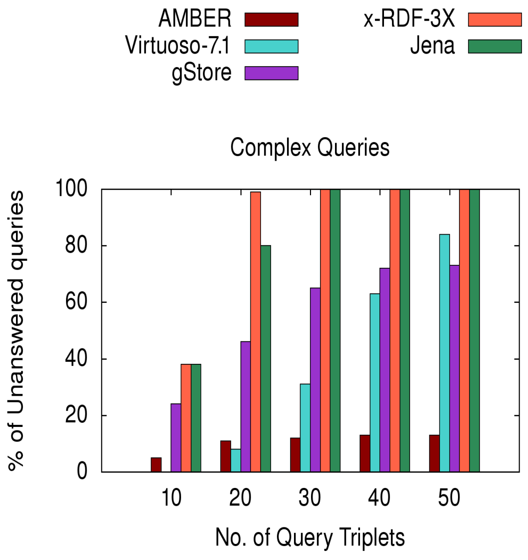
The experimental results for *DBPEDIA* are depicted in Figure 4.7 and Figure 4.8. The time performance (averaged over 200 queries) for *Star-Shaped* queries (Figure 4.7a), affirm that AMBER clearly outperforms all the competitors. Further the robustness of each approach, evaluated in terms of percentage of unanswered queries within the stipulated time, is shown in Figure 4.7b. For the given time constraint, *x-RDF-3X* and *Jena* are unable to output results for size 20 and 30 onwards respectively. Although *Virtuoso* and *gStore* output results until query size 50, their time performance is still poor. However, as the query size increases, the percentage of unanswered queries for both *Virtuoso* and *gStore* keeps on increasing from ∼0% to 65% and ∼45% to 95% respectively. On the other hand AMBER answers >98% of the queries, even for queries of size 50, establishing its robustness.

Analysing the results for *Complex-Shaped* queries (Figure 4.8), we underline that AMBER still outperforms all the competitors for all sizes. In Figure 4.8a, we observe that *x-RDF-3X* and *Jena* are the slowest engines; *Virtuoso* and *gStore* perform better than them but nowhere close to AMBER. We further observe that *x-RDF-3X* and *Jena* are the least robust as they don't output results for size 30 onwards (Figure 4.8b); on the other hand AMBER is the most robust engine as it answers >85% of the queries even for size 50. The percentage of unanswered queries for *Virtuoso* and *gStore* increase from 0% to ∼80% and 25% to ∼70% respectively, as we increase the size from 10 to 50.

The results for *YAGO* are reported in Figure 4.9 and Figure 4.10. For the *Star-*



(a) Time performance



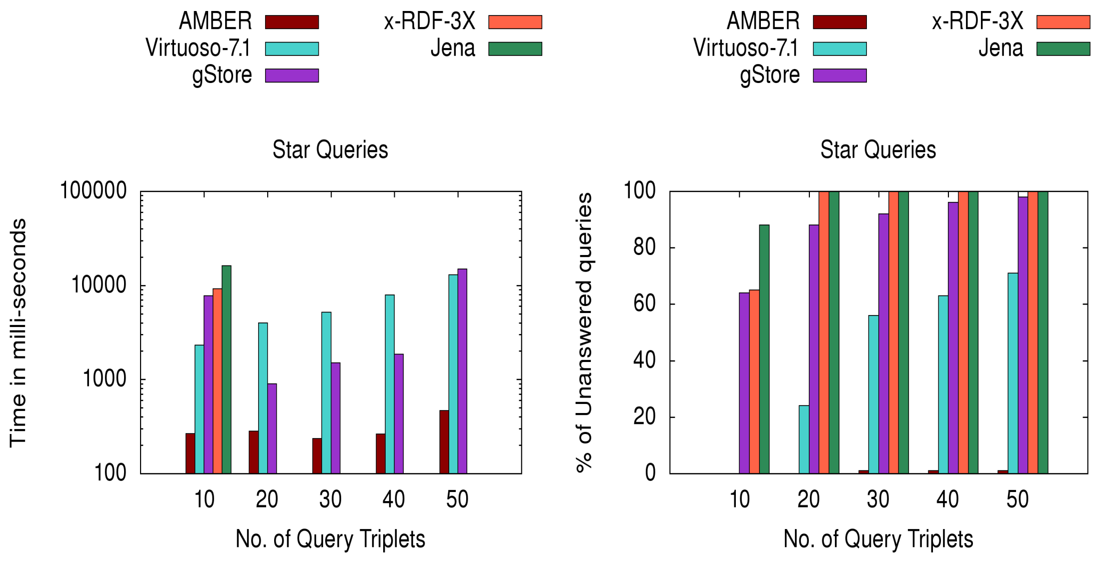
(b) % Unanswered queries

Figure 4.8: Evaluation of (a) time performance and (b) robustness, for *Complex-Shaped* queries on *DBPEDIA*.

Shaped queries (Figure 4.9), we observe that AMBER outperforms all the other competitors for any size. Further, the time performance of AMBER is 1-2 order of magnitude better than its nearest competitor *Virtuoso* (Figure 4.9a), and the performance remains stable even with increasing query size (Figure 4.9b). *x-RDF-3X*, *Jena* are not able to output results for size 20 onwards. As observed for *DBPEDIA*, *Virtuoso* seems to become less robust with the increasing query size. For size 20-40, time performance of *gStore* seems better than *Virtuoso*; the reason seems to be the fewer queries that are being considered. Conversely, AMBER is able to supply answers most of the time (>98%).

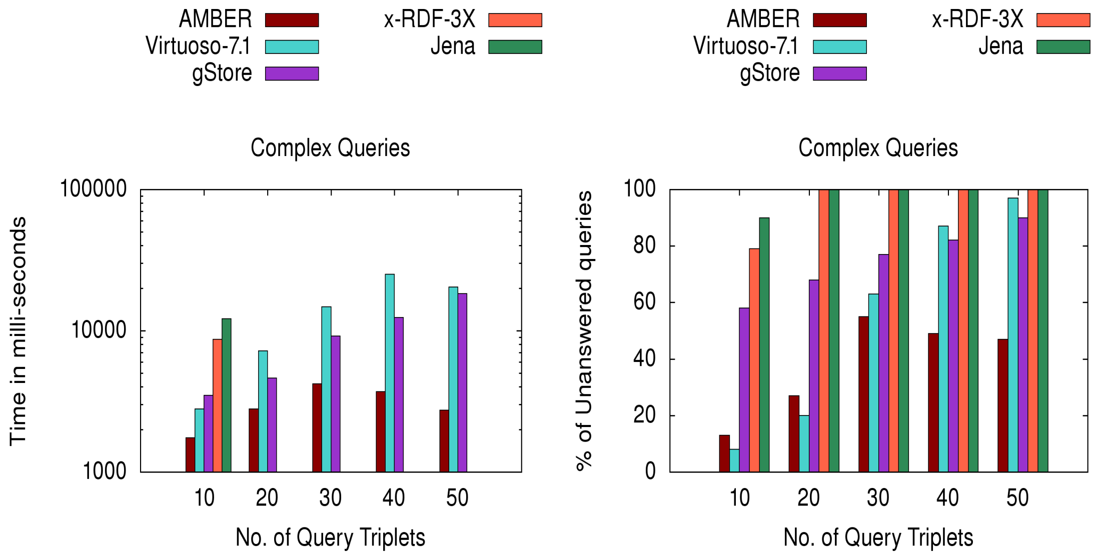
Coming to the results for *Complex-Shaped* queries (Figure 4.10), we observe that AMBER is still the best in time performance; *Virtuoso* and *gStore* are the closest competitors. Only for size 10 and 20, *Virtuoso* seems robust than AMBER. *Jena*, *x-RDF-3X* do not answer queries for size 20 onwards, as seen in Figure 4.10b.

The results for *LUBM100* are reported in Figure 4.11 and Figure 4.12. For the *Star-Shaped* queries (Figure 4.11), AMBER always outperforms all the other competitors for any size (Figure 4.11a). Further, the time performance of AMBER is 2-3 orders of magnitude better than its closest competitor *Virtuoso*. Similar to the *YAGO* experiments, *x-RDF-3X*, *Jena* are not able to manage queries from size 20 onwards; the same trend is observed for *gStore* too. Further, *Virtuoso* always looses its robustness as the query size increases. On the other hand, AMBER answers queries for all sizes.



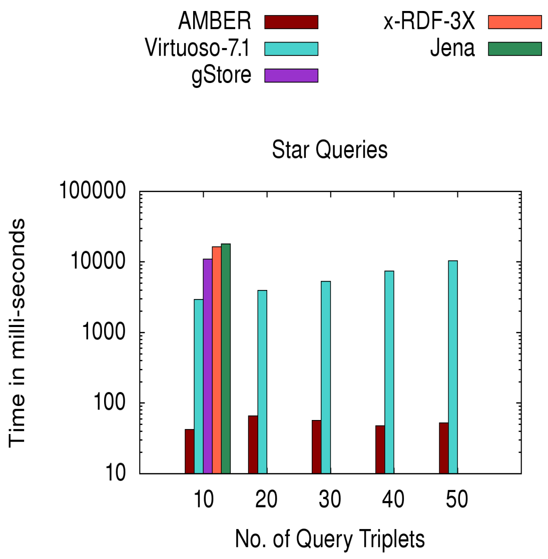
(a) Time performance (b) % Unanswered queries

Figure 4.9: Evaluation of (a) time performance and (b) robustness, for *Star-Shaped* queries on *YAGO*.

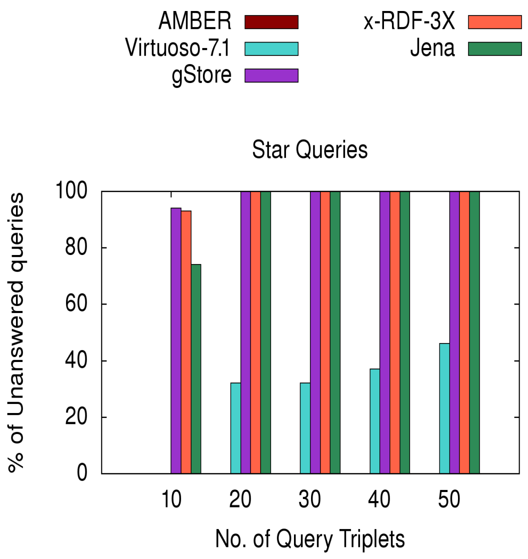


(a) Time performance (b) % Unanswered queries

Figure 4.10: Evaluation of (a) time performance and (b) robustness, for *Complex-Shaped* queries on *YAGO*.

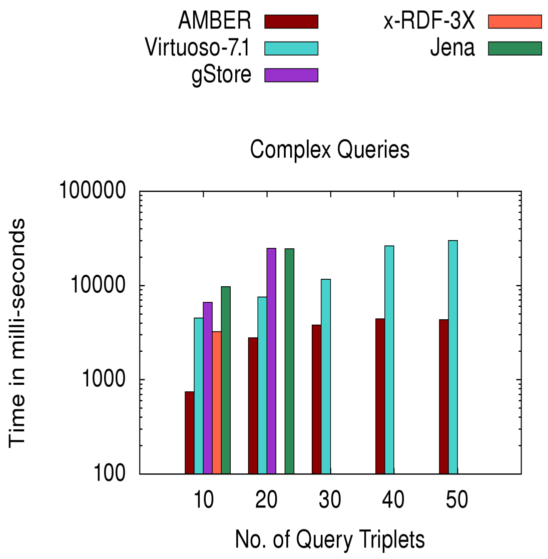


(a) Time performance

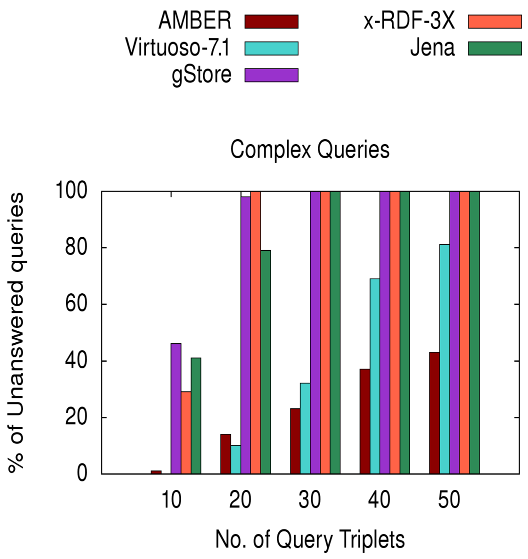


(b) % Unanswered queries

Figure 4.11: Evaluation of (a) time performance and (b) robustness, for *Star-Shaped* queries on *LUBM100*.



(a) Time performance



(b) % Unanswered queries

Figure 4.12: Evaluation of (a) time performance and (b) robustness, for *Complex-Shaped* queries on *LUBM100*.

Considering the results for *Complex-Shaped* queries (Figure 4.12), we underline that AMbER has better time performance as seen in Figure 4.12a. *x-RDF-3X*, *Jena* and *gStore* did not supply answer for size 30 onwards (Figure 4.12b). Further, *Virtuoso* seems to be a tough competitor for AMbER in terms of robustness for size 10 and 20. However, for size 30 onwards AMbER is more robust.

To summarise, we observe that *Virtuoso* is enough robust for *Complex-Shaped* smaller queries (10-20), but fails for bigger (>20) queries. *x-RDF-3X* fails for queries with size bigger than 10. *Jena* has reasonable behavior until size 20, but fails to deliver from size 30 onwards. *gStore* has a reasonable behavior for size 10, but its robustness deteriorates from size 20 onwards. To summarize, AMbER clearly outperforms, in terms of time and robustness, the state-of-the-art RDF engines on the evaluated datasets and query configuration. Our proposal also scales up better than all the competitors as the size of the queries increases.

4.8 Summary

In this chapter, a multigraph based engine AMbER has been proposed in order to answer complex SPARQL queries over RDF data. The multigraph representation has bestowed us with two advantages: on one hand, it enables us to construct efficient indexing structures, that ameliorate the time performance of AMbER; on the other hand, the graph representation itself motivates us to exploit the valuable work done until now in the graph data management field. Thus, AMbER meticulously exploits the indexing structures to address the problem of sub-multigraph homomorphism, which in turn yields the solutions for SPARQL queries. The proposed engine AMbER has been extensively tested on three well established RDF datasets. As a result, AMbER stands out w.r.t. the state-of-the-art RDF management systems considering both the robustness regarding the percentage of answered queries and the time performance.

As already discussed in the beginning of the chapter, we have addressed the SPARQL queries with 'SELECT/WHERE' clause of the SPARQL language, that constitutes the most important operation in any RDF query engines. However, as an extension of this work, we can consider operators like `FILTER`, `UNION` and `GROUP BY`, thereby making AMbER a much more generic RDF querying platform. Another significant extension could be to mange the updating of RDF data. Since RDF data forms the basis of knowledge graphs, over the time, the RDF repositories are updated, with some information being removed and some being added.

Another potential extension of AMbER is to handle *property paths*. Property Paths give a more succinct way to write parts of basic graph patterns and also extend matching of triple pattern to arbitrary length paths. Since the initial SPARQL language for querying RDF provided only limited navigational functionalities, the new version of SPARQL 1.1[5] includes the feature of property paths. However, since it has plenty of lapses w.r.t the performance, as reported in [Arenas et al., 2012], it is compelling to improve the performance of SPARQL querying systems to efficiently answer property path queries.

[5]https://www.w3.org/TR/sparql11-property-paths/

Conclusions Part I

In this part, we focused on efficient computation of subgraph query matches in single large multigraphs. In Chapter 3, we proposed an efficient subgraph matching algorithm - SuMGra, that works on undirected multigraphs. This work has potential applications in many domains where the data can be modelled as undirected multigraphs, and one is interested in finding the matches that are isomorphic to the query. Thus, the problem of *subgraph isomorphism* was at the crux of this work. The proposed indexing structures followed by an efficient backtracking procedure helped in efficient enumeration of subgraph matches.

In Chapter 4, we focused on a particular kind of graphs called knowledge graphs, which are represented as directed multigraphs. In particular, we focused on RDF graphs, where the most prevalent form of SPARQL querying requires to discover homomorphic matchings of the embeddings. The crux of this work was not only the problem of *subgraph homomorphism*, but also managing complex and big queries on large RDF graphs. The proposed approach AMbER, incorporated the index structures from the previous chapter, but was modified according to the problem requirements. AMbER further incorporated efficient query decomposition techniques to answer the SPARQL queries.

Although, to our knowledge, we have proposed efficient subgraph querying approaches for multigraphs, there still remains a plenty of areas to be explored. Focussing on the indexing part, recall that the proposed set of synopses features vary for both isomorphic matching (6 features) and homomorphic matching (4 features). This seems reasonable, since isomorphism is a stronger constraint matching than homomorphism and hence we can afford to have more features. We also recall that these synopses features capture the structural property of the data multigraph only at the vertex granularity. And it is here, where we see an opportunity to capture information which is based on substructure granularity. For example, consider a *triangle* substructure; we could add another synopsis feature that could identify if a data vertex is part of any triangle substructure; this could help in refining the candidates, when the subgraph query itself has a triangle substructure. Many such incremental improvements can be proposed to be part of the synopses features. Further, we observe that using offline index structures in both isomorphic and homomorphic matching, has been proven to be an effective way of approaching the problems of query matching.

One of the major advantages of representing the real world data as graphs and performing query matching on them is that the graph based approaches can naturally be incorporated to perform parallel processing and even distributed computing. Since we follow backtracking approach using DFS search to explore the search space for potential query matches, one can distribute the DFS search process starting from several initial roots. Thus, in combination with the proposed exploration of search space in an efficient manner, distributed approach can be adopted in a seamless manner to address massive multigraphs.

Part II

Mining Multigraphs

Overview Part II

This part of the thesis focuses on the mining aspects of knowledge discovery. In particular, we are interested in discovering frequent subgraph patterns in multigraphs. Given a multigraph data and a user defined threshold value of a frequency measure, the problem of frequent pattern mining is to discover all the patterns that exist in the data multigraphs, that obey the frequency threshold value.

Mining frequent subgraphs or patterns has been one of the most researched topics among the fields of graph data mining [Han et al., 2011, Aggarwal and Han, 2014]. Discovering interesting knowledge in graphs is crucial, since much of the data from various fields (social networks, remote sensing, biochemistry, bioinformatics, etc.) can be represented as graph data. Further, graph mining has become a powerful tool to discover patterns that exist in the data, thanks to the structural property of the graph data.

Frequent pattern mining in multigraphs is motivated by the fact that the existing approaches are not appropriate to handle multigraph data, and hence this contribution fills the void. In Chapter 5, we introduce a novel algorithm for frequent pattern mining in multigraphs. The proposed algorithm not only incorporates an efficient computation of support of a pattern - that helps to quickly decide if a pattern is frequent or not, but also introduces several optimized traversal of search space.

Chapter

5

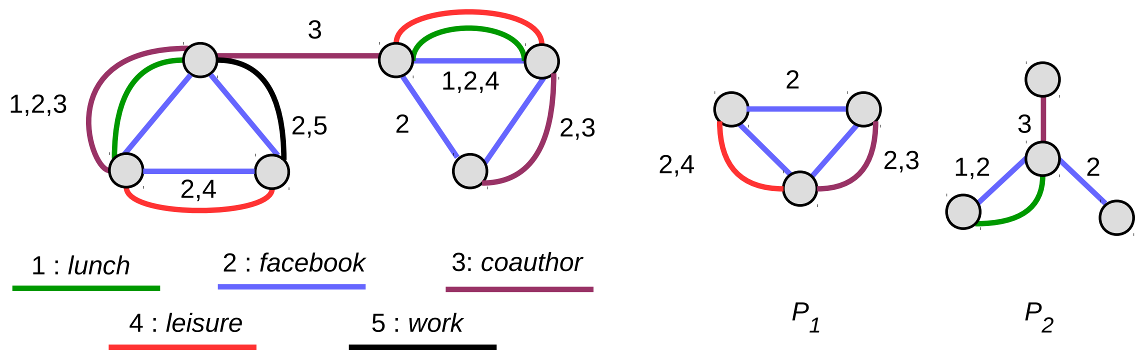
Frequent Pattern Mining in Multigraphs

In this chapter we address the problem of frequent pattern mining in multigraphs. We introduce the concept of support measure that is paramount in deciding if a pattern is frequent or not. We propose a novel algorithm MuGraM *to discover the frequent patterns by optimized search space exploration and numerous pruning techniques. We then perform both qualitative and quantitative analyses on a set of real world datasets.*

5.1 Introduction

Discovering patterns that occur *frequently* in a graph database is the problem of Frequent Subgraph Mining (FSM). We recall that many FSM approaches have been proposed to address graph data that exist in different forms [Inokuchi et al., 2003, Jiang et al., 2013]. Further, several works have been proposed that efficiently discover frequent subgraphs in single large graphs [Kuramochi and Karypis, 2005, Elseidy et al., 2014]. In this chapter we propose a novel FSM approach for multigraphs.

This work is motivated by the fact that the existing FSM approaches cannot be applied to multigraph data. That is, whenever multiple relations (multiedges) exist between a pair of nodes, in order to use the existing FSM approaches one has to map the multiple relations (multiedge) to a unique value (distinct edge label) and then perform FSM, which however, does not yield desirable results, thereby making the existing approaches rather incomplete. For example, Figure 5.1 depicts a typical scenario when we attempt to perform FSM on a multigraph. The data multigraph in Figure 5.1a is an extract of the real world AUCS dataset [Kim and Lee, 2015] that has



(a) A sample data multigraph (b) A set of frequent multigraph patterns

Figure 5.1: (a) A data multigraph and (b) a set of multigraph patterns with frequency $\delta = 2$

five different relations (edge types) namely, *lunch, facebook, coauthor, leisure, work,* defined among a set of university employees (nodes). If we perform FSM on this dataset by setting a frequency threshold equals to 2, the existing FSM approaches output *no* patterns, since they treat a set of relations between a pair of nodes as a unique identifier, rather than treating it as a set of multiple relations. And thus, they are unable to discover those frequent patterns that are spanned from a subset of the relations, as depicted in Figure 5.1b. The objective of this work is to fill the gap in the field of FSM by proposing an approach that is able to extract frequent patterns from multigraph data considering patterns that can span over a subset of multigraph relations.

In this chapter we propose an exact algorithm called MuGRAM (*Frequent* ***Mu****lti****Gra****ph* ***M****iner*) that enumerates all frequent subgraph patterns in a single large multigraph.

5.2 Related Work

A plenty of literature exists for Frequent Subgraph Mining (FSM) for both transactional graph databases and single graph databases. For the transactional graph database setting, the work of Inokuchi et al. [Inokuchi et al., 2000] shapes the foundation for many later works. This work proposes an approach called AGM to efficiently mine the association rules among the frequently appearing substructures in a given graph data set, by treating a transaction as an adjacency matrix. Among the many later works, few notable works are FSG by Kuramochi and Karypis [Kuramochi and Karypis, 2001] that models the problem of finding frequent graphs as a subgraph discovery problem and gSpan by Yan and Han [Yan and Han, 2002], that discovers frequent substructures without candidate generation. Other related works include significant pattern mining Leap [Yan et al., 2008], maximal frequent subgraph mining Margin [Thomas et al., 2010].

Works on single large graph databases can be traced back to Subdue [Holder et al., 1994], that uses minimum description length principle to discover substructures that compress the database, and which was again improved upon later [Gonzalez et al., 2000]. The work of Kuramochi and Karypis [Kuramochi and Karypis, 2005] propose two approaches of vSiGraM and hSiGraM that are depth first approach and breadth first approach respectively. The underlying principle followed is to grow only frequent patterns starting from the smallest sized frequent patterns. The frequency of a subgraph is computed by addressing the problem of subgraph isomorphism. Since the problem of subgraph isomorphism is NP-complete, they propose efficient ways of computing support by storing exact location of each frequent subgraph, which results in speed to memory trade off. In this work, they also introduce the notion of canonical representation to check if two patterns are isomorphic or not, in order to avoid any redundant pattern generation.

One of the most recent FSM approaches for single large graphs is GraMi, wherein they model the problem of frequency evaluation, that requires a costly subgraph isomorphic matching operation, as a constraint satisfaction problem, thereby avoiding the enumeration of all the embeddings, which was deemed expensive in Kuramochi et al. [Kuramochi and Karypis, 2005]. In addition they also propose an approximate version of the algorithm. The latest work in FSM is DistGraph, proposed by Talukder and Zaki [Talukder and Zaki, 2016], which is a distributed approach to address FSM in single large graph settings.

5.3 Preliminaries and problem definition

In this chapter we address the problem of mining single large multigraphs with undirected edges and unlabelled vertices, which will now on be simply referred to as *multigraphs*. We recall the definition of such a multigraph G as a four-tuple (V, E, L_E, T), where V is a set of vertices, T is a set of edge types, $E \subseteq V \times V$ is a set of undirected edges, and $L_E : V \times V \rightarrow 2^T$ is a labelling function that assigns a subset of edge types to each edge E it belongs to. With the labelling function L_E, the edge E becomes a multiedge, and thus G is a multigraph.

One of the major problems during the Frequent Subgraph Mining (FSM) process is to check if a subgraph exists in a given data graph or not, which is the subgraph isomorphism problem. For the ease of readability, we recall the definition of subgraph isomorphism for undirected multigraphs (from Chapter 2).

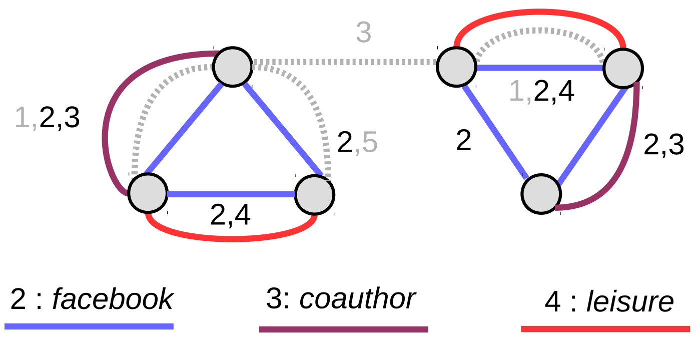


Figure 5.2: Embeddings of pattern P_1 (rest of the data graph is in dotted lines)

Definition 5.1. Subgraph isomorphism for multigraphs. *Given a multigraph pattern $P = (V^p, E^p, L_E^p, T^p)$ and a multigraph $G = (V, E, L_E, T)$, the subgraph isomorphism from P to G is an injective function $\psi : V^p \to V$ such that,*

$$\forall (u_m, u_n) \in E^p,\ \exists\ (\psi(u_m), \psi(u_n)) \in E \textit{ and } L_E^p(u_m, u_n) \subseteq L_E(\psi(u_m), \psi(u_n)).$$

If a pattern P, which itself a multigraph, exists in a single large graph G, then P is subgraph isomorphic to G. Further, if P is subgraph isomorphic to G, then there exists a set of embeddings of P in the multigraph G. The problem of enumerating all the embeddings of P in G is a classic problem of subgraph matching [Lee et al., 2012], reintroduced later in the chapter. From now on, a sub-multigraph pattern P will be simply referred to as a *pattern*.

The embeddings of a pattern P in a given multigraph G play a crucial role for the task of graph mining. To evaluate the frequency of a subgraph, one needs to check if the number of isomorphic embeddings of a pattern in a given graph is atleast δ. Owing to the computational feasibility, we consider *MNI support* measure (Definition 2.12), in order to perform FSM in multigraphs.

For an intuitive understanding of the MNI support measure, consider the example in Figure 5.1. To measure the MNI support for the pattern P_1, we enumerate the embeddings of the pattern P_1, as depicted in Figure 5.2. As we observe, every vertex of pattern P_1 has a unique image in the data graph and 2 distinct embeddings imply that for each pattern vertex, the node image is 2. Thus the MNI support is $\Delta(P_1) = 2$, since the minimum of the image size of all the nodes is 2.

With all this necessary background, we arrive at the problem that we are interested in tackling. Mining multigraphs to discover frequent sub-multigraphs is the problem of *frequent sub-multigraph pattern mining*, formally defined as:

Problem 5.1. Frequent sub-multigraph pattern mining. *Given a multigraph G, and a support threshold δ, discover the entire set of patterns $\mathcal{P}$ in G, such that $\Delta(\mathcal{P}) \geq \delta$.*

In the following section, we propose an efficient graph mining algorithm MuGraM, that seamlessly extracts frequent multigraph patterns from a single large multigraph.

5.4 MuGraM: A frequent pattern mining algorithm

The objective of this work is to address the problem of exact frequent sub-multigraph mining (Problem 5.1) by efficiently discovering all the frequent sub-multigraph patterns that exist in a given single large multigraph G. In the following sections we provide a generic overview of frequent pattern mining algorithms and then look into the challenges faced by addressing multigraphs. We then propose a frequent subgraph mining approach MuGraM, and discuss several optimization techniques to improve its performance.

Our approach towards mining multigraphs follows a similar framework of already existing mining approaches for single large graph settings, as described in [Kuramochi and Karypis, 2005, Elseidy et al., 2014]. A generic framework of mining single large graphs involves the following steps: (i) enumerate the frequent edges (frequent patterns of size $s = 1$) (ii) extend each frequent pattern by successively adding the frequent edges recursively (iii) avoid repeated pattern generation (iv) compute the support of the newly generated patterns to decide if the pattern is frequent or not. Before getting into the details, we introduce some concepts to understand the operations of mining multigraphs.

5.4.1 Multi-edge representation and pattern enumeration

Since in a multigraph, the basic pattern is a multiedge, one might compute the MNI support of each multiedge E in a given multigraph G, to discover the frequent multiedges. However, this set of frequent multiedges is rather incomplete. To perform exact mining, it is necessary to enumerate the subsets of the multiedges, and then compute their support to decided if they are frequent or not; such frequent subsets of the multiedges are what we refer to as a set of *frequent seeds* $\mathscr{P}^1$.

Definition 5.2. Frequent seeds. *A set of frequent seeds $\mathscr{P}^1$ is a union of each frequent subset f of all the distinct multiedges E. Given a multigraph G with n distinct multiedges, a set of frequent seeds is represented as:*
$\mathscr{P}^1 = \cup_{i=1}^{n} \{f \subseteq E_i : \Delta(f) \geq \delta\}$.

For example, consider a multigraph G with two distinct multiedges $E_1 = \{e_1, e_2, e_3\}$ and $E_2 = \{e_2, e_3, e_4\}$, that occur exactly once. If one is interested in finding patterns with support threshold $\delta \geq 2$, then without enumerating the subsets of multiedges, we would have no frequent patterns, which is incorrect. By enumerating the subsets of multiedges, we discover three frequent patterns, each of size $s = 1$, which have multiedges $\{e_2\}$, $\{e_3\}$ and $\{e_2, e_3\}$ respectively. Thus, the set of frequent seeds for G is represented as $\mathscr{P}^1 = \{\{e_2\}, \{e_3\}, \{e_2, e_3\}\}$.

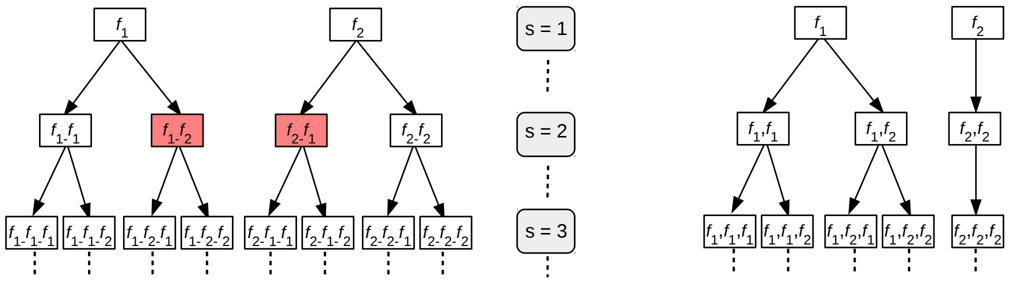
5.4.2 Overview of MuGraM

The proposed algorithm MuGraM is summarised in Algorithm 5.1. The algorithm takes a multigraph G and the support threshold measure δ as the inputs. MuGraM outputs a set of patterns $\mathscr{P}$ that satisfy the minimum frequency threshold δ w.r.t. MNI support measure. As a first step, we enumerate all the frequent seeds, which are nothing but frequent patterns of size-1 $\mathscr{P}^1$, as shown in line 4. To enumerate the frequent seeds, we firstly collect the existing multiedges in G along with the subsets of multiedges. Then we compute the number of occurrences of each multiedge and quickly verify if they appear at least δ times, with respect to the MNI measure. Thus, the set of frequent seeds is represented as $\mathscr{P}^1 = \{f_1, \dots, f_n\}$.

Algorithm 5.1: MuGraM

```
1  INPUT: A multigraph $G$, support threshold $\delta$
2  OUTPUT: All frequent patterns $\mathscr{P}$ in $G$
3  INITIALIZE: $\mathscr{P} = \emptyset$, $\mathscr{S} = \emptyset$, $\mathscr{N}_{\mathscr{P}} = \emptyset$, $ne = 1$
4  ENUMERATE: All frequent seeds (multiedges) $\mathscr{P}^1$
5  ORDERING: Maintain a partial order $(\mathscr{P}^1, \preceq)$, where $\preceq$ is a relation on
   frequency value
6  for every $f \in \mathscr{P}^1$ do
7      $\mathcal{EP} = \{f_e \in \mathscr{P}^1 : \forall f \in \mathcal{P}_C, f_e \succeq f\}$
8      $\mathscr{S}.push(f)$                          /* Stack of $f$ for DFS mining */
9      $\mathscr{P} = \mathscr{P} \cup$ FINDFREQPATTERNS($\mathcal{EP}$, $ne$, $\delta$, $\mathscr{S}$, $\mathscr{N}_{\mathscr{P}}$, $G$)
10 return $\mathscr{P}$
```

Since an efficient way of exploring the search space for single large graphs is to traverse in a DFS manner, as endorsed in [Kuramochi and Karypis, 2005], we follow the *Depth First Search* (DFS) traversal of the search space. MuGraM then discovers frequent patterns by recursively extending the frequent size-1 patterns $\mathscr{P}^1$, by traversing the search space in a DFS manner (lines 6-9). Further, we propose to order the set of frequent seeds $\mathscr{P}^1$ with the increasing order of their frequency of occurrence. Thus, we define the partial order $(\mathscr{P}^1, \preceq)$, where $\preceq$ is a relation on the frequency of occurrence of the elements of $\mathscr{P}^1$. Since we generate new patterns by extending the frequent patterns, the ordering of the frequent with the increasing order of their frequency helps in deciding the infrequent patterns much sooner.



(a) Original search space (b) Compact search space

Figure 5.3: Search space for a set of frequent seeds $\mathscr{P}^1 = \{f_1, f_2\}$

The procedure FINDFREQPATTERNS discovers the frequent patterns in a DFS manner, as depicted in Algorithm 5.2. A stack data structure $\mathscr{S}$ manages the nodes of the DFS tree traversal, which is initialized in line 8. Thus for each frequent multiedge $f \in \mathscr{P}^1$, we invoke the procedure FINDFREQPATTERNS (Algorithm 5.2) to find the subsequent frequent patterns of size $s > 1$ (line 9). All discovered frequent patterns are collected in $\mathscr{P}$. Before getting into the details of FINDFREQPATTERNS procedure, we discuss about the search space spanned by MUGRAM, which will in turn help in understanding the FINDFREQPATTERNS procedure.

5.4.3 Search space spanned by DFS traversal

The search space of MUGRAM starts with a set of frequent seeds $\mathscr{P}^1 = \{f_1, \ldots, f_n\}$, where f_i is a frequent seed (multiedge). In order to discover patterns of size $s > 1$, we extend each frequent seed f_i with the set of frequent seeds $\mathscr{P}^1$. For example, if we have a set of frequent seeds $\mathscr{P}^1 = \{f_1, f_2\}$, in order to discover all the frequent patterns, one needs to explore the entire search space (generated by pattern extension) as shown in Figure 5.3a. In Figure 5.3a, each node in the search tree for size $s > 1$ represents a set of child patterns of size $s + 1$ generated by extending its parent of size s. The search space grows from one level s to the next $s+1$ only if the patterns at the s^{th} level are frequent. This exploration follows the antimonotonic property of the support measure; i.e., if a pattern P of size s is not-frequent, then any pattern that is an extension of P can not be frequent. For example, the node f_1-f_2-f_1 is generated by extending the pattern f_1-f_2, only if at least one pattern in f_1-f_2 is frequent.

In simple terms, each frequent seed $f \in \mathscr{P}^1$ is extended with the entire set of frequent seeds $\mathscr{P}^1$, where repeated extension are allowed; thus f_1-f_1 is a valid extension. However, we propose that it is possible to reduce the search space traversal by making an interesting observation.

Theorem 5.1. *Given a partially ordered set* $(\mathscr{P}^1, \preceq)$*, where* $\preceq$ *is a relation on the set of frequent seeds* $\mathscr{P}^1$*, it is sufficient to extend a frequent seed* $f \in \mathscr{P}^1$ *with the set of extendible seeds* $\{f_e \in \mathscr{P}^1 : f_e \succeq f\}$*.*

Proof. Since the extension of frequent seeds generates pattern children of size $s = 2$, it would be sufficient to prove that at level $s = 2$, we would generate all possible patterns. And due to the antimonotonicity of the support measure, all possible frequent patterns would be discovered for $s > 2$. Now, given a multigraph G with no vertex labels and a set of n frequent seeds $\mathscr{P}^1$, ordered by a relation $\preceq$, pattern extension generates a set of patterns of size $s = 2$, $\mathscr{P}^2 = \cup_{i=1}^{k}\{(f_i \bowtie f_j) : f_i, f_j \in \mathscr{P}^1, j = 1 \to k\}$, where $\bowtie$ is a join operation on two patterns so that all possible structures are enumerated. For level $s = 2$, we observe that any two patterns, $P_1 = (f_i \bowtie f_j)$ and $P_2 = (f_j \bowtie f_i)$ with $i \neq j$ are pairwise isomorphic, and hence redundant. Thus, given a relation $\preceq$, it would be sufficient to retain a pattern $P = (f_i \bowtie f_j)$ w.r.t. the order $f_i \preceq f_j$. □

As a consequence of Theorem 5.1, we do not need to extend each frequent seed with all possible frequent seeds $\mathscr{P}^1$; this reduces the traversal search space, and therefore avoid many redundant mining operations. As depicted in Algorithm 5.1, line 7, every frequent seed f is extended only by a set of extendible frequent seeds $\mathcal{EP}$. Intuitively, if we observe Figure 5.3a, two patterns (shaded) f_1-f_2 and f_2-f_1 are pairwise isomorphic and hence we can only retain one pattern f_1-f_2, by following the relation $\preceq$. Thus, the pattern f_2-f_1 can be safely discarded; further the search space extending from f_2-f_1 can be discarded thanks to antimonotonic property of the support measure, resulting in a much compact search space as depicted in Figure 5.3b.

5.4.4 Discovering frequent patterns

Frequent patterns are discovered in a recursive manner, which follows DFS traversal of search space as depicted in Algorithm 5.2. Since the stack structure $\mathscr{S}$ stores the frequent patterns for the entire search space, we perform DFS exploration until the stack $\mathscr{S}$ is emptied. When the stack $\mathscr{S}$ becomes empty in line 3, we have visited all possible patterns that are an extension of $f \in \mathscr{P}^1$, which itself is repeated until all elements have been extended as observed in Algorithm 5.1, line 6.

As we traverse the search space in a DFS manner (lines 3-14), we keep on extending the initial frequent seed $f \in \mathscr{P}^1$. A pattern P to be extended is fetched from the stack $\mathscr{S}$ (line 4) and the corresponding multiedge P_e which is used for extending P (line 5), is chosen from the set of extendible edges $\mathcal{EP}$. The set of extendible edges $\mathcal{EP}$ is nothing but the frequent patterns $\mathscr{P}^1$ of size $s = 1$. When a pattern P of size s is extended, a set of new patterns $\mathcal{P}_{\mathcal{N}}$ of size $s+1$ are generated.

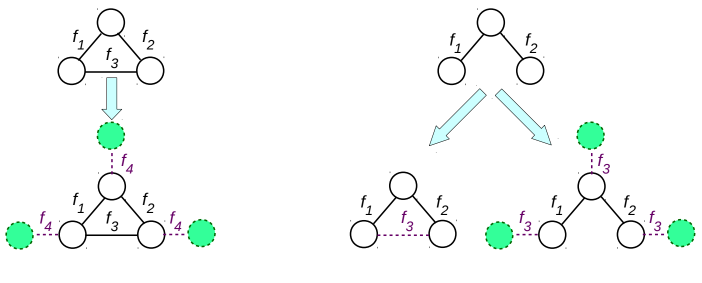
Algorithm 5.2: FINDFREQPATTERNS($\mathcal{EP}$, ne, δ, $\mathscr{S}$, $\mathscr{N}_{\mathscr{P}}$, G)

```
1  if $ne > \mathcal{EP}.size()$ then
2      return                                              /* All edges extended */
3  while $\mathscr{S}$ is not empty do
4      $P := \mathscr{S}.pop()$                            /* The pattern to be extended */
5      $P_e := \mathcal{EP}.ne$                            /* The multiedge used for extension */
6      $\mathcal{P}_{\mathcal{N}}$ := EXTENDPATTERN($P$, $P_e$, $\mathscr{C}$)   /* New patterns */
7      $\mathcal{P}_{\mathcal{F}}$ := COMPUTESUPPORT($\mathcal{P}_{\mathcal{N}}$, $G$)   /* New frequent patterns */
8      if $\mathcal{P}_{\mathcal{F}} \neq \emptyset$ then
9          $\mathscr{S}.push(\mathcal{P}_{\mathcal{F}})$    /* Stack grows with new FSGs */
10         $\mathscr{P} = \mathscr{P} \cup \mathcal{P}_{\mathcal{F}}$
11     else
12         $ne := ne + 1$
13         FINDFREQPATTERNS($\mathcal{EP}$, $ne$, $\delta$, $\mathscr{S}$, $\mathscr{N}_{\mathscr{P}}$, $G$)
14     RESET: $ne = 1$
```

The EXTENDPATTERN procedure performs this extension as shown in line 6. For this set of new patterns $\mathcal{P}_{\mathcal{N}}$, the COMPUTESUPPORT (line 7) procedure computes the support value and checks if there are any frequent patterns, which results in a set of frequent patterns $\mathcal{P}_{\mathcal{F}} = \{P \in \mathcal{P}_{\mathcal{N}} : \Delta(P) \geq \delta\}$; thus $\mathcal{P}_{\mathcal{F}} \subseteq \mathcal{P}_{\mathcal{N}}$. The new frequent patterns in $\mathcal{P}_{\mathcal{F}}$ are added to the stack $\mathscr{S}$, for further pattern extension and added to the repository of frequent patterns $\mathscr{P}$ (lines 8 - 10). However, if $\mathcal{P}_{\mathcal{F}}$ has no frequent patterns, we choose to extend the pattern P, with the next multiedge by incrementing the pointer to choose the next multiedge ne (line 12), and the FINDFREQPATTERNS procedure is called recursively. Once all the frequent seeds $f \in \mathcal{EP}$ have been used for extension, i.e., when $ne > \mathcal{EP}.size()$ (lines 1 - 2), the recursive call is returned. The entire recursive procedure is repeated until the stack $\mathscr{S}$ is empty, thereby traversing the entire search space that spans from the initial frequent pattern $P \in \mathscr{P}$.

5.4.5 Pattern extension

In Algorithm 5.3, we extend a pattern P with an extendable multiedge P_e, resulting in a set of new patterns $\mathcal{P}_{\mathcal{N}}$. A pattern P of size s can be extended into several patterns of size $s + 1$, by attaching a multiedge P_e. However, depending on the structure of pattern P, we can attach P_e in two ways: 1. by introducing an additional vertex 2. without introducing any additional vertex. For example, in Figure 5.4a, we can only extend the pattern P by introducing a node; whereas in Figure 5.4b we need to extend the pattern P by introducing a node as well as without any extra



(a) Extension *only with* an extra node

(b) Extension *with and without* an extra node

(c) Extension considering automorphic groups

Figure 5.4: Various possible extensions of a pattern and automorphic grouping

node (the new nodes used for extension are shaded, while the dotted lines indicate all possible edge extensions). In Algorithm 5.3, the extension is done for every vertex of the pattern (lines 3 - 4), where the addition of an edge is done through $P.add(P_e)$ operation.

Optimization techniques during pattern extension

We now propose a few optimization techniques that we employ during the pattern extension procedure ExtendPattern.

Automorphic extension: While extending a pattern P *with an extra node* by an extendible edge P_e, instead of choosing each vertex $v \in V^p$, it is sufficient to choose a set of vertices for extension that belong to distinct automorphic groups of V^p [Kuramochi and Karypis, 2005]. This in turn reduces the overhead caused by computing the canonical representation for the extended new patterns P_{new}, that is deemed redundant. For example, in Fig 5.4c, we are interested in extending a pattern P that has a multiset of multiedges $\{f_1, f_2, f_2\}$ with an extendible frequent seed f_3, using an extra node. The set of automorphic groups for the given pattern P can be computed as $\{\{u_1, u_3\}, \{u_2\}\}$. Since u_1 and u_3 belong to the same automorphic group, it is sufficient to consider any of them; thus we can either use $\{u_1, u_2\}$ or $\{u_3, u_2\}$ as the set of vertices for extending with P_e. Thus, V_a consists a set of vertices belonging to the distinct automorphic group as shown in Algorithm 5.3, line 2. The procedure ExtendPattern returns a set of new patterns $\mathcal{P}$ and the corresponding canonical representations $\mathscr{C}$.

Canonical verification: One of the vital problems that is often faced in FSM approaches is the repeated generation of patterns. When we are traversing the search space by extending the frequent patterns, it is highly likely that the same pattern was already generated. Such repeated generations can be avoided by checking if the new pattern is isomorphic to the already generated patterns. One of the efficient ways to perform this task is to assign each pattern with a canonical representation and store it in a repository called *canonical repository*. Whenever a new pattern is generated, we check it against the canonical repository; if there is a match in canonical representations, then the new pattern was already generated; else, the repository is updated with the canonical representation of the new pattern. This approach is proposed in [Kuramochi and Karypis, 2005], which we extend for multigraphs.

Canonical verification: Each time a new pattern is generated by an edge extension, we generate canonical representation of P_{new} (line 5) and check if it has already been generated (lines 7-8). Thus, if the newly generated patterns is pairwise non-isomorphic to already generated patterns, we update the set of new patterns $\mathcal{P}_{\mathcal{N}}$ and the corresponding canonical repository $\mathscr{C}$; otherwise, we continue to generate the next new pattern (lines 7).

Thus, each time a new pattern is generated by an edge extension, we generate canonical representation of P_{new} (line 5) and check if it has already been generated (lines 7-8). Thus, if the newly generated patterns is pairwise isomorphic to any of the already generated patterns, we continue to generate the next new pattern (lines 7), since the new pattern has already been generated before; else, we update the corresponding canonical repository $\mathscr{C}$ (line 11).

Detecting infrequent patterns: When we traverse the search space in a DFS manner, we collect a set of infrequent patterns $\mathscr{N}_{\mathscr{P}}$ as we traverse a path, to be discussed later in Algorithm 5.4. Whenever a new pattern is generated we check if the new pattern P_{new} is a supergraph of any of the infrequent patterns in $\mathscr{N}_{\mathscr{P}}$, as depicted in lines 8-9. This avoids the expensive support computation, as we are able to quickly detect that the new pattern can not be frequent, thanks to the antimonotonicty property of the support measure.

Once it is confirmed that the newly generated pattern P_{new} is not infrequent and never generated before, we update the set of new patterns $\mathcal{P}$ (line 12). Once the pattern P has been extended in all possible ways, the procedure EXTENDPATTERN returns the set of newly generated patterns $\mathcal{P}$ and the updated canonical repository $\mathscr{C}$.

5.4.6 Support computation for multigraphs

Computing the support of a pattern is one of the most crucial aspects of any graph mining algorithm, as one has to address the NP-complete problem of subgraph isomorphism. This becomes even harder when we are dealing with multigraphs, as we have to tackle the more generic version of the problem of sub-multigraph isomorphism [Ingalalli et al., 2016]. Thus, in this section, we propose an algorithm for efficient support computation by addressing three vital questions: (i) How to efficiently enumerate the isomorphic embeddings of a multigraph pattern, in order

Algorithm 5.3: ExtendPattern(P, P_e, $\mathscr{C}$, $\mathscr{N}_{\mathscr{P}}$)

```
1  Initialize: $\mathcal{P} := \emptyset$
2  $V_a$ = ComputeAutoGroup($P$)        /* Generate new patterns that are distinct */
3  for every $\{v \in V_a\}$ and $\{[v_i, v_j] \in V^p : i \neq j \wedge i, j \leq |V^p|\}$ do
4      $P_{new} = P.add(P_e)$                 /* Node and edge extension */
5      Generate: $\mathscr{C}.P_{new}$ := CanRep($P_{new}$)
6      if $\mathscr{C}.P_{new}$ is contained in $\mathscr{C}$ then
7          continue;                          /* Skip to next pattern */
8      if HasInfrequentChild($P_{new}$, $\mathscr{N}_{\mathscr{P}}$) then
9          continue;                          /* Skip to next pattern */
10     InheritInvalid($P_{new}$, $\mathscr{P}$)
11     $\mathscr{C} := \mathscr{C} \cup \mathscr{C}.P_{new}$
12     $\mathcal{P} := \mathcal{P} \cup \{P_{new}\}$
13 return $\mathscr{C}$, $\mathcal{P}$
```

to compute the support of a pattern? (ii) Assuming that a pattern is frequent, how fast we can find a set of embeddings of a pattern such that MNI support threshold is reached quickly? (iii) How quickly can we decide if a pattern is *not* frequent? We will address these three issues respectively in the following sections.

5.4.6.1 Sub-multigraph matching

In this section, we address the first question of enumerating the isomorphic embeddings for a given multigraph pattern by formulating the problem of sub-mulitgraph matching, formally defined as follows.

Problem 5.2. *Given a pattern P, and a multigraph G, discover the entire set of embeddings $E = \{e_1, \ldots, e_l\}$ such that every $e_i \in E$ is a sub-multigraph of G and e_i is isomorphic to P.*

The sub-multigraph matching problem involves a much basic problem of solving sub-multigraph isomorphism as described in Definition 5.1. To address the problem of sub-multigraph matching, we take inspiration from Chapter 3, where a subgraph matching approach called SuMGra is proposed that is curated for multigraphs. We recall that SuMGra incorporates efficient indexing structures that leverage multiedge information, which is eventually used during the backtracking procedure to enumerate the isomorphic matches.

However, in this work, since we focus on checking if a pattern P appears in G at least δ times w.r.t. the MNI measure, we do not need to exhaustively compute all the embeddings. Instead, we only need to carefully span the search space in order to count the isomorphic embeddings of a pattern until the support threshold value is reached; however, this counting can not be merely incremental, but has to comply with the MNI support measure. Thus, in the following sections, we propose the optimization techniques required to reach the support threshold value as quickly as possible, by respecting the MNI measure constraints.

5.4.6.2 Optimised support computation

In this section, we discuss about the second question posed to enumerate a set of embeddings of a multigraph pattern such that MNI support threshold is reached quickly. This question is tackled by optimizing the problem of sub-multigraph matching to compute the MNI support measure. To achieve efficient computation of MNI support value, we modify the backtracking approach of sub-multigraph matching [Ingalalli et al., 2016].

For the ease of readability, in this section, we refer to the vertex set of pattern P as $V^p = \{u_1, u_2, \ldots, u_n\}$, and the corresponding vertex set of a matched embedding $e \in E$ as $V^e = \{v_1, v_2, \ldots, v_n\}$. Now consider a pattern P for which we want compute the support; in order to reach the support threshold value of δ, we must enumerate at least δ isomorphic embeddings of the pattern P. In the best case scenario, δ number of embeddings would be enough to reach the MNI threshold value of δ; however, to achieve this, the δ isomorphic embeddings should be non-overlapping (two embeddings e_1 and e_2 are overlapping, if they share at least one vertex) enough as to achieve MNI measure count. This, however, poses a challenge for any subgraph matching approach to enumerate such embeddings in an efficient manner.

To address this issue, we propose a heuristic search procedure that enumerates the isomorphic embeddings that overlap in a least possible manner. In this direction we introduce the concept of what we refer to as the problem of *bin filling*, formally defined as follows.

Problem 5.3. Bin filling. *Consider a set of n bins $\mathscr{B} = \{b_1, \ldots, b_n\}$, where $|b_i| = \delta$ is the capacity of each bin, and a set of l embeddings $M = \{m_1, \ldots, m_l\}$, where each embedding $m_k \in M$ is represented by a vertex set $V^e = \{v_1, \ldots, v_n\}$. Further, an embedding m_k is placed in the bin $\mathscr{B}$ in such a way that each vertex $v_i \in V^e$ occupies a position $b_i \in \mathscr{B}$. The problem of bin filling is to fill all the bins in $\mathscr{B}$ placing the vertex set of least possible set of embeddings $m \in M$, where any two embeddings could be overlapping.*

To understand *bin filling*, let us consider an example where a pattern P has $n = 3$ vertices, and the support threshold value is $\delta = 2$. In this case, we will have to fill a set of $n = 3$ bins $\mathscr{B}$, each with a capacity of $\delta = 2$. Further, let us assume that we have three embeddings each with a vertex set $V_1^e = \{v_1, v_2, v_3\}$, $V_2^e = \{v_1, v_4, v_5\}$ and $V_3^e = \{v_2, v_4, v_5\}$ respectively. Now, if we somehow choose V_1^e and V_3^e, we would fill the bin $\mathscr{B}$ with only 2 embeddings; any other order would require 3 embeddings to fill the bin. Further, more importantly, we do not want to enumerate the entire set of embeddings and then fill the bin $\mathscr{B}$; instead, we are interested in filling the bin $\mathscr{B}$, as soon as an embedding is discovered. Thus, the challenge in designing the heuristic involves the discovery of the isomorphic embeddings in such a way that they are overlapping to the least.

Let us now focus on the backtracking approach of sub-multigraph matching problem SuMGra [Ingalalli et al., 2016] that discovers a set of embeddings by traversing the subgraph search space in a DFS manner. One of the basic steps in a sub-multigraph matching procedure is to maintain an ordering on the vertex set V^p of the subgraph pattern P; subsequently an isomorphic set of vertices V^m of an embedding $m \in M$ are enumerated in that order. Let us assume a linearly order set $(V^p, <)$, where the relation $<$ is defined on the set of pattern vertices $V^p = \{u_1, \ldots, u_n\}$; with this linear order, the corresponding embeddings of pattern P are enumerated, where u_1 is matched first and u_n is the last vertex to be matched. Since we follow backtracking approach, the distinct matched vertices for u_n are replenished faster than the distinct matches for u_1; thus if we allow the embeddings discovered through backtracking approach to fill the bins, we observe a skewed filling of the bins, since bin b_n is filled much faster than bin b_1. We observe that this is an inefficient way to fill the bins in order to compute threshold and hence propose the heuristic search procedure ComputeSupport (Algorithm 5.4).

In Algorithm 5.4, we speed up the filling of the bins, by making the bin filling distribution more symmetric. For a pattern P with $|V^p| = n$ vertices, we compute n distinct vertex order permutations $\sigma = \{\sigma_1, \ldots, \sigma_n\}$, where a permutation σ_i represents a vertex ordering with u_i as its initial vertex (line 3). Thus, by considering every permutation $\sigma_i \in \sigma$, we would have each vertex $u \in V^p$ as the initial vertex for the backtracking procedure. Before applying the backtracking procedure, the potential matches for each vertex $u \in V^p$ are enumerated using the FindInitCandidates procedure (line 4) as proposed in [Ingalalli et al., 2016]; we call these matches as the initial candidate set $Cand$. Then we initialize the set of bins $\mathscr{B}$ for every vertex $u \in V^p$ (line 5).

We now follow the backtracking approach to discover isomorphic embeddings by following a set of vertex orderings σ (lines 7-15). For each pattern vertex permutation σ_i, the FillPatternBins procedure (Algorithm 5.5) is invoked, in order to collect the embeddings of pattern P, thereby filling the set of bins $\mathscr{B}$. If the entire set of bins $\mathscr{B}$ have been filled with δ vertices in them, then the MNI support has been reached and hence we collect the frequent pattern P and return to the previous procedure to fetch the subsequent pattern (line 14-15).

In Algorithm 5.5, we propose the FillPatternBins procedure that rapidly fills

Algorithm 5.4: ComputeSupport($\mathcal{NP_C}$, $\mathscr{N_P}$, δ, G)

```
1  Initialize: $\mathcal{FP_C} = \emptyset$                    /* A set of frequent patterns */
2  for each $P \in \mathcal{NP_C}$ do
3      $\sigma :=$ ComputePerm($V^p$)   /* Permutation ordering of vertices in P:  $V^p$ */
4      $Cand :=$ FindInitCandidates($V^p$, $P$, $G$)
5      Initialize: $\mathscr{B} := \{b_i = \emptyset : i = 1 \rightarrow |V^p|\}$   /* Bins for each vertex in P */
6      Set: $P.frequent = False$;
7      for each permutation $k \in \sigma$ do
8          Fetch: $im$ = InvalidMatches($P$, $u_k$)
9          $Cand(\sigma_k) := \{Cand(\sigma_k) \setminus im \setminus b_i : i = k\}$
10         if $|Cand(\sigma_k)| < \delta$ then
11             break;                                     /* The pattern in infrequent */
12         if FillPatternBins($P$, $\sigma_k$, $Cand(\sigma_k)$, $G$, $\mathscr{B}$) then
13             $P.frequent = True$;
14             $\mathcal{FP_C} = \mathcal{FP_C} \cup P$                 /* Pattern P is frequent */
15             break;                                     /* Skip to next pattern */
16     if $P.frequent = False$ then
17         $\mathscr{N_P} = \mathscr{N_P} \cup P$                       /* Pattern P is not frequent */
18 return $\mathcal{FP_C}$
```

the set of bins $\mathscr{B}$. For every matched initial vertex $v \in Cand(\sigma_k)$, we recursively find an isomorphic match (lines 7-8). Whenever a match is found, i.e., when $|Emb| = |V^p|$, we fill the bin $\mathscr{B}$ with the solutions; and when the bin is filled, we terminate the procedure as the pattern P is frequent (lines 9-12). One prime factor where the proposed FillPatternBins procedure differs from a backtracking procedure to discover isomorphic embeddings is that every time an embedding is found, instead of backtracking to enumerate the next embedding, we terminate the backtracking process and restart the matching procedure for the next vertex $v \in Cand(\sigma_k)$, ensures symmetric distribution of bin filling.

Theorem 5.2. *Considering a pattern with n vertices and the corresponding set of vertex permutations σ that has n distinct orderings, where each ordered vertex set begins with a distinct pattern vertex u_i, it is sufficient to terminate the backtracking procedure once an embedding is discovered in a particular DFS traverse; i.e., by restarting the backtracking procedure for each vertex permutation $\sigma_i \in \sigma$, we will enumerate all the embeddings of the pattern.*

Proof. For a given linearly ordered vertex set $V^p, <$ of a pattern P with n vertices, the backtracking procedure traverses the entire search space by recursively performing DFS search for the patter vertex matches. To achieve this, a single vertex set ordering, say σ_i is sufficient to span the entire search space, since each pattern vertex is recursively allowed to find the corresponding isomorphic match. Now, if we allow the backtracking to be performed for a permutation of vertex set ordering σ of size n, where each vertex order permutation σ_i has u_i as its initial vertex in the ordering, we can then terminate the backtracking process for that DFS search path once an embedding is found. Th rest of the embeddings that could be existing in this DFS path will be discovered by following another vertex order permutation σ_j. □

Algorithm 5.5: FillPatternBins(P, σ_k, $Cand(\sigma_k)$, G, $\mathscr{B}$)

```
1  PotentialMatchesSize: $PM := |Cand(\sigma_k)| + |b_k|$
2  for each $v \in Cand(\sigma_k)$ do
3      if $PM < \delta$ then
4          $P.frequent = false$;                    /* Pattern $P$ is infrequent */
5          return false;
6      $Emb = \{v\}$                                /* A match for initial vertex $u_1$ */
7      for each pattern vertex $u \in \{\sigma_k \setminus u_1\}$ do
8          $Emb = Emb \cup$ RecursiveMatching($u$, $P$, $G$)
9      if $|Emb| = |V^p|$ then
10         $\mathscr{B} = \mathscr{B} \cup Emb$          /* Add a new embedding to the bin */
11         if $\{|b_i| \geq \delta : i = 1 \rightarrow |V^p|\}$ then
12             return true;                         /* Pattern $P$ is frequent */
13     else
14         AssignInvalidMatches($P.u_k$, $v$)
15         Decrement $PM = PM - 1$
16 return false;
```

However, even after the RecursiveMatching procedure, no embedding is discovered, i.e., $|Emb| < |V^p|$, it implies that for the initial pattern vertex u_k, the initial vertex match $v \in Cand(\sigma_k)$, is not a valid match since, there exists no embedding in G with the initial match v. Thus, the vertex v can be considered as an invalid match for the pattern vertex $P.u_k$ and hence the AssignInvalidMatches procedure assigns vertex match v as an invalid match for the pattern vertex $P.u_k$ (line 14). Once all the initial matches have been checked for the embeddings using the RecursiveMatching procedure, the algorithm returns back to Compute-Support to fetch the next vertex order permutation, since the bin is still partially filled.

As a consequence of Theorem 5.2, we can rapidly fill the bins in a more symmetric manner, which in turn helps us to decide if a pattern P is frequent. In the following section, we discuss on the various optimizations that we adopt in computing the support.

5.4.6.3 Optimization techniques during support computation

In this section, we address the third question that was raised in the beginning as to how quickly can we decide if a pattern is *not* frequent? Since the search space of mining frequent patterns grows huge, especially in the case of multigraphs, the number of infrequent patterns collected over the number of frequent patterns discovered also increases. Thus, we spend more time on infrequent patterns to confirm their infrequency than to decide a pattern is frequent. While in the previous section, the problem of bin filling promoted the idea of quickly deciding if a pattern is frequent, also to some extent helps in deciding if a pattern in infrequent. We now discuss few details on optimization strategies, which have been proposed as a part of Algorithms 5.4, 5.5, that help us to predict that a pattern is infrequent, without spanning the entire search space to decide so.

Managing infrequent patterns Recalling Algorithm 5.4, whenever we find a pattern P to be infrequent, we add it to the repository of infrequent patterns $\mathscr{N}_{\mathscr{P}}$ (line 16-17). This repository is maintained as a hierarchical collection of infrequent pattern structures w.r.t. their sizes. Whenever a new pattern is generated during the pattern extension procedure EXTENDPATTERN (Algorithm 5.3), we fetch all the infrequent patterns that are smaller than the new pattern, and check if any of them is subgraph isomorphic to the new pattern; if so, then the new generated pattern can not be frequent due to anti-monotonicity principle.

Managing invalid matches of pattern vertices As we just discussed on leveraging the information of infrequent patterns for early prediction of further infrequent patterns, we now focus on how can we leverage crucial information from frequent patterns and use them to decide the infrequency of subsequent patterns. When we observe the procedure FILLPATTERNBINS (Algorithm 5.5), we initially have a set of potential candidate matches $Cand.\sigma_k(u_0)$ for the initial vertex u_0 of a pattern P. The initial vertex u_0 is fetched from a permutation of vertex order σ_k. The RECURSIVEMATCHING procedure on line 8, manages to find the matches to the rest of the vertices $\sigma_k \setminus u_0$ of pattern P, using backtracking approach, which yields an isomorphic solution of the pattern P. However, if RECURSIVEMATCHING procedure can not find an isomorphic matching, then the potential match $v \in Cand.\sigma_k(u_0)$ for the initial vertex u_0 is indeed an invalid match, since no embedding exists in the entire search space. Thus, we can safely assign v as an invalid match for the initial vertex u_0 of pattern P. Note that we can *assign the invalid matches only for the initial vertex* u_0 since the search space exploration for an embedding begins from that vertex.

However, thanks to the antimonotonicity property, we can push the invalid matches of a pattern at level s to a pattern of level $s+1$, if the pattern at level $s+1$ is an extension of pattern at level s. And since we traverse the search space in DFS manner, we cumulatively collect the invalid matches as we go down the search space. Thus, whenever a new pattern is created during the pattern extension procedure FINDFREQPATTERNS as depicted in Algorithm 5.2, we pass on the invalid matches as depicted by the INHERITINVALID in line 14. Thus when a new pattern P_{new} is evaluated for its support measure by Algorithm 5.4, it already contains a set of invalid matches for all of its vertices V^p. This information is exploited to refine the candidate matches in Algorithm 5.4, line 9. Thus is the candidate matches for the initial vertex $u_0 \in \sigma_k$ are pruned by the operation: $Cand(\sigma_k) := \{Cand.\sigma_k(u_0) \backslash InvalidMatches(\sigma_k(u_0))\}$, thereby reducing the number of candidate matches. This significantly reduces the computational overhead on the backtracking procedure RECURSIVEMATCHING in Algorithm 5.5 (line 8). Further, the reduction in the size of potential matches $Cand$ also helps us in deciding if a pattern can reach the support threshold; i.e., if for a given permutation ordering σ_k, the size of the potential matches is less than the support threshold $Cand.\sigma_k < \delta$, then there is no need to compute the support since, the pattern cannot have δ embeddings. This optimization is exploited in Algorithm 5.4, lines 10-11, and in Algorithm 5.5, lines 3-5.

5.5 Experimental Evaluation

In this section, we evaluate the time performance as well as perform qualitative analysis of the proposed MUGRAM for mining multigraphs. The quantitative analysis is done w.r.t. the time performance, by comparing the performance of MUGRAM with the current state-of-the-art approach GRAMI. Further, the qualitative analysis is done on select datasets of real-world, to realize the importance of multigraph mining.

All the experiments were carried out on a server, with 64-bit Intel 6 processors @ 2.60GHz, and 250GB RAM, running on a Linux OS - Ubuntu 14.04 LTS. MUGRAM is implemented in C++.

5.5.1 Quantitative analysis: Time performance evaluation

In this section, we compare the performance of MUGRAM with a recent state of the art approach GRAMI [Elseidy et al., 2014]. Although GRAMI can not be employed to work on multigraphs, it can be employed to work on edge labelled graphs that have exactly one label per edge.

In order to evaluate the time performance of MuGraM, we use a variety of real world datasets, ranging in different sizes and densities. Table 5.1 lists the datasets and their properties that we use for the time performance of MuGraM and compare it with the recent state-of-the-art approach GraMi.

Dataset	# Vertices	# Edges	# Edge types	Density
DBLP-Mappedgraph	83 901	141 471	910	Medium
Citeseer	3 312	4 732	27	Medium
Microsoft	100 000	1 080 298	240	Dense
Amazon	334 863	925 872	1	Medium

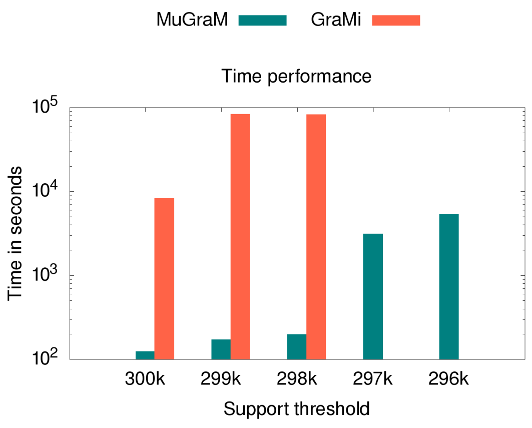
Table 5.1: Properties of the graph datasets

DBLP-Mappedgraph is built from the original *DBLP-Multigraph* dataset by mapping the multiedges in the original dataset to a set of distinct values. The original dataset *DBLP-Multigraph* is built by following the procedure adopted in [Boden et al., 2012], where the vertices correspond to different authors and each edge type represents one of the top 50 Computer Science conferences. Two authors are connected by an edge type if they co-authored at least one paper together in that conference. Thus, in the original dataset, a pair of authors can be co-authors for more than one conference and hence can share more than one edge type. For example, the original dataset can have a multiedge with three edge types $\{e_1, e_3, e_8\}$. Since the existing approaches can discover patterns with only one edge type, we use the mapped dataset, so that we can compare MuGraM with GraMi.

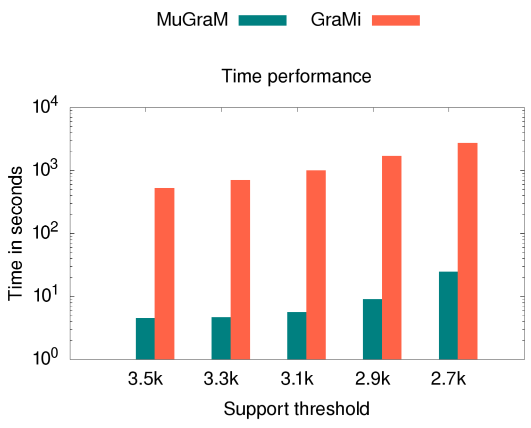
CiteSeer dataset is a graph created with 'publications' as nodes and 'citations' between a pair of publications as edges. Each node has a single label representing the field of Computer Science. The distinct edge types (27) are the similarity measures between the corresponding pair of publications, which vary in the range 0 to 100, where a smaller value implies stronger similarity. *Microsoft* dataset models the Microsoft co-authorship information and consists of an undirected graph, where unlabelled nodes represent the authors and edges represent collaboration between two authors. The different edge types correspond to the number of co-authored papers (240). Both the *CiteSeer* and *Microsoft* datasets are provided by the authors of GraMi [Elseidy et al., 2014].

Amazon[1] dataset was created by crawling Amazon website. It is based on the '*Customers Who Bought This Item Also Bought*' feature of the Amazon website. If a product p is frequently co-purchased with product p', then graph contains an undirected edge from p to p'. This dataset contains only unlabelled edges and has been chosen to evaluate the performance of MuGraM when dealing with simple

[1]http://snap.stanford.edu/data/com-Amazon.html



(a) *Amazon* dataset



(b) *DBLP* dataset

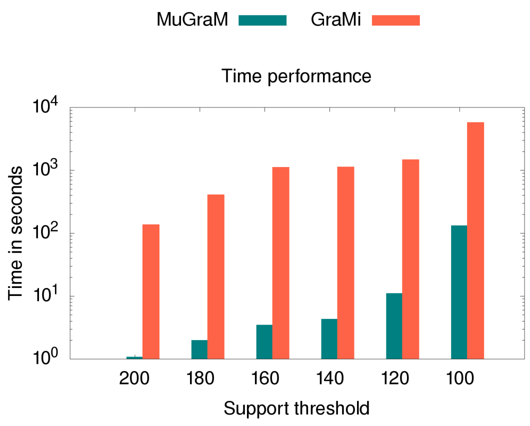
Figure 5.5: Time performance of *Amazon* and *DBLP* datasets

graphs.

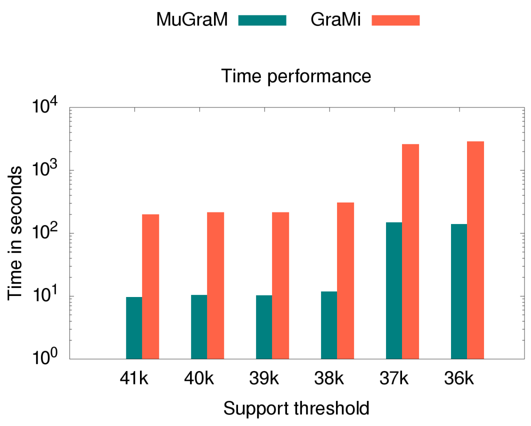
All the experiments have been conducted for varying values of support measure threshold δ, depending on the dataset. Further, for each support threshold value, a maximum execution time of 1 day was allotted in order to discover the frequent patterns; we report the time taken in seconds (on a logarithmic scale) to discover patterns only after the respective algorithms terminate their execution, where 1 day is the maximum time allowed.

The time performance for *Amazon* dataset is depicted in Figure 5.5a, where we vary the support value from 300k to 296k. As we observe, although *Amazon* is a simple graph with only one edge type (not a multigraph), MuGraM outperforms GraMi by ∼10 orders of magnitude. Further, for support values from 297k, GraMi fails to terminate within 1 day of maximum execution time allotted, while GraMi successfully discovers frequent patterns. Results for *DBLP-Mappedgraph* dataset are depicted in Figure 5.5b. We recall that although *DBLP-Mappedgraph* dataset has 50 edge types, it has 910 distinct multiedges, which we map to unique identifiers in order to run both MuGraM and GraMi. For *DBLP-Mappedgraph*, we vary the support values from 3.5k to 2.7k and output the time in seconds. From Figure 5.5b we observe that MuGraM outperforms GraMi with almost 1-2 orders of magnitude.

For the *Citeseer* dataset, from Figure 5.6a we observe that the time performance of MuGraM is 2-3 orders of magnitude better than GraMi for higher support values, although for relatively lower support value of 100, we outperform GraMi by ∼1 order of magnitude. For the *Microsoft* dataset, both MuGraM and GraMi are able to terminate within the permissible time of 1 day for the entire support range of 41k to 36k. For higher support values, MuGraM is ∼1 order of magnitude better than GraMi, while the performance gap slightly shrinks for the lower support values.



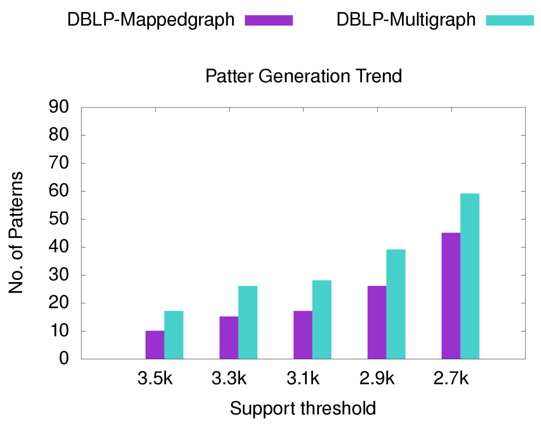
(a) *Citeseer* Dataset



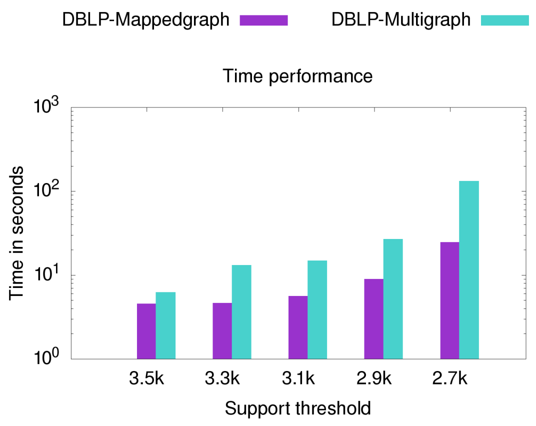
(b) *Microsoft* Dataset

Figure 5.6: Time performance of *Citeseer* and *Microsoft* datasets

We perform an additional quantitative analysis in order to test the potentiality of the proposed MuGraM algorithm. We perform this test by employing MuGraM for a real multigraph data. We recall that the original *DBLP-Multigraph* dataset is inherently a multigraph, and we have mapped the distinct multiedges to unique identifiers in order to compare the time performance as already discussed in Figure 5.5b. We are now interested in comparing the performance of MuGraM on the original multigraph - *DBLP-Multigraph* and the mapped graph - *DBLP-Mappedgraph*. Figure 5.7 depicts the behaviour of multigraph and mapped graph performances; in Figure 5.7a, we compare the number of patterns discovered for the two versions of the same *DBLP* dataset, and we observe that we are able to discover patterns for *DBLP-Multigraph* that were not by using *DBLP-Mappedgraph*. It is to be noted that the patterns from mapped graph are a subset of the patterns from its multigraph, and thus we infer that mining multigraphs has the advantage of discovering the frequent patterns which are otherwise not discovered by mapping them. Thus, our experiments demonstrate the importance of the multigraph mining approach MuGraM over the existing state-of-the-art- approaches. Further, we also evaluate the time performance of MuGraM when employed to mine multigraphs. In Figure 5.7b, we observe that *DBLP-Multigraph* needs some amount of additional time to discover multigraph patterns which did not exist in the case of *DBLP-Mappedgraph*. However, it is important to note that this additional time grows only linearly for multigraphs when compared to the corresponding mapped-graphs, although multigraphs pose a tougher challenge in terms of search space than the corresponding mapped-graphs.



(a) No. of patterns outputted



(b) Time performance

Figure 5.7: Comparison of time performance and # of outputted patterns for DBLP dataset

5.5.2 Qualitative analysis

We conduct qualitative analysis for the following datasets, for which we employ MuGram to discover the below mentioned patterns. It is important to learn that these patterns are frequent patterns and hence they reveal the majoritarian behaviour of a set of users/entities (nodes).

AUCS: The *AUCS* dataset [Kim and Lee, 2015] is a multi-layered dataset, where each layer represents the relationships among 61 employees of a University department in five different aspects of (i) coworking, (ii) having lunch together, (iii) friends on facebook, (iv) spending leisure time (v) have coauthorship. We model this dataset as a multigraph with 61 nodes and a set of edge types $T := \{lunch, facebook, coauthor, leisure, work\}$. In the original dataset there are 620 edges, where each edge belongs to one of the above mentioned five layers; we map all the edge types T that belong to different layers of a pair of nodes (say, v_i and v_j) into a multiedge (e.g., $E_i := \{facebook, leisure\}$) which results in 353 multiedges.

For *AUCS* dataset, we extract multigraph patterns by setting MNI support threshold $\delta = 30$, and report few interesting patterns as depicted in Figure 5.8. Since the dataset has very few coauthors, for $\delta = 30$, we do not find any patterns that have the edge type *coauthor*. Pattern P_1 depicts a frequent behaviour, where we observe that the employees who lunch together also work together. Also, by the relation of transitivity, we can conclude that every employer in the pattern work at the same place. Since the employee a only works with b and c and does not share any other relations, all the three employees a, b and c seem not to know each other. On the other hand, b and e as well as c and f seem to be close to each other as they not only lunch together, but also spend leisure time after work. However, b and d

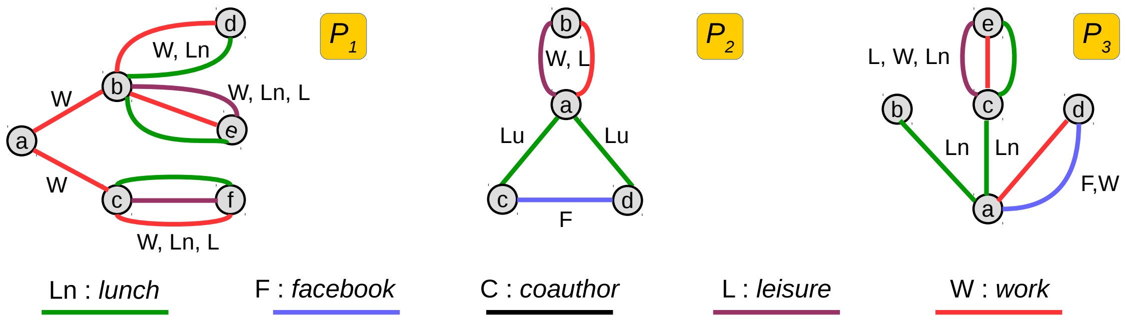


Figure 5.8: Few interesting patterns of size $s = 4$ and $s = 5$ for *AUCS* dataset

seem to have a limited acquaintance, as they only lunch together, apart from being co-workers.

Pattern P_2 depicts a frequent behaviour where two employees (a and b) seem to be good friends as they share leisure time, although do not lunch together. Further, a shares lunchtime with c and d, who are not co-workers implying that c and d do not work in the same place, which further helps us to understand as to why b does not lunch with a. Interestingly, both c and d who share lunchtime with a on separate occasions, are friends on facebook. In pattern P_3, the relation between a and d is interesting to observe, as although they share the workplace and are friends on facebook, they never have any other relations, which helps us deduce that they are not very close friends. Further, a prefers to lunch with non-coworkers b and c, whereas c and e share many relations and hence are good friends.

ATN: The *ATN* [Cardillo et al., 2012] dataset is an Air Transportation Network data that describes airline companies operating in Europe during the year 2011. The dataset is represented as a multi-layered graph, where the nodes and edges represent airport locations and routes, respectively, and each layer corresponds to a different airline company. In the original dataset, there are 417 nodes, 3 588 edges and 37 layers. We map this data into a multigraph dataset, with 417 nodes and 2 953 multiedges.

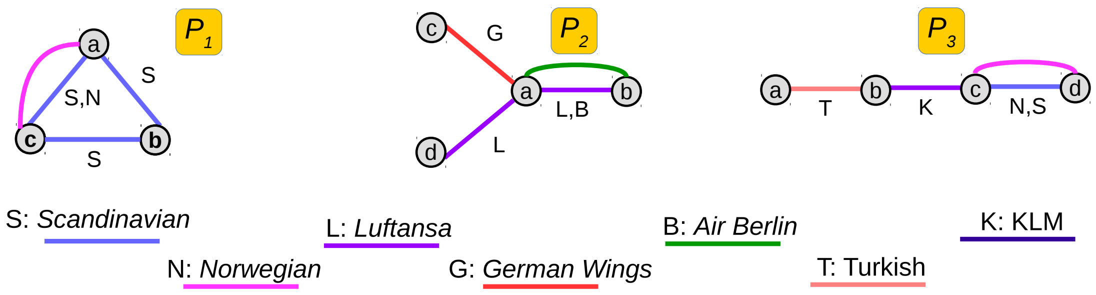


Figure 5.9: Few interesting patterns of size $s = 3$ for *ATN* dataset

For *ATN* dataset, we extract multigraph patterns by setting MNI support threshold $\delta = 15$, and report a select few interesting patterns as depicted in Figure 5.9. Pattern P_1 depicts a frequent behaviour of three airports a, b and c interconnected by *Scandinavian* airlines, where a and c are additionally connected by *Norwegian* airlines. Since these three airports are strongly connected by forming a clique, we can deduce that such pattern could be revealing a common behavior where the three airports could belong to bigger cities (with more passenger traffic), and hence more likely to be capital cities. And since the carriers are Scandinavian and Norwegian, the three cities probably belong to Scandinavian countries.

Pattern P_2 depicts a frequent situation in which four airports (a,b,c and d) are involved. Airport a appears to be a major airport hub form where many links exist (from a to b, c and d). With this pattern being frequent, we can deduce that a major airport (a) often offers connectivity to airports with relatively lesser air traffic (b, c, d). In this particular pattern, since all the airlines belong to Germany, a could be the busiest airport in Germany.

MRM: The *MRM* [Kim and Lee, 2015] dataset is provided by MIT Media Lab. It represents human interactions using mobile phones. The dataset is a multi-layered graph where the nodes represent users and the edges represent the way they interact with each other. The dataset has four different layers that represent different ways of communication namely, *phone calls*, *friendship* claims, *bluetooth* proximity scans, and *text message* exchanges. We map the original dataset to a multigraph with 94 nodes and 3 079 multiedges.

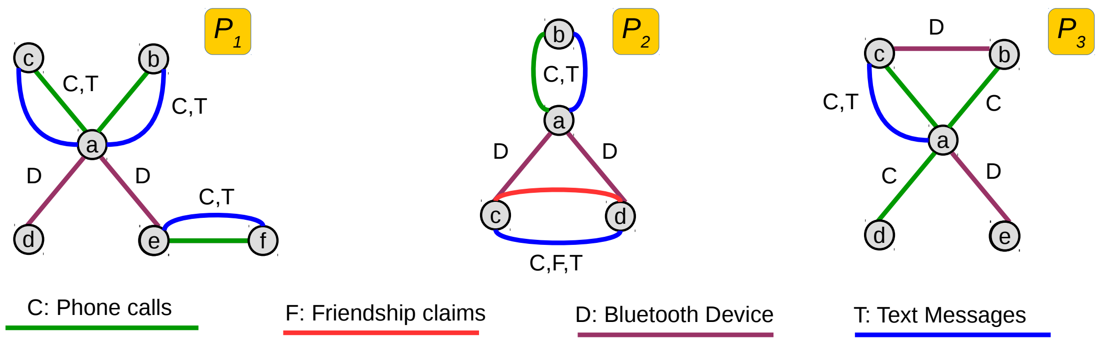


Figure 5.10: Few interesting patterns of size $s = 4$ and $s = 5$ for *MRM* dataset

For *MRM* dataset, we extract multigraph patterns by setting MNI support threshold $\delta = 30$, and report a select few interesting patterns as depicted in Figure 5.10. In In Figure 5.10, pattern P_1 depicts a frequent behaviour involving six users (nodes), where one user is pivotal (here a) to the interaction pattern. Although the user a has a direct interaction with b and c through *phone calls* and *text messages*, a is in proximity with d and e, who together form a 2-degree interaction. Further, e usually performs *phone calls* with f, indicating that friendship interaction occur as small communities (two in the depicted patterns: i) a, b, c and ii) e, f), that are in turn loosely connected. Pattern P_2 portrays a situation, where four users are interconnected; in such frequent structure two users (here c and d) interact very closely, since they claim the *friendship* along with *phone calls* and *text messages*; the other two users (here a and b) interact with *phone calls* and *text messages*. These two small groups often have a bluetooth proximity, indicating that they hangout around same places (in particular a, b and c) but they are isolated in term of direct communications (*phone calls*, *friendship* claims and *text message*).

5.6 Summary

In this chapter, we proposed a generic multigraph mining algorithm called MuGraM that can discover frequent patterns in multigraphs. We then discussed the challenges offered by multigraphs, in terms of search space complexity as well as the complexity due to subgraph isomorphism and proposed optimized solutions to tackle them. We performed experimental analyses (both quantitative and qualitative) on various datasets and compared the performance of MuGraM with the existing state-of-the-art approach GraMi. On one hand, the qualitative analysis exhibits the power of MuGraM in discovering interesting multigraph patterns, which can not be discovered by any existing graph mining algorithms; on the other hand, our quantitative analysis shows that MuGraM outperforms the prevalent graph mining approaches, even when handling simple graphs.

Conclusions Part II

Frequent subgraph mining (FSM) is one of the most prevalent problems in the field of data mining, and certainly so, under the gamut of knowledge discovery. In this part we learnt the importance of FSM problem in the context of multigraphs, and the importance of frequent patterns thus discovered in multigraph data.

In Chapter 5, we understood the core principles of the proposed algorithm MuGraM, that discovers the exact frequent subgraph patterns in multigraphs, given a user defined frequency threshold value. We incorporated a support measure called Minimum Node Image (MNI), which is recognized for its linear time complexity. We proposed a backtracking procedure that amalgamates support measure evaluation and DFS search space exploration, in order to determine if a pattern is frequent. Qualitative analysis of the experiments revealed that MuGraM is able to discover patterns in multigraphs, that the existing FSM approaches are not able to discover. When tested on simple graphs, in terms of time performance, MuGraM outperformed the existing approaches.

Although we believe that the proposed approach MuGraM has filled the void in the graph data mining community by addressing the problem of FSM for more generic class of graphs, we do observe a plenty of potential extensions. The current version of MuGraM is a standalone system algorithm, and we intend to extend the work for distributed systems, thereby enabling MuGraM to be employed for massive multigraphs. In this work, we consider multigraphs without any vertex labels; for the sake of completeness of the approach, we can perform a trivial extension of MuGraM to consider vertex labelled multigraphs. In this work, although we consider a computationally lighter support measure of MNI, there are a plenty of other support measures such as *Maximum Independent Set* measure, *Overlap Hypergraph* measure [Wang et al., 2013], that could be considered under MuGraM. Although these support measures are computationally expensive, they could discover more interesting patterns.

Case Study

Chapter 6

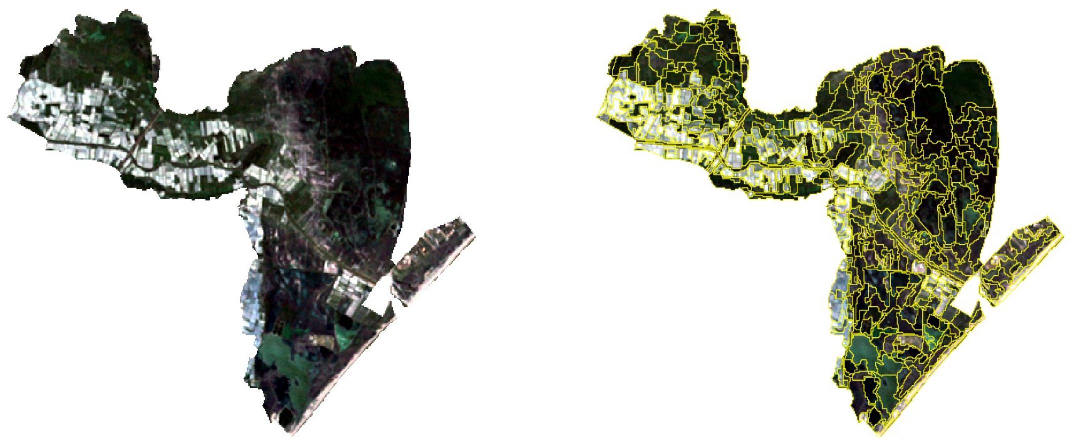
Knowledge Extraction for Remote Sensing Data

In this chapter, we perform a case study analysis on a remote sensing data that harnesses the contributions of this thesis in terms of both knowledge discovery and retrieval.

We perform a case study analysis of the works proposed in this thesis for the application of remote sensing data. In Figure 6.1a, we observe a remote sensing image taken with a pixel resolution of 30 meters, over the 'Basse Plaine de l'Aude' (BPA) or 'Lower Aude Valley', in France on 7^{th} of May 2009. This region is covered with a plenty of natural areas like river, forest, lakes, etc., along with semi-natural/cultivated areas like vineyards. In the image, the greener areas represent the natural vegetation, darker areas represent the water bodies like lakes and rivers and silver/whitish areas represent the semi-natural areas like vineyards.

The 'Basse Plaine de l'Aude' (BPA) dataset has four different relations namely $\{NDVI, NDWI, VSDI, Brightness\}$: the Normalized Difference Vegetation Index ($NDVI$) is a vegetation indicator; the Normalized Difference Water Index ($NDWI$) is an indicator on plant water content; the visible and shortwave infrared drought index ($VSDI$) is a drought indicator; and the *Brightness* relation is an indicator of soil reflectance, where *Brightness* is calculated as a weighted sum of all the bands and is defined in the direction of principal variation in soil reflectance.

This BPA satellite image undergoes segmentation operation that results in an image as observed in Figure 6.1b. The segmented image can then be readily modelled as a graph by mapping each segmented object as a vertex and the relations between any two objects is represented as a multiedge. Since the relations in BPA dataset have continuous values, each relation is allocated with a discrete interval; and two



(a) Satellite image (b) Satellite image after segmentation

Figure 6.1: A satellite image of *Basse Plaine de l'Aude* (BPA) or *Lower Aude Valley, France*', and the same image after segmentation

objects share an edge if they share the same discreet interval value. Finally, when two vertices (objects) share more than one edge (relation), the such a scenario is represented as the vertices having a multiedge, and the entire graph can then be termed as a multigraph.

Discovering frequent patterns

In order to discover interesting patterns on the multigraph data of the remote sensing data BPA, we run the proposed multigraph mining algorithm MuGraM to discover a set of frequent patterns. These patterns are the frequent patterns that occur at least 150 times in the given BPA dataset. The frequency threshold was chosen arbitrarily in order to discover frequent patterns. For case study analysis, we consider two frequent patterns as depicted in Figure 6.2.

In order to retrieve useful knowledge from the muligraph data of the remote sensing data BPA, we now run the proposed subgraph pattern matching algorithm SuMGra that unearths all the embeddings of the earlier discovered pattern, in our case for P_1 and P_2. The pattern P_1 has 2 232 instances in the original BPA image and the pattern P_2 has 1 704 instances in the original BPA image. The images with corresponding embeddings for both pattern P_1 and P_2 are depicted in Figure 6.3.

In Figure 6.2, we observe that pattern P_1 has lesser information (and hence lesser constraints) when compared to P_2 and hence P_1 is more generic in covering the zone of the satellite image when compared to P_2 as evident in Figure 6.3, where

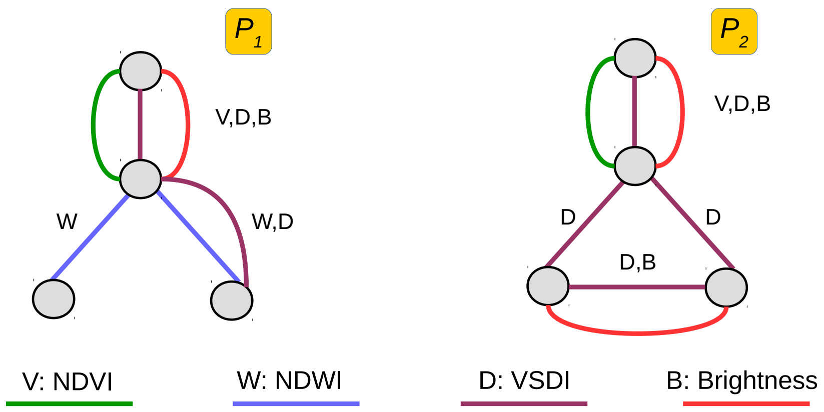


Figure 6.2: Two frequent patterns for the BPA dataset that occur at least 150 times

the locations of embeddings are marked in light green color.

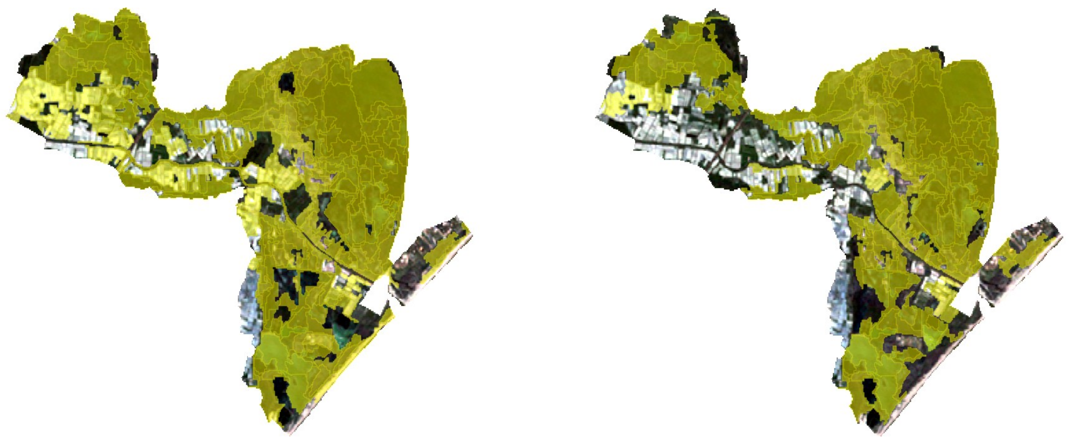
Pattern P_2 is more specific for naturally occurring zones like forests, lakes and rivers (green and dark colours in Figure 6.1a), and the pattern does not cover the semi-natural regions like vineyards (silver coloured in Figure 6.1a).

In Figure 6.4 we observe a zoomed part of the original image and the corresponding coverage of the embeddings of pattern P_2. Referring to Figure 6.4b, we observe that while the embeddings have been correctly located (green coloured on the lower right section of the image), the segmentation itself that was done as a preprocessing, is not accurate. This is because, by comparing Figure 6.4b with the original image Figure 6.1a, we observe that the two rightmost objects in Figure 6.4b are segmented, although they should be the same, as observed in Figure 6.4a.

Possible extensions

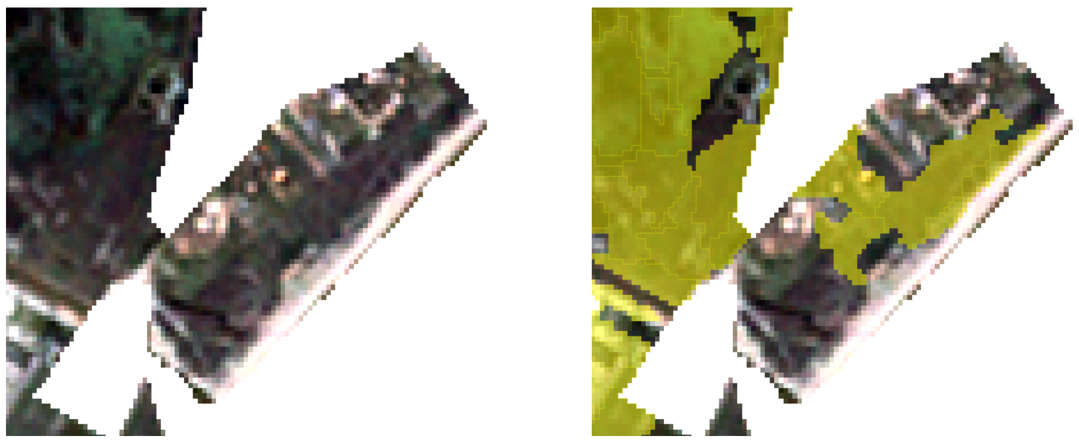
In the field of environmental sciences, simulation models are an important way to study systems inaccessible to scientific experimental and observational methods, and also an essential complement to those more conventional approaches. Our pattern mining framework can be employed to support spatial simulation analysis. In particular, the extracted patterns can be supplied as input for a spatial simulation model with the objective of guiding the simulation process. The frequent patterns not only represent some kind of spatial constraints, but also introduce knowledge about the interactions that appear between the involved spatial entities. For example, one could be interested in simulating the agricultural landscape in order to understand the interactions between the various fields.

In the above discussions, all the tasks were operated in an unsupervised manner.



(a) Embeddings of pattern P_1 in BPA image (b) Embeddings of pattern P_2 in BPA image

Figure 6.3: The images showing the embeddings of the frequent patterns in the original BPA image



(a) Zoomed part of the original imaged (b) Zoomed image to visualize embeddings

Figure 6.4: Zooming the original image to study the validity of the extracted knowledge

If we have ground truths available for the remote sensing image, then we can perform supervised classification on original image and then, we could look for interesting patterns in a particular class. For example, if we are interested in knowing interesting patterns that could exist in only the vegetation zone, then we would be able to restrict our search only to that extent, thereby yielding more specific and practical knowledge.

Summary

Chapter

7

Summary and Future Directions

Thesis Summary

This thesis makes contributions in the field of knowledge retrieval and discovery by addressing the problems of subgraph query matching and frequent subgraph mining respectively. These problems have been addressed for data from various domains, that can be represented as graphs. Graphs - a class of semi-structured data - have garnered the attention for several decades now, owing to their inherent property of modelling complex relations among the entities in a succinct manner, and yet retaining the structural properties of the data.

As already discussed in the introduction, extracting knowledge has evolved to become a necessity ever since there has been an unprecedented growth in the amount of data collected, and the information retrieved. Although the field of knowledge extraction is determined to automate the process of discovering knowledge, several challenges are encountered. Pattern enumeration and mining plays almost an intermediate role in this regard, by capturing the instances of patterns or discovering the patterns hitherto unknown, that exist in the vast information. However, on one hand, the pattern enumeration and mining approaches have to be time efficient owing to the huge amount of information to be processed, on the other hand, the discovered patterns have to be interesting enough to be deemed as knowledge. Thus, discovering such interesting patterns is indeed a challenge in the field of frequent subgraph pattern mining.

The contributions in this thesis have been in the direction of efficient retrieval of patterns and mining frequent patterns in multigraphs. These contributions fill the gap in the graph analytics community, by addressing these problems in the case of multigraphs, which are a generic class of graphs.

In Part I of the thesis, we focused on the querying aspects of multigraphs. In the beginning we learnt the challenges posed by the multiedges in the multigraphs, and proposed efficient indexing structures to capture the vital information in the multigraph dataset, which are ultimately utilised by the subgraph query matching procedures. The proposed algorithm SuMGra in Chapter 3 is one such novel approach, that retrieves isomorphic matches for a given query subgraph in undirected multigraphs. The subgraph matching procedure is in particular, a backtracking approach, that explores the search space in a DFS manner; however, in doing so, it is important to begin the exploration form a vertex (a point in search space) which guarantees to discover the matches, which is partially achieved by the ordering of the query vertices. In this direction, we proposed heuristics for ordering the query vertices. Further, we proposed a backtracking procedure that is coupled with the proposed indexing structures, to quickly discover the matched embeddings for the subgraph query.

The proposed subgraph isomorphic matching algorithm SuMGra has many potential applications in the fields of bioinformatics, social network and many others. In [Bonnici et al., 2013], the subgraph matching approach is applied to biochemical data, where the authors enumerate the subgraphs matches of patterns to get insight into evolutionary mechanisms, as these patterns could be repeated in the same network or in different networks. Similarly, a subgraph matching tool for biological graphs has been proposed in [Tian et al., 2007], as they are convinced that graphs provide a powerful primitive for modelling biological data.

After addressing the problem of querying isomorphic matches in undirected multigraphs, in Chapter 4, we moved to the problem of querying homomorphic matches in directed multigraphs. This problem has potential applications in the domain of *Linked Data* - a method of publishing structured data so that it can be interlinked and become more useful through semantic queries. This has an advantage of enabling data from different sources to be connected and queried [Bizer et al., 2009]. For such Linked Data, open standards exists, where the data is stored in the form of RDF triples and queried using SPARQL language. The field of relational databases has made a plenty of contributions for efficiently performing SPARQL query operations on an RDF data [Erling, 2012, Carroll et al., 2004, Neumann and Weikum, 2010]. Since very few graph database approaches exist, even though RDF data is inherently a graph structure, in this thesis, we discuss on our proposed approach - AMbER, that performs SPARQL querying on RDF data as a graph database approach.

In AMbER, we firstly model an RDF data as a multigraph, since RDF data is rich with multiple relations between a pair of entities, which allows for a succinct representation. We employ the index structures that were proposed for SuMGra, but we modify them in accordance with homomorphic constraint. Since the SPARQL queries can grow bigger in size, and also become more complex in terms of structure, we proposed an efficient query decomposition approach, that leverages the proposed index structures for a faster traversal of search space. Our experimental results proved that the proposed AMbER is not only faster when compared with the existing approaches, including the relational database approaches, but also robust in answering SPARQL queries.

In Part II of the thesis, the focus was on mining frequent subgraphs (FSM) in multigraphs. The main motivation for this part of work has been the inefficacy of the existing FSM approaches in discovering patterns in multigraphs. We learnt that the existing approaches are unable to discover the patterns that are spanned by various combinations of subsets of the multiedges. Furthermore, we realized that the sheer extension of the existing approaches for multigraphs is non-trivial, and hence the problem had to be tackled with a different approach, which led to the emergence of the proposed approach MuGraM.

In MuGraM, we outlined the generic algorithm in the beginning, followed by more detailed description and analysis. Since in a multigraph, the pattern growth expands exponentially due to multiedges, we proposed a possible reduction of search space, thereby improving the time performance for multigraphs. Further, we coupled the evaluation of MNI support measure with the pattern matching procedure, that is necessary to compute the number of embeddings of a pattern. This greatly helped to quickly decide if a pattern is frequent or not; we formulated this problem as *bin filling* problem, where the vertices of patterns are represented as a set of bins, and the task is to fill each bin with a set of vertex matches, so that all the bins are filled with distinct matched vertices. Several other optimization approaches were also incorporated to boost the performance of the proposed MuGraM. In the experiments we observe that MuGraM not only discovers patterns in multigraphs, but also outperforms the existing FSM approaches, when tested on simple graphs.

In Chapter 6, we performed a case study analysis on a remote sensing data, which is modelled as a multigraph. In this case study, we analysed the application of the proposed frequent pattern mining algorithm MuGraM and the subgraph pattern matching algorithm SuMGra to the remote sensing data. This case study analyses sheds some light on the utilitarian aspects of the contributions made in this thesis. Modelling the remote sensing data as multigraphs, and the application of the proposed approaches in this thesis on the multigraph data yielded useful knowledge that the experts approve of. Having made useful contributions and having put them in a nutshell, we now discuss on a few future directions, that this thesis makes conceivable in an optimistic manner.

Future Directions

Although in this thesis, contributions have been made in the field of subgraph querying and frequent pattern mining, many potential future works are possible. Even though we have discussed about the possible extensions of the corresponding work at the end of Part I and II, we now discuss on the future directions in a much broader sense.

In this thesis, we have considered static multigraphs for both querying and mining operations. With the ever increasing amount of data, it has become essential to perform these tasks on time evolving or dynamic multigraphs, since much of the real world data like, social network data, RDF data, and remote sensing data are constantly updated over time. We have learnt that multigraphs capture the real world data in a succinct manner; more so, dynamic multigraphs capture the real time data in a continuous manner, and hence the potential extension of the proposed works is necessary to address dynamic multigraphs.

Analysis and behaviour of time evolving graphs has been studied by [Leskovec et al., 2005]; these insights would be helpful in the process of knowledge extraction too. Further, several subgraph mining approaches already exist [Tong et al., 2008, Bogdanov et al., 2011], that are able to manage dynamic graphs. As we have already worked on multigraphs, we believe that it is relatively easier, though non trivial, to extend the current work for multigraphs; this can be achieved by modelling the time evolving graphs as multigraphs, where we can consider the time stamps as a set of multiedges. However, the challenge remains w.r.t. the applications; in some domains, say social network, the timestamps could be of longer intervals, thus having fewer multiedges, and in other domains, say a telecommunication network, a plenty of time stamps can be recorded within a short interval. Thus, a structured approach is needed in order to address several domains of time evolving graphs.

Another future extension could be in the direction of approximate subgraph matching. Since exact matching is too expensive to be computed, there are several domains as in bioinformatics [Tian et al., 2007], which are interested in enumerating the similar subgraph structures rather than discover only exact subgraph matches. Many works have been proposed to perform such similarity matches [Yan et al., 2005b], but in case of simple graphs. Thus, performing approximate subgraph querying for multigraphs in order to obtain similar subgraphs is a potential future work.

In the field of multigraph mining, several follow up works are possible. While dealing with Frequent Subgraph Mining (FSM), the amount of patterns generated poses a challenge. Although in this thesis, we have justified the approaches for mining frequent patterns in multigraphs, we are also convinced that several other mining approaches, described below, lead to potential future works.

Maximal and Closed frequent subgraph mining. Since the problem with the FSM approach is the huge number of patterns discovered, restricting the number of frequent patterns by evoking some constraints would make the discovered patterns more interesting. A *closed FSM* outputs a smaller set of frequent patterns [Yan and Han, 2003, Wang et al., 2006] and a *maximal FSM* outputs much fewer patterns [Thomas et al., 2010]. Maximal patterns are indeed discovered during an FSM procedure; however they are located deep down in the search space, at the end of the search path. Owing to a huge search space for multigraphs, it will be a challenge to find maximal patterns in multigraphs. For closed pattern mining, the research is already making progress towards multigraphs; the most recent work being addressed for edge labelled graphs [Karabadji et al., 2016].

Top-k frequent subgraph mining. Although FSM is one of the most widely explored mining approaches, it suffers from lack of scalability even for relatively larger input graphs, which is simply due to the inherent intractability of the FSM task as it is defined. Even in the proposed graph mining approach MuGraM, we have witnessed how the subgraph space grows exponentially with the size of input graph, despite all the pruning techniques. To overcome the exhaustive exploration of subgraph search space, many alternative paradigms have been proposed, which are not complete and hence approximate. One such approach is FS-3 [Saha and Al Hasan, 2015], which is a Fixed Size Subgraph Sampler, that proposes a sampling based method for mining top-k frequent subgraphs. Although this work is a recent contribution, it is meant for transactional graph databases. We can take inspiration from such works, and propose novel approaches to discover top-k frequent subgraphs for single large multigraphs.

Statistically significant pattern mining. In this approach, patterns with low p-values are discovered, that occur at low frequencies. For some applications, it is not enough just to find the frequent patterns as they may not provide the best characterization of the dataset. However, significant patterns have the potential to unearth properties where the data deviates from the expected behaviour. Works already exist in this direction; for transactional graph databases, GraphSig has been proposed by [Ranu and Singh, 2009] and for single graph setting, [Arora et al., 2014] and [Lee et al., 2016] propose approaches when vertex attributes exist. Since no attempt has been made to address mining statistically significant patterns in multigraphs, we behold it as an open problem. Multigraphs in particular would be a curious case, since we can observe if the patterns spanned by the subsets of multiedges can be statistically significant.

The field of *spectral graph theory*, where the properties of a graph are studied w.r.t. the eigenvalues and eigenvectors of the Laplacian matrix of the associated graph, would be interesting to explore for managing multigraphs. Further, concepts already exist for matrix methods in pattern mining approaches as introduced in [Eldén, 2007]. This would give an impetus for tensor modelling of multigraphs, thereby leading to another systematic study of multigraphs. In this direction, although few works already exists for clustering in multigraphs [Dong et al., 2012], a plenty of work has to be done. Thus, a systematic modelling of generic graphs for knowledge extraction can be accomplished in the near future.

Bibliography

Aggarwal, C. C. and Han, J. (2014). *Frequent pattern mining*. Springer.

Aggarwal, C. C., Wang, H., et al. (2010). *Managing and mining graph data*, volume 40. Springer.

Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. In *Acm sigmod record*, volume 22, pages 207–216. ACM.

Agrawal, R., Srikant, R., et al. (1994). Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499.

Aluç, G., Hartig, O., Özsu, M. T., and Daudjee, K. (2014). Diversified stress testing of RDF data management systems. In *ISWC*, pages 197–212.

Aluç, G., Özsu, M. T., and Daudjee, K. (2014). Workload matters: Why RDF databases need a new design. *PVLDB*, 7(10):837–840.

Arenas, M., Conca, S., and Pérez, J. (2012). Counting beyond a yottabyte, or how sparql 1.1 property paths will prevent adoption of the standard. In *Proceedings of the 21st international conference on World Wide Web*, pages 629–638. ACM.

Arora, A., Sachan, M., and Bhattacharya, A. (2014). Mining statistically significant connected subgraphs in vertex labeled graphs. In *Proceedings of the 2014 SIGMOD international conference on Management of data*, pages 1003–1014. ACM.

Babai, L., Dawar, A., Schweitzer, P., and Torán, J. (2016). The graph isomorphism problem (dagstuhl seminar 15511). *Dagstuhl Reports*, 5(12).

Baget, J.-F. (2005). Rdf entailment as a graph homomorphism. In *International Semantic Web Conference*, pages 82–96. Springer.

Bellinger, G., Castro, D., and Mills, A. (2004). Data, information, knowledge, and wisdom.

Berlingerio, M., Pinelli, F., and Calabrese, F. (2013). Abacus: frequent pattern mining-based community discovery in multidimensional networks. *Data Mining and Knowledge Discovery*, 27(3):294–320.

Bizer, C., Heath, T., and Berners-Lee, T. (2009). Linked data-the story so far. *Semantic Services, Interoperability and Web Applications: Emerging Concepts*, pages 205–227.

Boden, B., Günnemann, S., Hoffmann, H., and Seidl, T. (2012). Mining coherent subgraphs in multi-layer graphs with edge labels. In *KDD*, pages 1258–1266.

Bogdanov, P., Mongiovì, M., and Singh, A. K. (2011). Mining heavy subgraphs in time-evolving networks. In *2011 IEEE 11th International Conference on Data Mining*, pages 81–90. IEEE.

Bonchi, F., Gionis, A., Gullo, F., and Ukkonen, A. (2014). Distance oracles in edge-labeled graphs. In *EDBT*, pages 547–558.

Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D., and Ferro, A. (2013). A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics*, 14(Suppl 7):S13.

Bringmann, B. and Nijssen, S. (2008). What is frequent in a single graph? In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 858–863. Springer.

Broekstra, J., Kampman, A., and van Harmelen, F. (2002). Sesame: A generic architecture for storing and querying RDF and RDF schema. In *ISWC*, pages 54–68.

Bunke, H. (2000). Graph matching: Theoretical foundations, algorithms, and applications. In *Proc. Vision Interface*, volume 2000, pages 82–88.

Cabrio, E., Cojan, J., Aprosio, A. P., Magnini, B., Lavelli, A., and Gandon, F. (2012). Qakis: an open domain QA system based on relational patterns. In *ISWC*.

Cardillo, A., Gómez-Gardenes, J., Zanin, M., Romance, M., Papo, D., del Pozo, F., and Boccaletti, S. (2012). Emergence of network features from multiplexity. *arXiv preprint arXiv:1212.2153*.

Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. (2004). Jena: implementing the semantic web recommendations. In *WWW*, pages 74–83.

Chakrabarti, D. and Faloutsos, C. (2012). Graph mining: laws, tools, and case studies. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 7(1):1–207.

Cheng, J., Ke, Y., Ng, W., and Lu, A. (2007). Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, pages 857–872. ACM.

Cook, D. J. and Holder, L. B. (2006). *Mining graph data*. John Wiley & Sons.

Cordella, L. P., Foggia, P., Sansone, C., and Vento, M. (2004). A (sub) graph isomorphism algorithm for matching large graphs. *IEEE TPAMI*, 26(10):1367–1372.

Das, S., Srinivasan, J., Perry, M., Chong, E. I., and Banerjee, J. (2014). A tale of two graphs: Property graphs as rdf in oracle. In *EDBT*, pages 762–773.

Dong, X., Frossard, P., Vandergheynst, P., and Nefedov, N. (2012). Clustering with multi-layer graphs: A spectral perspective. *IEEE Transactions on Signal Processing*, 60(11):5820–5831.

Eldén, L. (2007). *Matrix methods in data mining and pattern recognition*, volume 4. SIAM.

Elseidy, M., Abdelhamid, E., Skiadopoulos, S., and Kalnis, P. (2014). Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment*, 7(7):517–528.

Erling, O. (2012). Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8.

Fan, W., Li, J., Ma, S., Wang, H., and Wu, Y. (2010). Graph homomorphism revisited for graph matching. *Proceedings of the VLDB Endowment*, 3(1-2):1161–1172.

Fayyad, U., Piatetsky-Shapiro, G., and Smyth, P. (1996). From data mining to knowledge discovery in databases. *AI magazine*, 17(3):37.

Fiedler, M. and Borgelt, C. (2007). Subgraph support in a single large graph. In *Seventh IEEE International Conference on Data Mining Workshops (ICDMW 2007)*, pages 399–404. IEEE.

Fortin, S. (1996). The graph isomorphism problem. Technical report, Technical Report 96-20, University of Alberta, Edomonton, Alberta, Canada.

Garey, M. R. and Johnson, D. S. (1979). A guide to the theory of np-completeness. *WH Freemann, New York.*

Godehardt, E. (2013). *Graphs as structural models: The application of graphs and multigraphs in cluster analysis.* Springer Science & Business Media.

Gonzalez, J., Jonyer, I., Holder, L. B., and Cook, D. J. (2000). Efficient mining of graph-based data. In *Proceedings of the AAAI Workshop on Learning Statistical Models from Relational Data*, pages 21–28.

Gubichev, A. and Neumann, T. (2014). Exploiting the query structure for efficient join ordering in sparql queries. In *EDBT*, pages 439–450.

Han, J., Pei, J., and Kamber, M. (2011). *Data mining: concepts and techniques.* Elsevier.

Han, J., Pei, J., Yin, Y., and Mao, R. (2004). Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data mining and knowledge discovery*, 8(1):53–87.

Han, W.-S., Lee, J., and Lee, J.-H. (2013). Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, pages 337–348. ACM.

He, H. and Singh, A. K. (2008). Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, pages 405–418. ACM.

Holder, L. B., Cook, D. J., Djoko, S., et al. (1994). Substucture discovery in the subdue system. In *KDD workshop*, pages 169–180.

Hopcroft, J. E. and Karp, R. M. (1973). An nˆ5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231.

Hsieh, S.-M., Hsu, C.-C., Ti, Y.-W., and Kuo, C.-J. (2014). Reducing the bottleneck of graph-based data mining by improving the efficiency of labeled graph isomorphism testing. *Data & Knowledge Engineering*, 91:17–33.

Huan, J., Wang, W., and Prins, J. (2003). Efficient mining of frequent subgraphs in the presence of isomorphism. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 549–552. IEEE.

Huan, J., Wang, W., Prins, J., and Yang, J. (2004). Spin: mining maximal frequent subgraphs from graph databases. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 581–586. ACM.

Huang, J., Abadi, D. J., and Ren, K. (2011). Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134.

Ingalalli, V., Ienco, D., and Poncelet, P. (2016). Sumgra: Querying multigraphs via efficient indexing. In *International Conference on Database and Expert Systems Applications*, pages 387–401. Springer.

Inokuchi, A., Washio, T., and Motoda, H. (2000). An apriori-based algorithm for mining frequent substructures from graph data. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 13–23. Springer.

Inokuchi, A., Washio, T., and Motoda, H. (2003). Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning*, 50(3):321–354.

Jiang, C., Coenen, F., and Zito, M. (2013). A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 28(01):75–105.

Kamel, M. and Quintana, Y. (1990). A graph based knowledge retrieval system. In *Systems, Man and Cybernetics, 1990. Conference Proceedings., IEEE International Conference on*, pages 269–275. IEEE.

Karabadji, N. E. I., Aridhi, S., and Seridi, H. (2016). A closed frequent subgraph mining algorithm in unique edge label graphs. In *Machine Learning and Data Mining in Pattern Recognition*, pages 43–57. Springer.

Kenji, A., Kawasoe, S., Sakamoto, H., Arimura, H., and Arikawa, S. (2004). Efficient substructure discovery from large semi-structured data. *IEICE TRANSACTIONS on Information and Systems*, 87(12):2754–2763.

Kim, J. and Lee, J.-G. (2015). Community detection in multi-layer graphs: A survey. *ACM SIGMOD Record*, 44(3):37–48.

Kim, S., Song, I., and Lee, Y. J. (2011). An edge-based framework for fast subgraph matching in a large graph. In *Database Systems for Advanced Applications*, pages 404–417. Springer.

Kudo, T., Maeda, E., and Matsumoto, Y. (2004). An application of boosting to graph classification. In *Advances in neural information processing systems*, pages 729–736.

Kumar, R., Raghavan, P., Rajagopalan, S., Sivakumar, D., Tompkins, A., and Upfal, E. (2000). The web as a graph. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–10. ACM.

Kuramochi, M. and Karypis, G. (2001). Frequent subgraph discovery. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 313–320. IEEE.

Kuramochi, M. and Karypis, G. (2004). Grew-a scalable frequent subgraph discovery algorithm. In *Data Mining, 2004. ICDM'04. Fourth IEEE International Conference on*, pages 439–442. IEEE.

Kuramochi, M. and Karypis, G. (2005). Finding frequent patterns in a large sparse graph. *Data mining and knowledge discovery*, 11(3):243–271.

Lee, J., Han, W.-S., Kasperovics, R., and Lee, J.-H. (2012). An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *PVLDB*, pages 133–144.

Lee, J., Park, K., and Prabhakar, S. (2016). Mining statistically significant attribute associations in attributed graphs. *arXiv preprint arXiv:1609.08266*.

Leskovec, J., Kleinberg, J., and Faloutsos, C. (2005). Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187. ACM.

Li, Y. and Li, J. (2012). Disease gene identification by random walk on multigraphs merging heterogeneous genomic and phenotype data. *BMC genomics*, 13(Suppl 7):S27.

Libkin, L., Reutter, J., and Vrgoč, D. (2013). Trial for rdf: adapting graph query languages for rdf data. In *PODS*, pages 201–212. ACM.

Lin, Z. and Bei, Y. (2014). Graph indexing for large networks: A neighborhood tree-based approach. *Knowledge-Based Systems*.

Martin, P. and Eklund, P. W. (2000). Knowledge retrieval and the world wide web. *IEEE Intelligent Systems*, 15(3):18–25.

McKay, B. D. et al. (1981). *Practical graph isomorphism*. Department of Computer Science, Vanderbilt University Tennessee, US.

McKay, B. D. and Piperno, A. (2014). Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112.

Morsey, M., Lehmann, J., Auer, S., and Ngomo, A. (2011). Dbpedia sparql benchmark performance assessment with real queries on real data. In *ISWC*, pages 454–469.

Neumann, T. and Weikum, G. (2010). x-rdf-3x: Fast querying, high update rates, and consistency for RDF databases. *PVLDB*, 3(1):256–263.

Raghavan, S. and Garcia-Molina, H. (2003). Representing web graphs. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 405–416. IEEE.

Ranu, S. and Singh, A. K. (2009). Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In *2009 IEEE 25th International Conference on Data Engineering*, pages 844–855. IEEE.

Rattigan, M. J., Maier, M., and Jensen, D. (2007). Graph clustering with network structure indices. In *Proceedings of the 24th international conference on Machine learning*, pages 783–790. ACM.

Ren, X. and Wang, J. (2015). Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *PVLDB*, 8(5):617–628.

Saha, T. K. and Al Hasan, M. (2015). Fs3: A sampling based method for top-k frequent subgraph mining. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 8(4):245–261.

Shang, H., Zhang, Y., Lin, X., and Yu, J. X. (2008). Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375.

Shasha, D., Wang, J. T., and Giugno, R. (2002). Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52. ACM.

Talukder, N. and Zaki, M. J. (2016). A distributed approach for graph mining in massive networks. *Data Mining and Knowledge Discovery*, pages 1–29.

Tang, L., Wang, X., and Liu, H. (2012). Community detection via heterogeneous interaction analysis. *Data Min. Knowl. Discov.*, 25:1–33.

Terrovitis, M., Passas, S., Vassiliadis, P., and Sellis, T. (2006). A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *CIKM*, pages 728–737. ACM.

Thomas, L. T., Valluri, S. R., and Karlapalem, K. (2010). Margin: Maximal frequent subgraph mining. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(3):10.

Tian, Y., Mceachin, R. C., Santos, C., Patel, J. M., et al. (2007). Saga: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(2):232–239.

Tong, H., Papadimitriou, S., Sun, J., Yu, P. S., and Faloutsos, C. (2008). Colibri: fast mining of large static and dynamic graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 686–694. ACM.

Ullmann, J. R. (1976). An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42.

Unbehauen, J., Hellmann, S., Auer, S., and Stadler, C. (2012). Knowledge extraction from structured sources. In *Search Computing*, pages 34–52. Springer.

Wang, J., Zeng, Z., and Zhou, L. (2006). Clan: An algorithm for mining closed cliques from large dense graph databases. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 73–73. IEEE.

Wang, X., Ding, X., Tung, A. K., Ying, S., and Jin, H. (2012). An efficient graph indexing method. In *2012 IEEE 28th International Conference on Data Engineering*, pages 210–221. IEEE.

Wang, Y., Ramon, J., and Fannes, T. (2013). An efficiently computable subgraph pattern support measure: counting independent observations. *Data Mining and Knowledge Discovery*, 27(3):444–477.

West, D. B. et al. (2001). *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River.

Wu, Y., Yang, S., and Yan, X. (2013). Ontology-based subgraph querying. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 697–708. IEEE.

Yan, X., Cheng, H., Han, J., and Yu, P. S. (2008). Mining significant graph patterns by leap search. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 433–444. ACM.

Yan, X. and Han, J. (2002). gspan: Graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 721–724. IEEE.

Yan, X. and Han, J. (2003). Closegraph: mining closed frequent graph patterns. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 286–295. ACM.

Yan, X., Yu, P. S., and Han, J. (2004). Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 335–346. ACM.

Yan, X., Yu, P. S., and Han, J. (2005a). Graph indexing based on discriminative frequent structure analysis. *ACM Transactions on Database Systems (TODS)*, 30(4):960–993.

Yan, X., Yu, P. S., and Han, J. (2005b). Substructure similarity search in graph databases. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 766–777. ACM.

Yang, J., Zhang, S., and Jin, W. (2011). Delta: indexing and querying multi-labeled graphs. In *CIKM*, pages 1765–1774. ACM.

Yao, Y. (2002). Information retrieval support systems. In *Fuzzy Systems, 2002. FUZZ-IEEE'02. Proceedings of the 2002 IEEE International Conference on*, volume 2, pages 1092–1097. IEEE.

Yao, Y., Zeng, Y., Zhong, N., and Huang, X. (2007). Knowledge retrieval (kr). In *Web Intelligence, IEEE/WIC/ACM International Conference on*, pages 729–735. IEEE.

Zaki, M. J. (2001). Spade: An efficient algorithm for mining frequent sequences. *Machine learning*, 42(1-2):31–60.

Zhang, A. (Cambridge, 2009). Protein interaction networks: Computational analysis.

Zhang, S., Li, S., and Yang, J. (2009). Gaddi: distance index based subgraph matching in biological networks. In *EDBT*, pages 192–203. ACM.

Zhao, P. and Han, J. (2010). On graph query optimization in large networks. *PVLDB*, 3(1-2):340–351.

Zhao, P., Yu, J. X., and Yu, P. S. (2007). Graph indexing: tree+ delta $\geq$ graph. In *PVLDB*, pages 938–949.

Zou, L., Huang, R., Wang, H., Yu, J. X., He, W., and Zhao, D. (2014a). Natural language question answering over RDF: a graph data driven approach. In *SIGMOD Conference*, pages 313–324.

Zou, L., Özsu, M. T., Chen, L., Shen, X., Huang, R., and Zhao, D. (2014b). gstore: a graph-based SPARQL query engine. *VLDB J.*, 23(4):565–590.