IMPROVING THE PERFORMANCE AND SCALABILITY OF PATTERN SUBGRAPH QUERIES

FOTEINI KATSAROU

SUBMITTED IN FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy

SCHOOL OF COMPUTING SCIENCE

COLLEGE OF SCIENCE AND ENGINEERING
UNIVERSITY OF GLASGOW

APRIL 2018

Abstract

Graphs have great representational power, and can thus efficiently represent complex structures, such as chemical compounds and social networks. A common problem that often arises to graphs is the subgraph pattern matching querying problem, where given a graph DB and a query in the form of a graph, the graphs from the DB that contain the query are returned. In some algorithms, all possible occurrences of the query graph in the DB graphs are additionally returned. The subgraph matching problem entails subgraph isomorphism which is known to be NP-Complete. To alleviate the problem, a large number of methods has been proposed over the years that can be classified in two major categories: (i) the filter-then-verify (FTV) and (ii) the subgraph isomorphism (SI) methods. Specifically, the FTV methods rely on a constructed index with the aim to filter out graphs from the DB that definitely do not contain the query graph as an answer. On the remaining set of graphs, which form the so-called candidate set, a subgraph isomorphism algorithm is applied to verify whether the query graph is indeed contained in the DB graph. SI methods target in optimizing their subgraph isomorphism testing process by suggesting different heuristics.

With our work, we confirm that both FTV and SI methods suffer from significant performance and scalability limitations, stemming from the NP-complete nature of the subgraph isomorphism problem. Instead of trying to devise new algorithms with better performance compared to the already existing ones, we take a different approach. We suggest a number of solutions to improve their performance and to extend their scalability limitations.

In more detail, we conduct a comprehensive analysis of the state of the art FTV methods. We initially identify a set of key-factor parameters that influence the performance of related methods, namely the number of nodes and density per graph, the number of distinct labels and graphs in the graph DB, and the size of the query. Subsequently, using the aforementioned parameters, we perform a large number of experiments with both real and synthetic datasets in a systematic way, where we report on indexing time and size, query processing time and filtering power. We analyze the sensitivity of the various FTV methods. Our analysis helps us draw useful conclusions about the algorithms relative performance. In parallel, we stress-test them and thus, we recognize different scalability limitations, i.e., points where some algorithms operate while others break.

One of the conclusions drawn from our experiments with the FTV methods is that as the graphs in the dataset grow large in the number of nodes and/or density and as the query size increases query processing becomes harder. Thus, we additionally bring into the play the state of the art SI methods and along with the top-performing FTV methods as indicated by our aforementioned analysis, we investigate whether all queries of the same size are equally challenging. First, our experiments reveal that all proposed methods suffer from stragglers, i.e., queries with execution times many orders of magnitude worse compared to the majority of them. Second, through our experiments we have seen that isomorphic queries can have widely and wildly different execution times on the various algorithms. Thus, we propose our own isomorphic query rewritings that can introduce large performance gains. Third, we observe that stragglers are algorithm specific, i.e., a straggler query on one algorithm can be a typical query on some other algorithm. We incorporate our findings in a novel proposed framework, coined Ψ-framework that runs in parallel different isomorphic instances of the original query and/or different algorithms. Such parallel executions of various algorithms have been used for other NP-hard problems and are known as portfolios of algorithms. Our framework introduces large performance gains in the subgraph matching problem, on both FTV and SI methods across all employed datasets, where some combinations of algorithms perform better than others. Similar to Ψ-framework, some portfolios are more favorable than others.

Recent proposed methods tend to totally dismiss FTV methods and employ SI methods instead, with the claim that the SI methods enjoy shorter query execution times and that managing the index-based FTV methods is too costly. With our work, we investigate this claim. We initially quantify the constructed index of state of the art SI methods and the top performing FTV method in terms of time and size and we evaluate the efficiency of the constructed indices in filtering out graphs that do not contain the query. Based on our experiments, in both real and synthetic datasets, SI methods fail to avoid a large number of redundant subgraph isomorphism tests. Additionally, our experiments on the SI methods fail to indicate a single-winner. Thus, we propose a hybrid FTV-SI method, as a combination of the filtering achieved by the top-performing FTV method and the verification of various SI methods. This hybrid FTV-SI combination was not studied before, perhaps surprisingly for the problem at hand. Based on our experiments, such a hybrid combination brings high speedups in the subgraph matching problem. In an attempt to reduce even more the underlying indexing costs, we additionally experiment with different values of the enumerated features. Our experiments reveal that we can still achieve high quality filtering, even with smaller features, whereas the overall query execution time is still significantly boosted.

With our research results, we hope to open up a whole new research trend where community will benefit from already existing solutions by combining them appropriately to achieve large performance gains.

Acknowledgements

At this point I would like to express my sincere gratitude to my supervisors Prof. Peter Triantafillou and Dr. Nikos Ntarmos for the opportunity they gave me to work with them and for their help and guidance through all these years. My PhD studies was a long journey, but with their support and patience, I was able to face numerous challenges. Their dedication to excellence as well as the effort they made and the time they sacrificed not only for me but for all the members of the team are priceless. Among others, through their guidance I succeeded in expanding my knowledge and improving my programming and presentation skills, which are invaluable assets to pursue any career in the future.

Additionally, I would like to thank my examiners, Prof. Phil Trinder and Prof. Alexandra Poulovassilis, during my PhD viva, for reading my thesis so carefully, for their precious advice for improvement and for the stimulating discussion we had during the viva. It was a pleasure meeting them.

I would also like to thank my office mates Dr. Jing Wang, Atoshum Cahsai, Fotis Savva and Wei Ma for their friendship and kindness and for their willingness and patience to discuss and analyze academic issues. Special thanks to Dr. Christos Anagnostopoulos and Dr. Yashar Moshfeghi for their guidance and invaluable advice through my PhD, and to Bessy Mousioni for proof-reading my PhD thesis.

During my PhD studies, I made good friends around the department, who gave me the necessary distractions from my research. Thus, I would like to thank Dr. Baharak Rastegari, Dr. Ornela Dardha, Dr. Rosanne English, Dr. Natalia Chechina, Dr. Oana Andrei, Fatma Amin Ibrahim, Gözel Shakeri, Frances Cooper, David Maxwell, Stuart Mackie, Oseghale Osezua Igene, Maria Evangelopoulou, Teresa Bonner, Gail Reat, Aileen Orr, Steven Kendrick, Helen McNee, Anastasia Fliatoura, Lydia Marshall, and John Hunter, and also my flatmates and good friends Georgios Sfakianakis and Susanne Oehler that made my stay in Glasgow memorable.

Finally, I would like to thank the people who with their support helped me reach my destination, and persuaded me to try this whole new experience. Thus, I would like to thank my professors from the Department of Electrical & Computer Engineering from University of Thessaly and especially Dr. Dimitrios Katsaros for all the knowledge I gained from them, my parents, Theodosios and Panagiota, who stand by my side in every choice I make, support and guide me, and Georgios Kazanidis, my partner in life, who not only shares with me beautiful moments and concerns and stands by also my side with a lot of patience and care.

“Τοῖς τολμῶσιν ἡ τύχη ξύμφορος”
Θουκυδίδης, 460-394 π.Χ.

“*Fortune is by the side of those who dare to try*”
Thucydides, 460-394 BC

To my parents:
Theodosios and Panagiota

Author’s Declaration

I declare that, except where explicit reference is made to the contribution of others, that this dissertation is the result of my own work and has not been submitted for any other degree at the University of Glasgow or any other institution.

Foteini Katsarou

Contents

1 Introduction **1**

1.1 Graphs and the Subgraph Pattern Querying Problem 1

1.1.1 Thesis Statement 3

1.2 Research Questions and Contributions 4

1.3 Thesis Outline 9

1.4 Publications 11

2 Related Work & Basic Definitions **12**

2.1 Graphs and Networks 12

2.1.1 Graph Data Models 13

2.2 Basic Definitions 13

2.3 Subgraph Matching problem 15

2.3.1 FTV methods 16

2.3.2 SI methods 19

2.4 Other types of queries 22

2.5 Graph Databases 23

2.6 Graph Generators and Graph Visualization 24

2.7 Branch and bound paradigm 25

3 Experimental Setup **27**

3.1 Competing algorithms 27

3.1.1 Competing FTV methods 28

3.1.2 Competing SI methods 30

3.2 Datasets 32

3.2.1 Graph Generation 33

3.2.2 Characteristics of Real and Synthetic Datasets 33

3.2.3 Query Workloads 35

3.3 Metrics 35

3.3.1 Time and Size metrics 36

3.3.2 Quantifying the Filtering Power 36

3.3.3 Speedup 37

3.3.4 WLA and QLA Performance Metrics 37

4 Performance and Scalability of Indexed Subgraph Query Processing Methods **39**

4.1 Introduction 39

4.2 Related Work and Contributions 41

4.3 The Experimental Framework 43

4.3.1 Competing Algorithms 43

4.3.2 Setup 43

4.3.3 Real and Synthetic Datasets 44

4.3.4 Query Workloads 46

4.4 Evaluation Results 46

4.4.1 Real Datasets 46

4.4.2 Synthetic datasets 46

4.5 Lessons Learned 60

4.5.1 Effect of key dataset/workload characteristics 60

4.5.2 Sancta Simplicitas 61

4.5.3 Choosing the right index method for user needs 62

4.5.4 Scalability limits 63

4.6 Conclusions 63

5 Subgraph Querying with Parallel Use of Query Rewritings and Alternative Algorithms 65

5.1 Introduction 66

5.2 Related Work and Contributions 66

5.3 Experimental Setup 67

5.3.1 Algorithms 67

5.3.2 Setup 68

5.3.3 Datasets 69

5.3.4 Query Workloads 69

5.3.5 Performance Metrics 70

5.4 Straggler Queries 71

5.5 Isomorphic queries 76

5.6 Graph query rewriting 80

5.7 Algorithm-specific Stragglers 87

5.8 The Ψ-framework 90

5.9 Conclusions 98

6 Hybrid Algorithms for Subgraph Pattern Queries in Graph Databases: An Evaluation 100

6.1 Introduction 101

6.2 Related Work and Contributions 102

6.3 Experimental Setup 104

6.3.1 Algorithms 104

6.3.2 Setup 104

6.3.3 Datasets 105

6.3.4 Query Workloads 105

6.4 Index construction 107

6.5 Filtering power 108

6.6 Performance of SI methods 110

6.7 Evaluating the hybrid FTV-SI method 112

6.7.1 Performance Metrics 112

6.7.2 Performance Results 113

6.8 Reducing filtering time with parallelism 115

6.9 Index Time/Size - Filtering Power Tradeoff 118

6.10 Conclusions 125

7 Conclusion and Future Steps 127

7.1 Summary of Contributions 127

7.2 Limitations and Future Work 132

Bibliography 134

List of Tables

3.1 Characteristics of 4 Real datasets and the Synthetic dataset for FTV methods 34

3.2 Dataset characteristics for SI methods 34

5.1 Results for SI methods on the yeast dataset (AET: Average exec time) . . . 73

5.2 Results for SI methods on the human dataset (AET: Average exec time) . . 73

5.3 $(max/min)_{QLA}$ statistics for FTV methods 77

5.4 $(max/min)_{QLA}$ statistics for SI methods 79

5.5 Percent reduction of straggler queries for FTV and SI methods using isomorphic query counterparts 84

5.6 $speedup^*{}_{QLA}$ statistics for FTV methods across rewritings 85

5.7 $speedup^*{}_{QLA}$ statistics for SI methods across rewritings 86

5.8 $speedup^*{}_{QLA}$ statistics when utilizing different algorithms on SI methods for yeast and human 87

5.9 $speedup^*{}_{QLA}$ statistics when utilizing different algorithms on SI methods for wordnet 89

5.10 Percentage of killed queries of FTV methods and different versions of our Ψ-framework. (Or stands for original query.) 92

5.11 Percentage of killed queries of SI methods and different versions of our Ψ-framework. (Or stands for original query.) 94

5.12 Percentage of killed queries of SI methods and on running multiple SI algorithms on Ψ-framework. (Or stands for original query.) 97

6.1 $speedup^*{}_{QLA}$ statistics for FTV-SI combination with 1 thread 114

6.2 $speedup^*{}_{QLA}$ statistics for FTV-SI combination with 4 threads 118

6.3 $speedup^*{}_{QLA}$ statistics for FTV-SI combination with 1 and 4 threads, $maxL = 2$ 123

6.4 $speedup^*{}_{QLA}$ statistics for FTV-SI combination with 1 and 4 threads, $maxL = 3$ 124

List of Figures

1.1 The subgraph querying problem. In the example, the graph DB consists of 4 graphs. The different colors on the nodes represent different labels. The query graph is found in G_1 (one occurrence) and G_4 (two occurrences). . . 2

1.2 Features of different sizes of a given graph. 7

2.1 Summary of the stages of FTV methods 18

2.2 Example of a hypothetical FTV method that enumerates exhaustively paths and organizes them in a trie index structure 19

2.3 Summary of the stages of SI methods 20

4.1 Indexing results over the real datasets 47

4.2 Query processing results over the real datasets 48

4.3 Indexing performance results for varying number of nodes 49

4.4 Query processing performance results for varying number of nodes 50

4.5 Indexing performance results for varying density values 52

4.6 Query processing performance results for varying density values 53

4.7 Query processing times for individual query graph sizes and varying density values 54

4.8 Indexing performance results for varying number of distinct labels 55

4.9 Query processing performance results for varying number of distinct labels 56

4.10 Indexing performance results for varying number of graphs in the dataset . 58

4.11 Query processing performance results for varying number of graphs in the dataset 59

5.1 WLA-Average exec time (s) in FTV methods 72

5.2 Percentages of *easy*, *2”-600”*, and *hard* queries in FTV methods 73

5.3 WLA-Average exec time (s) in SI methods 74

5.4 Percentages of *easy*, *2”-600”*, and *hard* queries in SI methods 75

5.5 Average $(max/min)_{QLA}$ for FTV methods 77

5.6 Average $(max/min)_{QLA}$ for different query sizes on FTV methods and for PPI dataset 78

5.7 Average $(max/min)_{QLA}$ for SI methods 78

5.8 Average $(max/min)_{QLA}$ for different query sizes on SI methods and for human dataset 79

5.9 Isomorphic queries generated with different rewritings (assuming the label frequencies in the stored graph are: “A”=20, “B”=15, “C”=10) 82

5.10 Results for individual query rewritings for FTV on PPI dataset 82

5.11 Results for individual query rewritings for SI methods on yeast dataset . . . 83

5.12 Average $speedup^*_{QLA}$ for FTV methods across rewritings 84

5.13 Average $speedup^*_{QLA}$ for different query sizes on FTV methods 85

5.14 Average $speedup^*_{QLA}$ for SI methods across rewritings 86

5.15 Average $speedup^*_{QLA}$ when utilizing different algorithms on SI methods . . 88

5.16 Average $speedup^*_{QLA}$ across different versions of our framework on the FTV methods 92

5.17 Average $speedup^*_{WLA}$ across different versions of our framework on the FTV methods 93

5.18 Comparison of average execution time over the PPI dataset, for Grapes/4 against the Ψ-framework with 4 rewritings (ILF, IND, DND, ILF+IND) over Grapes/1 94

5.19 Average $speedup^*_{QLA}$ across different versions of Ψ-framework on the SI methods 95

5.20 Average $speedup^*_{QLA}$ for running multiple algorithms against SI methods on Ψ-framework 96

5.21 Average $speedup^*_{WLA}$ for running multiple algorithms against SI methods on Ψ-framework 97

6.1 Indexing time and size of Grapes and SI methods 106

6.2 Indexing time and size of subsets of the Synthetic dataset 108

6.3 Pruning Power of Grapes and SI methods 109

6.4 Avg query exec time (ms) of SI methods 110

6.5 Avg query exec time (ms) of SI methods for PPI and Synthetic datasets and for different query sizes 111

6.6 Average $speedup^*_{QLA}$ & $speedup^*_{WLA}$ of the hybrid FTV-SI method . . . 113

6.7 Avg query exec time (ms) of hybrid FTV-SI methods for PPI and Synthetic datasets and for different query sizes 115

6.8 Avg query exec time (ms) of the FTV-SI hybrid methods 116

6.9 Example on parallel execution of the verification stage of the hybrid FTV-SI combination with number of threads $N = 2$. (We assume that graphs g_1, g_2, g_4, and g_8 formed the candidate set after the filtering stage. The red X is used to represent the removal of a grpah from the queue or the completion of a thread execution.) 116

6.10 Average $speedup^*_{QLA}$ & $speedup^*_{WLA}$ of the hybrid FTV-SI method, 4 threads 117

6.11 Tweaking the maxL parameter, index construction 119

6.12 Tweaking the maxL parameter, filtering power 120

6.13 Tweaking the maxL parameter, achieved $speedup^*$, 1 thread 121

6.14 Tweaking the maxL parameter, achieved $speedup^*$, 4 threads 122

Abbreviations

Avg exec time	Average execution time
FTV	Filter-then-Verify
SI	Subgraph Isomorphism
GGSX	GraphGrepSX
GR	Grapes
GQL	GraphQL
SP	sPath
QSI	QuickSI
TI	TurboIso
BTI	BoostIso over TurboIso
FPR	False Positive Ratio
QLA	Query-Level Average
WLA	Workload-Level Aggregation

Chapter 1

Introduction

This chapter contains an introduction to graphs, their efficacy to represent complex structures, the graph databases and an informal definition of the subgraph pattern querying problem, along with useful applications. It also contains the thesis statement along with the basic research questions and research contributions of this thesis. Subsequently, an outline of the chapters that follow is presented. Finally, we provide the list of publications that came out of this work.

1.1 Graphs and the Subgraph Pattern Querying Problem

Graphs have great representational power. They are ideal for representing complex entities and their relationships / interactions, such as social networks, chemical compounds and protein-protein interaction networks. Both nodes and edges that connect the nodes allow labels that characterize them, in such a way that repetitions of the labels are allowed. For example, in a chemical compound the node labels are the names of the molecules, whereas the edge labels could characterize the type of the chemical bonds. In a social network the node labels could be various properties such as the name, age, location, profession and the edge labels could be the existence of friendship, reactions to posts and others. Thus, the graphs that constitute the graph datasets, e.g., [1, 2, 3], can vary widely in numerous graph characteristics, such as the number of graphs in the dataset, the size of the graphs and the number of distinct labels. As a result, a large number of graph databases (graph DBs) has been developed to efficiently store, handle and process the ever increasing graph data, such as Neo4j[4] and OrientDB[5].

A common query pattern that arises in such a graph DB, which is essential to graph analytics, is finding the occurrence(s) of a pattern graph within the various graphs in the graph DB. Specifically, in *subgraph pattern matching* or *subgraph querying* (for short), given a graph DB and a pattern query graph, we want to locate which graphs in the DB contain the query (the decision problem) and/or find all its occurrences (the matching problem). Subgraph querying entails the subgraph isomorphism problem, which is known to be NP-complete [6]. Over the years, subgraph querying has received and continues to receive a lot of attention, as is evident by the numerous new methods that are added in the bibliography annually. Furthermore, four recent experimental and analysis papers ([7, 8, 9, 10]) compare and stress-test the proposed methods, thus providing interesting insights about the performance of the various solutions. Figure 1.1 presents a simple example of the subgraph querying problem. In the presented example, the graph DB consists of four graphs, and the different colors on the nodes represent different labels. Thus, the whole dataset consists of four distinct labels. In this example, the query graph (which is usually a much smaller graph compared to those stored in the DB) is found in G_1 (one occurrence) and G_4 (two occurrences), as highlighted with red color.

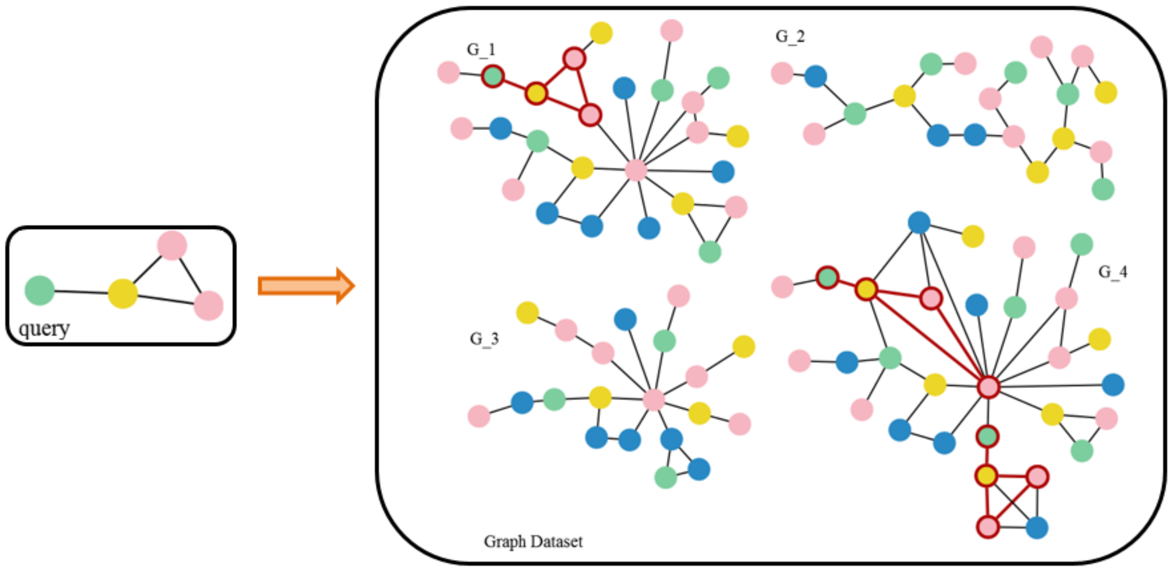


Figure 1.1: The subgraph querying problem. In the example, the graph DB consists of 4 graphs. The different colors on the nodes represent different labels. The query graph is found in G_1 (one occurrence) and G_4 (two occurrences).

The various proposed methods can be classified in two major categories: the *filter-then-verify (FTV)* and the *subgraph isomorphism (SI)* methods. Specifically, the FTV methods, that address the decision problem, mainly focus on filtering out graphs from the DB that definitely do not contain the query graph as an answer. Then, in the remaining set of graphs FTV methods employ a "standard" SI algorithm for performing the final verification, to confirm that the query graph is indeed located in those larger graphs. However, the SI methods, that usually address the matching version of the problem, mainly neglect indexing and filtering in order to focus on providing different subgraph isomorphism heuristics.

1.1.1 Thesis Statement

The subgraph matching problem entails the subgraph isomorphism which is known to be NP-Complete.. For the purpose of this thesis, we have conducted a large number of experiments with existing FTV and SI methods employing both real and synthetic datasets designed for the subgraph matching problem. However, both FTV and SI methods show significant limitations in their performance as the key parameters of the problem are increased, i.e., the number of nodes and / or density per graph, the number of stored graphs in the DB, and the size of the query. One solution is to try to devise new algorithms, with the aim to avoid these problems. Instead, in the current thesis, we take a completely different route. With our work we have come to the conclusion that these shortcomings were vanished when we applied various simple techniques such as reformulating the query to an equivalent one and/or combining existing algorithms to form hybrids or employing them in a framework. This gave rise to the following thesis statement.

State of the art methods for the subgraph querying problem, which include both the FTV and SI methods, do not scale when the graph DB grows large in terms of number of nodes or density per graph and/or in number of graphs in the DB. Additionally, their performance is seriously affected by increasing the query size. Instead of devising new algorithms with the aim of better performance, to extend the scalability of existing methods, one should consider rewriting the original query and/or combining the top-performing methods appropriately. By such simple techniques, one is able to achieve large performance gains.

The above thesis statement can be further analyzed to the following statements:

- Both existing FTV and SI methods have serious shortcomings stemming from the NP-Complete nature of the underlying subgraph isomorphism.
- Key parameters that influence the algorithms performance are: the number of nodes and density per graph, the number of graphs and the number of distinct labels in the dataset, and the size of the query.
- Proposed FTV and SI methods suffer from straggler queries, i.e., queries with execution time much larger compared to the majority of them.
- Isomorphic queries to the original query can have widely and wildly different execution times.

- Challenging queries are algorithm-specific.
- Thus, by executing in parallel isomorphic instances of the original query and/or different algorithms in the proposed Ψ-framework, we are to able to achieve large performance gains. Such parallel executions of algorithms are widely used for other NP-hard problems, as we will discuss in §2.7 and §5.8. As in the case of the Ψ-framework, some combinations of algorithms are more beneficial than others.
- FTV methods were designed to prune out graphs from the dataset that definitely do not contain the query graph as an answer and thus they possess high filtering power. However, their overall performance is diminished because of their underlying isomorphism algorithms, which can be replaced with newer SI methods to achieve large performance gains. Such a hybrid FTV-SI combination can prove to be highly beneficial.

1.2 Research Questions and Contributions

A number of different FTV and SI methods is presented annually, extending the relevant bibliography for the subgraph matching problem, with the purpose of surpassing the performance of older methods. But before totally dismissing older proposed methods, it is essential that we better analyze existing ones and study their performance, stress-test them, bring out their good qualities, and combine them (when appropriate) thus forming new better performing hybrids or execute them in parallel and achieve large performance gains. Such a work has not been conducted properly so far and this blind spot is investigated by the current thesis.

In the current work, we tried to answer a set of fundamental questions for the subgraph matching methods. The knowledge we have gained from our experiments in its turn generated other fundamental questions. Specifically, we know that both FTV and SI methods rely on a constructed index to facilitate the query processing. The index is formed by various features either maintained or encoded in a compressed format. Thus, our initial question was the following:

Question 1: What is the time and space required to construct the index of related subgraph matching methods and how effective is the constructed index in the query processing?

All proposed methods claim that they exhibit better performance results compared to preceding ones. However, existing comparative studies ([7, 8]) claim that this is not the case. With our experiments, we also investigated the aforementioned claim (§4.4 and §4.5 for the FTV methods, §6.6 for the SI methods).

Question 2: Does a performance analysis of both FTV and SI methods reveal a single-winner?

The quick answer to this question is that there was no algorithm that was the clear winner across the spectrum. As a matter of fact and especially in the case of the SI methods, there were cases that the best algorithm changed even for the same dataset and different query workload sizes (§6.6). Additionally, our intuition was confirmed; as we were increasing the parameters of the problem such as the number of nodes of the stored graph or the size of the query, query processing was becoming harder (§4.4 and §5.4). Thus, our findings triggered the following two questions:

Question 3: Are all queries of the same size equally challenging for a specific algorithm?

Question 4: In the case that a query is found to be challenging for one algorithm, should we conclude that the query is challenging for all algorithms or is this related to the algorithm's specificity?

The brief answer to these is that there is a small portion of queries whose execution time dominates the overall processing time (§5.4) and that different algorithms are challenged by different queries (§5.7). Triggered from all of our findings, our last question was the following:

Question 5: How to combine / exploit existing algorithms appropriately to achieve large performance gains?

Motivated from the above questions, we studied the details (both theoretical and their implementation) of existing methods, and we performed a large number of experiments. With the knowledge we gained, we were able to point out the real assets of existing work. We exploited them appropriately to achieve large performance gains in the subgraph matching problem on both FTV and SI methods (§5.8, §6.7.2, §6.8 and §6.9).

Overall, the current thesis makes the following key research contributions, that were so far lacking from the bibliography.

Contribution 1: Identification of a set of key factor-parameters that influence the performance of subgraph matching methods.

In chapter §4, and specifically in §4.3, we identify the number of nodes and density per graph, the number of distinct labels and graphs in the dataset and the size of the query that influence the performance of both FTV and SI methods either positively or negatively, as these factors influence the performance of the underlying subgraph isomorphism test. In §4.4, we use these parameters to perform experiments and analyze the sensitivity on existing FTV methods in a systematic manner, i.e., we maintain all parameters fixed except for one that we gradually increase and we study the effect of the said parameter on the performance of the various algorithms. Furthermore, we stress-test them and we study the scalability limitations of these algorithms by pinpointing points where some algorithms break whereas others still operate by efficiently constructing their index and by answering queries.

Contribution 2: Choosing the right algorithm for our application through the quantification of indexing time, index size, query processing time and filtering power of top-performing FTV and SI methods.

In §4.4 and §4.5, in order to identify top-performing methods, we have quantified the constructed index in time and size and we have studied its efficiency in answering queries both in time and in filtering power (when applicable). For this task, we have used both well-known real and numerous synthetic datasets. Such an analysis is essential as newly proposed methods utilize different metrics to compare with each other, and thus it is very difficult to decide which method performs best and based on which criteria. All in all, all proposed methods have their advantages and disadvantages and when we need to choose the right method to use, we need to consider optimizing different aspects of the problem and these are the indexing time, the index size, the query processing time and scalability. Although our experiments in §6.6 were inconclusive for pointing out the top-performing SI method, for the FTV methods our experiments revealed two winners with a common attribute, the simplicity. Specifically, features are the fundamental components for constructing the index; the term *feature* refers to a connected subgraph structure of the initial graph, and the *size of the enumerated features* refers to the size of the subgraph structure in number of edges. Figure 1.2 illustrates an example of features produced from a given graph. Related FTV methods employ various features for constructing their index; i.e., paths, trees, graphs, cycles or a combination of them, up to maximum size. Among these features, paths is the simplest form because of the underlying procedure for extracting them. Additionally, paths are also considered to be trees and graphs. Various methods employ different maximum sizes that in bibliography (§2.3.1 and §3.1) varies between 4 and 10 edges. Based on our experiments. Grapes[11] and GGSX[12], which employ the simplest form of features, the paths, are the clear winners from the set of the various FTV methods by performing the best in terms of indexing time, query processing time and scalability limitations. However, we also see that Grapes significantly outperforms GGSX in filtering power, especially when the stored graphs in the dataset increase in size, i.e., in number of nodes and/or density. Contributions 1 and 2 are additionally discussed in [9].

Contribution 3: Confirmation of the existence of straggler-queries and the role of isomorphic instances of the same query.

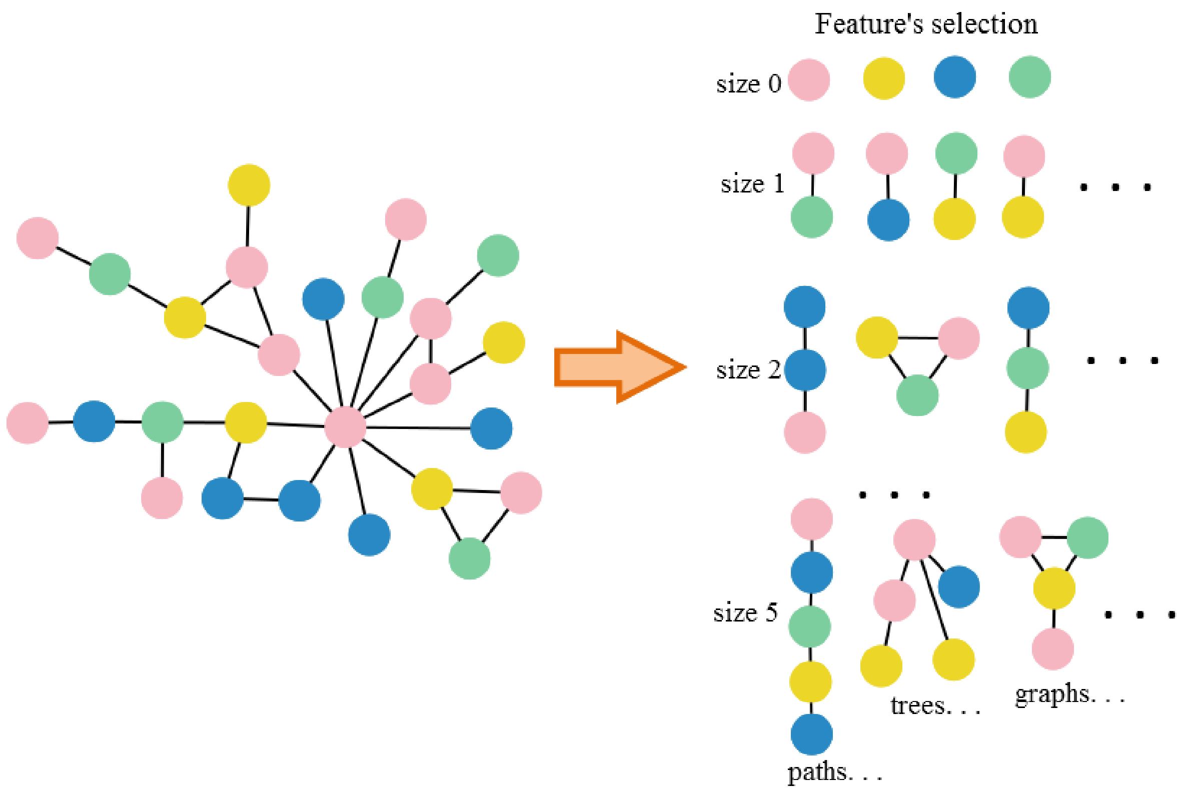


Figure 1.2: Features of different sizes of a given graph.

So far, related work, e.g. [7, 8] was limited in employing workload metrics; i.e., the average query execution time (calculated as the total time to execute all queries in the workload divided by the number of the queries in the workload) as a representative metric of the algorithms' performance (§5.4). In other cases, queries that were identified as outliers, i.e., too time-consuming to execute, were totally removed from the query workloads. With our experiments, we show that such assumptions are erroneous and improper. Specifically, we show that given a large stored graph and some query graphs, the query execution times for a specific algorithm (among queries of the same size) can vary widely, with the majority of the queries being very easy to execute; i.e., their execution time is <2". However, there is a small percentage of queries with execution times many orders of magnitude higher compared to the rest, which we call "straggler" queries. In such a query workload, different algorithms have different percentages of straggler queries. This finding holds for both FTV and SI methods (§5.4). Given the existence of straggler queries, averaging execution times over all queries in the workload can lead to a misinterpretation of the algorithms' performance; i.e.: the average execution time can be artificially inflated or the straggler queries can disappear because of the possible accumulated number of non-straggler queries.

In both the stored and the query graph, a unique number (ID) is used to identify the nodes; this numbering is used by some algorithms during query processing and can affect the order in which the nodes of the query are matched to the nodes of the stored graph. Given a query graph q, we can produce an isomorphic query q' by maintaining the structure and labels of the query the same (i.e., the nodes with their labels and edges among the nodes), and by interchanging the node IDs (definition 3 and figure 5.9). We call this process *query rewriting*. In §5.5 and §5.6 we see that if we rewrite the query to an isomorphic one, we might get completely different execution times. In other words, isomorphic queries can have widely and wildly different execution times. Thus, we propose and implement our own isomorphic query rewritings (on top of any other rewriting imposed internally by each algorithm), by permuting the node IDs in a specific manner, to achieve large performance gains (§5.6).

Contribution 4: Discovery of algorithm specific stragglers.

With our experiments in §5.4, we see that all proposed methods suffer from stragglers, which appear in different percentages depending on the dataset. We also know that the various SI methods employ different heuristics to perform the subgraph isomorphism test (§2.3.2). With our experiments in §5.7, we see that different algorithms are challenged by different queries. In other words, a straggler query on one algorithm is a typical query on some other algorithm.

Contribution 5: The Ψ-Framework.

Instead of totally dismissing prior work and trying to devise new straggler-free algorithms, with all the aforementioned findings, we show that related work already performs very well in the majority of queries, i.e.: although proposed algorithms suffer from straggler queries, a large number of queries is actually executed very fast, as discussed in §5.4. Thus, we need to exploit existing methods appropriately in order to achieve large performance gains in the subgraph matching problem. This is the task of our novel proposed Ψ-Framework (§5.8), which stands for Parallel Subgraph Isomorphism Framework. As the name suggests, we generate and execute in parallel different isomorphic instances of the same query and/or different algorithms by instantiating executions in different threads. After the completion of any first thread, the rest of them are killed. Although such an execution can have a large memory footprint, which depends on the number of the instantiated threads, it is highly beneficial across algorithms and datasets and leads to a performance improvement of many orders of magnitude compared to the original proposed methods. Contributions 3, 4 and 5 are also discussed in [10].

Contribution 6: Creation of hybrid FTV-SI methods for the matching problem.

The smart indexing of FTV methods is mainly used to prune out graphs that definitively do not contain the query as an answer and is thus useless in scenarios of a dataset consisting of a single large stored graph. Of course in such scenarios, SI methods surpass FTV methods without doubt [8]. However, recent works ([8]) tend to totally dismiss FTV methods even in datasets consisting of a large number of graphs, with the claim that the fast subgraph isomorphism test of the SI methods can significantly outperform the overall performance of the FTV methods. With our work, we investigate the above claim. To better test this, we study whether the benefits of the filtering of the FTV methods can be really offset by the fast SI methods in datasets consisting of a large number of graphs and under which circumstances. In chapter §4 and in [9], we identify Grapes as a top-performing FTV method in terms of indexing time, index size, scalability limitations and filtering power. Our experiments are inconclusive in identifying a single top-performing SI method in §6.6. Knowing that Grapes has a much stronger filtering compared to the filtering, if any, performed by the SI methods (§6.5), we set out to construct a hybrid FTV-SI method as a combination of the top performing FTV method (Grapes) and any SI method that would replace the underlying subgraph isomorphism test used in Grapes (§6.7 and §6.8). Such a hybrid combination is able to achieve large performance gains in the subgraph matching problem. This contribution is also discussed in [13].

Contribution 7: Identifying and extending scalability limitations.

With our comprehensive and systematic conducted experiments on the FTV methods, we show that all proposed methods suffer from scalability limitations as the dataset grows large in terms of number of nodes / density per graph or as the number of graphs in the dataset increases (§4.5). We identify cases where (i) the index creation required excessive amount of time or was not even possible due to excessive memory requirements and (ii) the queries could not be answered in reasonable time (§4.4). Similar conclusions hold for the SI methods (§5.4). However, we are able to extend these scalability limitations by applying a set of different techniques, presented in this thesis. Specifically, although Grapes is one of the top performing FTV methods, the size of the enumerated features can restrict its scalability. By tweaking this parameter with smaller values in §6.9, we can still achieve high filtering power but with a non-negligible reduced cost of index time and size. Also, our proposed hybrid FTV-SI method aims to solve scalability limitations in query processing. Finally, our Ψ-Framework, proposed in §5.8 is another solution to the scalability limitations problem in the identified cases where the index could be constructed efficiently but the queries were not answered in reasonable time.

1.3 Thesis Outline

The current thesis is organized as follows:

- Chapter 1 serves as the introductory material of this thesis, by presenting the efficacy of graphs in representing complex structures and the subgraph pattern matching problem. It presents the thesis statement and it identifies the fundamental research questions and contributions of this thesis. Finally, it provides the list of peer-reviewed publications that came out of this thesis.

- Chapter 2 provides the literature review of the most recent and well-known work conducted in the field of graph DB and specifically in the subgraph pattern querying problem, and emphasizes on the various FTV and SI methods that were introduced over the years. It also provides some useful definitions.

- Chapter 3 provides details for the experimental setup for the chapters that follow. Specifically, it provides details on the employed algorithms, the used datasets for the subsequent experiments (both real and synthetic ones). It also provides details on how we generate the queries and some useful metrics for evaluating the performance of the existing algorithms and the introduced solutions.

- Chapter 4 includes results from the comparison of various well-known and top-performing FTV methods. After a set of experiments with both real and synthetic datasets, where we vary parameters of interest, i.e. number of nodes, density, number of labels of graphs and number of graphs in the dataset, and the query size it also provides useful insights about their performance and scalability.

- Chapter 5 focuses on the subgraph isomorphism test of both FTV and SI methods. This chapter identifies the straggler queries, i.e. queries with execution times much greater than the majority of queries in the workload. Subsequently, it employs isomorphic instances or alternative algorithms to efficiently cope with straggler queries. Finally, it employs these findings in the novel proposed Ψ-framework to achieve large performance gains on both FTV and SI methods.

- Chapter 6 studies the indexing time and size and the achieved filtering of a top-performing and well-known FTV method (Grapes) with top-performing SI methods. Equipped with this knowledge, we combine them to form a hybrid FTV-SI method that is able to achieve large performance gains on a graph dataset that consists of many large graphs. Finally, to reduce the cost of index time/size, we tweak the size of the enumerated features and we study the related trade-offs.

- Chapter 7 concludes this thesis by presenting an overview of the contributions. Additionally, it sets the future steps through the questions arising from the already conducted work.

1.4 Publications

The majority of the content of this thesis has been peer-reviewed and published in academic conference proceedings as follows:

- Foteini Katsarou, Nikos Ntarmos, Peter Triantafillou, "Performance and Scalability of Indexed Subgraph Query Processing Methods", *Proceedings of the VLDB Endowment, (P/VLDB 2015)*, vol. 8, no. 12, pp. 1566-1577, August 2015.
- Foteini Katsarou, Nikos Ntarmos, Peter Triantafillou, "Subgraph Querying with Parallel Use of Query Rewritings and Alternative Algorithms", *20th International Conference on Extending Database Technology, (EDBT17)*, pp. 25-36, March, 2017.
- Foteini Katsarou, Nikos Ntarmos, Peter Triantafillou, "Towards Hybrid Methods for Graph Pattern Queries", *6th International Workshop on Querying Graph Structured Data (GraphQ 2017) co-located with EDBT2017*, March 2017.
- Foteini Katsarou, Nikos Ntarmos, Peter Triantafillou, "Hybrid Algorithms for Subgraph Pattern Queries in Graph Databases", *In Proc. IEEE International Conference on Big Data, (BigData17)*, December 2017.

Chapter 2

Related Work & Basic Definitions

In this chapter, we will introduce the background information and we will provide a literature review of the most relevant related work for this thesis. Specifically, we will first provide a formal definition of the problem along with other useful basic definitions. We will then define the graphs by giving representative examples of some real world, well-known graphs and networks and we will explore the expressive power of graphs, i.e., their efficiency in representing complex structures. Armed with this basic knowledge, we will set the context of the pattern subgraph querying problem, which is the main scope of this thesis. We will discuss extensively the various proposed FTV and SI methods and we will focus on their differences on addressing the subgraph queries. Subsequently, we will briefly discuss other types of queries addressed by the graph DBs. Additionally, we will provide an overview of the various well-known graph DBs. As visual representation of graphs is essential in graph analysis, we will name some well-known tools that are publicly available for graph visualization, along with some tools for graph generation that try to emulate the characteristics of real-world graphs.

2.1 Graphs and Networks

A *graph* is a collection of *vertices / nodes* and *edges*. Assuming that the nodes are different entities and the edges are the various relationships among them, then graphs are ideal for representing complex entities and their relationships / interactions.

Examples that can be modeled as graphs are abundant in both real life and in computer and communication systems. One of the most characteristic examples is the various online social networks [14], where the entities are the users of the social network and the relationships are the *friendship*, *follow*, *like* or other kinds of interactions among them. Similar to that, the World Wide Web can be modeled as a huge graph with the nodes being the various pages and the edges being the links among different pages. Accordingly, the set of research papers and their citations could model a similar network. In biology, we have biological networks such as chemical compounds[15, 1, 16, 2] and protein-protein interaction networks[17, 18, 19]. Even the various transportation networks can form large graphs. In all the aforementioned examples of graphs, the various components / entities interact with other components / entities and thus they formulate interconnected and heterogeneous networks of various scales [20].

2.1.1 Graph Data Models

Depending on the purposes of the application, several different graph data models have been developed and these include (among others): property graphs, hypegraphs and triples.

Property graphs contain nodes and relationships, where both nodes and relationships can contain properties in the form of key-value pairs. Relationships are always directed and named, but also nodes can be labeled with one or more labels.

Hypergraphs are especially popular when modeling a many-to-many relationship, where a special relationship, called hyper-edge, can connect any number of nodes. In other words, the hyper-edge allows as starting point multiple start vertices and as ending point multiple end nodes.

Triples are developed to model short statements in the form of subject - predicate - object. Triples originate from the semantic web and the idea behind them is to harvest useful relationship information from the Web.

2.2 Basic Definitions

The aforementioned algorithms can in theory support arbitrary graphs with labels on both vertices and edges; however, the available implementations of several of them can only handle undirected graphs with labels only on vertices (e.g. [11, 12, 8]). We thus focus on graph datasets with such graphs.

Definition 1 (Graph) *A graph $G = (V, E, L)$ is defined as the triplet consisting of the set $V = \{v_i\}, i = 1, ..., n$ of vertices of the graph, the set $E \subseteq \{(v, u) : v, u \in V\}$ of edges between vertices in the graph, and a function $L : V \to \mathcal{L}$ assigning a label $l \in \mathcal{L}$ ($\mathcal{L}$ being the set of all possible labels) to each vertex $v \in V$.*

We assume that each node in a graph is assigned an integer in the interval $[1, n]$, so that no two nodes in a graph have the same number; we call this the node ID.

In this work we consider undirected graphs. We assume that, in each graph, each vertex has a unique identifier. Note that, by the above definition, each node in a graph can have only one label, but any given label can be assigned to multiple nodes in a graph.

Definition 2 (Graph Database/Dataset) *A graph DB or graph dataset $D = \{G_1, G_2, \ldots, G_m\}$ is a collection of vertex-labeled graphs as defined in definition 1.*

Definition 3 (Graph Isomorphism) *Two graphs $G = (V, E, L)$ and $G' = (V', E', L')$ are isomorphic iff there exists a bijection $I : V \rightarrow V'$ that maps each vertex of G to a vertex of G', such that if $(u, v) \in E$ then $(I(u), I(v)) \in E'$, $L(u) = L'(I(u))$, $L(v) = L'(I(v))$, and vice versa. The* ***isomorphism class of*** *G is the collection of graphs that are isomorphic to graph G and to each other.*

Note that, given a graph G, a graph G' isomorphic to G can be trivially produced by permuting the node IDs in G.

Definition 4 (Canonical Label) *A canonical labeling, or canonical form, of a graph G is a unique string representation which characterizes the whole isomorphism class of G.*

Definition 5 (Non-Induced Subgraph Isomorphism) *A graph $G = (V, E, L)$ is non-induced subgraph isomorphic to a graph $G' = (V', E', L')$, denoted by $G \subseteq G'$, iff there exists an injective function $I : V \rightarrow V'$ such that if $(u, v) \in E$ then $(I(u), I(v)) \in E'$ and $L(u) = L'(I(u))$ and $L(v) = L'(I(v))$. Graph G is then called a subgraph of G' and G' is called a supergraph of G; equivalently, we say that G is contained in G'. Subgraph isomorphism is injective; thus there may exist edges in E' for which there are no corresponding edges in E.*

Definition 6 (Induced Subgraph Isomorphism) *A graph $G = (V, E, L)$ is induced subgraph isomorphic to a graph $G' = (V', E', L')$, iff there exists an injective function $I : V \rightarrow V'$ such that (i) if $(u, v) \in E$ then $(I(u), I(v)) \in E'$ for $L(u) = L'(I(u))$ and $L(v) = L'(I(v))$ and (ii) if $(u, v) \notin E$ then $(I(u), I(v)) \notin E'$ for $L(u) = L'(I(u))$ and $L(v) = L'(I(v))$.*

In other words, induced subgraph isomorphism is different than the non-induced subgraph isomorphism in that the absence of an edge in G also implies the absence of the corresponding edge in G', whereas in subgraph isomorphism these “extra” edges may be present. Checking for either induced or non-induced subgraph isomorphism is known to be NP-Complete [6]. Although the problem of induced subgraph isomorphism seems to be only slightly different from that of subgraph isomorphism, the "induced" restriction introduces enough changes that have major implications for the computational complexity. There are pairs of G and G', that when the subgraph isomorphism problem is applied, the problem is NP-Complete, whereas when the induced subgraph isomorphism is applied, the problem can be solved in polynomial time, while the opposite is not valid [21, 22]. Much like all of the FTV and SI methods discussed in §2.3, we focus on the non-induced subgraph isomorphism problem, or simply subgraph isomorphism, and omit any further discussion of the induced subraph isomorphism.

Definition 7 (Graph Density) *The density d of a graph $G = (V, E, L)$ is defined as the quotient of the division of the number $|E|$ of edges in the graph over the number of edges in a complete graph with the same number of vertices. In an undirected graph with $|V|$ vertices, the latter is equal to $\frac{|V|\times(|V|-1)}{2}$ edges, and thus:*

$$d = \frac{2 \times |E|}{|V| \times (|V| - 1)}, d \in [0, 1] \tag{2.1}$$

Definition 8 (Average Degree) *The degree of a node v in a graph $G = (V, E, L)$ is defined as the number of edges in the graph having v as an endpoint. The average degree avg_{deg} of graph G is then defined as the average of the degrees of all vertices in the graph. For undirected graphs:*

$$avg_{deg} = 2 \times \frac{|E|}{|V|} \tag{2.2}$$

Finally, we define the *subgraph matching problem*, that we will discuss later in chapters §4, §5 and §6.

Definition 9 (Subgraph Matching Problem) *Given a set of graphs $D = G_1, ..., G_n$, and a query graph q, the subgraph matching problem determines all graphs $G_i \in D$ such that $q \subseteq G_i$ and finds all the occurrences of q within each G_i.*

2.3 Subgraph Matching problem

Armed with the knowledge of examples of real graphs, it is essential to set the context of typical queries addressed to these graphs. Thus, in the *exact subgraph pattern querying* problem, given a pattern graph (query) and a graph DB, we want to locate which graphs in the DB contain this query. In some algorithms, all occurrences of the query graph in the DB graphs are additionally returned. Exact subgraph matching is one of the most fundamental operators in many applications that handle graphs (as discussed in [23]). Typical application fields, among others, are biomedicine and the protein-protein interaction networks [11, 24, 25], knowledge bases [26, 27], program analysis [28, 29], social network search and graph analytics applications [23, 30, 31]. Another manifestation of the importance of the problem is the large set of already proposed solutions and comparative studies, e.g.: [7, 8], which is annually enhanced by at least 2 new proposed methods. Other types of queries addressed in a graph DB will be discussed in §2.4.

The subgraph querying problem entails subgraph isomorphism which is known to be NP-complete [6]. Thus, over the years numerous methods have been proposed to alleviate the problem with related work being fragmented in two different categories that examine different versions of the subgraph querying problem: the *decision* and the *matching* version. In the decision version, given a DB of many (typically small) graphs and a query/pattern graph q, the method decides whether q is contained in any graph in the dataset and returns the IDs of those graphs. The decision version is typically addressed by the so called *filter-then-verify (FTV)* or *indexed subgraph query processing* methods and work in two stages. In the matching version, the method finds *all* embeddings of the query graph q in a typically large, stored graph g or in each graph of a graph DB. The matching version is usually addressed by *subgraph isomorphism (SI)* algorithms that employ different heuristics.

2.3.1 FTV methods

With a typical graph DB consisting of a large number of graphs and subgraph isomorphism being NP-complete, the philosophy of FTV methods relies on the attempt to reduce the number of graphs that undergo subgraph isomorphism. Specifically, all such algorithms construct an index with the attempt to reduce the set of graphs against which to test for containment and during query processing they operate in two stages: filtering (where they create a set of candidate matching graphs) and verification (of the query graph in the candidate set). Central to the whole procedure is the use of *features*, where the term “feature” refers to substructures of DB graphs used to produce the index, regardless of whether these are then stored in the index or not.

The wide design space is formed through the different design options of the numerous proposed FTV methods. In total, the design space is characterized through a classification of related works in 4 major categories: (i) type of indexed features: paths, trees, simple cycles, or graphs; (ii) approach for extracting said features from indexed graphs: i.e., exhaustive enumeration or frequent mining techniques; (iii) index data structure: hash table, tree, trie; and (iv) whether the index stores location information or not.

We will now present in detail the distinct stages of the various FTV methods:

Index Construction, which is a pre-processing step before the actual query processing. Specifically, The various FTV methods extract features from the DB's graphs and index them in an appropriate data structure. Depending on the algorithm, these features can be (a) simple paths[32, 12, 33, 11, 34], (b) trees[35, 36, 37], (c) graphs[38, 39, 40, 41, 42, 43], or (d) a combination of trees and graphs/cycles[44, 43]. Additionally, the features can be extracted from the graphs by either (i) exhaustively enumerating all such features across all graphs[12, 44, 39, 34], or (ii) mining the dataset graphs for frequent patterns[38, 36, 40, 41, 42, 37, 43, 45, 46]. LIndex[42] reuses the frequent feature extraction primitives of previous algorithms (e.g., [38, 35, 41, 43]), and is thus able to function with several feature types.

In the case of frequent feature mining algorithms a larger feature can be produced as the union of several smaller features, and thus, the number of graphs that contain the former is a subset of those that contain the latter. Thus, frequent mining techniques employ the *support ratio* metric of a feature which is defined as the percentage of graphs in the dataset containing it, where the feature is considered *frequent* if its support ratio is above some algorithm-specific threshold. Correspondingly, the *discriminative ratio* of a feature is a metric characterizing the pruning power of a feature compared to its sub-features [1]. Finally, in order to be able answer all possible queries, frequent mining techniques index all features of size 1.

In all cases, an upper limit is imposed on the size of the indexed features, where the size of a feature is defined as the number of edges comprising it. Features are identified by their canonical label; i.e., a unique string representation of each feature, computed on the labels of the vertices of the feature using an algorithm appropriate for the feature's structure (path, tree, etc.). In the end of the indexing phase, the stored index contains the features along with graph ID lists, i.e. a list of the IDs of graphs containing this specific feature. Additionally, some of the algorithms choose to further enhance their index with *location information*, such as the id of the first node in each path feature [11], or the id of the node at the center of a tree feature [37], whereas others [41] prefer not to do so for space reduction purposes. Last, all this is organized in algorithm-specific structures, such as hash tables, prefix trees, tries, or lattices.

During *query processing*, the stages of the various FTV methods are:

Filtering. In this stage, the query graph is first looked up in the index. If an exact match is found, the related graph IDs are returned. Otherwise, the query graph is broken up into features of the same form as those used to create the index. The query index is matched with the dataset's index, filtering out unmatched branches. Subsequently, in the majority of algorithms, an intersection of the sets of graphs containing each feature of the

[1]All frequent mining-based works mentioned above provide differing formulas for this metric; we thus do not provide a formula here but rather refer interested readers to the cited papers for more details.

query graph (i.e. an intersection of the returned graph ID lists) is performed, resulting in a set of graphs possibly containing the query graph, called the *candidate set* for the given query. Additionally, the algorithms that also store location information take advantage of this added knowledge at this stage for further filtering.

Verification. The above filtering stage may well produce false positives as graphs in the dataset may contain all (size-limited) features of a query graph but not the query graph itself. To this end, a final verification step is necessary, consisting of testing the query graph for subgraph isomorphism against only those graphs in its candidate set. The vast majority of FTV algorithms opt to employ the VF2 subgraph-isomorphism algorithm[47] mainly because of its public availability, with the exception of [44], [35] and [37] which employ algorithm-specific tests.

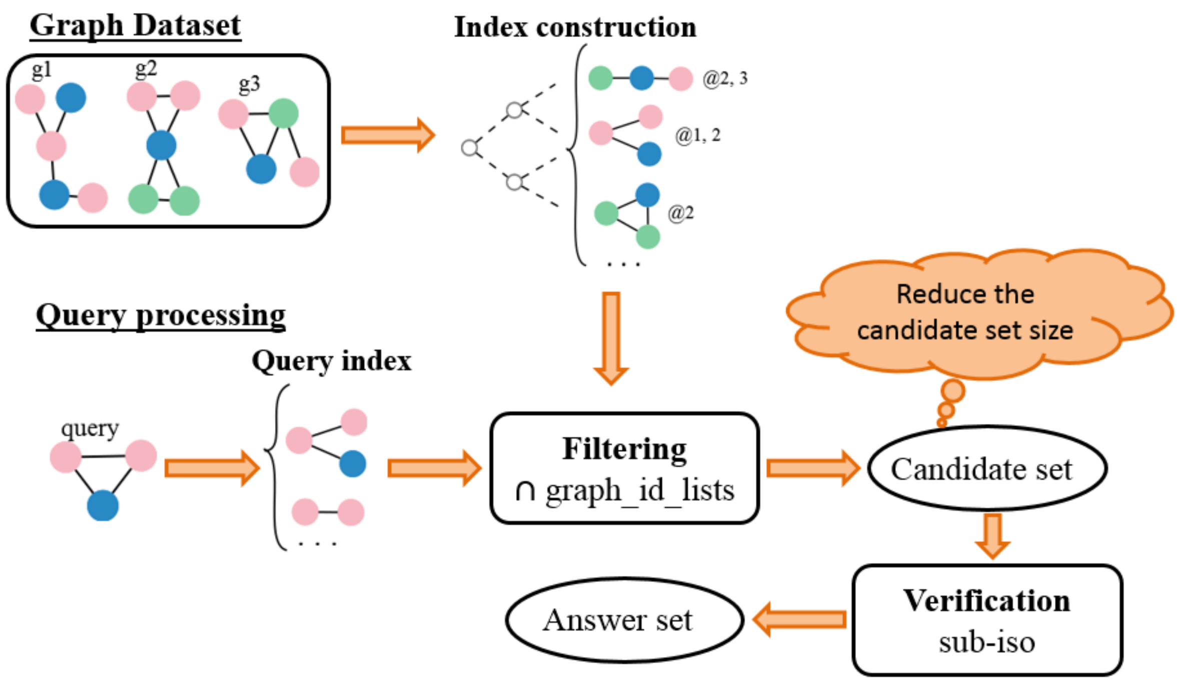


Figure 2.1: Summary of the stages of FTV methods

Figure 2.1 summarizes the stages of the various FTV methods. FTV methods are extensively discussed in [7, 42, 9, 13]. To better comprehend the stages of the various FTV methods, figure 2.2 presents a hypothetical FTV method that exhaustively enumerates paths up to maximum length $maxL = 2$ for a graph dataset that consists of graphs $\{g1, g2, g3\}$. The enumerated paths are organized in a trie and the graph-id lists are represented with the '@' sign followed by the ids of graphs. In query processing, the query index is matched with the dataset's index (highlighted with orange color in the figure), and the intersection of the graph-id lists reveal the candidate set; i.e. $\{g1, g2\}$. The final isomorphism test reveals $g1$ as an answer to the query.

The main target of all these methods is to prune the candidate set and thus to reduce the

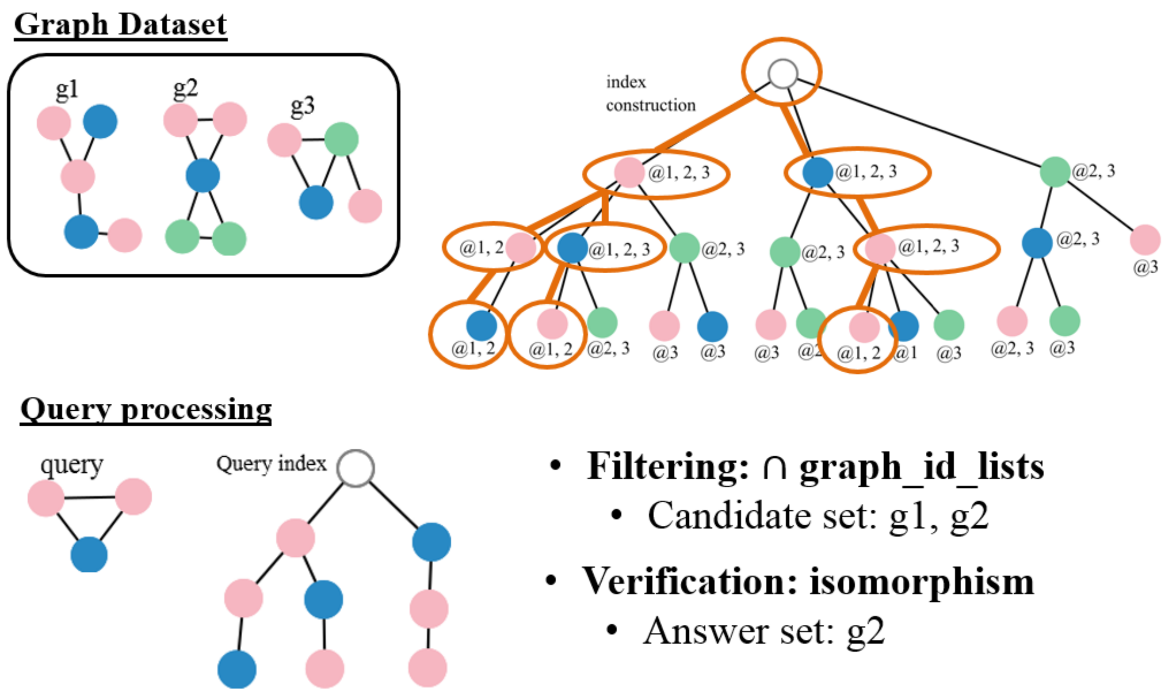


Figure 2.2: Example of a hypothetical FTV method that enumerates exhaustively paths and organizes them in a trie index structure

number of (expensive) subgraph-isomorphism tests performed. However, with their design options the various proposed methods try to optimize 4 different criteria: (i) the indexing time, (ii) the index size, (iii) the query processing time and (iv) the candidate set size. Their design options reflect on the scalability of these methods, i.e., the ability of constructing the index in reasonable time and size and answering queries in reasonable time. Our findings on the above criteria are extensively discussed in chapter §4 and specifically in §4.4 and §4.5. In brief, with our work we concluded that Grapes[11] and GGSX[12] are the best solutions in terms of index construction time, query processing time, and scalability limitations. We also showed that both Grapes and GGSX enjoy similar filtering power for datasets consisting of relatively small graphs. However, when the graph sizes increase, Grapes outperforms GGSX in filtering power.

2.3.2 SI methods

Some early SI methods include [48, 49, 50, 51], approaches [24, 47, 52] are widely used and later work include [53, 54]. The main focus of SI methods, is not to filter out graphs in the dataset that definitely do not contain the query as an answer, but for each DB graph (i) to locate the best candidate vertices to expedite the sub-iso test, and (ii) to decide the optimal join plan to follow; i.e., the sequence in which the query vertices will be matched to those of the stored graph. Thus, proposed SI methods, apart from the sub-iso heuristic algorithm, additionally comprise of a pre-processing/indexing step where they maintain a feature-based index consisting of: (i) vertices and edges [35, 49], (ii) shortest paths [29] or (iii) subgraphs [24, 52] up to a certain size. The algorithms additionally store vertex label lists along with additional information to facilitate the sub-iso test. During query processing, they apply different heuristics and define different join operations to match the query. Figure 2.3 summarizes the stages of the various SI methods.

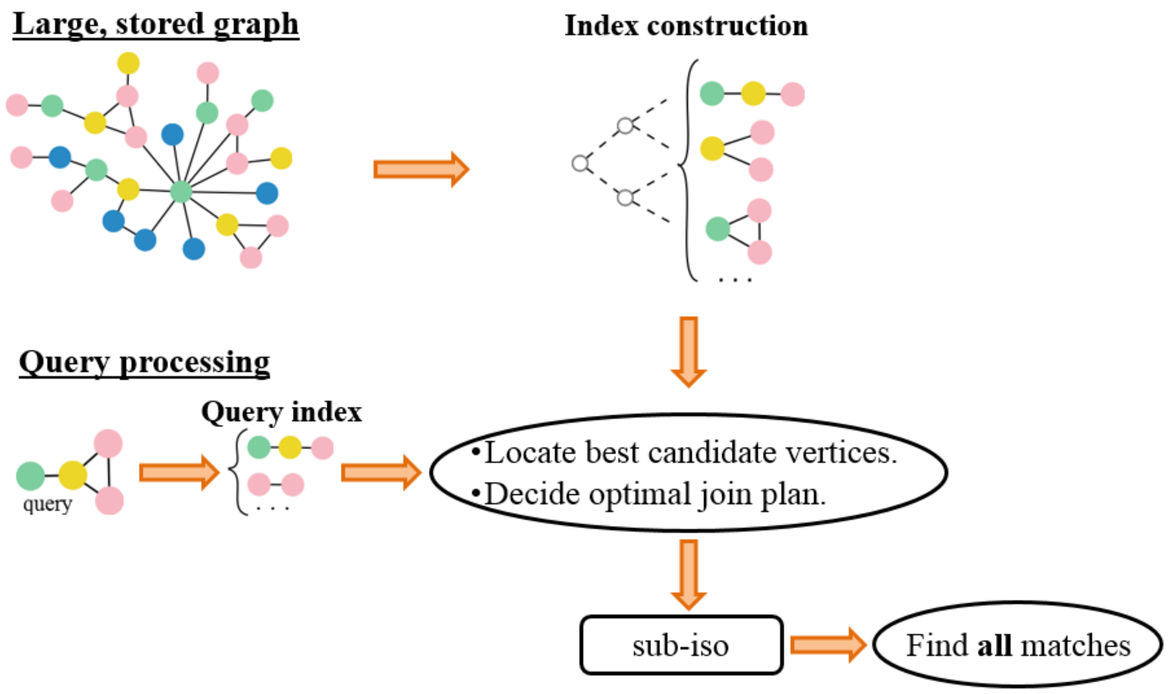


Figure 2.3: Summary of the stages of SI methods

A number of such methods were presented and compared in [8], concluding that (i) although there was no single algorithm to outperform all others on all occasions, GraphQL[24] was the only one that managed to complete all tested query workloads; (ii) GraphQL and sPath[29] showed very good performance; but also that (iii) all existing algorithms have weaknesses in the way they apply their join selection and pruning heuristics, leading to the need for new SI methods with improved performance. Following the publication of [8], several subgraph isomorphism tests were proposed. Specifically, TurboIso[55] rewrites the query by merging vertices that share the same label and neighborhoods. BoostIso[56] applies the aforementioned rewriting technique to the stored graph and dynamically reduces the duplicate computations. Thus, BoostIso claims that it can be applied on top of any SI algorithm and that it can accelerate all proposed subgraph isomorphism techniques. CFL-Match[57] applies decomposition of the query in dense subgraph and forest and unlike other methods, CFL-Match processes the dense subgraph first. Finally, Peng et al.[58] decompose the query in adjacent edge pairs or star-style patterns and propose an Edge Join algorithm for matching the query.

Our recent work[10], also discussed in chapter §5, provided key insights about the performance of both FTV and SI methods, and complements [8] with the inclusion of more recent SI algorithms. In brief, our experiments first showed that all existing SI algorithms suffer from straggler-queries; i.e., queries whose processing time is many orders of magnitude worse compared to the rest (§5.4). Second, that isomorphic queries, generated by simply permuting the node IDs, can have widely and wildly different execution times. The fact that all proposed methods do not define an absolutely strict order in which the nodes of the query will be matched, constitutes to this end (§5.5). Thus, straggler queries may have isomorphic instances which are not stragglers. Finally, we observed that stragglers are algorithm-specific, i.e. a straggler-query on one algorithm can be a typical query on some other algorithm (§5.7). These findings yielded the Ψ-framework (Ψ for Parallel Subgraph Isomorphism), which executes in parallel threads of different query rewritings and/or alternative algorithms to achieve large performance gains on both SI and FTV methods (§5.8).

There is nothing obstructing the SI methods being applied for the decision problem or the FTV methods for the matching problem. Both FTV and SI methods initially construct an index. FTV methods were originally proposed to work with datasets consisting of numerous, relatively small graphs, whose effectiveness relies on their achieved filtering, whereas SI methods employ the constructed index to primarily locate candidate vertices of the query in a large stored graph.

Finally, McCreesh [59], proposes additional heuristics to the subgraph matching problem. Unfortunately, [59] did not consider as many/large real datasets (yeast[8], human[8], wordnet[23]) or synthetic datasets (constructed using GraphGen[60] – see section §3.2.1), as we did. Additionally, the state of the art SI methods, namely GraphQL[24], sPath[29], QuickSI[35], TurboIso[55] and BoostIso[56] over TurboIso, are totally ignored and no comparison with them is considered.

Distributed SI methods

All the aforementioned FTV and SI methods are designed for subgraph matching query processing over graph DBs that can reside in the memory of a single commodity computer. In other words, they are not designed to tolerate any graph partitioning mechanisms or processing of the graphs in the DB in batches.

Thus, in addition to the above directly relevant research, recent research has expanded its scope in various directions. Below we refer to some interesting representative examples. Methods, such as sTwig[23], TwinTwig[30], SEED[31], and ParMa[61] deal with a single, very large graph, stored in a distributed infrastructure, and rely on parallel computing algorithms and infrastructures to perform the subgraph isomorphism testing.

Specifically, sTwig[23] relies on Trinity[62] for storing and partitioning the large graph. During query processing, the query is decomposed into smaller subqueries and sTwig utilizes mainly graph exploration, whereas the expensive joins are necessary only in the case of cycles existing in the query. TwinTwigJoin[30] is implemented over MapReduce[63]. Similar to sTwig, the query is decomposed in smaller subqueries, but the main join unit is a TwinTwig, i.e., either an edge or two incident edges of a node. The query is reconstructed following left-deep-joins of intermediate results of matching the TwinTwigs against the stored graph. SEED[31] is an improvement over TwinTwigJoin (by the same authors), where not only TwinTwigs are considered for the intermediate joins, but also stars and cliques. Additionally, the left-deep joins are replaced with a dynamic-programming algorithm and cliques are compressed to reduce the intermediate results. ParMa[61], unlike all other previous methods, tries to optimize not only the number of intermediate results produced during the join operations but also the number of iterations through the query decomposition. Therefore, the query is decomposed in such a way that overlaps are allowed. Finally, all aforementioned methods employ a mechanism in such a way to estimate intermediate produced results and thus they attempt to define the optimal order in which the join operations will be applied in the decomposed query.

2.4 Other types of queries

Apart from the subgraph pattern queries, other types of queries could be addressed in a graph DB. Therefore, there has been considerable work on the subjects of approximate graph pattern matching and of supergraph query processing. In the first case, related techniques (e.g, cTree[36], CT-Index[44], Tale[64], GD-Index[39], Grafil[65], GiS[66], SAPPER[28], SAGA[67], gSimJoin-minEdit[68], APPSUB[69], NeMa[70], GrafD-Index[71], etc.) do perform subgraph matching, but with support for wildcards and/or approximate matches. A characteristic use case is the 2017 Pulitzer Prize-winning Panama Papers investigation[72] initiated by the International Consortium of Investigative Journalists, which revealed highly connected networks of offshore tax structures, created in Neo4j[4]. In the second case, the related algorithms (e.g., LW-Index[73], cIndex[74], prefIndex[75], igQuery[76]) return those graphs in the dataset which are contained in the query graph (as opposed to containing the query graph; see [73] for an overview of related approaches). All these algorithms are not directly related to our work, as we focus on exact-match subgraph query processing.

Methods, like iGQ[77] and GraphCache[78], employ caching on top of any proposed FTV method to improve performance and study the architecture, system and algorithms for a graph cache for both subgraph and supergraph queries for FTV and SI methods, whereas GraphCache+[79] proposes different approaches to ensure consistency in GraphCache. Similarly, PatternTree$_{ISO}$[80] utilizes pattern correlations of preceding queries to expedite subgraph isomorphism for subsequent ones. Ren et al.[81] study the problem of multi-query optimization (MQO) on a system where multiple subgraph matching queries arrive simultaneously; first the system detects useful common subgraphs on the queries, then it proposes a sequence in which the queries should be executed and a caching mechanism to exploit intermediate results. Lin et al.[82] address the problem of generalized subgraph query processing. Finally, Semertzidis et al.[83] considered pattern queries over time-evolving graphs.

Significant work has been conducted in graph mining, which is essential in frequent mining FTV methods (as presented in §2.3.1). Proposed algorithms can be classified in the apriori approach (e.g. AGM[84], FGS[85]) or the pattern growth approach (e.g. gSpan[46], SPIN[86], GASTON[87]). For more information, [88] is a recent comparative study of various subgraph mining methods. Unlike, other mining methods, GraMi[89] is a system that allows frequent subgraph mining in a single large graph.

Finally, as we already mentioned in §2.3, canonical labels (or canonical forms), i.e., a unique string representation of a feature employing the labels of the vertices of the feature, are essential in the indexing process. The indexed features can be paths, trees or graphs. For paths, the generation of canonical labels is straight forward and is computed as the concatenation of labels of participating nodes in the path. The canonical label of a tree is calculated in linear time. Specifically, first we need to calculate the center of the tree by repeatedly removing the leaf nodes till 1 or 2 nodes are left. Then, as proposed in [90], the vertices of the rooted unordered tree are sorted / ordered level-by-level and bottom to up following a recursive procedure. Subsequently, the canonical label is produced by concatenating the labels of the vertices in a DFS traversal. Canonical labels for graphs is known to be NP-complete. Thus, efficient algorithms have been proposed that scale reasonably in moderate size graphs such as Nauty[48], Bliss[91] and Traces[92].

2.5 Graph Databases

There are various types of *databases (DBs)*, that could be used to store graph data with many different database management systems (*DBMS*) developed over the years to manage the DB; i.e., traditional *relational DBs* with PostegreSQL[93] being a characteristic DBMS, and *NoSQL DBs*, with the DBMS MongoDB[94]. However, both relational DBs and NoSQL DBs struggle to model the complex represented relationships [95]. Specifically, the "third normal form (3NF)" is crucial in a relational DB, where a series of transformations in tables is applied in order to avoid replication of data and thus redundancy, and to enforce logical consistency among the data of the table [96]. Thus, exploring relationships in a graph stored in a relational DB translates into joining different tables, where additional overheads are added by the foreign key constraints and the special checking of nullable columns [95]. Similarly, the NoSQL DBs are used to store sets of disconnected values/ columns/ documents, and in order to impose some order in the stored data, they can add aggregate identifiers inside these fields. Thus, when exploring relationships in a graph stored in a NoSQL DB, that translates into expensive join of aggregates which become too costly too soon.

On the contrary, a *graph DB* is more appropriate to efficiently store and process graphs and the queries addressed to the said graphs, by storing the data using graph structures, i.e., the entities are represented as vertices and the relationships/ interactions are represented as edges. Graph DBs are well-known for 2 basic characteristics: (i) the *index-free adjacency*, i.e., every element contains a pointer to its adjacent element and thus no index lookups are necessary, and (ii) *native graph processing*, the graph traversal is achieved by literally chasing pointers [97].

Over the years, several graph DBs have been proposed such as Neo4j[4] (which is open-source and well documented [98, 99, 100]), OrientDB[5], Titan[101], Trinity[62], FlockDB[102] and others. We omit any further analysis here, as it is beyond the scope of this thesis. Addtionally, for the same reason we just reference here Pregel[103], and its open-source counterpart Apache Giraph[104], Horton[105] and GraphLab[106] which are different graph processing systems that offer high scalability, whereas GraphChi[107] was designed to perform processing of large graphs over a single computer.

For the current thesis, and for simplicity reasons, we omit the use of the above systems for storing and processing the graphs. Instead, graphs reside in plain text files, which are loaded into memory at the beginning and before any processing. Graphs in the DB are also static, i.e. no modifications such as deletion/ additions of graphs/ nodes/ edges are considered.

2.6 Graph Generators and Graph Visualization

The Stanford Network Analysis Project [3] provides some real-world graphs, which among others represent social networks, communication networks, online reviews and communities. However, the number and/or size of graphs publicly available are not suitable for experimentation. Thus, over the years various graph generators have been proposed that capture properties of these real-world graphs, which allow various parameters of interest to be set such as the number of nodes and edges of the generated graphs. In other words, such a parametrization is suitable to generate graphs of different scale and size.

GraphGen[60] is a standard tool for constructing datasets suitable for graph mining techniques and subgraph queries, as it allows the parametrization of various parameters of interest; namely number of graphs, average number of nodes and density per graph, number of labels in the dataset, etc. As GraphGen is essential in our experiments (see chapters §4, §5 and §6), we include a more detailed description of how it generates a graph dataset in §3.2.1. R-MAT[108] follows the Erdös-Rényi model and is especially popular for generating power-law degree distributions graphs, such as the social network graphs or the World Wide Web, in a recursive manner by specifying a small number of parameters. However, neither Stanford Network Analysis Project nor R-MAT contain any labels on the nodes and/or the edges of the graph.

Visual representation of graphs is essential in graph analysis. GraphTool[109] is a very powerful tool that enables graph visualization, along with other capabilities such as statistical analysis of graphs. It is a Python[110] module with many algorithms implemented in C++, which benefits from the Boost Graph Library[111]. A similar tool for graph visualization, but with less algorithms ready implemented is SNAP[112]. Pajek[113] is one of the oldest graph visualization tools which is publicly available but not open-source and additionally offers some real graphs with various characteristics. Pegasus[114] is an open-source graph mining library which is able to perform statistical analysis of graphs in the size of Tera- or even Peta-bytes, and is implemented over Hadoop[115].

2.7 Branch and bound paradigm

Branch and bound refers to the exploration of the tree-shaped search space by subdividing it and branching recursively into each sub-space [116, 117]. The concept of branch and bound search is equivalent to the case where many paths from a root node to a goal exist. What we want to achieve is to locate the optimal path; i.e. the path with the smallest cost. Thus, we perform a depth-first traversal from the root node to the goal, by extending by one edge each time, and we maintain the path with the lowest cost discovered so far. While forming the path from root to goal, if we encounter a path with higher cost than the currently optimal one, this path can be safely pruned. Otherwise, the current best path and cost is updated. Branch and bound is applied in solving NP-hard optimization problems such as constraint satisfaction and mixed integer programming problems.

Proposed algorithms for these problems can show a high variance in their execution times. In their core, they employ different heuristics and are significantly affected by the search order they impose. Additionally, during the process of defining a search order, they may encounter ties, that when randomized can even further affect the algorithm's performance.

Given an algorithm, its execution time is a random variable and independent of the execution time of another algorithm for the same problem. Thus, in order to achieve large performance gains, the algorithms are combined in parallel searches and/or re-uses of the same algorithm, which is known as *portfolios of algorithms* [116]. It can be easily conceived that portfolio approaches can have a practical pay-off when combining methods with a high variance in their execution time and when there is not a single winning algorithm across the whole spectrum of problem instances. Thus, some portfolios of algorithms are favorable when compared to others. Additionally, the design of the optimal portfolio is quite sensitive to the distribution of underlying execution times of the various algorithms.

Chapter 3

Experimental Setup

In the current chapter, we will set the experimental framework for the chapters that follow. Specifically, we provide a detailed description of the competing algorithms of both FTV and SI research camps. We also provide the description and statistics of the employed datasets (both real and synthetic) for our experiments and the procedure we follow to generate our query workloads. We conclude this chapter with the presentation of the main performance metrics we employ for the evaluation of our experiments. We note that additional details for the experimental setup of each chapter will be introduced in the individual chapters.

3.1 Competing algorithms

For the experiments carried out for the purpose of this thesis, we used a set of well-established FTV and SI algorithms. To facilitate the reader, the competing algorithms' details are presented here.

For the FTV methods, our main focus was to cover as much of the design space as possible, and thus, we opted to perform an extensive comparison of six of the above algorithms. These algorithms were purposefully selected to represent different points in the design space, characterized by their use of: different feature types, different feature extraction approaches, different index data structures, and different filtering and verification processes. We chose to evaluate a set of representatives of the various regions in the design space, as opposed to providing an exhaustive (and unwieldy) examination of all available FTV algorithms. More specifically, we compare CT-index[44], GCode[34], gIndex[41], Grapes[11], GraphGrepSX[12], and Tree+Δ[43]. We justify our selection of algorithms and discuss their characteristics below.

For the experiments with the SI methods, we opted for methods (i) whose code is publicly available or made available to us by the authors upon request, so any conclusions would not be implementation dependent and (ii) that were well recognized as well performing. We selected GraphQL[24], sPath[29], QuickSI[35], TurboIso[55], and BoostIso[56] over TurboIso. With respect to CFL-Match[57]: we did not employ the algorithm as its authors did not respond to our request for their code.

3.1.1 Competing FTV methods

gIndex[41] uses a frequent mining approach, indexing graph-structured features. It uses both the features' support ratio and the discriminative ratio to decide whether a feature is frequent or not, and indexes these features in a prefix tree. gIndex uses no location information, and thus it only stores a graph ID list per feature. During query processing, gIndex enumerates all graph-structured fragments of the query graph up to a maximum fragment size, in a way which ensures that (a) smaller fragments are enumerated before larger ones, by starting with fragments of size one and expanding each fragment with one additional edge at a time, and (b) if a fragment does not appear in the index, no supergraphs of that fragment will be produced. Then, the candidate set of the query is computed as the intersection of the graph ID lists of the largest fragments along each expansion path. Finally, the verification is performed by comparing the query graph against all candidate graphs using the VF2 algorithm.

Tree+Δ[43] also uses a frequent mining approach, but initially indexes only tree-structured features of up to a predefined size. The feature information is stored in a hash table. Like gIndex, no location information is maintained. In the query processing phase, all tree-structured fragments of the query graphs are enumerated and looked up in the index; the candidate set is then computed as the intersection of the graph ID lists corresponding to these fragments, and a final verification step is performed using the VF2 algorithm. However, Tree+Δ takes an extra step: in addition to trees, the algorithm also enumerates simple cycles found in query graphs, which then extends by adjacent edges. Those cycle-based structures that are found to be discriminative enough (based on a predefined threshold on their discriminative ratio) are used to enhance the index structure and serve just like the initially indexed tree-structured features do for the subsequent queries.

gCode[34] takes a different route and chooses an exhaustive enumeration approach. First, it enumerates all paths of up to a predefined size. Given these paths, it then produces vertex *signatures*, consisting of three components. The first two components are a counter-string encoding of the labels of vertices in each path[1], and a counter-string encoding of the neighbors of each vertex in each path. The third component is computed as follows: (i) first, for each node in the database, the algorithm creates a "*level-N path tree*" rooted at said node and consisting of all length-N paths starting at that node; (ii) this tree is encoded in an adjacency matrix form; (iii) the eigenvalues of the matrix are computed and sorted by value; and (iv) the top-m such values (for some user-configurable m) are used as the third component of the vertex signature. For every graph in the database, all these vertex signatures are then combined to form the graph's *code*. All graph codes are finally stored in a balanced search tree. When a query comes in, first gCode follows the same process as above to construct a graph code for the query graph. This code is then compared against the codes of graphs in the database. This results in a first set of candidate graphs, which is then further pruned by comparing the individual vertex signatures of the query graph and candidate graphs. Finally, verification is performed by comparing the query graph against all graphs in the final candidate set using the VF2 algorithm.

CT-Index[44] also uses an exhaustive enumeration approach to build its index. It uses path-, tree- and cycle-structured features (of up to a user-configurable size). The canonical labels of all such features are then combined and hashed, producing a fixed-size bit array *fingerprint* for each graph in the database. Because of the hash-key compression applied, additional collisions are introduced in the case of high numbers of different features and small number of used bit array size. This algorithm, too, does not maintain any location information. During query processing, a similar fingerprint is created for the query graph and is then compared against the fingerprints of all graphs in the database via a bitwise-AND operation. This produces a candidate set which is then fed to a verification stage utilizing a modified VF2 algorithm with additional heuristics. The leading heuristic is that the frequent labels have bigger priority than the less frequent ones.

GraphGrepSX[12] (**GGSX** for short) enumerates all paths up to a maximum length using depth first search (DFS) and organizes them in a suffix tree. Each node of the suffix tree also stores the graph ID list along with the number of occurrences of the corresponding path feature in each graph it appears in. During query processing, maximal paths (of the same maximum length as above) of the query graph are extracted and organized in a query suffix tree, which is then compared against the index structure. Unmatched branches of the index are pruned away, and a further filtering is performed based on the frequencies of features in each graph. This process produces the candidate set, which then undergoes a verification

[1] gCode utilizes a fixed-length vector representation, of size n of each node label l, let $X_l = \{x_0^l, x_1^l, ..., x_{n-1}^l\}$, that is computed via hashing. Let also P the set of paths of maximum length L, starting from a node n of the graph, and $path_i \in P$. Then, a counter-string encoding C_n for a node n is calculated as $C_n = \left\{ \sum_{l \in path_i}^{P} x_0^l, \sum_{l \in path_i}^{P} x_1^l, ..., \sum_{l \in path_i}^{P} x_{n-1}^l \right\}$. In other words, C_n is the vector produced as the sum of all the X_l representations of all labels appearing in all $path_i \in P$ starting from node n. A similar definition applies to the counter-string encoding of the neighbors of each vertex in each path.

stage using VF2.

Finally, **Grapes**[11] (**GR** for short) also uses an exhaustive enumeration approach, indexing the simplest of features – i.e., paths of up to a maximum length, denoted $maxL$. However, in addition to the paths' canonical labels, Grapes also maintains location information in the form of the ID of the starting node of each path for each graph it appears in, as well as a counter denoting how many times each feature appears in each such graph. This information is then indexed using a trie. An added benefit of Grapes is that it is the only of the above algorithms which was designed specifically to support parallel execution of both its indexing and query processing chores. In the former case, this is accomplished via a smart assignment of graph nodes to threads so that each thread can produce a complete and disjoint part of the final trie, without needing to synchronize with the rest. In more depth, from the pool of all distinct labels, every initialized thread is assigned some labels to perform the indexing and every new path that is enumerated is added on the local tries created by the thread. When the indexing of a specific label is terminated, the next "free" label is assigned to that thread. In the end, all the local tries are concatenated very fast, as no merging of the inner nodes of the tries is required. During query processing, the query graph also undergoes the same (parallel) process of path enumeration and trie construction, with the sole exception that only maximal paths are considered. The query trie is then compared against the index trie and the non-matching parts are pruned away. The "surviving" parts of the trie are further reduced by taking into account the aforementioned location information, and are then translated into a set of connected components per database graph. This set then forms the candidate set of this algorithm. In the verification stage, the query graph is tested for subgraph isomorphism against each of these connected components in parallel, with each such component being assigned to a different thread.

3.1.2 Competing SI methods

The underlying isomorphism algorithm for the majority of FTV methods is **VF2**[47]. VF2 does not define any order in which query vertices are selected. Given a query graph q and a dataset graph g, the algorithm chooses a vertex from q to match to vertices in g, and proceeds by then trying to match still unmatched vertices adjacent to the matched ones in q. Given an unmatched vertex in q, the set of candidate vertices of g is defined as the set of all vertices in g with the same label as the unmatched vertex in q. VF2 then employs 3 pruning rules to reduce the number of candidate vertices. The first rule removes candidates that are not directly connected to the already matched vertices of g. The second rule removes all candidates for which the number of adjacent unmatched nodes which are also adjacent to matched nodes of g, is smaller than the corresponding figure for the matched vertex of q. The final rule removes all g candidates with less adjacent (matched/candidate) nodes than the corresponding figure in q.

In the indexing phase of **GraphQL**[24] (**GQL** for short), the labels of all vertices along with the neighborhood signatures, which capture the labels of neighboring nodes in a radius i in lexicographical order, are indexed. In the subgraph matching phase, the algorithm starts by retrieving all possible matches for each node in the pattern. Subsequently, 3 rules are applied in order to prune the search space. First, the indexed vertex labels and neighborhood signatures are used to remove infeasible matches. Then a pseudo subgraph isomorphism algorithm is applied to the problem iteratively up to level l; i.e., for every pair of possible graph-query vertex matches, the nodes adjacent to the query node should be matched to the corresponding neighbors of the graph. Finally, the algorithm needs to optimize the search order in the query before proceeding with the actual sub-iso test, which in turn consists of a number of *joins* of the candidate node lists. This optimization is based on an estimation of the result-set size of intermediate joins, and as it would be very expensive to enumerate all possible search orders, only left-deep query plans are considered.

sPath[29] (**SPA** for short), similarly to GraphQL, also maintains a neighborhood signature comprised of shortest paths organized in a compact indexing structure. Specifically, in order to reduce the storing space, shortest paths are not really maintained, but they are decomposed in a distance-wise structure. In the query processing, the query is initially decomposed in shortest paths that are then matched to the candidate shortest paths from the stored graph. From all possible candidate shortest paths, those that (i) can cover the query and (ii) provide good selectivity, i.e. minimize the estimated result-set size of each join operation, are selected as candidates. For each one of the selected paths, an edge-by-edge verification is then used to perform the sub-iso test.

In the sub-iso test of **QuickSI**[35] (**QSI** for short), priority is given to the vertices with infrequent labels and infrequent adjacent edge labels. In the indexing phase, QuickSI precomputes the frequencies of labels and edges and uses them to compute the “average inner support” of a vertex or an edge; i.e., the average number of possible mappings of the vertex or edge in the graph. The inner support is later used in the graph matching process to assign weights on the edges of the query graph and to construct a rooted minimum spanning tree (MST). In case of symmetries, edges are added in such a way that they will make the MST denser. The order in which vertices are inserted to the MST defines the order in which they are then matched in the sub-iso test.

TurboIso[55] (**TI** for short), utilizes 2 data structures as its index: (i) an inverse vertex label list that allows easy access to the vertices that share the same label, and (ii) a list of adjacent vertices for every vertex. TurboIso defines the Neighborhood Equivalent Class (or NEC for short) as the class of vertices that share the same structure, i.e. the same labels. In the query processing, for a given query a starting (root) vertex is chosen based on a ranking function that favors low label frequencies and high node degrees and the query is rewritten to the equivalent NECtree. Initiating from the root query vertex, TurboIso identifies candidate regions to the stored graph by performing a DFS search on the query's NECtree. As in all aforementioned methods, the matching order for the query vertices is chosen so that the intermediate candidate results are minimized. However, TurboIso defines a better matching order because of the more precise candidate regions estimation. Specifically, TurboIso exploits the paths of the NECtree from the starting vertex to every leaf node of the NECtree, and calculates the cardinalities of their candidate regions. Based on that, a matching order is defined in ascending order to the vertices of the NECtree.

BoostIso[56] (**BI** for short) can be applied on top of every proposed back-tracking algorithm and is based on the use of 4 types of relationships: (i) syntactic containment (SC), (ii) syntactic equivalence (SE), (iii) query-dependent containment (QDC), and (iv) query-dependent equivalence (QDE). Empirically, the SC is evident in the case that two nodes have the same label and the neighboring set of nodes on the second node is contained in the first node. In the SE, two nodes share the same label and the same neighboring set of nodes. SC and SE relationships are combined to form the Syntactic Equivalence Class (SEC), where vertices in the same SEC form a clique or are pairwise adjacent (they are either 1-step or 2-step reachable from each other respectively). QDC and QDE are similar conditions to SC and SE between the nodes of the query and the nodes of the stored graph. In brief, BoostIso employs the aforementioned relationships as follows: the SC and SE relationships are used to transform the stored graph to a new graph called the adapted hypergraph G_{sh} (sh stands for stored hypergraph), whereas QDC and QDE are only applied in query processing and are employed to further reduce duplicate computations. In more detail, as a pre-processing step, BoostIso transforms the stored graph to the adapted hypergraph, by utilizing the SEC. This process is called graph adaptation. Thus, the adapted hypergraph G_{sh} captures the structure of the original graph along with the SE and SC relationships between vertices. In the query processing, BoostIso searches for hyperembeddings of the query graph G_q in the adapted hypergraph G_{sh} which are then translated to embeddings. Duplicates can be further reduced using the QDC and QDE relations along with the SC and SE relations. For our experiments, we employ BoostIso over TI (**BTI** for short).

3.2 Datasets

For our experiments, we have employed both real world and synthetic datasets (generated with GraphGen). The DOI of data used for the current thesis can be found in `http://dx.doi.org/10.5525/gla.researchdata.588`. In the current subsection, we will present their characteristics.

3.2.1 Graph Generation

GraphGen[60] allows the parametrization of a large number of key graph parameters, such as the number of the graphs in the dataset, the average number of edges per graph, the number of unique node labels in the whole dataset and the average density of the graphs. We obtained the binary files of GraphGen from `http://www.cse.ust.hk/graphgen/` (authors of FG-Index[38]) where it was publicly available, and through reverse engineering we obtained the source code. GraphGen employs the following algorithm to generate a graph DB:

1. The user specifies the number of distinct labels, of distinct edges, and of graphs in the dataset, as well as the average graph density and graph size;
2. First, GraphGen produces an *alphabet* of distinct edges, in our case consisting of all possible pairs of distinct node labels;
3. Then, for every new graph, GraphGen:
   (a) Computes a random size (number of edges) and density, following a normal distribution around the aforementioned averages and a standard deviation of 5 and 0.01 respectively, and
   (b) Iteratively selects a (uniformly distributed) random edge from the *alphabet*, adding it to the current graph, until the requested size/density is reached or the system runs out of edges to use.

Concerning the alphabet, let $\mathcal{L}$ be the number of distinct node labels. Then the maximum number of distinct edge labels is: $\frac{\mathcal{L}\times(\mathcal{L}-1)}{2}+\mathcal{L}=\frac{\mathcal{L}\times(\mathcal{L}+1)}{2}$. To better justify this formula, let a graph dataset with 3 distinct node labels, i.e., "A", "B", "C". Then the possible edge labels are the combination in pairs of the above labels (in lexicographical order): "AA", "AB", "AC", "BB", "BC", "CC". Note that "BA" is the same as considering "AB" and is thus omitted. Finally, all graphs generated by GraphGen are connected.

3.2.2 Characteristics of Real and Synthetic Datasets

We have chosen datasets which (a) have also been used by other studies, so as to enable possible direct comparisons, and (b) have key characteristics covering a large part of the design space (e.g., regarding graph size and density).

We have performed experiments with datasets consisting of a large number of graphs, both real and synthetic ones and their characteristics are found in table 3.1 and datasets consisting of a single graph with their characteristics summarized in table 3.2.

Table 3.1 summarizes the characteristics of the 4 real datasets and a synthetic dataset that consist of many graphs. All these datasets differ across all characteristics of interest.

		AIDS	PDBS	PCM	PPI	Synthetic
Dataset	# graphs	40000	600	200	20	1000
	# disconnected graphs	3157	360	200	20	0
	# labels	62	10	21	46	20
Per Graph	Avg # nodes	45	2939	377	4942	1100
	StdDev # nodes	21.7	3215	186.7	2648	483
	Avg # edges	46.95	3064	4340	26667	12487
	Avg density	0.0475	0.0007	0.0612	0.0022	0.020
	Avg node degree	2.09	2.06	23.01	10.87	24.5
	Avg # labels	4.4	6.4	18.9	28.5	20

Table 3.1: Characteristics of 4 Real datasets and the Synthetic dataset for FTV methods

The AIDS antiviral dataset was used (as a whole or in subsets) by all the aforementioned FTV algorithms and it consists of 40000 small graphs, with only 45 nodes on average each. PDBS, PCM and PPI are 3 real datasets that were used in [11, 9]. PDBS and PCM represent chemical compounds comprising of 600 and 200 graphs respectively, whereas PPI represents 20 different protein-protein interaction networks. The majority of existing real datasets comprise of relatively small and sparse graphs.

In the lack of real datasets publicly available that preserve the required properties (i.e., many large graphs), for this thesis we employ an additional synthetic dataset of 1000 graphs generated with GraphGen[60], which is further described in table 3.1.

	yeast	human	wordnet
#nodes	3112	4674	82670
#edges	12519	86282	120399
Avg degree	8.04	36.91	2.912
StdDev degree #nodes	14.50	54.16	7.74
Density	0.00258	0.0079	0.000035
#labels	184	90	5
Avg frequency labels	127	240	16534
StdDev frequency labels	322.5	430	152

Table 3.2: Dataset characteristics for SI methods

The primary task of the SI methods is to find all occurrences of the pattern graph in a large stored graph, as mentioned in §2.3.2. Table 3.2 summarizes the characteristics of the three real datasets – namely yeast, human and wordnet — that we have used for the SI methods.

Yeast and human were previously used in [8] and we obtained them from the authors of [8], while wordnet was used in [23]. We obtained wordnet from `http://vlado.fmf.uni-lj.si/pub/networks/data/dic/Wordnet/Wordnet.htm`. Both yeast and human represent protein-protein interaction networks for budding yeast and for homo sapiens accordingly. Wordnet represents relations of the variable words. It has only 5 distinct labels which represent nouns, verbs, adjectives, adverbs or adjective satellites. Here, we need to note that unlike wordnet, yeast and human, allow multiple labels on each node. Finally, all three datasets vary widely in their size and the number of distinct labels.

3.2.3 Query Workloads

For the query formation, the following process is followed. Given the number of query graphs and their desired size (in terms of number of edges), a query workload is constructed as follows:

1. If the dataset consists of many graphs, we first select a graph uniformly and at random from the dataset. Otherwise, we proceed to the next step.
2. We maintain two sets: one for visited nodes and one for traversed edges that will formulate the query graph; both sets are initially empty.
3. We select a node uniformly at random from said graph and we add the said node to the set of visited nodes.
4. Starting from that node, we perform a random walk, by choosing one of the edges of that node. The sets of visited nodes and traversed edges are updated accordingly.
5. If the query graph size has not been reached, a new starting node is chosen from the set of visited nodes as long as possible edges exist from that node, i.e., not yet considered in the set of traversed edges and the previous step is repeated.
6. When the desired query graph size is reached, the procedure is terminated and the new query graph is returned as the union of visited nodes and traversed edges and in a compatible format for each algorithm.

Although the graphs in the dataset are not connected, the formulated query graphs are always connected. Additionally, because of the procedure followed to formulate the query, all queries are matched to at least one graph in the dataset. Finally, the formulated queries can be paths, trees (stars) or graphs (containing loops).

3.3 Metrics

This subsection presents the metrics used throughout this thesis and are essential for the evaluation of our experiments in the subsequent chapters along with other metrics that have been used in related work.

3.3.1 Time and Size metrics

Indexing time, index size and average query processing time are the three standard metrics that are employed by all relevant publications, as presented in §2.3 and are also reported in the current thesis, to study algorithms' relevant performance and to allow direct comparisons with previous work. Indexing time and size are reported in §4.4, §6.4 and §6.9. The average query execution time is calculated as the query workload execution time divided by the number of queries in the workload and is reported in §4.4, §5.4, §5.6, §5.8, §6.6 and §6.7.2. We are interested in the aforementioned index and query processing time metrics because we are looking at the subgraph pattern matching problem from the systems' perspective, and thus, it is important to measure the overall (clock) execution time. Additionally, we are interested in measuring the index size because all methods discussed in §2.3 and especially in §3.1 load all data in memory, and thus, in cases where the constructed index size is close to the machine's available memory, then the index size is a limiting factor for scalability.

Apart from the aforementioned metrics, some prior works report on the average number of recursive calls during query processing and the number of maintained features during indexing. The average number of recursive calls can be used to compare different query processing instances of the same algorithm, but it is not a good metric to compare across algorithms because any recursive algorithm could be transformed to an equivalent iterative one through the use of stack [118], without really giving any information about how expensive each step is. For this reason, we did not utilize the average number of recursive calls.

3.3.2 Quantifying the Filtering Power

To quantify the filtering power, we utilize 2 different metrics.

The first metric is the percentage of graphs that constitute the candidate set for each algorithm, before proceeding with the final subgraph isomorphism test. This metric is employed in §6.5 and §6.9. It is a standard metric used by prior work, e.g. [41, 7, 44].

The second metric is the false positive ratio, defined as:

$$FPR = \frac{1}{|\mathcal{Q}|}\sum_{q\in\mathcal{Q}}\frac{|C\{q\}| - |A\{q\}|}{|C\{q\}|} \tag{3.1}$$

where $|\cdot|$ denotes set cardinality, $\mathcal{Q}$ is the set of all queries in each query workload, and $C\{q\}$ and $A\{q\}$ are the candidate set and answer set respectively for query q, with $A\{q\} \subseteq C\{q\}$. Additionally, $FPR \in [0, 1]$. We note that $FPR = 0$ means that the candidate set is exactly the same as the answer set and $FPR = 1$ that although some/all of the graphs in the dataset belong in the candidate set, none of them is found to be an answer in the query, i.e. the answer set size is 0.

Similar metrics to the false positive ratio have been used by prior work to quantify the pruning power of the algorithms, such as: $\frac{1}{|\mathcal{Q}|}\sum_{q\in\mathcal{Q}}\frac{|C\{q\}|}{|D|}$ (in [44]) and $\frac{1}{|\mathcal{Q}|}\sum_{q\in\mathcal{Q}}\frac{|D|-|C\{q\}|}{|D|-|A\{q\}|}$ (in [34]), where $|D|$ denotes the number of graphs in the dataset. For our experiments, we employ the FPR metric for comparing results across algorithms. We note that we can still reach the same conclusions for the relative order of the algorithms with any of the these formulas. Thus, we used the FPR in §4.4, §6.5 and §6.9.

3.3.3 Speedup

For every query against a stored graph, we measure the *Execution Time*, denoted *exec time*, for both FTV and SI methods, while *avg exec time* denotes the average execution time.

In many cases, we need to compare the performance of two different methods M and M'. Let q_i be a given query. Let also t_i^M and $t_i^{M'}$ the exec time of q_i over method M and M' respectively. Thus, we employ the $speedup^*$ metric calculated as: $\frac{t_i^M}{t_i^{M'}}$. In other words, $speedup^*$ represents what we lose in performance if we choose method M over an alternative method M'. More details about the calculated $speedup^*$ will be introduced in the following chapters. Finally, the $speedup^*$ metric is used in §5.6, §5.7, §5.8, §6.7, §6.8 and §6.9.

3.3.4 WLA and QLA Performance Metrics

Fundamentally, FTV and SI solutions must confront the NP-Completeness of the underlying sub-iso testing. That is, all solutions are heuristics whose main achievement is that *in the majority of cases* performance is very good. However, in any workload of non-negligible size there will always be queries whose execution time may take a very large amount of time to complete (called straggler queries). And it has been discovered (see §5) that different queries may be stragglers for different methods. All related work so far utilize only workload-based metrics: they run a workload of a number of queries where they compute the average query execution time. Also, they (unavoidably) set an upper bound for the execution time of each query. If a query in the workload takes longer, it is simply discarded and not included in the results. Thus the workload-based metric in itself (unavoidably) is not entirely reliable. In addition, such metrics provide only one point of view: that of the system. However, to the average user, this is not particularly informative. To her the question is what is the best method for her query and by how much; or, put differently, what is the probability that a given method will perform best for her query. To address these concerns we will provide, additionally to workload-based metrics, query-based metrics which will reflect the (average) per-query performance of each method. This has the effect of treating all queries (and thus users) equally, despite the time taken by each query to execute. Such query-based metrics have unfortunately so far escaped all related work.

When comparing two sets of measurements $A = \{A_i\}$ and $B = \{B_i\}$, for $i = 1, ..., n$, we can compute their average (mean) ratio in two ways:

- *Workload-Level Aggregation (**WLA**)*, given by $\frac{\frac{\sum_{i=1}^{n} B_i}{n}}{\frac{\sum_{i=1}^{n} A_i}{n}} = \frac{\sum_{i=1}^{n} B_i}{\sum_{i=1}^{n} A_i}$. When A and B contain query response times, the WLA computation would give the improvement in the overall average execution time. This metric is important from the *system* perspective as it encapsulates the overall performance change.
- *Query-Level Aggregation (**QLA**)*, computed as $\frac{1}{n}\sum_{i=1}^{n}\frac{B_i}{A_i}$. When applied to query processing times, the QLA computation would give the average of per-query improvements. This metric is *user-centric* in the sense that each user cares what the performance improvement for his query is using different methods.

Let a set $X = \{X_i\}$, where $X_i = \frac{B_i}{A_i}$ for $i = 1, ..., n$. In other words, $speedup^*{}_{QLA}$ = $avg(X_i)$ (the average/mean over all items X_i in the set X). Thus, the QLA variant can also be carried over to other computations; for example, the standard deviation of the ratio of A and B would be computed as $stdDev(X_i)$ under QLA. However, unless stated otherwise, we shall use QLA to denote averages. Finally, we note that the QLA and WLA subscripts are widely used in chapters §5 and §6.

Chapter 4

Performance and Scalability of Indexed Subgraph Query Processing Methods

In the current chapter, we identify a set of key factors-parameters, that influence the performance of related FTV index-based methods: namely, the number of nodes per graph, the graph density, the number of distinct labels, the number of graphs in the dataset, and the query graph size. We then conduct comprehensive and systematic experiments that analyze the sensitivity of the various methods on the values of the key parameters. Our aim are twofold: first to derive conclusions about the algorithms' relative performance, and, second, to stress-test all algorithms, deriving insights as to their scalability, and highlight how both performance and scalability depend on the above factors. We choose six well-established indexing methods, namely Grapes, CT-Index, GGSX, gIndex, Tree+Δ, and gCode, as representative approaches of the overall design space, including the most recent and best performing methods. We report on their index construction time and index size, and on query processing performance in terms of time and false positive ratio. We employ both real and synthetic datasets. Specifically, four real datasets of different characteristics are used: AIDS, PDBS, PCM, and PPI. In addition, we generate a large number of synthetic graph datasets, empowering us to systematically study the algorithms' performance and scalability as they depend on the aforementioned key parameters.

4.1 Introduction

We have already seen that real graph datasets (Table 3.1) can vary wildly on several key characteristics, such as the number of graphs in the dataset, the number of nodes per graph, the average density of the graphs in the dataset, and the size of the set of distinct node labels and these are typical datasets handled by the graph data management systems. One of the main problems addressed by the current graph data management systems is the *decision subgraph query processing problem* (as discussed in §2.3), where given a graph dataset consisting of numerous graphs and a query graph, all graphs that contain the query are returned. A straight forward solution for processing such queries is to perform a subgraph isomorphism test against each graph in the dataset, which is known to be NP-Complete and thus could be computationally expensive. With the claim that expensive subgraph isomorphism tests can be avoided for those graphs that definitely do not contain the query, a research trend has been formulated; many index-based methods or filter-then-verify (FTV) methods have been proposed to reduce the number of candidate graphs that have to underpass the subgraph isomorphism test. As a reminder, in brief, FTV solutions utilize an index based on features (i.e., substructures) of the graphs to filter out some of those that definitely do not contain the query q; however, the graphs remaining after the filtering – called the *candidate set* – may not actually contain q (i.e., the filtering process can produce false positives). Due to this, a verification stage is required, during which q is tested for subgraph isomorphism against all of these remaining graphs. The main premise of these algorithms is that the candidate set is usually much smaller in size than the complete dataset, and the number of redundant isomorphism tests is significantly reduced.

Given the importance of the problem and the large attention it has received in the research community, with this chapter we provide a systematic and comprehensive evaluation of the performance and scalability of a representative set of related methods, which includes the most recent and most competitive approaches and also approaches representing different key decisions in the design space. More specifically, the contributions of this chapter are the following:

1. *Algorithm Performance*. A comprehensive study of the performance of related methods. A set of methods is selected to represent different key design decisions with respect to the type of graph features indexed (i.e., paths, trees, cycles, subgraphs) and the method selected for generating graph features (i.e., based on frequency mining or exhaustive enumeration of graph features). Further, this set of methods includes the most recent and higher-performing methods (such as GGSX, Grapes, and CT-Index). Our performance metrics include query indexing time and space, as well as query processing time and false positives.
2. *Algorithm Scalability*. Our experiments also aim to stress-test the above methods, deriving conclusions with respect to the methods' scalability. The existing work focuses only on performance issues – i.e., time and space comparison of the algorithms – and fails to look at scalability issues – i.e., what the performance of the algorithm is when both the dataset and the graphs grow large and/or more complex.

3. *A systematic evaluation of performance and scalability*. We employ both real and synthetic graph datasets. Specifically, four real datasets of different characteristics are used: AIDS[1], PDBS[16], PCM[15], and PPI[18, 19]. Furthermore, a very large number of synthetic datasets is generated which facilitates a systematic study on the dependence of the algorithms' performance and scalability on the key problem parameters (e.g., number of nodes, graph density, number of distinct labels, number of graphs, and query graph size).

Such a scalability and performance showdown is currently very much lacking. Most related works are tested against the AIDS antiviral dataset and synthetic datasets, formed of many small graphs. These sets are not adequate to provide definitive conclusions on how an algorithm is influenced by the characteristics of the graphs. Of these works, Grapes[11] alone used several real datasets; however, the authors did not evaluate scalability. Also, their performance evaluation did not include a systematic exploration of the effect of the key problem parameters. The iGraph comparison framework [7], which implements several such techniques, compared the performance of older algorithms (up to 2010). Since then, several, more efficient algorithms have been proposed (e.g. GGSX[12], Grapes[11], CT-Index[44]). Finally, virtually all of these works report on different metrics; thus, no concrete conclusion can be reached regarding their relative scalability.

In order to address all above problems, we conduct a systematic and comprehensive evaluation on existing implementations and we report the results. Specifically, we use several real datasets (AIDS, PDBS, PCM, PPI [11]), as well as several synthetic datasets varying in all the factors of interest (created using GraphGen[60]). Because it is very difficult to exhaustively run experiments for variating simultaneously all factors of interest (number of nodes, number of graphs, density, distinct labels), we choose to fix all the parameters apart from one for every set of experiments and decide about how influential a specific factor is for every algorithm.

4.2 Related Work and Contributions

As mentioned previously, one approach to process a subgraph query is to check the query graph for subgraph isomorphism against each graph in the dataset. As subgraph isomorphism is an NP-complete problem, and graph datasets may contain a large number of graphs, this procedure can get too time consuming. To this end, many indexing methods, also known as FTV methods, have been proposed over the years, attempting to reduce the set of graphs against which to test for containment. The numerous proposed FTV methods were extensively discussed in §2.3. For completeness, we additionally reference here earlier works that also considered larger graphs and/or a comparison with related FTV methods. Thus, we note that Lindex[42] provides an extensive discussion of related methods (although some of the methods considered in our work were not mentioned in Lindex). Additionally, Lindex reuses the frequent feature extraction primitives of previous algorithms (e.g., [38, 35, 41, 43]), and is thus able to function with several feature types. CP-index [40] was designed to solve the frequent mining problem for larger graphs but not necessarily for bigger datasets (in number of graphs).

All aforementioned algorithms reported performance results against datasets consisting of a large number of small graphs (e.g., the AIDS antiviral dataset containing 40,000 graphs each consisting of 45 nodes and 47 edges on average (Table 3.1), the PubChem dataset containing 1 million graphs each consisting of 24 nodes and 26 edges on average ([7]), etc.), often providing no proper insight on the performance and scaling of the algorithms against considerably larger and more complex datasets and query workloads. Of these works, Grapes[11] was the only one to provide results against datasets with larger graphs (PDBS, PCM, PPI, Table 3.1), but the number of graphs in these datasets was quite small (PDBS contains 600 graphs, PCM contains 200 graphs, and PPI only 20 graphs). Moreover, virtually every such work reported on a different set of metrics, thus making comparisons between the various algorithms a very hard task.

iGraph[7] provided a comprehensive comparison of indexing methods for subgraph query processing. Our work replicates some of the experiments in [7]. However, our work complements and surpasses iGraph in several ways. First, we considered several indexing algorithms that were published after [7] and were proven to be significantly superior to those tested by the latter (often by orders of magnitude). Second, in iGraph, scalability was not addressed; all the previous papers, and specifically iGraph, focus on performance evaluation against small graphs (especially in terms of number of nodes), while their "large datasets" are typically only large in terms of the number of graphs in the dataset and not with respect to the size/complexity of the graphs. Third, we performed a systematic study on performance and scalability based on 5 characteristics of a graph dataset/workload: the number of nodes, the density of the graphs, the number of distinct labels, the number of graphs in the dataset, and the query size.

Finally, sTwig[23] has looked at subgraph query processing for datasets consisting of a single very large (billion-node) graph, and the same research paradigm was later followed by TwinTwig[30], and SEED[31]. sTwig takes a totally different approach, by not building an index at all and instead utilizing a memory cloud and massively parallel computing primitives. Its authors motivate their approach by claiming that index-based solutions do not scale, and by providing theoretical arguments based on the asymptotic complexity of the latter (but no experimental evaluation of their approach against index-based techniques). Our work complements and substantiates this claim, by providing hard numbers and a systematic examination of the breaking points of each index algorithm type, across a large number of datasets of varying characteristics.

4.3 The Experimental Framework

4.3.1 Competing Algorithms

For our study, we chose different six algorithms so that they represent different points in the design space, and a range of publication dates with an emphasis on newer and improved approaches:

- gIndex[41] to represent frequent-mining algorithms, utilizing graph-structure features, storing the index in a prefix tree data structure;
- Tree+Δ[43] to represent frequent mining algorithms, utilizing tree- and cycle-structured features, storing the index in a hash table;
- CT-Index[44] to represent exhaustive enumeration algorithms utilizing tree- and cycle-structured features, encoding indexed features as fingerprints, and storing them in a hash table;
- gCode[34] to represent exhaustive enumeration algorithms utilizing path-structured features, encoding vertex- and neighborhood-related information in bit strings, and storing their combination in a balanced search tree data structure;
- GGSX[12] to represent exhaustive enumeration algorithms utilizing path-structured features, and storing them in a prefix tree data structure; and
- Grapes[11] to represent exhaustive enumeration algorithms utilizing path-structured features along with location information and storing them in a trie data structure.

A more detailed description of the in-use algorithms can be found in §3.1.1. In the following section, we report the indexing time, index size, the query processing time and filtering power of the above algorithms by employing the false positive ratio metric as discussed in §3.3.2.

4.3.2 Setup

All experiments were conducted on a Windows 7 SP1 host, featuring 2 Intel Xeon E5-2660 CPUs (2.20GHz, 20MB Cache, 8 cores/16 threads per CPU) and 128GB of RAM. For each experiment, a time limit of totally 8 hours[1] was imposed, after which the experiment was terminated.

[1]As a matter of fact, we waited for more than 24 hours before terminating those experiments exceeding the 8-hour limit, but to no avail.

For Tree+Δ, gIndex and gCode we used the implementations provided by [7]. For all remaining algorithms, we used the implementations provided by their respective authors. In the case of Grapes, we had to alter the source code so that the VF2 verification step returns after the first match of the query graph against a connected component for an indexed graph, as opposed to the original implementation which was returning all possible matches; this was necessary as all other algorithms return only the first match by default.

We used the default values for the input parameters of compared algorithms, as they were defined by their respective authors in the relevant publications and/or in their implementation code. Specifically:

- For gIndex, the maximum feature size was set to 10, the support ratio to 0.1, and the discriminative ratio to 2.0. The same parameters were used for both indexing and query processing.
- For Tree+Δ the maximum feature size was set to 10, the support ratio to 0.1, and the discriminative ratio[2] to 0.1. For query processing, the support ratio threshold to add new features to the index is set to 0.8.
- For gCode, paths of up to size 2 were used to construct the vertex signatures, and the top 2 eigenvalues are maintained. These values are based on the experiments performed by the respective authors of gCode, where their experiments reveal that the extra values maintained do not improve the filtering power. Additionally, the maintained vertex label and neighbor label bit-strings were both 32 bits long.
- For CT-Index, we created 4096-bit fingerprints by exhaustively enumerating trees and cycles of up to length 4. The respective authors of CT-Index propose the use of trees of size 6 and cycles of size 8, but [11] showed that a size of 4 for the features results in a somewhat worse filtering power but a significantly lower indexing and query processing time.
- For GGSX, we enumerated paths of up to a size of 4.
- For Grapes, we used 6 threads and enumerated paths of up to a size of 4.

4.3.3 Real and Synthetic Datasets

We tested the performance and scalability of these algorithms against (a) a set of real datasets provided by [11] (i.e., AIDS, PDBS, PCM, PPI), and (b) synthetic datasets created using the widely used GraphGen[60] generator.

Table 3.1 summarizes the characteristics of the real datasets. All four datasets differ across all of the characteristics which we identified as significant. Specifically, AIDS consists of a large number of small and low average degree graphs, PDBS contains a moderate

[2]Tree+Δ uses a different formula than gIndex to compute the discriminative ratio, hence the different parameter value.

number of large but low average degree graphs, PCM is comprised of a moderate number of medium-sized but high average degree graphs, and PPI includes few large but medium average degree graphs. Thus, they provide individual data points across the evaluation space, but are not adequate to examine the algorithms' scaling across datasets of varying sizes and complexities. Please note that the AIDS antiviral dataset was used by all the aforementioned algorithms as a whole or as a subset of graphs that form the database and thus, it can be used to compare results with previous work.

We used GraphGen (see §3.2.1) for the generation of our synthetic graphs, as it allows the parameterization of various parameters of interest. The parameters used are: number of graphs in the dataset, number of distinct labels (that will define the size of the alphabet), mean number of nodes and mean density per graph. Given a mean number of nodes and mean density per graph, and using the equation 2.1, the mean size of the graph in number of edges is retrieved, which is the input parameter used in GraphGen (instead of mean number of nodes). In brief, based on the aforementioned input parameters, GraphGen initially produces the alphabet of possible distinct edges. Then, for every new graph that will be added in the dataset, GraphGen computes a random number of edges and density, following a normal distribution around the user's corresponding input parameters, with a standard deviation of 5 and 0.01 respectively. Subsequently, GraphGen iteratively selects a random edge from the alphabet to add to the current graph until the desired size/density for this graph is reached.

Regarding the generation of the synthetic datasets, we took a rather systematic approach. First, we examined the values of the core input parameters for the four real datasets (AIDS, PDBS, PCM, PPI) (table 3.1) and we established an initial set of relevant values:

- Mean number of nodes per graph: {50, 200, 400, 4000},
- Mean graph density: {0.005, 0.025, 0.05, 0.075},
- Number of labels in the dataset: {10, 20, 40, 60},
- Number of graphs in the dataset: {1000, 10000}.
- Query size: {4, 8, 16, 32}.

We then tested the algorithms using all possible combinations of these parameter values ($4 * 4 * 4 * 2 = 128$ cases in the indexing phase and $4 * 128 = 512$ in the query processing phase). Alas, the results revealed that many of the algorithms could not produce an index or process queries for most of these combinations. We then computed a set of "sane defaults", so that they represent a challenging case but for which all algorithms could produce results; namely, 200 nodes per graph, average density 0.025, 20 distinct labels and 1000 graphs in the dataset. Subsequently, we executed several experiments to study the scalability of the various algorithms, varying one parameter at a time to examine its effect on the various metrics and algorithms.

It is also worth mentioning that several of the graphs in the real datasets are disconnected, whereas all generated graphs in the synthetic datasets are connected. Moreover, for the vast majority of the input parameter values considered in our setting, the datasets generated by GraphGen consisted almost exclusively – more than 95% of the graphs in the dataset – of graphs with cycles (i.e., not trees or paths). There were only two exceptions to this rule: (i) datasets with only 50 nodes per graph, where almost half of the graphs were tree-shaped, and (ii) datasets with average graph density of 0.005, where 8.5% of the graphs contained no cycles.

4.3.4 Query Workloads

We describe the procedure for generating query workloads in §3.2.3. For the purpose of our experiments, we created query graphs with 4, 8, 16, and 32 edges, to match the query graph sizes used by related work. As these query graphs are actually subgraphs of the various datasets, they have the same characteristics (on average) as the latter with regard to density and distribution of labels.

4.4 Evaluation Results

4.4.1 Real Datasets

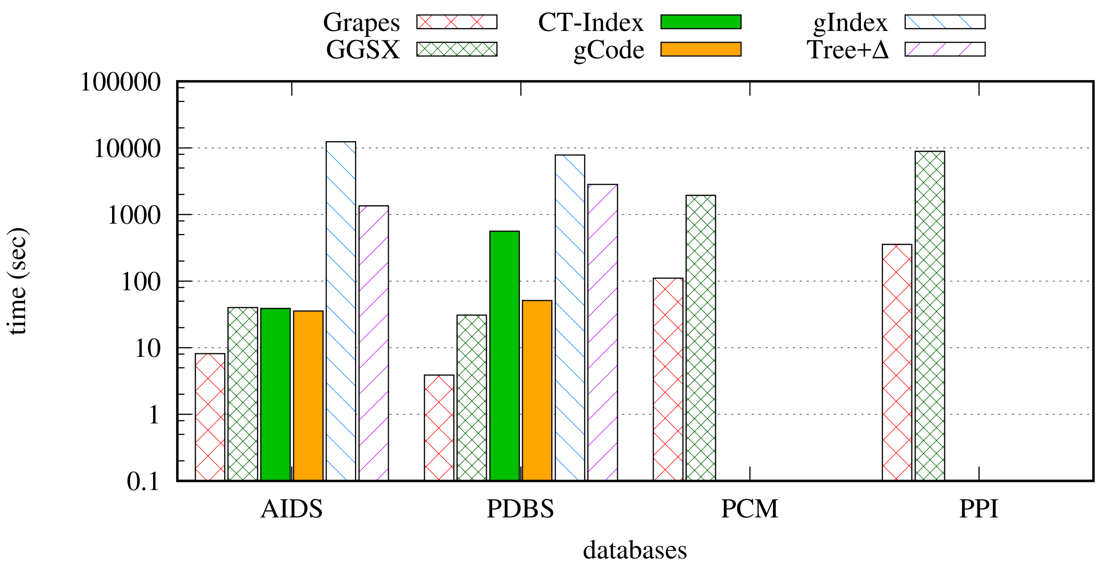
Figures 4.1(a) and 4.1(b) present the time and size requirements to perform the index creation with all algorithms. Grapes and GGSX are the only algorithms which managed to complete indexing for all datasets in the 8-hour time limit. Grapes consistently outperformed the other methods in terms of indexing time, often by at least one order of magnitude; conversely, its index size grows quite large compared to all but Tree+Δ's.

Figures 4.2(a) and 4.2(b) present the query processing time and false positive ratio respectively. Again, Grapes outperforms all contenders in processing time, with the sole exception of GGSX on the PPI dataset. The fact that there is no result for gCode for the PDBS dataset, is due to its implementation not being able to handle signatures of the size required for this dataset and thus crashing.

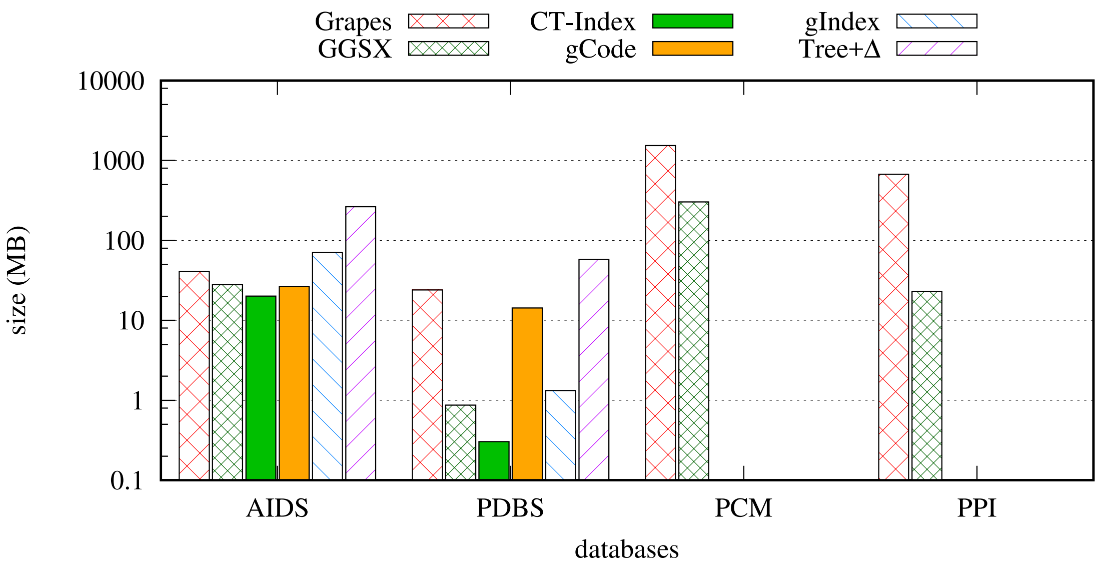
Overall, the results agree with what was reported in the respective papers for these algorithms.

4.4.2 Synthetic datasets

We will now focus on stress-testing the various algorithms using synthetic datasets, with the intent of both covering the space of possible parameter value combinations, and explor-



(a) Indexing Time



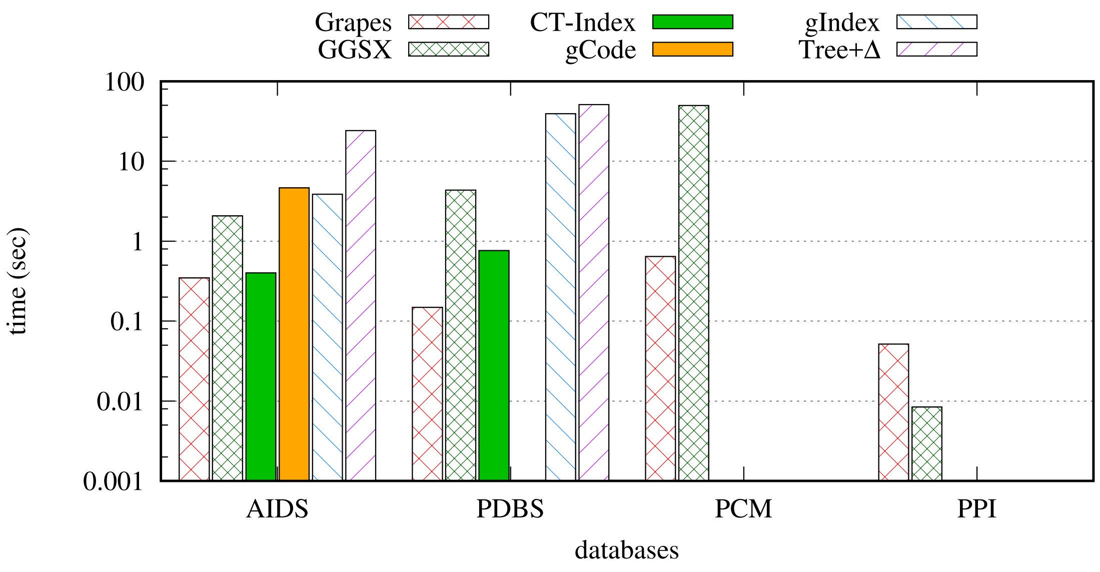
(b) Index Size

Figure 4.1: Indexing results over the real datasets

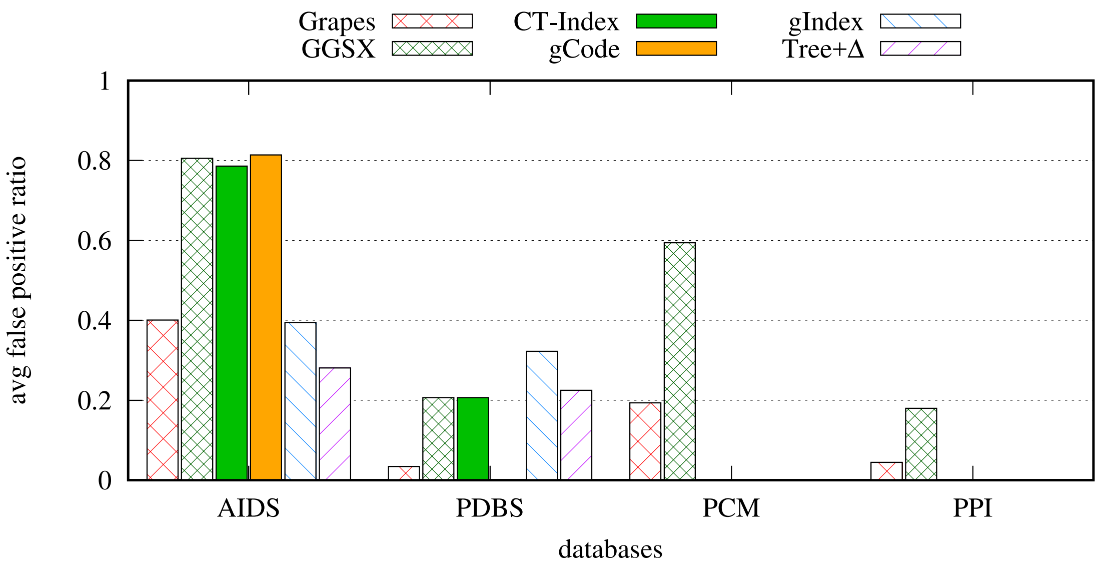
ing the breaking points of the various indexing and query processing algorithms. Unless otherwise noted, we are using the "sane" defaults mentioned above to generate the graph datasets and query workloads.

Number of nodes

First, we vary the number of nodes per graph in the dataset in an almost linear way so as to capture the different breaking points of the various algorithms for both indexing and query processing phases. Thus, we initially increase the number of nodes using smaller intervals



(a) Query Processing Time

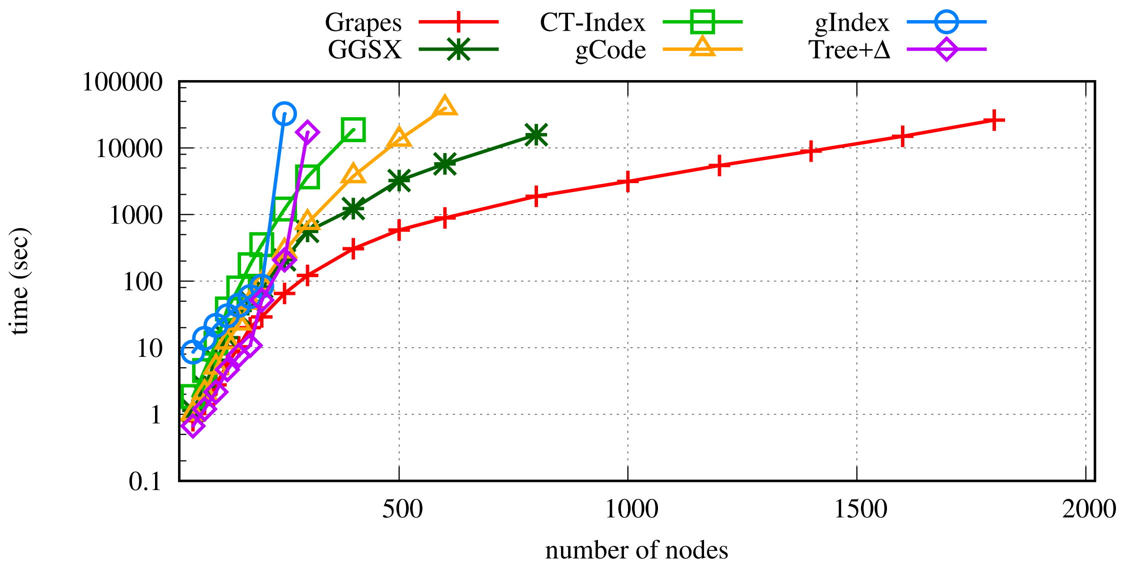


(b) Query Processing False Positive Ratio

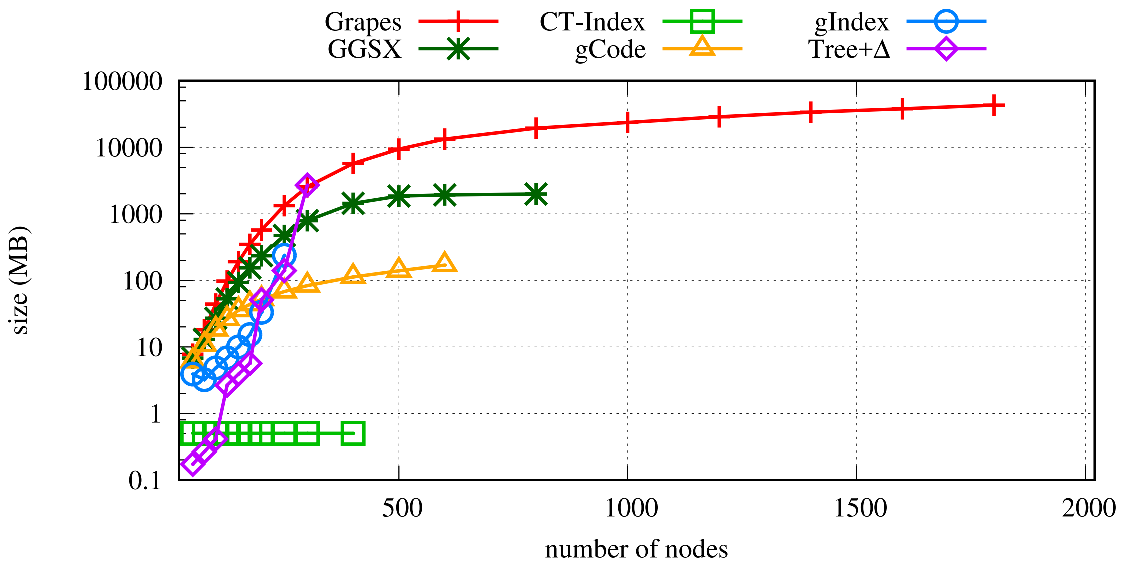
Figure 4.2: Query processing results over the real datasets

to provide sufficient points for all algorithms and then we gradually increase the intervals. Specifically, we have created datasets consisting of graphs with 50, 75, 100, 125, 150, 175, 200, 250, 300, 400, 500, 600, 800, 1000, 1200, 1400, 1600, 1800, and 2000 nodes. Given a fixed value for the average graph density, a linear increase in the number of nodes translates into a quadratic increase in the number of edges in the graph (see equation (2.1)); that is, we can also think of this scenario as consisting of graphs with an increasing number of edges.

Figures 4.3(a) and 4.3(b) present the time and size results for the index construction for each algorithm. For clarity, we provide the breaking points of the algorithms; gIndex can efficiently construct an index for only up to 250 nodes, Tree+Δ for up to 300 nodes, CT-



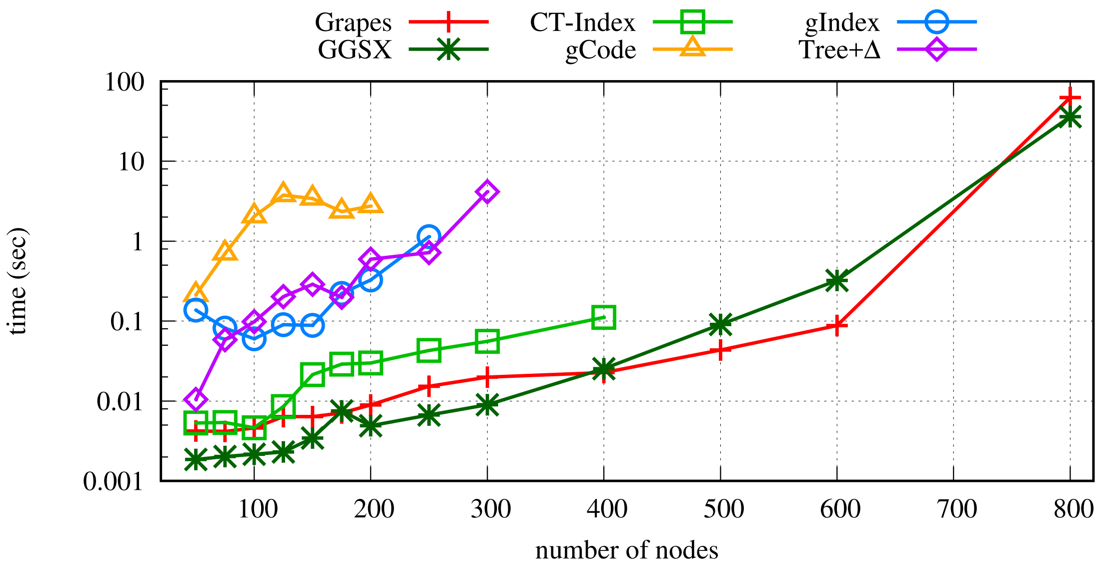
(a) Indexing Time



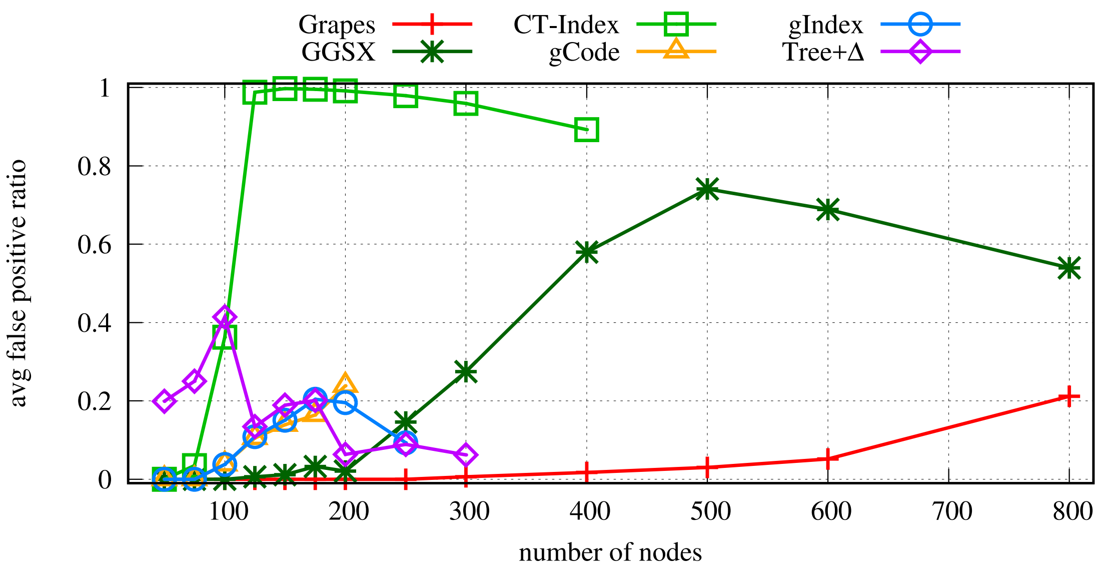
(b) Index Size

Figure 4.3: Indexing performance results for varying number of nodes

Index for up to 400 nodes, gCode for up to 600 nodes, GGSX for up to 800 nodes and finally Grapes for up to 1800 nodes. For small graphs (less than 175 nodes), Tree+Δ is marginally better than Grapes in index construction time. For larger graphs, Grapes takes the lead, being faster than the rest by at least one order of magnitude. GGSX comes second, with gCode and CT-Index being third and fourth; gIndex and Tree+Δ fail to produce an index even for as few as 250-300 nodes per graph. This result is expected and is a direct artefact of the complexity of the indexed features and the methods of feature extraction. Frequent feature mining is known to be a very computationally costly process[7] and thus gIndex and Tree+Δ have the worst running times; moreover, as graphs are more complex (and more numerous) than



(a) Query Processing Time



(b) Query Processing False Positive Ratio

Figure 4.4: Query processing performance results for varying number of nodes

trees, gIndex fairs worse than Tree+Δ. CT-Index and gCode exhaustively enumerate their features and are thus faster than the frequent mining approaches; however, the computation of fingerprints/signatures is non-trivial and this is evident in the results. Moreover, tree features are more complex (and numerous) than paths, and thus CT-Index fairs worse than gCode. Last, Grapes and GGSX both exhaustively enumerate paths (leading to the lowest running times), with the former having an edge due to its multi-threaded implementation.

On the index size front, CT-Index and gCode have the smallest indices since these algorithms only store fixed-size fingerprints/signatures per graph (gCode's index is larger as it also stores node signatures). The index size for GGSX and Grapes levels out after some point; as these algorithms use a prefix tree/trie to store indexed paths, as soon as all possible paths up to the size limit have been produced, the index structure doesn't grow any further (other than location/frequency information being recorded). Last, the frequent mining algorithms start off with a small index, but the larger the graphs the more the frequent features, and this exponential increase is evident in both of these figures.

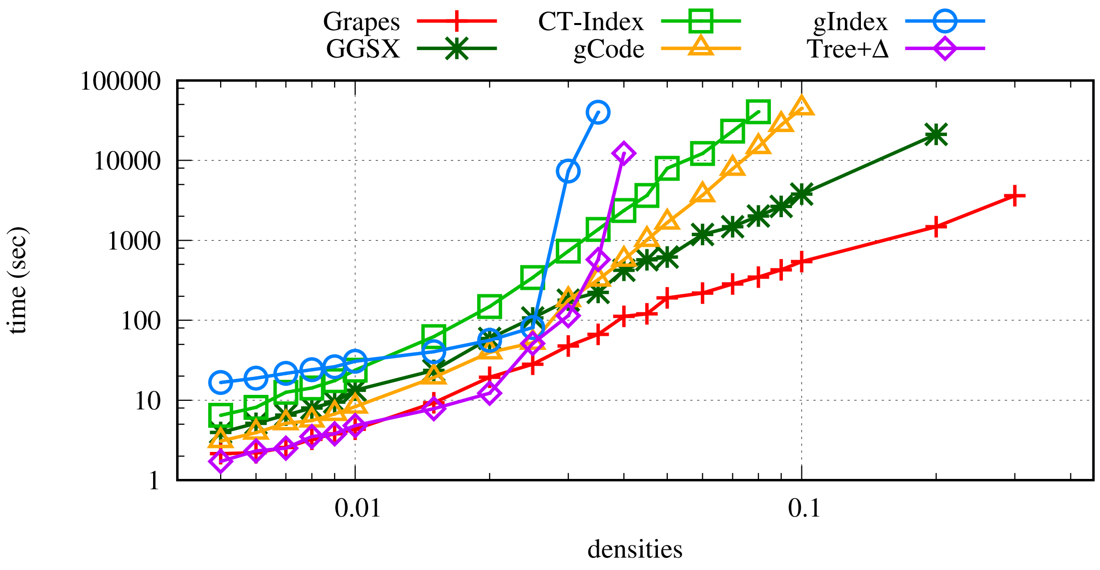
Figures 4.4(a) and 4.4(b) depict the query processing time and false positive ratio results for this case, averaged over all query sizes. The x-axis extends only up to 800 nodes due to the fact that no algorithm could handle queries on datasets with larger graphs within the 8-hour limit. Although Grapes managed to complete the indexing stage for larger cases (up to 1800-node graphs), in the query processing phase the increase in the number of candidates and in the average size of each candidate graph made the subgraph isomorphism time take more than the 8-hour limit. Moreover, gCode was failing for graphs larger than 200 nodes with error messages indicating that the implementation was unable to handle signatures of the produced sizes. Trend-wise, the "simple" algorithms (exhaustive enumeration of paths) were the clear winners, with the other algorithms following in a similar order as in the indexing time case. The exception here is gCode, whose convoluted signature generation algorithm resulted in a larger execution time than even the frequent mining algorithms. In conclusion, the order of the algorithms from the fastest to the slowest is: $($Grapes, GGSX$) <$ CT-Index $<$ $($Tree+Δ, gIndex$) <$ gCode.

Last, as far as the algorithms' filtering power is concerned, figure 4.4(b) shows an interesting trend: for all algorithms, the false positive ratio increases initially with the number of nodes, but then decreases again after some point: This "knee" appears around the 100-node mark for CT-Index and Tree+Δ, around the 200-node mark for gIndex and gCode, on the 500-node mark for GGSX, while Grapes doesn't reach its turning point in the x-axis range depicted in the figure.

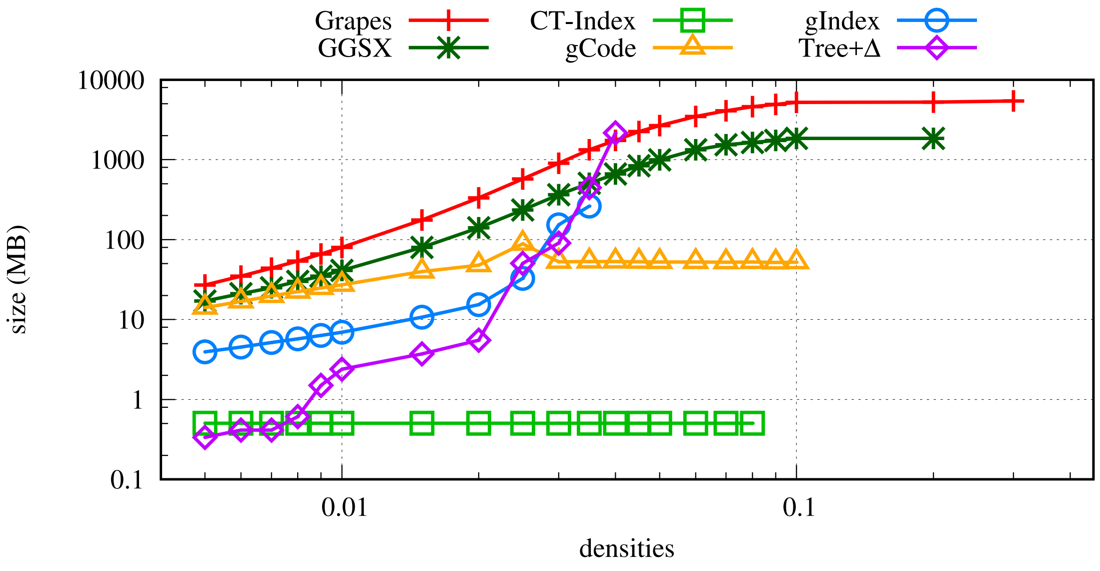
Density

Next, we vary the density of the graphs in the dataset. We used four intervals with different increasing steps each, denoted $[starting\_point : increasing\_step : ending\_point]$, and these are: [0.005:0.001:0.01], [0.01:0.005:0.05], [0.05:0.01:0.1], [0.1:0.1:0.3]. Thus, we used the following density values: 0.005, 0.006, 0.007, 0.008, 0.009, 0.01, 0.015, 0.02, 0.025, 0.03, 0.035, 0.04, 0.045, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.2, 0.3. The chosen values follow an almost exponential series with the exception of the second interval. We opted for the aforementioned density values for practical purposes, as in the first 2 intervals executions were fast and thus we were able to get more results. This allowed us to also reveal interesting patterns appearing in the performance sequence of algorithms in both indexing and query processing. Additionally, we identify the breaking points of all competing algorithms to appear in between density values 0.035 and 0.3.

Figures 4.5(a) and 4.5(b) depict the indexing time and size results, while figures 4.6(a) and 4.6(b) show the query processing results. Apart from Grapes and GGSX, no other algorithm could produce an index for density values above 0.1 within the 8-hour limit, and Grapes was the only one capable of dealing with densities above 0.2. With a fixed number of graph nodes, increasing density results in a proportional increase in the number of graph edges (see equation (2.1)). We would thus expect to see similar behavior for both indexing and query processing as in the previous case, only with a less dramatic effect as the dependency is now proportional, not quadratic, and this is exactly what can be seen in these figures.

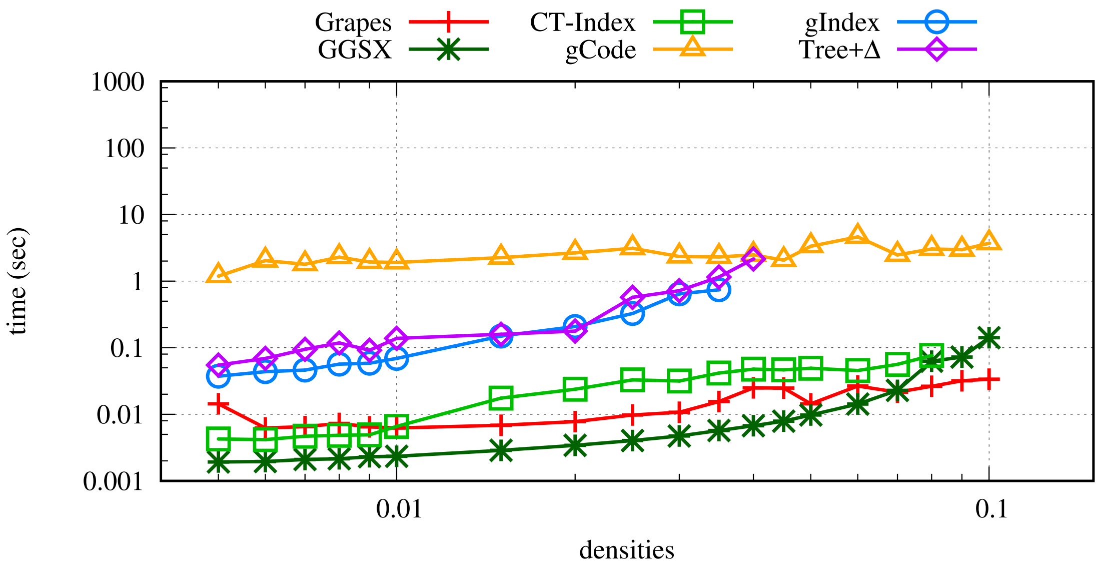


(a) Indexing Time

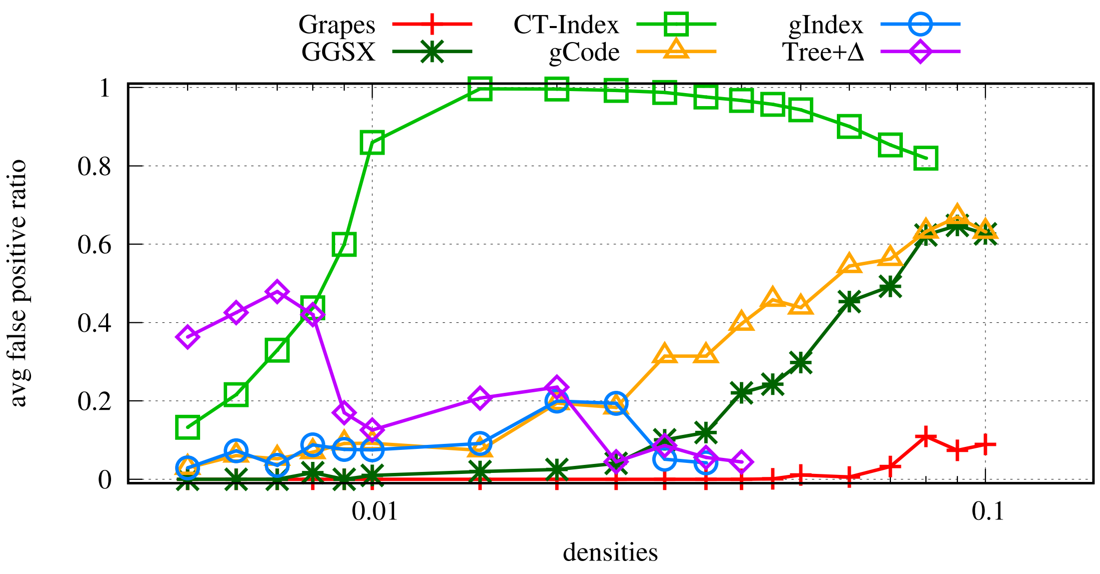


(b) Index Size

Figure 4.5: Indexing performance results for varying density values



(a) Query Processing Time

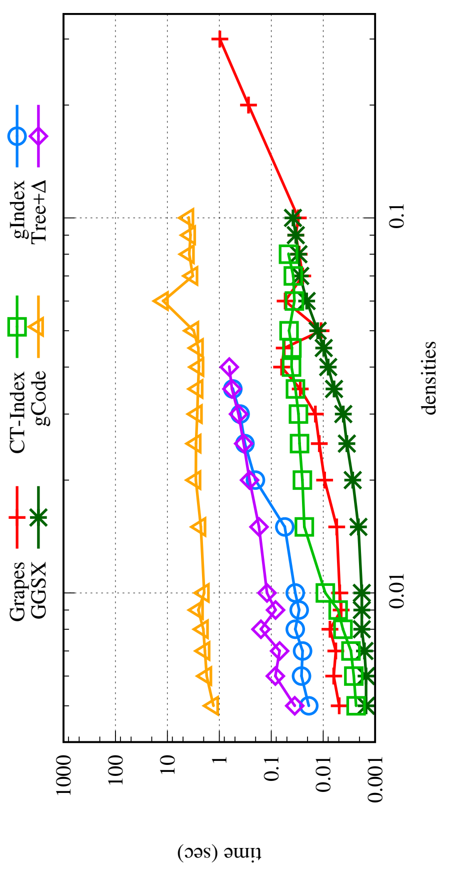


(b) Query Processing False Positive Ratio

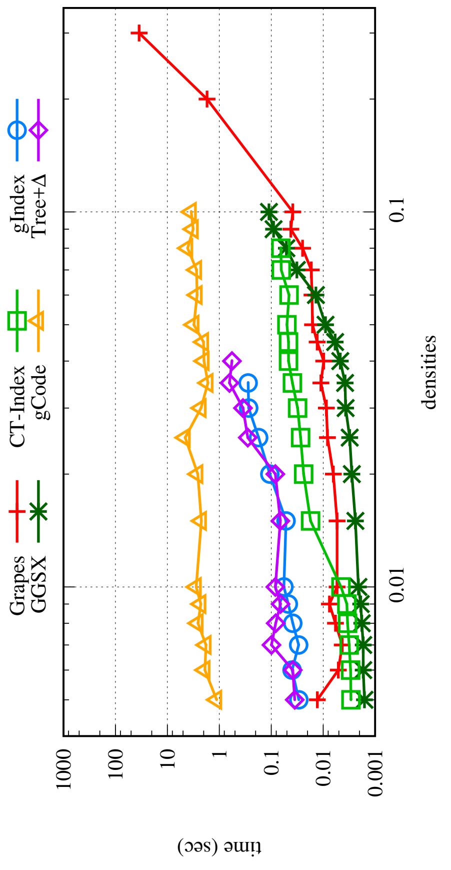
Figure 4.6: Query processing performance results for varying density values

For clarity, we provide the breaking points of the algorithms; gIndex can efficiently construct an index for only up to density 0.035, Tree+Δ for up to 0.04, CT-Index for up to 0.08, gCode for up to 0.1, GGSX for up to 0.2 and finally Grapes for up to 0.3.

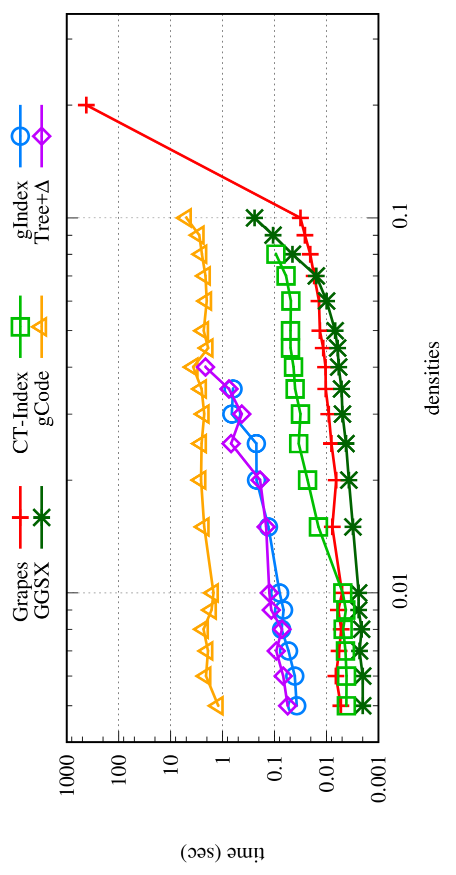
The astute reader will note that although GGSX and Grapes did produce an index for density values up to 0.3, the x-axis in Figures 4.6(a) and 4.6(b) does not extend beyond the 0.1 point. This was due to the fact that, when increasing densities, these algorithms didn't manage to produce results for densities above 0.1 within the 8-hour limit. To this end, we also show per-query-size query processing time results in figure 4.7. An interesting observation stemming from this figure is that, for density values up to 0.1, the exhaustive



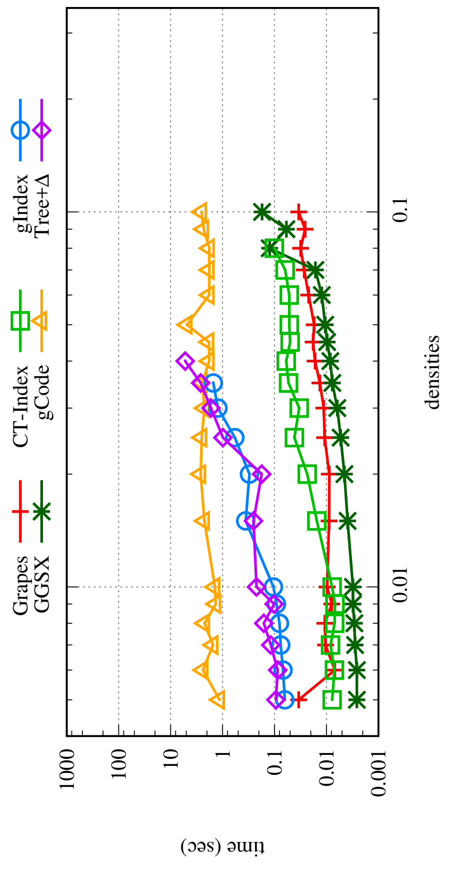
(a) Query Size: 4



(b) Query Size: 8



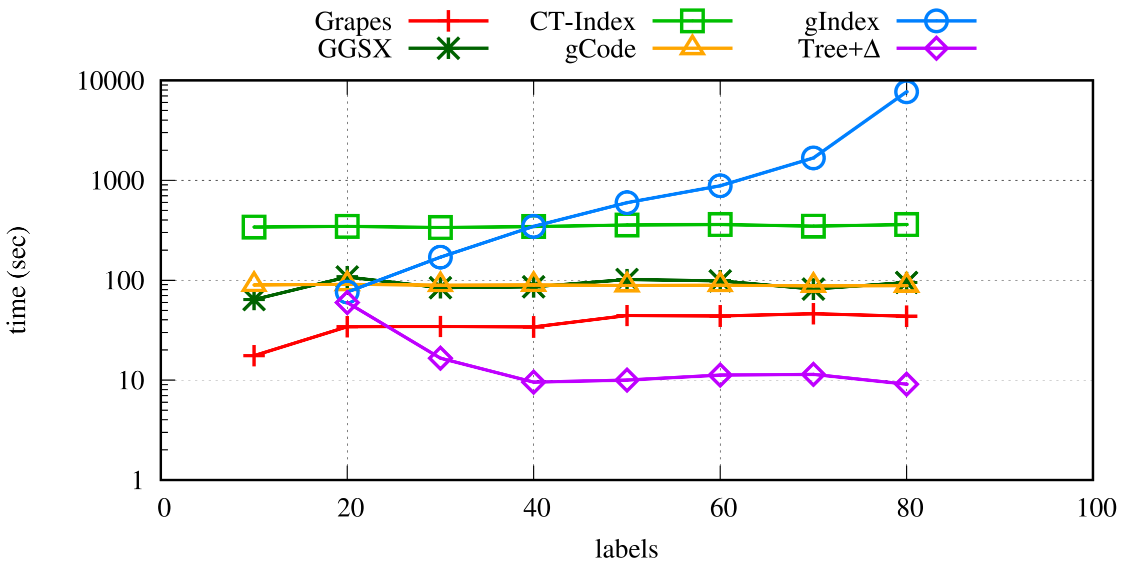
(c) Query Size: 16



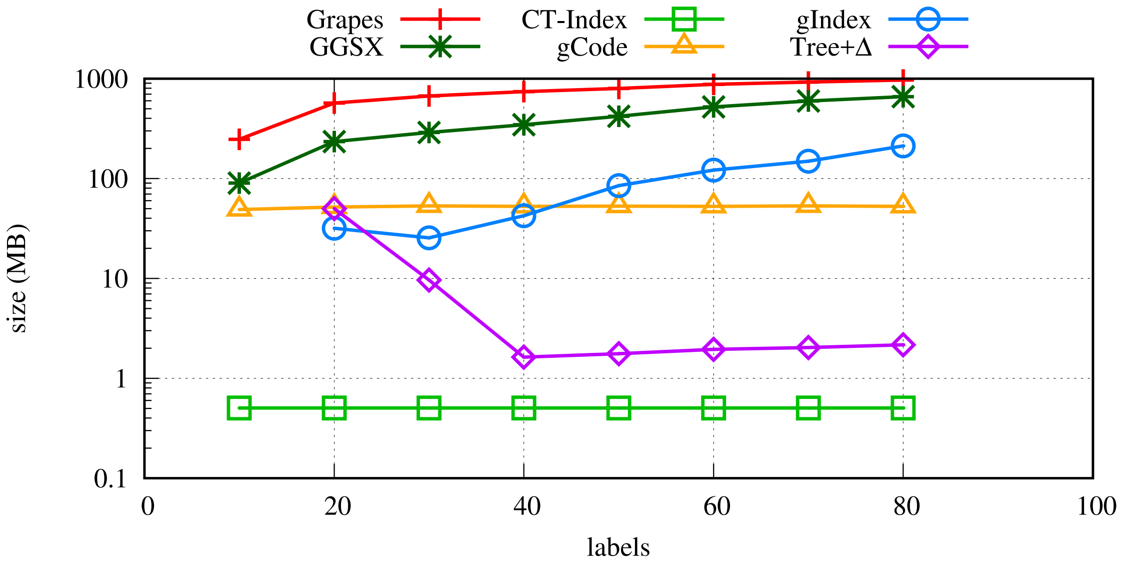
(d) Query Size: 32

Figure 4.7: Query processing times for individual query graph sizes and varying density values

enumeration approaches are rather insensitive to the size of the queries, whereas the frequent mining approaches show a small but noticeable increase in their query processing times. Grapes was the only algorithm capable of producing some query results for densities above 0.1. Moreover, we can see how the increase in query processing time becomes more abrupt as the query size gets larger, with Grapes producing results within the 8-hour limit for density values up to only 0.2 for 16-edge queries, and only up to 0.1 for 32-edge queries.

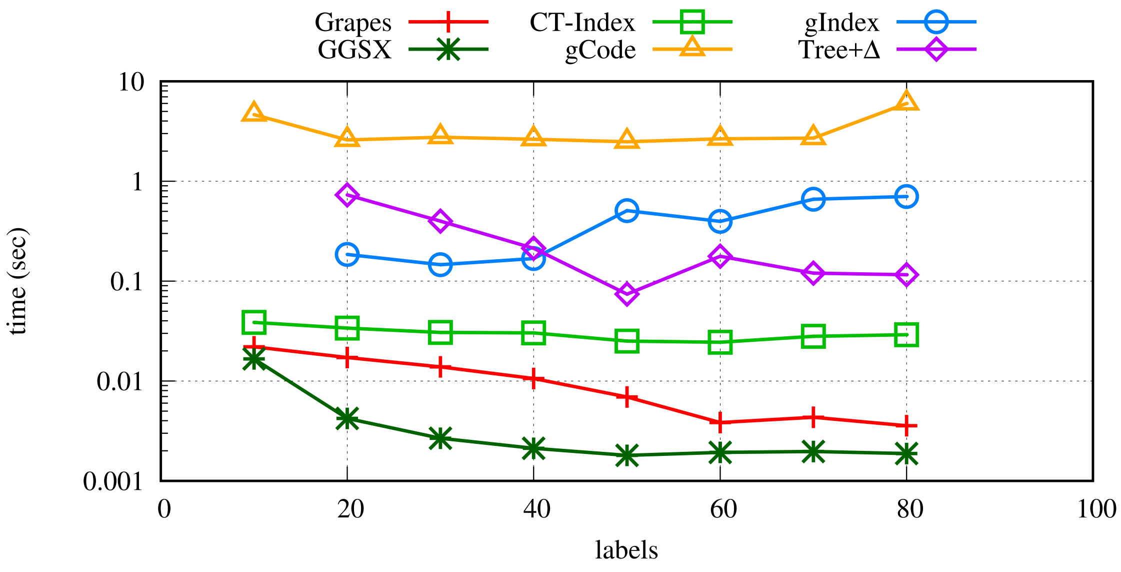


(a) Indexing Time

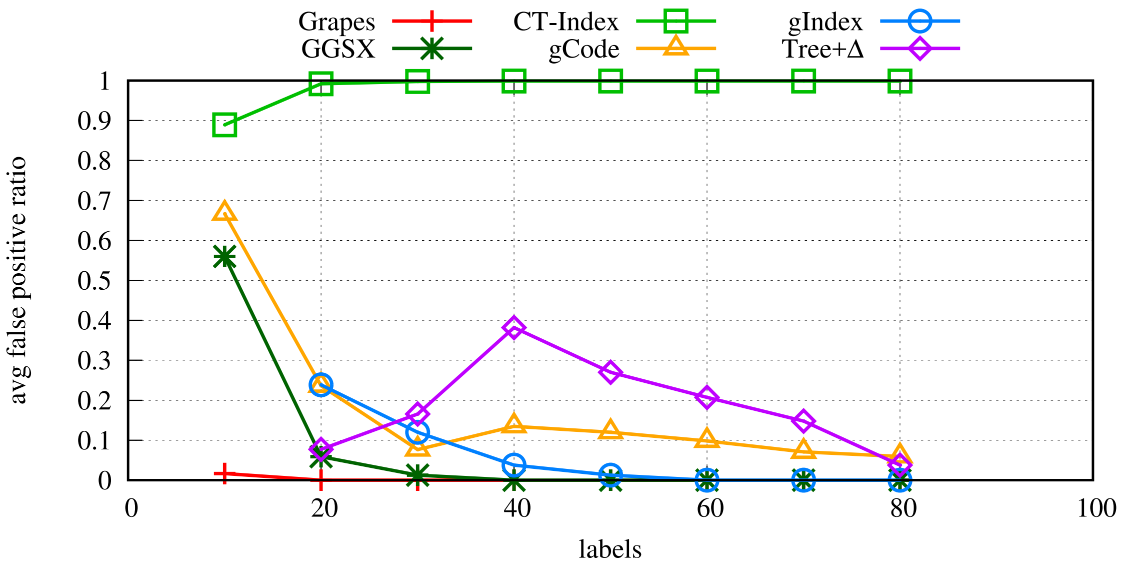


(b) Index Size

Figure 4.8: Indexing performance results for varying number of distinct labels



(a) Query Processing Time



(b) Query Processing False Positive Ratio

Figure 4.9: Query processing performance results for varying number of distinct labels

Number of distinct node labels

Varying the number of nodes per graph or the average graph density, resulted in larger graphs with more nodes and more edges. However, other than the graph size, the number of distinct labels in the dataset can also affect the indexing and query processing performance of the algorithms. Specifically, the *alphabet* of edges produced by GraphGen grows quadratically to the number of distinct labels (see §3.2.1). With the rest of the parameters constant, a larger alphabet translates into less overlap in the edges across graphs of any given dataset, inadvertently affecting approaches based on frequent pattern mining. We opted for a linear increase in the number of distinct labels, with the aim to include the number of distinct labels found in the real datasets, as presented in table 3.1. To this end, we performed an evaluation over datasets with 10, 20, 30, 40, 50, 60, 70, and 80 distinct labels.

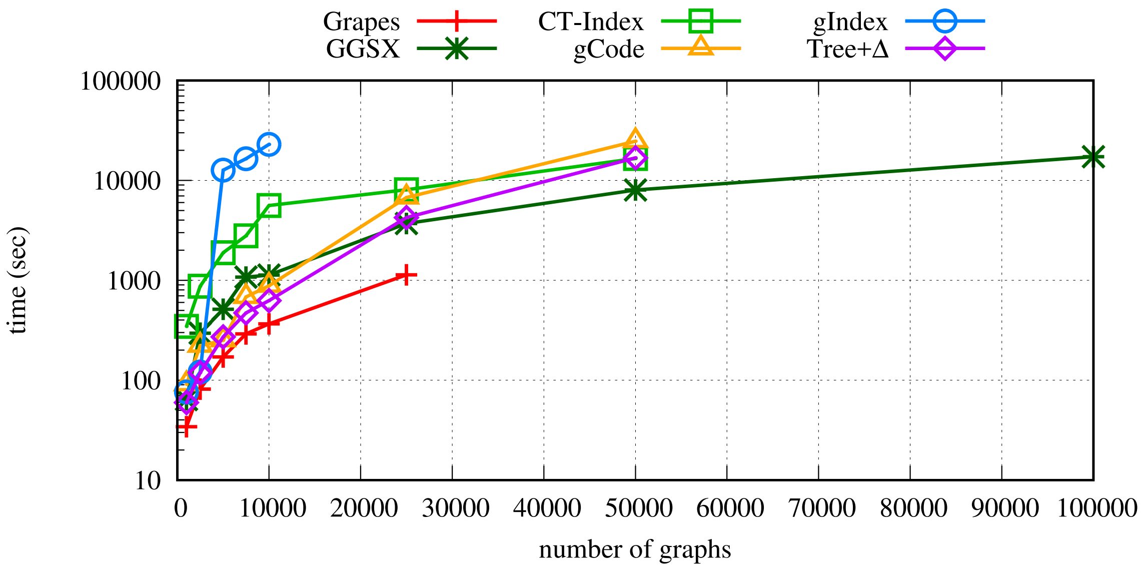
Figures 4.8(a) and 4.8(b) present the indexing time and index size for all algorithms. True to our above intuition, the indexing time of approaches using exhaustive enumeration, i.e. in Grapes, GGSX, CT-Index and gCode, is relatively unaffected by the increase in the number of distinct labels. On the other hand, the two frequent-mining algorithms are definitely affected, albeit curiously in completely opposite ways; whereas gIndex's indexing time increases with more distinct labels, that of Tree+Δ's decreases. We attribute this discrepancy in the different heuristics implemented by each algorithm; both algorithms set different discriminative ratios and employ different mining features, i.e., the former mines graphs while the latter mines trees. Tree canonical labels are less computationally expensive to be produced than graph canonical labels and require less space for storage. Also notably, the frequent mining techniques could not construct an index within the 8-hour limit for the case of 10 distinct labels. We speculate the following: these mining techniques start from small features of size 1 (1 edge), that are expanded by one edge at every stage. If all features are found to be frequent because of the small number of labels, then they all need to be expanded in the next step, leading to an exponential increase of combinations to be considered.

Figures 4.9(a) and 4.9(b) present the query processing time and false positive ratio results, averaged over all different query sizes. We can see that the processing time of all algorithms seems to improve when utilizing more distinct labels, with the sole exception of gIndex. Similarly, the filtering power of the algorithms also improves when increasing the distinct labels, with the exception of Tree+Δ and CT-Index. Intuitively, the more the distinct labels, the less repetition there is of distinct edges (i.e., pairs of labels) from the *alphabet*, so the number of false positives is expected to decrease. CT-Index again has the worst filtering power of all contenders, since its fixed-size hash-based fingerprints seem to suffer from "collisions" (considerably different graphs producing very similar fingerprints); however, what it loses in filtering power, it gains in processing time, with the simplicity of its hash-based approach and its tweaked verification algorithm. In this case as well, the general pattern regarding query processing time is: (Grapes, GGSX) $<$ CT-Index $<$ (Tree+Δ, gIndex) $<$ gCode.

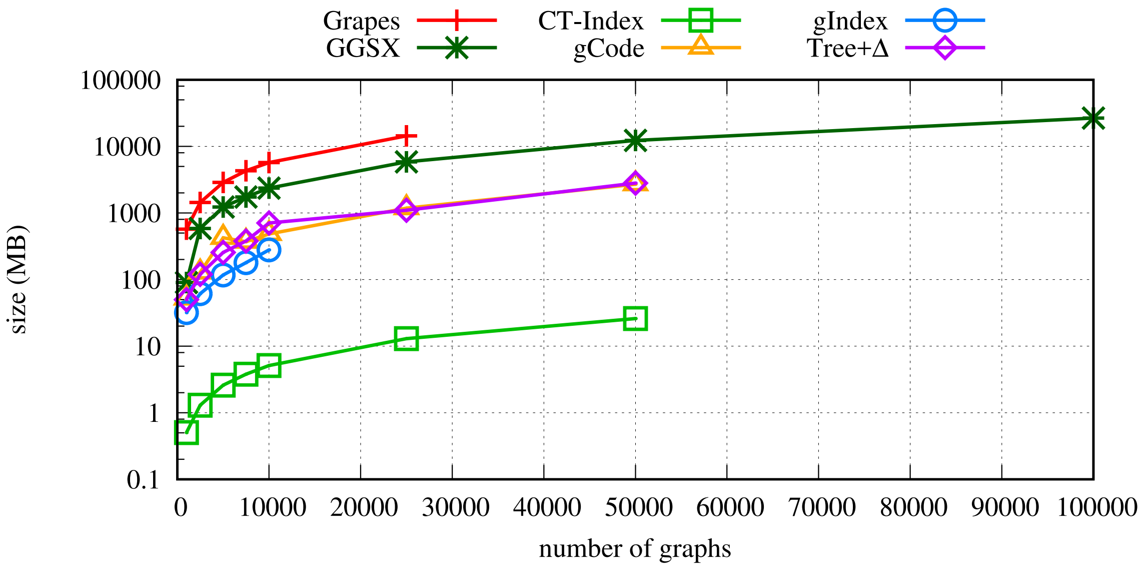
Number of graphs in the dataset

Last, we vary the number of graphs in the dataset. This parameter takes on values of 1000, 2500, 5000, 7500, 10000, 25000, 50000, 100000 and 500000 graphs. Similar to the density, we also opted for the semi-exponential series. We would expect all performance metrics (indexing time, index size, processing time) to scale linearly to the number of graphs, as the latter does not affect in any way the complexity or size of individual graphs in the dataset. Along the same lines, we would expect the query processing false positive ratio to be relatively unaffected, for the exact same reason. These intuitions are indeed verified by the results depicted in figures 4.10 and 4.11, where these tendencies are rather prominent and clear.

Although the increase in performance metrics is linear to the number of graphs, we can see that all algorithms other than GGSX didn't manage to produce an index for very large numbers of graphs. For gCode, CT-Index, and Tree+Δ, this was a case of the indexing pro-

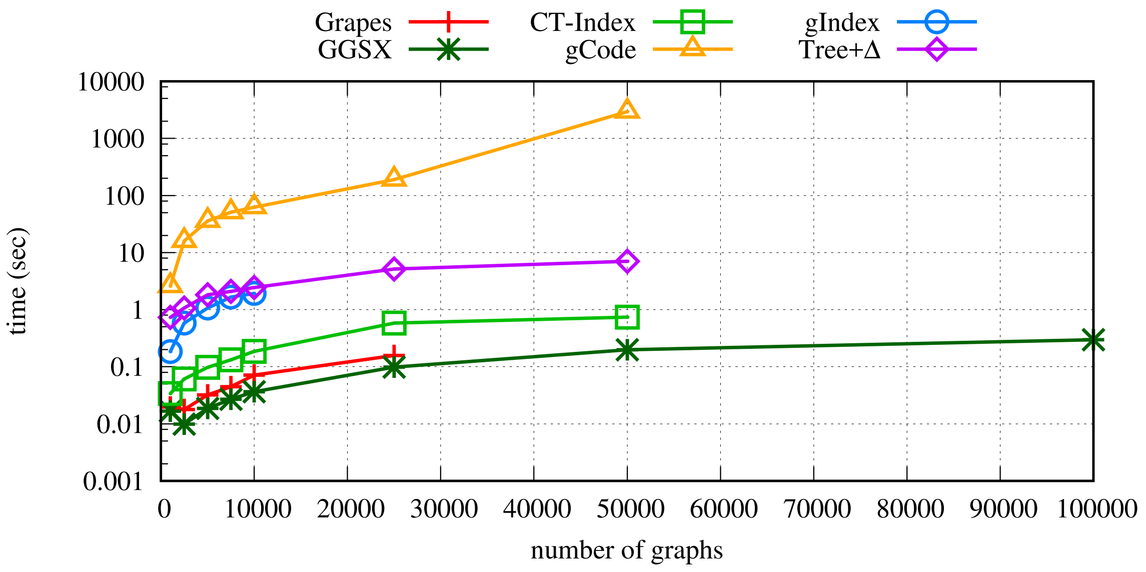


(a) Indexing Time

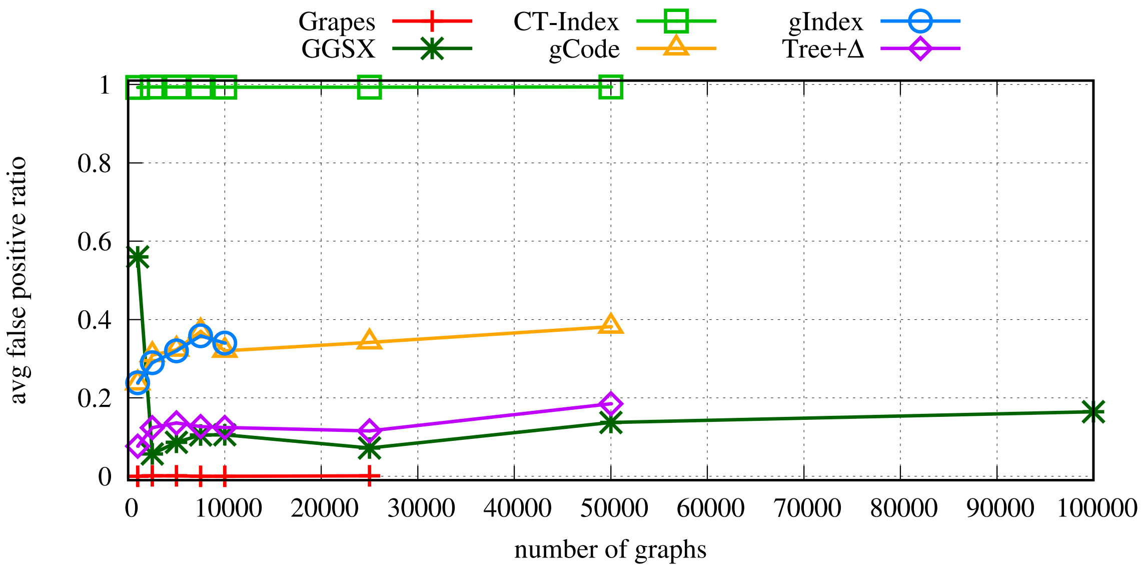


(b) Index Size

Figure 4.10: Indexing performance results for varying number of graphs in the dataset



(a) Query Processing Time



(b) Query Processing False Positive Ratio

Figure 4.11: Query processing performance results for varying number of graphs in the dataset

cess taking more than 8 hours for datasets with more than 50,000 graphs. gIndex also failed to produce indices due to excessive indexing time; moreover, its average time was much higher than that of other algorithms (an artefact of frequent mining and graph features) and thus exceeded the 8-hour limit much earlier than the rest (around the 10,000-node mark). Furthermore, this is the only case in our experiments where Grapes didn't manage to complete all indexing chores; the reason for this was excessive memory usage (its index size curve alludes to this), leading to thrashing even in our 128GB RAM host. However, for the cases where it managed to produce an index, its query processing performance was on par with that of GGSX and considerably better than the rest. GGSX towards the indexing of the last paths in the case of 100,000 graphs was also using an excessive amount of memory with the thrashing not being very evident in that stage. Last, we can see again the same paradox as before for CT-Index; although its filtering power was the worst by a large margin, its query processing time was second only to Grapes and GGSX, due to the speed of using hash-based fingerprints and implementing a smarter subgraph isomorphism algorithm. In summary, the general pattern regarding query processing time is: $(\text{Grapes, GGSX}) <$ CT-Index $< (\text{Tree}+\Delta, \text{gIndex}) <$ gCode.

4.5 Lessons Learned

We strived for comprehensiveness; first, we used four real datasets representing different points in the dataset complexity, to gain some first insights into the performance of the various approaches and to validate our experimental infrastructure against results published in the respective papers for the competing algorithms. We then identified five key characteristics of the graph datasets and corresponding query workloads that influence the performance of the underlying subgraph isomorphism algorithm; namely, (i) the number of nodes/ edges and (ii) average density per graph, (iii) the number of distinct labels and (iv) the number of graphs in the database, and (v) the number of nodes/ edges in the query graphs. Subsequently, we performed an extensive and systematic exploration of this space, by using a large number of purposefully generated synthetic datasets generated so as to cover all aspects of the design space, employing a well-known synthetic generator. We focused not only on the performance of the various algorithms, but also on their scalability, in terms of constructing the index in reasonable time and size and answering queries in reasonable time; by stress-testing the various approaches, we attempted to identify tendencies in their performance as well as their breaking points. Based on our experiments, the following is a summary of the insights we gained from the above evaluation.

4.5.1 Effect of key dataset/workload characteristics

Our findings of the effect of the dataset characteristics on the performance and scalability of the various algorithms are summarized below:

- As indicated by equation (2.1), when density is kept constant, a linear increase in the *number of nodes* results in a quadratic increase in the number of edges. As the number of features is exponential to the number of nodes in the graph, the increase on the number of nodes leads to a detrimental increase in the indexing time, with the frequent mining techniques being more severely affected (figures 4.3 and 4.4).

- Along the same lines by equation (2.1), given a constant number of nodes, the number of edges increases linearly to the *graph density*. Thus, when increasing the graph density, indexing and query processing times are affected in a similar manner as in the case of increasing the number of nodes, only with a less dramatic effect, because the dependency is now proportional and not quadratic (figures 4.5 and 4.6).
- The *number of graphs* increases the overall complexity only linearly (albeit the frequent mining techniques are more sensitive because more features have to be located across more graphs) (figures 4.10 and 4.11).
- The increase in the *number of distinct labels* leads to an easier dataset to index and an easier query workload to process (figures 4.8 and 4.9), as it results in fewer occurrences of any given feature and thus in a decrease in the false positive ratio of the various algorithms. Even relatively small changes in this characteristic affected drastically the performance of some of the algorithms.
- The *size of query graphs* affects all methods, even more so in the case where the datasets consist of dense graphs. This effect is more pronounced for frequent mining techniques, manifesting even for moderately dense or even sparse graphs (figures 4.4, 4.6, 4.7, 4.9 and 4.11).

4.5.2 Sancta Simplicitas

As shown by our experiments, on one hand, algorithms utilizing complex feature structures: (a) fail to produce indexes within the time limit for large/complex datasets, and (b) exhibit a high filtering power, but at the expense of a much higher overall query processing time. On the other hand, *algorithms with simpler feature structures* (i) have very fast indexing times, traded off for a higher index size, (ii) also exhibit a high filtering power by the sheer coverage provided by the larger number of indexed features, and (iii) achieve the lowest processing times of the lot; (iv) last, the additional use of location information seems to greatly increase the index's filtering power (e.g., see Grapes).

Overall, our findings give rise to the following adage: *"Keep It Simple and Smart"*. The general tendency is that, the simpler the feature structure and extraction process, the faster the indexing and query processing algorithm. Although seemingly counterintuitive, this conclusion is easily justifiable. Graphs are indeed more expressive than trees, which are in turn more expressive than paths, and thus a graph-based index would have a higher filtering power and lower processing time than a tree-based index etc. However, *the number of subgraphs of size n in a graph is significantly larger than the number of trees of size n, and the trees of size n is significantly larger than the number of paths of size n.* Due to this, on one side, indexes utilizing more features with complex structures are forced to maintain only a subset of them (frequent mining techniques) or apply some compression upon them

(CT-index). Thus, the expressive power gained by the more complex features is offset by the decrease in coverage and/or by the introduction of yet more false positives in the filtering stage. Moreover, the more complex features also translate into higher indexing times and similarly higher filtering times. On the other side, algorithms with simpler feature structures, resorting to exhaustive enumeration during their index construction, enjoy low indexing and filtering times, at the expense of considerably larger indexes.

4.5.3 Choosing the right index method for user needs

The various solutions tend to optimize different aspects of their operation. Answering which algorithm is best fit for any given case requires choosing an optimization criterion of interest, and these are the indexing time and size, the query processing time and the scalability.

If *index size* is of importance, algorithms utilizing fixed-width encodings (CT-Index, gCode) should be chosen first; this is especially true as the size and complexity of the input grows. Frequent mining algorithms (gIndex, Tree+Δ) may be competitive for small/sparse datasets, but they quickly lose their edge as the datasets grow. Last, techniques using exhaustive enumeration and no encoding of features (Grapes, GGSX) have by far the largest indexes in size. This is particularly important if the index is to reside in main memory, as it is usually desirable in most realistic use cases.

For the lowest *indexing time*, one should first look at techniques exhaustively enumerating their features (Grapes, GGSX, gCode, CT-Index). Approaches that utilize simpler features (paths; i.e., Grapes, GGSX) are considerably faster compared to those using more complex features (trees, cycles; i.e., CT-Index) and/or those using encoding (i.e., CT-Index, gCode). Again, frequent mining approaches (gIndex, Tree+Δ) are competitive only for small/sparse datasets, but their indexing times grow very high very fast.

For *query processing time*, again the approaches using exhaustive enumeration (Grapes, GGSX, CT-Index) are the definitive winners, with those indexing simple features (paths; i.e., Grapes, GGSX) having the edge over those with more complex features (trees, cycles; i.e., CT-Index). Frequent mining approaches (gIndex, Tree+Δ) are usually an order of magnitude slower than that. gCode here is the odd one out, as its encoding scheme seems to dominate the query processing time, hence it appears to be the slowest.

From a *scalability* point of view, as the input grows larger and/or denser, interesting trends evolve. For example, gCode, usually by far the slowest of the lot in query processing time, wins over the initially much faster frequent mining approaches as the dataset and graphs grow in size and density, as it exhibits a much better scaling. Conversely, Grapes, usually a very fast algorithm, fails to produce an index for certain very large datasets, and is routinely outscaled by GGSX. Notably, when the number of graphs goes beyond a few thousand, Grapes is also outscaled by Tree+Δ, gCode and CT-Index, as the additional location information in Grapes' index causes it not to fit in main memory.

4.5.4 Scalability limits

When the size of the dataset (in number of nodes or density per graph and/or in number of graphs in the dataset) grows very large, *none* of the above methods is capable of coping with it. Specifically, our results show that none of the studied methods can scale beyond graph datasets with 1000 graphs, with each graph having 800 nodes, of medium (0.025) density (figures 4.3 and 4.4). Reducing the average number of nodes per graph to 200 allows GGSX to operate with a dataset of up to 100000 graphs (figures 4.10 and 4.11). At larger scales, one should (i) vary input parameters of existing algorithms and/or combine existing top-performing algorithms to form a better one (this will be discussed in chapter §6), (ii) rethink anew indexing methods, (iii) adopt an index-less approach (e.g., [23]), or (iv) devise algorithms providing approximate answers.

Finally, comparing our conclusions against those reached by the iGraph[7] study, we note the following. For the smaller datasets that iGraph studied and for the common algorithms in iGraph and our paper, our results are in complete agreement. Specifically, the results for the index construction of gCode, gIndex and Tree+Δ show the same relative order and trends in both papers. Furthermore, for smaller datasets, our query processing results also coincide.

However, in our work, the algorithms were systematically studied as they depend on key workload and dataset characteristics and were stress-tested using 4 real and many synthetic datasets, revealing the new insights outlined above. Furthermore, with respect to the common algorithms with iGraph, although gCode can be up to orders of magnitude worse than gIndex and Tree+Δ at smaller scales, gCode can outscale both gIndex and Tree+Δ, as the dataset's density increases. gCode can also outscale gIndex and match the scalability of Tree+Δ as the number of dataset graphs increases. More importantly, in contrast to the iGraph paper's conclusion that there is no overall winner, our study reveals 2 methods, GGSX and Grapes, one of which is always the clear winner for query processing time and scalability!

4.6 Conclusions

The chapter reports a comprehensive and systematic evaluation of the performance and scalability of FTV methods for processing graph containment queries. Graph containment queries have received a lot of attention from our community, with research efforts spanning for over a decade. We have employed four real-world graph datasets and a large number of synthetic graph datasets. We have selected six methods to comparatively evaluate, representing different key design decisions and including also the most recent and higher performing methods. The performance and scalability of the methods was analyzed with respect to their sensitivity to different key dataset and workload characteristics that can affect the performance and scalability of methods; namely, the number of DB graphs, the size of DB graphs (as it depends on the graph density and number of graph nodes), the number of distinct labels per graph, and the size of query graphs. We provide detailed conclusions on (and explanations for) the comparative performance and scalability of the methods, as they depend on the key dataset and workload characteristics. In addition, we provide a set of major lessons learned. Although the results show no clear winner, we shed light into the characteristics of the winning methods for most of the datasets-workloads, the effect of the different characteristics of the performance/scalability of different methods, and on the differences between scalability and performance.

Limitations Despite the large number of experiments performed, there are limitations to the study. The initial target was to be able to answer exactly which algorithm to use given any set of parameters, i.e. average number of nodes and density, number of distinct labels and graphs in the dataset. Instead, we set the "sane" default parameters as discussed in §4.3.3 and vary only one parameter at a time to examine its effect on the various metrics and algorithms. Finding the exact algorithm to use for any set of given parameters proved to be impractical as the dataset workload characteristics are not enough to select the best algorithm. In more detail, as we will see in chapter §5, even different rewritings of a query – i.e., isomorphic instances of the same query – can lead to different algorithms performing the "best". For this reason, a more detailed model needs to be devised, which is left as the subject of future work.

Another limitation of our study is related to the values of the input parameters of the compared algorithms (see §4.3.2). Although these values are defined by the respective authors, the experiments reported in §6.9 indicate that the overall performance of the algorithms can be significantly affected. Defining the optimal values, requires deep knowledge of the algorithm, the underlying implementation and the problem to solve, and it is hard even for the authors of these methods to answer what are the optimal parameters to use.

Along the same lines of the previous limitation, the experimentation for finding the optimal values to use for every case is highly impractical. Thus, we need to devise a framework to use that would be able to auto-adjust the optimal values for answering queries in the most effective way on a specific dataset. That is also left for future work.

Chapter 5

Subgraph Querying with Parallel Use of Query Rewritings and Alternative Algorithms

Despite the large attention the problem of subgraph query processing has enjoyed within the data management/mining communities and the large number of methods that have been developed, scalability and efficiency remain an elusive goal, owing to the NP-Complete nature of the underlying subgraph isomorphism test. We analyze this problem and present key novel discoveries and observations on the nature of the problem which hold across query sizes, datasets, and top-performing algorithms. Firstly, we show that algorithms (for both the decision and matching versions of the problem) suffer from straggler queries, which dominate query workload times. If one were to cap query times and not to report results for queries exceeding the cap (personal communication with authors of [11], 2015), this can lead to erroneous conclusions of the methods' relative performance. Secondly, we study and show the dramatic effect that isomorphic graph queries can have on query times. Thirdly, we show that for each query, isomorphic queries based on proposed query rewritings can introduce large performance benefits. Fourthly, we observe that straggler queries are largely algorithm-specific: many challenging queries to one algorithm can be executed efficiently by another. Finally, the above discoveries naturally lead to the derivation of a novel framework for subgraph query processing, resting on query graph rewritings (producing graphs that are isomorphic to the original query) and on using multiple alternative algorithms. The central idea is to employ parallelism in a novel way, instead of trying to parallelize the subgraph isomorphism algorithm, whereby parallel matching/decision attempts are initiated, each using a query rewriting and/or an alternate algorithm. Such an approach is common to other NP-hard problems, where parallelizing the underlying algorithm is not easily feasible and thus running parallel instances of the same or alternative algorithms is a better approach, known as portfolios of algorithms. The framework is shown to be highly beneficial across algorithms and datasets.

5.1 Introduction

With our work, we explore the strengths and weaknesses of existing FTV and SI methods for all versions of the subgraph querying problem, and we exploit the strong points of every algorithm by combining them in parallel executions to achieve large performance gains in the subgraph querying problem. Our analysis aims to (i) lead to interesting novel findings about the nature of the problem and existing solutions, (ii) analyze and quantify said discoveries and their effect on well-established existing solutions, and (iii) show that the findings can be used to develop a framework that can offer large performance gains. Specifically, we first recognize the existence of "straggler" queries; i.e., queries whose execution time is dramatically higher than the rest. This holds for all query workloads and all datasets examined and across all tested FTV and SI algorithms. Subsequently, we reveal and quantify the interesting fact that isomorphic instances of queries can have a wild variation in querying times. Then we generate isomorphic instances of the original query using statistics on vertex-label frequencies and/or vertex degrees and we investigate their performance. Moreover, for SI methods in particular, we additionally show that challenging queries are algorithm-specific, with a straggler query for one algorithm possibly being easy for others. Finally, we incorporate these findings in a novel framework, coined the Ψ-framework, that exploits parallelism for both FTV and SI methods, achieving large performance gains. Specifically, instead of trying to come up with new algorithms for sub-iso testing, we utilize isomorphic query rewritings and existing alternative algorithms in parallel. Extensive experimentation shows that our framework can be highly beneficial across both real and synthetic datasets and query workloads, and for both FTV and SI methods.

5.2 Related Work and Contributions

The numerous proposed FTV and SI methods have been extensively discussed in §2.3. The target of the current chapter is to propose empirical ways in order to expedite the subgraph queries, originated from the strengths and weaknesses of all proposed methods. We mainly focus on the various SI methods and thus, we leave intact the indexing and filtering stage of the used FTV methods.

As subgraph querying is an important problem, we expect that many researchers will keep focusing on trying to improve upon existing algorithms in the future. Indeed, since the publication just a few years ago of iGraph-v2[8], comprehensively comparing the then state of art SI algorithms, newer algorithms have been proposed, e.g. [55], with better performance. Nonetheless all algorithms show exponential execution times even at small query sizes (up to 10 edges) (BoostIso[56]). Our contributions aim to help this process in two ways. First, by revealing key insights, based on comprehensive experimentation, about the problem itself and how they affect well-known algorithms. Second, by shedding light onto a novel overall approach to the problem and its benefits. Namely, instead of focusing solely on developing new solutions by improving earlier algorithms, try to benefit from the wealth of ideas already existing within previous algorithms! Specifically, our findings show that different algorithms are appropriate for different queries. Furthermore, they show that different query rewritings are appropriate for different queries and for different algorithms! Finally, the existence of straggler queries poses new challenges for the performance comparison of different algorithms, needing more detailed performance metrics and experimenting with more challenging queries. All current works miss the above points: (i) they only consider one query rewriting, if at all, for all queries, (ii) they use only one algorithm for all workload queries, and (iii) they do not stress-test their algorithms with more challenging queries (e.g., larger sizes). Our framework shows that such misses also lead to misses of dramatic performance improvements.

5.3 Experimental Setup

5.3.1 Algorithms

For the FTV methods, we chose Grapes[11] and GGSX[12] to be included in our experiments, accredited as top-performing based on our experiments in chapter §4.

For the SI methods, we opted for methods (i) whose code is publicly available or made available to us by the authors upon request, so any conclusions would not be implementation dependent, (ii) that were well recognized as well performing and (iii) used by many papers for comparison purposes. We selected QuickSI[35], GraphQL[24], sPath[29], TurboIso[55], and Boosted-TurboIso[56]. With respect to CFL-Match [57], its authors did not respond to our request for their code, and thus CFL-Match is not included in our experiments.

A more detailed description of the in-use algorithms can be found in §3.1.

5.3.2 Setup

Experiments with Grapes and GGSX were conducted on a small cluster consisting of 5 nodes, each featuring an Intel Core i5-3570 CPU (3.4GHz, 4 physical cores, 6MB cache), 16GB of RAM, 500GB disk per node, and running Ubuntu Linux 14.04. Experiments with QuickSI, GraphQL, sPath, TurboIso and BTI (i.e., the SI methods) were conducted on a Windows 7 SP1 host, with 2 Intel Xeon E5-2660 CPUs (2.20GHz, 20MB cache) with 8 cores/16 vcores per CPU, 128GB of RAM, and 3.5TB disk. With the term "experiment", we refer to the subgraph matching process of 1 query graph against 1 stored graph. For practical purposes, we allowed a maximum limit of 10 mins for each query to be processed. Beyond that time, the execution is terminated and we proceed with the next query in the workload. Please note that this 10' limit does not apply in the indexing phase of the individual algorithms.

For Grapes and GGSX we used the implementations provided by their respective authors. However, in the case of Grapes, we had to alter the source code so that the VF2 verification step returns after the first match of the query graph, as opposed to the original implementation which was returning all possible matches. The reason for this is that FTV methods are mainly designed to retrieve the graphs that contain the query as an answer. For QuickSI, GraphQL and sPath, we used the implementation provided by [8]. For TurboIso, we obtained the binary code from the authors and for BTI we obtained the source code from GitHub[1] where it was publicly available.

We used the default values for the input parameters of the compared algorithms, as they were defined by their respective authors in the relevant publications and/or in their implementation code. More specifically:

- For GGSX and Grapes, we enumerated paths of up to size of 4.
- We ran Grapes with 1 and 4 threads; results for executions with 1 (respectively 4) threads are denoted by Grapes/1 or GR/1 (respectively Grapes/4 or GR/4).
- For GraphQL, we used a refined level of iterations of pseudo-subgraph isomorphism $r = 4$.
- For sPath, we used a neighborhood radius of 4 and maximum path length 4.
- TurboIso does not require any input parameter. However, by default only queries up to 25 vertices are able to execute. We were not able to change this option because of the binary that was made available to us.
- BTI, too, does not require any input parameter. However, compared to the rest of the in-use SI methods, BTI only allows one label per vertex on the stored graph. Thus, we were not able to execute experiments with some of our datasets (see §5.3.3).
- For all SI methods the number of searched embeddings of the pattern graph on the stored graph is capped at 1000; i.e., after finding the first 1000 matches, the algorithms

[1] `https://github.com/UltraHector/BoostIsoGraphAdaptation`

terminate. We note that for GraphQL, sPath, QuickSI and BTI the number of embeddings is configurable, but for TurboIso it is hardcoded (an option we cannot change because of the binary made availabe to us).

A detailed description of the employed algorithms can be found in §3.1.

5.3.3 Datasets

We chose datasets which (a) have also been used by other studies, so as to enable possible direct comparisons, and (b) have key characteristics covering a large part of the design space (e.g., regarding graph size and density).

For the FTV methods, we have used PPI and the Synthetic datasets and their characteristics can be seen in Table 3.1. PPI (used in [11, 9]) is a real dataset representing 20 different protein-protein interaction networks, as discussed in §3.2.2. The majority of existing real datasets that were used for the FTV methods comprises of relatively small and sparse graphs. In [9] we showed that, for such datasets, both Grapes and GGSX perform adequately well. For our current study we are further interested in more challenging datasets and we thus employ the additional synthetic dataset generated with GraphGen[60], allowing various parameters of interest to be specified; namely, number of graphs and number of labels in the dataset, average number of nodes and density per graph). A more detailed description of how GraphGen constructs the dataset can be found in §3.2.1.

Datasets used for the SI methods consist of only one graph, as the primary task of these methods is to find all occurrences of the pattern graph in the large stored graph. Table 3.2 summarizes the characteristics of the three real datasets – namely yeast, human and wordnet – that we have used for the SI methods. Yeast and human were previously used in [8], while wordnet[2] was used in [23]. Finally, yeast and human, unlike wordnet, allow multiple labels on each node, and thus we were able to execute BTI only on wordnet, as discussed in §5.3.2.

5.3.4 Query Workloads

To generate each of the query graphs, we follow the procedure described in §3.2.3.

For the FTV methods, for the synthetic dataset, we used queries of size 24, 32 and 40 edges for Grapes/1 and Grapes/4. We did not run GGSX against the synthetic dataset, because of excessive amount of time required for the experiments to complete. For the PPI dataset, we used queries of size 16, 20, 24, and 32 edges.

For the SI methods, we used 200 queries of 10, 16, 20, 24 and 32 edges. For QuickSI we only report results against the yeast dataset, as (i) it was the easiest dataset used for

[2] `http://vlado.fmf.uni-lj.si/pub/networks/data/dic/Wordnet/Wordnet.htm`

the SI methods to process (smallest number of nodes with highest number of distinct labels compared to human and yeast as discussed in §3.2.2), and (ii) QuickSI always had many more cases, compared to GraphQL and sPath, where query processing exceeded the 10' cap (figures 5.4). Because of implementation restrictions with TurboIso and the binary that was made available to us, we were not able to execute queries with more than 25 vertices, and as a result, from the 200 constructed queries of 32-edges workload, none of them qualify for yeast and only 60 of them qualify for human dataset that we include on our experiments. BTI does not support the use of multiple labels per node. Thus, we do not present any results with BTI against the yeast and human dataset. Finally, to restrict the number of executed experiments, we opted to execute BTI instead of TurboIso as the third alternative algorithm in wordnet.

For all used methods, the majority of the queries completed in under 2". We call them *easy* queries. Another portion of queries had processing times in the 2" to 600" range; we denote these *2"-600"* queries. We use the term *completed* to refer to all queries that finished within the 10' limit; those that did not are called *hard* or *killed*.

5.3.5 Performance Metrics

For every query against a stored graph, we measure the *Execution Time*, denoted *exec time*, for both FTV and SI methods, while *avg exec time* denotes the average execution time. Specifically for this specific chapter, for FTV methods, this is the pure subgraph isomorphism time; i.e., excluding the index loading and filtering times, which add only a trivial overhead, unless stated otherwise. For FTV methods reported times are in seconds, while for SI methods times are in milliseconds, unless stated otherwise.

Let q_i be a given query and t_i^M the exec time of q_i over method M. Let also $q_{i,j}$ be the j-th isomorphic instance of q_i and $t_{i,j}^M$ the exec time of $q_{i,j}$ over method M. Finally, let $t_{i,j}^\Psi$ be the exec time of $q_{i,j}$ over our proposed Ψ-framework, and t_i^{GR-M} be the execution time of q_i over the hybrid combination of Grapes (GR) with method M over all the graphs in the dataset.

We define the (max/min) metric as: $\frac{\max_j(t_{i,j}^M)}{\min_j(t_{i,j}^M)}$. The minimum value of this metric is 1, indicating that there are no variations between the min and max exec time. The higher the value of this metric, the higher the differences between the min and max exec time achieved by the isomorphic query instances.

We also define the $speedup^*$ metric as: $\frac{t_i^M}{T}$, where T is set to: (i) $\min_j(t_{i,j}^M)$, when comparing against the various isomorphic instances of q_i, (ii) $\min_M(t_{i,j}^M)$, when comparing against different methods, and (iii) t_i^Ψ, when comparing against our Ψ-framework. $speedup^*$ represents what we lose in performance if we choose the original method over the various alternatives; i.e., $speedup^*$ equals the maximum attainable speedup over the original method, if we chose the best of the examined alternatives. For comparison purposes, for queries that were killed at the 10' limit we use this time (i.e., 600") as their minimum execution time.

Finally, both (max/min) and $speedup^*$ metric have their QLA and WLA version, as we discussed in §3.3.4, denoted with a matching subscript; e.g. $speedup^*_{QLA}$.

5.4 Straggler Queries

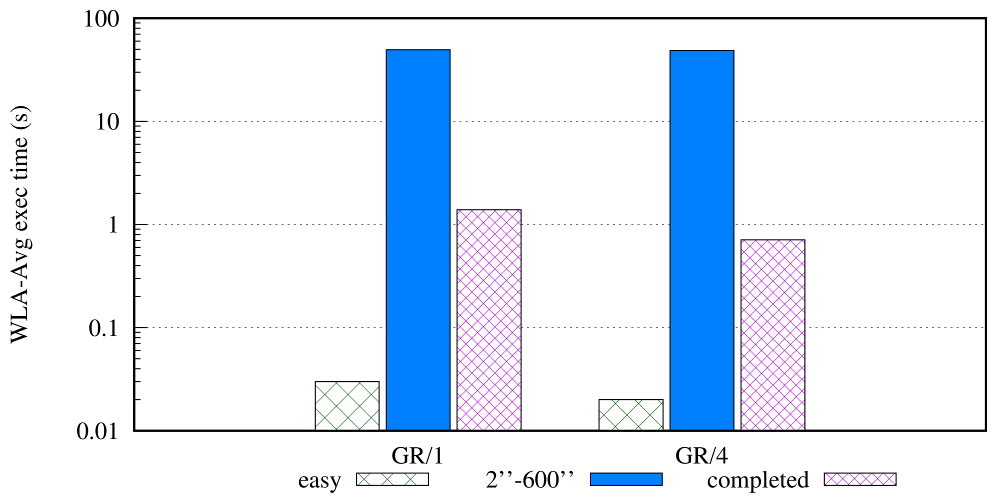
We know that as the dataset grows in terms of the size of graphs (i.e. density, number of nodes), query processing becomes harder; likewise, query processing becomes harder as the query graph incresases, as discussed in [9] and in chapter §4. But are these statements true for all queries-dataset stored graph combinations? Running many queries against the whole dataset can hide the details of how much time is required per individual graph-query pair. In the case that a small portion of such pairs dominates the whole execution time, then by just looking at the whole query workload execution times it is easy to draw wrong conclusions about the algorithms' performance. Also, several related works choose to ignore queries whose execution is much higher compared to the rest. To investigate the above, in this study we execute each individual query against a single stored graph at a time, for both FTV and SI methods.

Observation 1: In all workloads generated by us or found in other papers, our experiments show "stragglers"; i.e., queries whose processing time is many orders of magnitude higher compared to the rest.

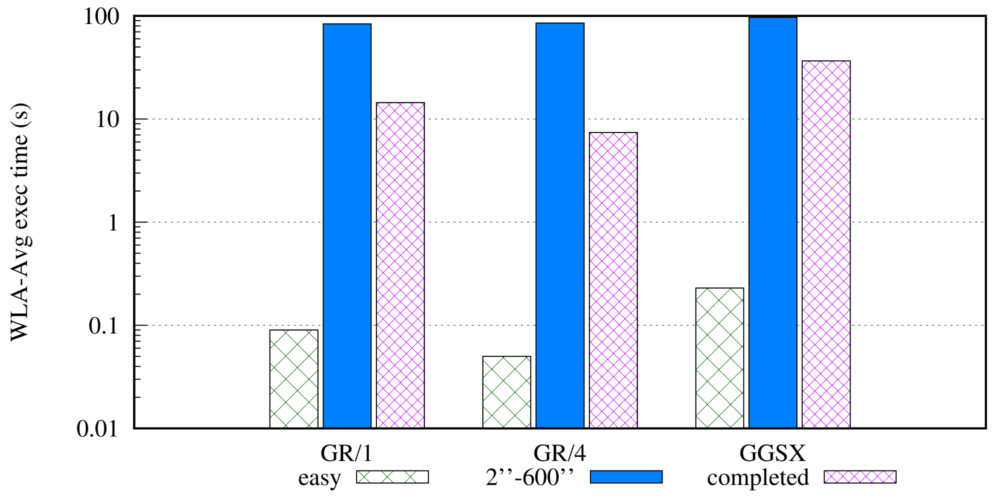
To support our observation, we present the results from our experiments on the aforementioned datasets against both FTV (figures 5.1 and 5.2) and SI methods (figures 5.3 and 5.4).

FTV methods

Figures 5.1 and 5.2 present the results from the query workloads on the FTV methods. Specifically, figures 5.1(a) and 5.1(b) show the average execution times for the corresponding algorithms for the synthetic and the PPI dataset accordingly (GGSX/synthetic results are omitted; see §5.3.2). Figure 5.2 presents the percentage of the sub-iso tests that were *easy*, *2"-600"*, and *hard* for both the synthetic and PPI datasets. As expected, Grapes/4 has a much smaller percentage of *killed* queries compared to Grapes/1 and GGSX. A notable thing here is that, the average execution time of *easy* queries is measured to some tens of



(a) Synthetic dataset, WLA-Average exec time (s)



(b) PPI dataset, WLA-Average exec time (s)

Figure 5.1: WLA-Average exec time (s) in FTV methods

milliseconds and the average execution time of *2"-600"* is hundreds of seconds, which results in an average execution time of *completed* queries measured in tens of seconds. In other words, although for both Grapes/1 and Grapes/4 the percentage of *2"-600"* queries is only $< 5\%$ in the synthetic dataset and $< 10\%$ in PPI, the average execution time across all completed queries is significantly affected; that is, the most expensive queries dominate the overall execution time.

SI methods

Figures 5.3 and 5.4 present the results from the query workloads on the SI methods (QuickSI-[human/wordnet], TurboIso-wordnet and BTI-[human/yeast] results are omitted; see §5.3.2), while tables 5.1 and 5.2 give the average execution times and percentages for

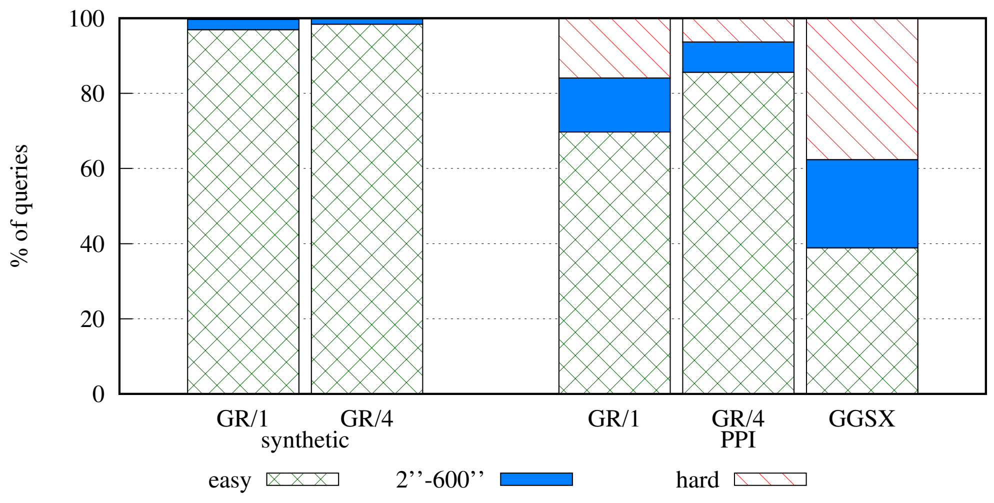


Figure 5.2: Percentages of *easy*, *2”-600”*, and *hard* queries in FTV methods

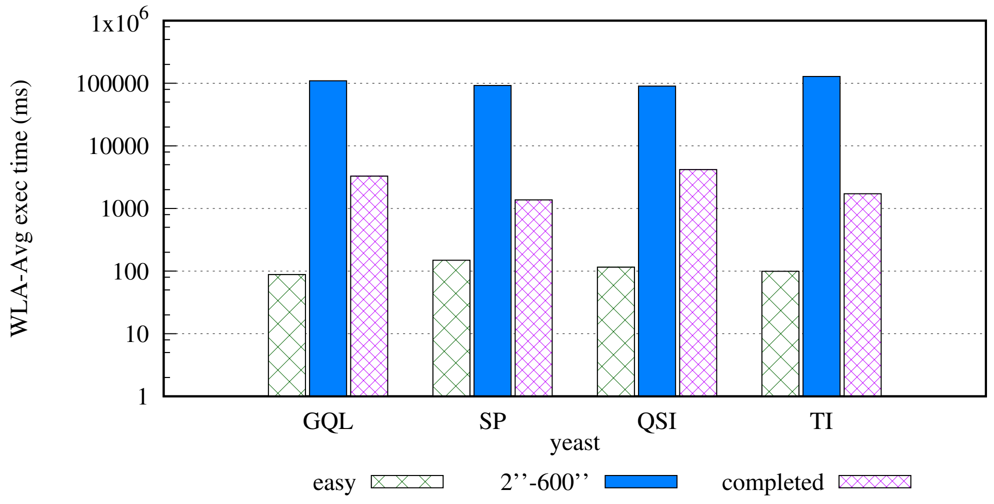
		GraphQL	sPath	QuickSI	TurboIso
10-edge *q*	AET *easy* (ms)	66.84	134.78	131.67	92.69
	% of *easy*	100	99.5	99	99.5
	AET *2”-600”* (ms)	-	2871.44	50367.40	-
	% of *2”-600”*	0	0.5	1	0
	% of *hard*	0	0	0	0.5
32-edge *q*	AET *easy* (ms)	130.66	120.71	96.62	-
	% of *easy*	80	91	67.5	-
	AET *2”-600”* (ms)	140812	140781	78917.2	-
	% of *2”-600”*	6.5	3	6	-
	% of *hard*	13.5	6	26.5	-

Table 5.1: Results for SI methods on the yeast dataset (AET: Average exec time)

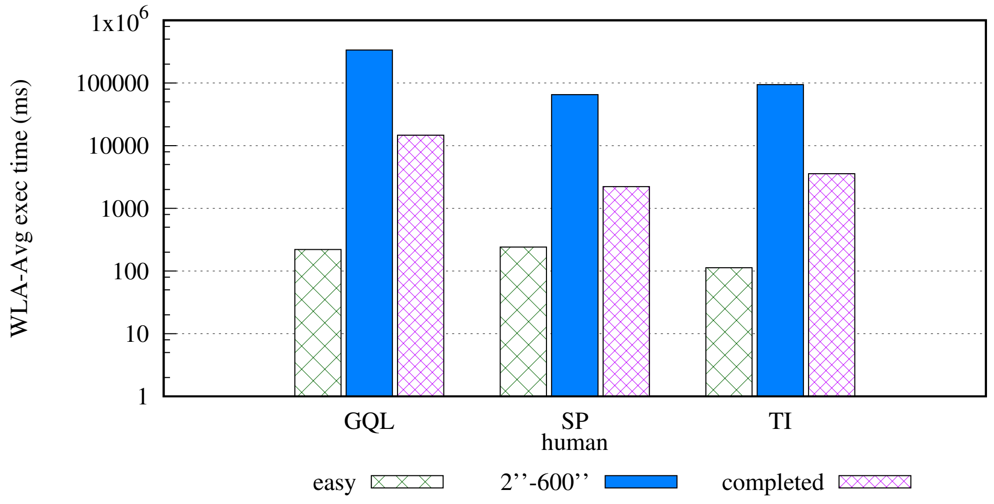
		GraphQL	sPath	TurboIso
10-edge *q*	AET *easy* (ms)	179.49	209.91	87.04
	% of *easy*	100	98	100
	AET *2”-600”* (ms)	-	182392	-
	% of *2”-600”*	0	1	0
	% of *hard*	0	0	0
32-edge *q*	AET *easy* (ms)	246.31	277.13	173.35
	% of *easy*	71.5	84.5	73.33
	AET *2”-600”* (ms)	93523.7	31817	148.695
	% of *2”-600”*	4.5	4.5	8.33
	% of *hard*	24	11	18.33

Table 5.2: Results for SI methods on the human dataset (AET: Average exec time)

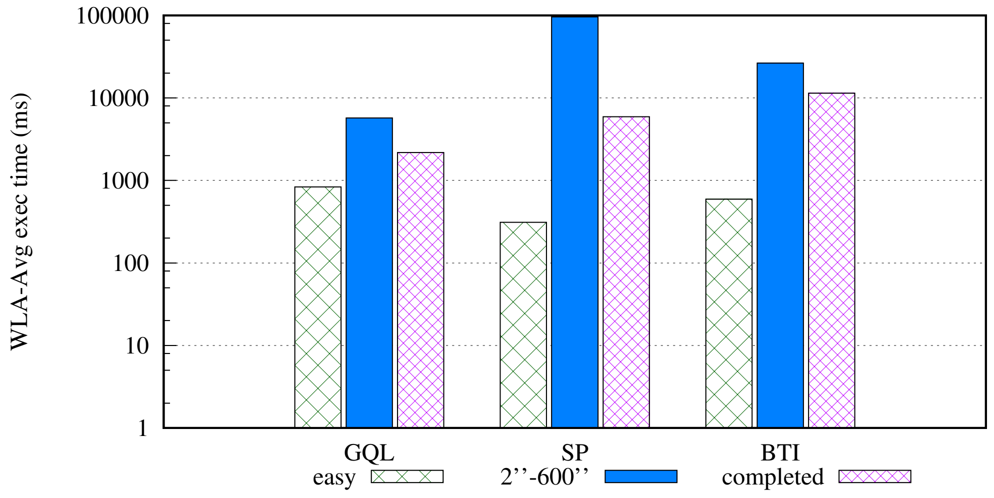
10- and 32-edge *easy*, *2”-600”* and *hard* queries for the yeast and human datasets. We use the 10-edge query results to compare our findings with those presented in iGraph-v2[8] and for the common algorithms, i.e., GraphQL, sPath and QuickSI. iGraph-v2 used small query



(a) yeast dataset, WLA-Average exec time (ms)



(b) human dataset, WLA-Average exec time (ms)



(c) wordnet dataset, WLA-Average exec time (ms)

Figure 5.3: WLA-Average exec time (s) in SI methods

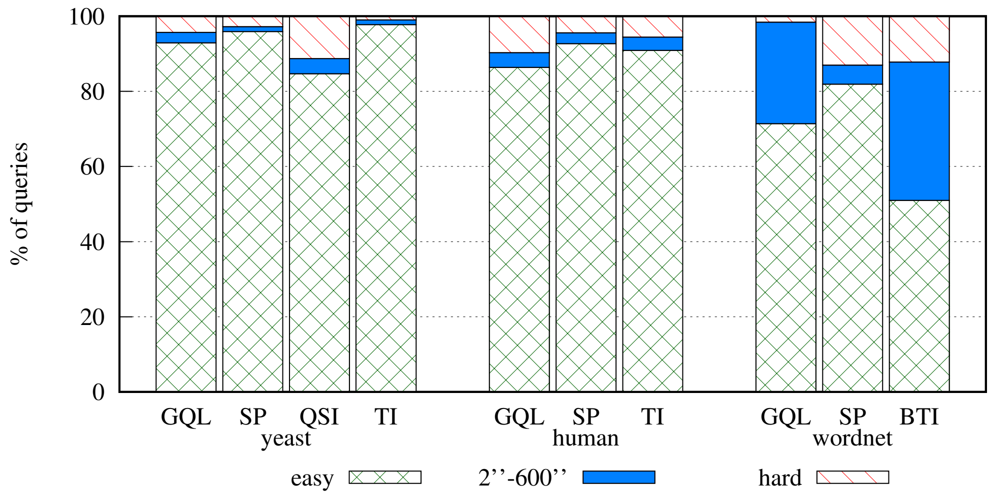


Figure 5.4: Percentages of *easy*, *2”-600”*, and *hard* queries in SI methods

sizes (up to 10 edges) and showed that the best performing algorithm is GraphQL, because it managed to complete all tested query workloads. With our experiments, we confirm this for both yeast and human datasets and for queries of size 10 edges. GraphQL performs the best among the three, having also 0% of *hard* queries. However, the picture is totally reversed when looking at the rest of the queries.

Specifically, regarding table 5.1 and the yeast dataset, for 10-edge queries, results for sPath and QuickSI are comparable, with GraphQL having 0% of *hard* queries, whereas TurboIso is already lagging behind with 0.5% of *hard* queries. The same is valid for the *easy* queries of 32 edges. However, the picture is totally reversed when looking at the rest of the queries. In this case, the percentage of *killed* queries is double for GraphQL and quadruple for QuickSI compared to sPath.

Additionally, regarding table 5.2 and the human dataset, where QuickSI is replaced with TurboIso (§5.3.2), GraphQL and TurboIso perform better compared to sPath that already has a 1% 10-edge query that takes *2”-600”* to execute. The picture is again reversed for all three competing algorithms as sPath proves to be the most robust at 32-edge queries, with only 11% of *hard* queries.

Because of implementation restrictions, we were not able to execute queries of $>$25 vertices with TurboIso (see §5.3.2); thus no queries of 32 edges were executed for yeast and only 60 queries of 32 edges qualified for human. As a result, the presented results for TurboIso on human and yeast are not directly comparable with the numbers provided for GraphQL and sPath. However, we note that for the same 60 32-edge queries executed over human dataset on all 3 algorithms, TurboIso encountered more *hard* queries than the other two contestants. Finally, although BTI is the latest proposed method, it suffers from more *hard* queries compared to GraphQL and sPath for the wordnet dataset.

GraphQL and sPath are the 2 alogrithms executed against all 3 datasets (figures 5.3 and 5.4, tables 5.1 and 5.2). Based on our experiments, sPath performs overall better than GraphQL having (i) smaller average execution times on the completed queries and (ii) smaller percentages of *hard* queries in yeast and human datasets. However, in wordnet this behavior is reversed. Thus, it's very difficult to claim that one algorithm is performing better than the other. In fact, in order to claim that, we need to define a performance metric of interest. Such a performance metric could be the percentage of killed queries, but that still depends on the time limit imposed on query processing. For example, in wordnet, if the threshold was 2", then sPath would be better than GraphQL, but if we change this threshold the picture changes.

We summarize our results to the following 3 conclusions: (1) Some queries are *hard*. (2) Different algorithms have different percentages of completed queries; thus, different algorithms find different queries hard. (3) As the most expensive queries dominate the average execution time, one must include a sufficient number of hard queries in order to draw conclusions about the relative performance of the algorithms.

5.5 Isomorphic queries

Various proposed SI methods ([24, 29, 35]), as well as iGraph-v2[8] that compares them, observe that the search order on the query can have a huge impact on query processing time. We agree with this claim. This behavior is typical on other NP-hard problems, as discussed in §2.7 ([116]). In the current study, we take a further step and instead of relying on the order that the individual method imposes, we generate our own isomorphic query rewritings. To achieve this, we maintain the structure of the query graph and the labels on the nodes unchanged, and we permute the node IDs. Subsequently, we transform the query graph to an input format compatible with each individual method and we perform the query processing. The same effect of constructing isomorphic query graphs to the original query graph can be achieved by permuting rows/columns of the original graph in an adjacency matrix representation. In the following experiments, we use a total of 6 different rewritings per query, leading to the following observation.

Observation 2: Queries which are isomorphic to the original query graph can have widely and wildly different execution times.

We attribute this behavior to the fact that all proposed methods do not define an absolutely strict order in which the nodes of the query are matched, as it would be too computationally expensive to compute a globally optimal join plan. Thus, all proposed methods, similarly to other NP-hard problems [116], rely on heuristics (see §5.3.1) in order to minimize the search space for the join plan. However, given the fact that SI methods define a more strict order in which the nodes of the query are matched, this wide execution time variation (as we will see later on in this chapter) was not initially expected.

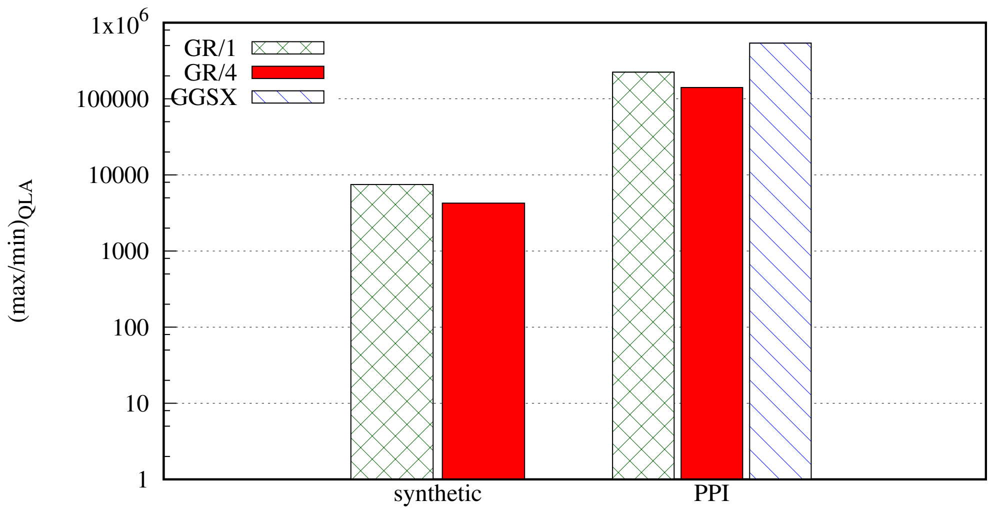


Figure 5.5: Average $(max/min)_{QLA}$ for FTV methods

		Grapes/1	Grapes/4	GGSX
synthetic	stdDev	86,700.40	65,988.40	-
	min	1.06	1.02	-
	max	3,820,000.00	3,490,000.00	-
	median	3.90	4.45	-
PPI	stdDev	469,934	395,285	1,020,000
	min	1.03	1.02	1.01
	max	3,680,000	3,160,000	12,000,000
	median	1,186.51	11.19	109,086.00

Table 5.3: $(max/min)_{QLA}$ statistics for FTV methods

FTV methods

Figure 5.5 depicts the QLA average value of the (max/min) metric for the synthetic and PPI datasets, for the FTV methods (GGSX results are omitted for the synthetic dataset; see §5.3.4), whereas figure 5.6 presents the $(max/min)_{QLA}$ for different query sizes on the PPI dataset. Table 5.3 additionally reports the stdDev, min, max and median values of $(max/min)_{QLA}$. In the calculations, we did not include a small number of queries that were not helped by any of the isomorphic instances tried; i.e., queries that were *hard* on all tested isomorphic instances of the query. This behavior occurred in 0.0036% and 1.4% of queries for Grapes/1 on the synthetic and PPI datasets respectively, in 0.37% of queries

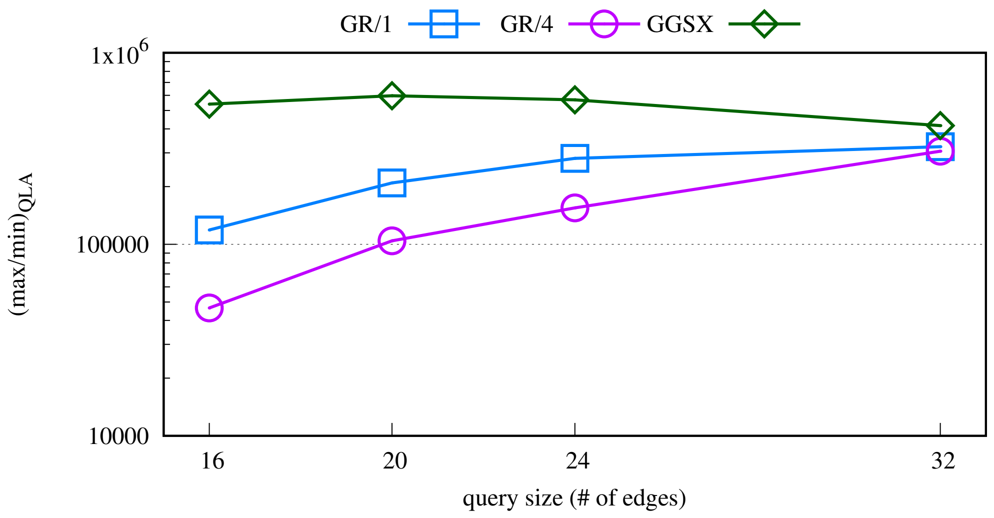


Figure 5.6: Average $(max/min)_{QLA}$ for different query sizes on FTV methods and for PPI dataset

for Grapes/4 and 1.96% of queries for GGSX for the PPI dataset. Note that the "max" and "average" values of $(max/min)_{QLA}$ are only lower-bound estimations, because of the 10' limit that we used instead of the actual verification time. In these results, we observe that there is an at least 6 orders of magnitude difference between the min and the max value of $(max/min)_{QLA}$, with the median (apart from GGSX) being closer to the min value. Along with the high stdDev, we can see that isomorphic instances of the same query can indeed have widely and wildly different verification times.

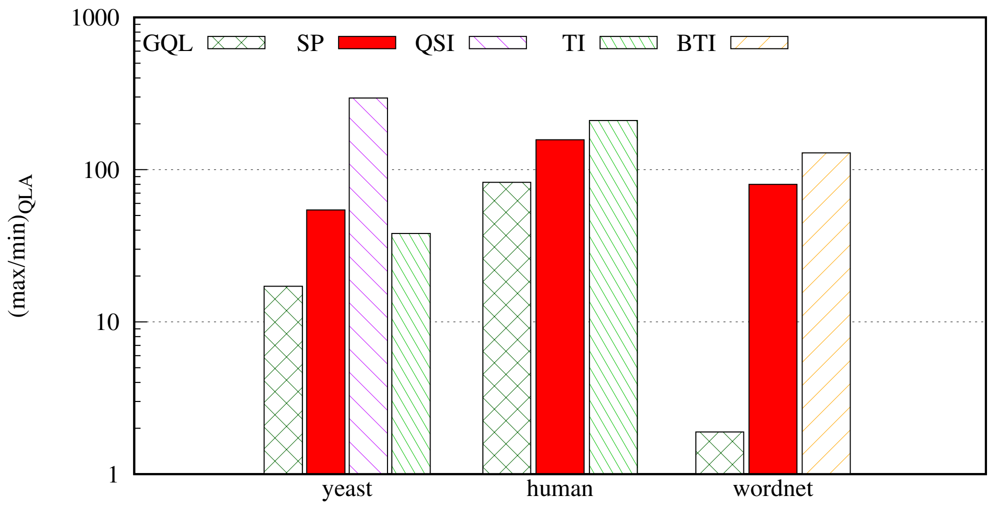


Figure 5.7: Average $(max/min)_{QLA}$ for SI methods

		GraphQL	sPath	QuickSI	TurboIso	BTI
yeast	stdDev	287.5	533.8	1685.7	704.7	-
	min	1.0	1.0	1.0	1.0	-
	max	7286.3	6695.8	15021.6	14053.4	-
	median	1.4	1.3	1.6	2.1	-
human	stdDev	440.1	662.7	-	1214.7	-
	min	1.0	1.0	-	1.0	-
	max	4115.0	4087.8	-	9030.0	-
	median	1.8	1.9	-	1.9	-
wordnet	stdDev	20.5	396.8	-	-	1635.1
	min	1.0	1.0	-	-	1.0
	max	646.4	3081.1	-	-	31121.8
	median	1.2	1.3	-	-	1.1

Table 5.4: $(max/min)_{QLA}$ statistics for SI methods

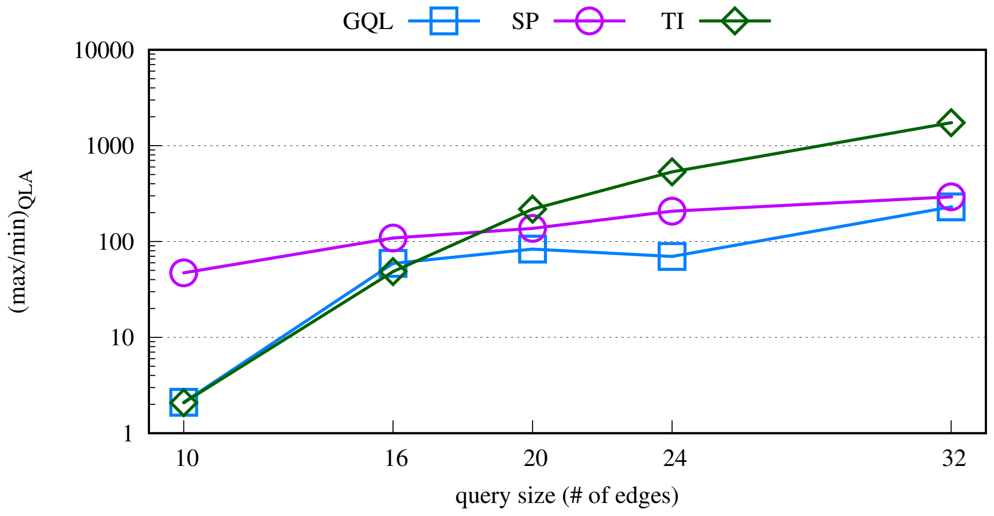


Figure 5.8: Average $(max/min)_{QLA}$ for different query sizes on SI methods and for human dataset

SI methods

Figure 5.7 reports the QLA-average values of the (max/min) metric for the yeast, human and wordnet datasets, for the tested SI methods (QuickSI-[human/wordnet], TurboIso-wordnet and BTI-[human/yeast] results are omitted; see §5.3.4). Additionally, figure 5.8 presents the $(max/min)_{QLA}$ for different query sizes on the PPI dataset. Table 5.4 reports the stdDev, min, max and median value of $(max/min)_{QLA}$. We report that 4.2%, 8.2% and 1.5% of queries were not helped by any tested isomorphic query instances for GraphQL and for yeast, human and wordnet respectively. For sPath the corresponding values are 2.1%, 1.4% and 11.8%, and for QuickSI 8.6% of the queries were not helped for the yeast dataset. For TurboIso 0.88% and 4.19% of the queries were not helped for yeast and human respectively and for BTI and wordnet the corresponding value is 7.7%.

The QLA-average (max/min) for the SI methods is up to 3 orders of magnitude lower than that of the FTV methods. This is somewhat expected as the SI methods define a more strict order in which the nodes of the query are matched and thus leave less space for wild variations. However, this order is still significantly affected by the initial node ids of the query, and thus we still see per-query (max/min) values of up to 2 orders of magnitude.

We summarize our overall results to the following conclusions: (1) For every isomorphic test to be executed, given a query graph q and a stored graph, there is an isomorphic version of q that can take anywhere from 2 to 6 orders of magnitude more time to execute compared to the least expensive version of the query. This holds across all algorithms and datasets tested. This is a typical behavior of other algorithms proposed for NP-hard problems, as discussed in §2.7. Specifically, we have seen that these algorithms also employ different heuristics and their performance is significantly affected by the search order they impose. (2) The harder the queries (higher query sizes), the higher these number are, as it can be seen by figures 5.6 and 5.8.

5.6 Graph query rewriting

Having established that isomorphic versions of a query can have dramatically different execution times, we set out to construct our own specific rewritings, constructing graphs isomorphic to the original queries, with the aim to capture these benefits. We have developed and experimented with several such query rewritings. We outline below five such rewritings, all performed by carefully permuting the node IDs in the query graph: The discussion assumes a query graph and a stored graph.

- `Query Rewriting ILF (`*`Increasing Label Frequency`*`):`
  In a preprocessing step, we compute the frequencies of all node labels in the stored graph, sorted in increasing frequency order. Given this order, we produce a rewriting of the query graph so if i, j are the node IDs of query graph nodes n_i, n_j, $L(n_i), L(n_j)$ are their labels, and $f(L(\cdot))$ is the frequency of a label $L(\cdot)$ in the stored graph, then $f(L(n_i)) < f(L(n_j)) \Rightarrow i < j$. Ties can appear in 2 cases: (i) two or more query nodes have the same label, or (ii) two or more query nodes have different labels but with the same frequency. These ties are broken arbitrarily.
- `Query Rewriting IND (`*`Increasing Node Degree`*`):`
  The nodes of the query are sorted in increasing node degree order; i.e., if n_i, n_j are two query graph nodes, and $d(\cdot)$ is the degree (number of edges) of a node, then $d(n_i) < d(n_j) \Rightarrow i < j$. In the case of nodes with the same number of edges, ties are broken arbitrarily.

- `Query Rewriting DND` (*Decreasing Node Degree*):
  This rewriting is similar to the IND rewriting but the nodes of the query are sorted in decreasing node degree and the node ids are assigned accordingly.
- `Query Rewriting ILF+IND`:
  This rewriting is the same as ILF above, with ties being broken in an IND manner: i.e., nodes with smaller outgoing degree get a lower node id.
- `Query Rewriting ILF+DND`:
  This rewriting is the same as ILF+IND, with ties being broken in a DND manner.

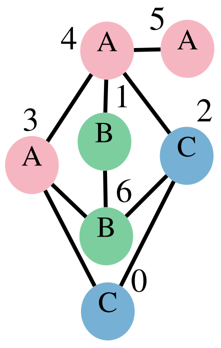
In order to conduct our experiments, and to perform our query rewriting, based on the initial query node IDs of the original query, we break ties by respecting the order of nodes on the original queries. In other words, let n_i and n_j be two query graph nodes with node IDs $i < j$, that belong in the same tie group; i.e. they have the same label or the same degree, or both the same label and degree. Then, in the rewritten query the node IDs will be $i' < j'$. Undoubtedly, this is not the only way to resolve ties; for example we could randomly assign node IDs on the nodes that belong in the same tie group. Additionally, although our current tie-breaking approach is deterministic, it is still based on the initial assignment of IDs to nodes, which is in turn arbitrary and random in its own right, e.g., any random rewriting could have been the original form of the query graph.

In §3.1.2 and §5.5, we discussed how the various methods decide the optimal join plan so as to match the query vertices to the vertices on the stored graph. However, we have seen that in some cases ties (symmetries) still exist, in other cases that not all possible join plans are considered. For example, QuickSI tries to resolve symmetries by making the constructed MST denser, whereas GraphQL considers only left-deep join plans to define a good search order. However, ties still exist and it is expected that these ties are resolved by the various algorithms by looking at the query node IDs. Thus, intuitively, and in case of ties, given that node ID $i < j$, it is expected that n_i will be matched earlier than node n_j. Time-wise, this might have a tremendous effect on the algorithm's performance depending on the number of the actual join operations to consider.

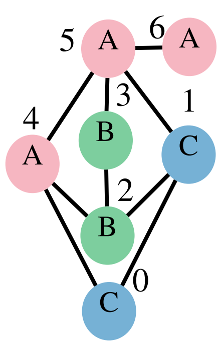
Figure 5.9 presents an example of the above rewritings. Note that the ILF+IND rewriting in 5.9(d) is another valid isomorphic ILF rewriting. Ties are (utterly) broken in an arbitrary way, and thus one may compute several different isomorphic graphs for the same rewriting.

Indicatively[3], in figures 5.10 and 5.11 we report the WLA average processing times of the original query and the 5 proposed query rewritings for the PPI and yeast datasets, as well as the corresponding percentages of the *hard* queries. For the FTV methods, the best performing rewritings are ILF and ILF+DND, with the percentage of *hard* queries be-

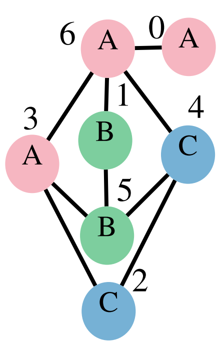
[3]We obtained similar results for the synthetic dataset for the FTV methods and the human and wordnet datasets for all used SI methods. The sole exception was sPath, whose percentage of *hard* queries increased slightly for the wordnet dataset.



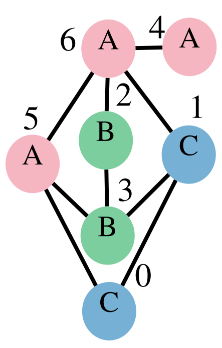
(a) Original



(b) ILF

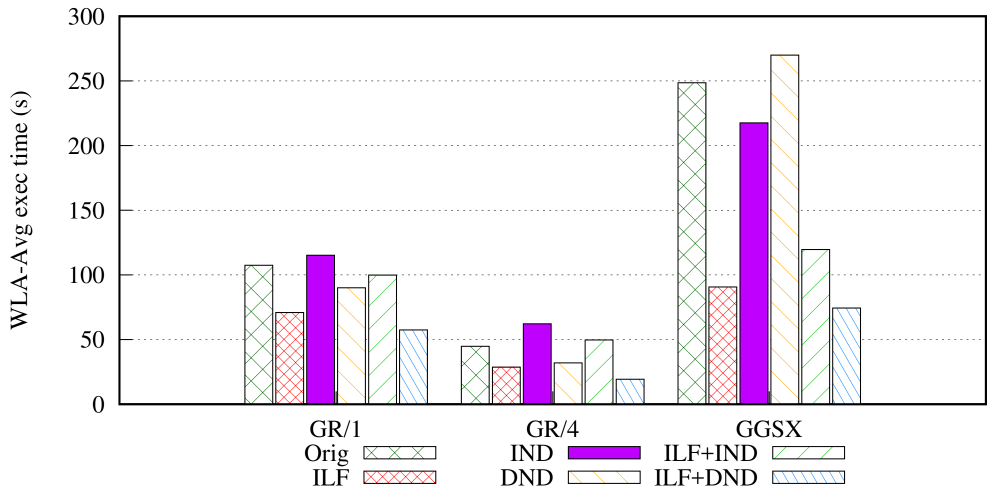


(c) IND

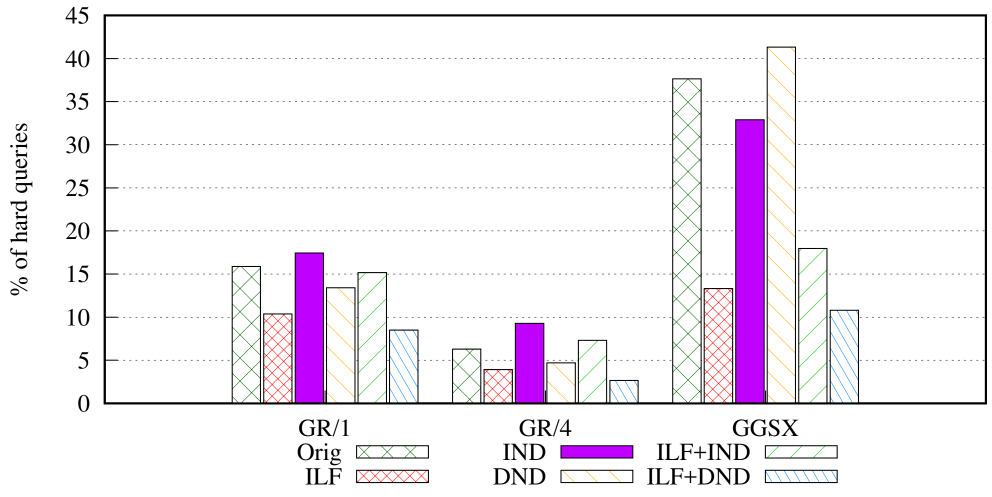


(d) ILF+IND

Figure 5.9: Isomorphic queries generated with different rewritings (assuming the label frequencies in the stored graph are: “A”=20, “B”=15, “C”=10)

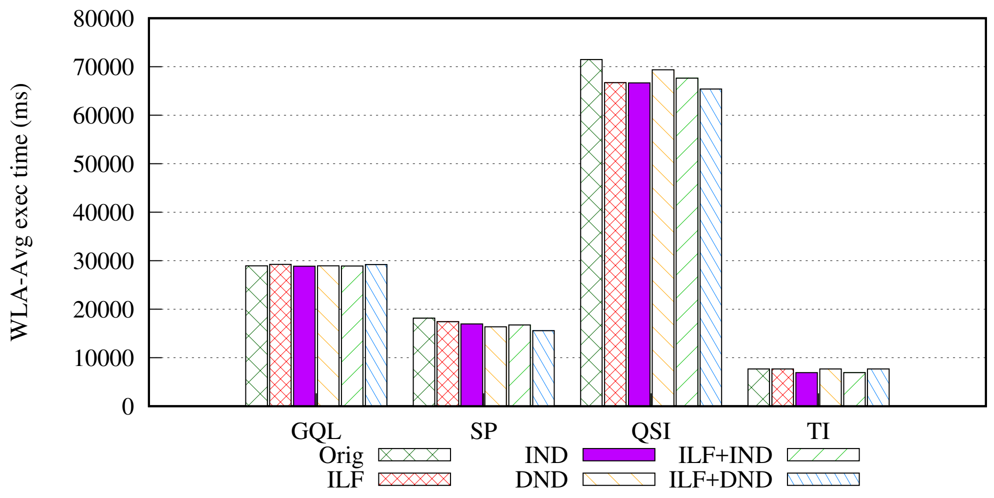


(a) WLA-Average exec time (s)

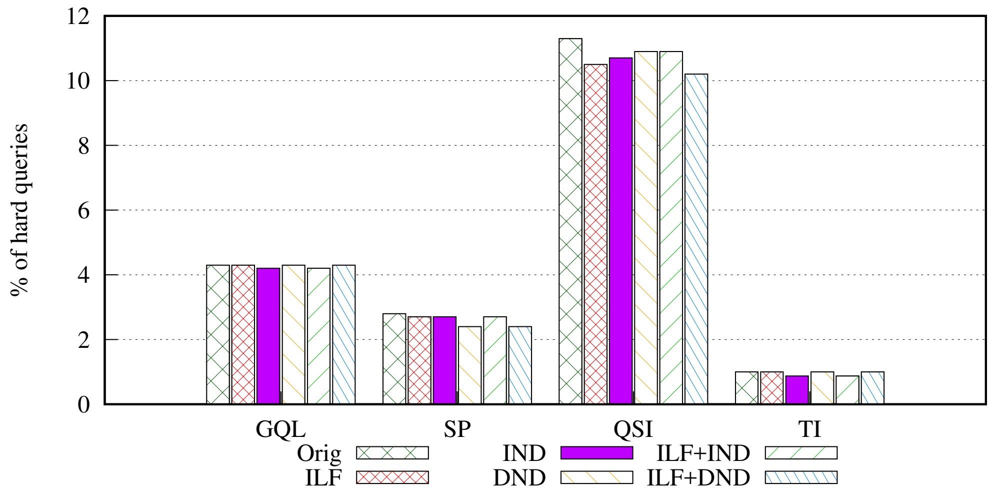


(b) percentage of *hard* queries

Figure 5.10: Results for individual query rewritings for FTV on PPI dataset



(a) WLA-Average exec time (ms)



(b) percentage of *hard* queries

Figure 5.11: Results for individual query rewritings for SI methods on yeast dataset

ing significantly improved. For the SI methods, the picture is slightly different. GraphQL shows no considerable improvement for each rewriting individually; as a matter of fact, there are rewritings leading to higher average execution times than the original query does. For sPath, the DND and ILF+DND rewritings reduce the percentage of killed queries from 2.8% to 2.4%. For QuickSI, ILF+DND reduced the percentage of killed queries from 11.3% to 10.2%, but DND only brings it down to 10.9%. For TurboIso, IND and ILF+IND reduce the percentage of killed queries from 1% to 0.875%. Note that in the case of TurboIso, results are not directly comparable with the rest of the contestants because no queries of 32 edges are included (see §5.3.4). More importantly, note that there is no single rewriting that manages to improve all algorithms across all datasets and workloads. However, if we were to execute in parallel various isomorphic rewritings, we would achieve large performance gains.

Observation 4: "Stragglers" can have isomorphic counterparts which are not stragglers. Table 5.5 quantifies the percent reduction of straggler queries for both FTV and SI methods when using isomorphic query counterparts for the tested datasets.

		GR/1	GR/4	GGSX	GQL	SP	QSI	TI	BTI
FTV	synthetic	98.8%	100%	-	-	-	-	-	-
	PPI	91.1%	94.1%	94.7%	-	-	-	-	-
SI	yeast	-	-	-	2.32%	25%	23.9%	12%	-
	human	-	-	-	15.5%	68.2%	-	24.9%	-
	human	-	-	-	6.3%	9.2%	-	-	36.9%

Table 5.5: Percent reduction of straggler queries for FTV and SI methods using isomorphic query counterparts

Figures 5.12 and 5.14 present the achieved $speedup^*$ of such an execution. Similarly, as in §5.5, the max and average reported $speedup^*$ represent a lower-bound estimation because of the value 10' that we use for the *hard* queries that were killed. Additionally, in our calculations we do not include the few queries that were killed in both the original instance and in all the rewritings of the query (see §5.5).

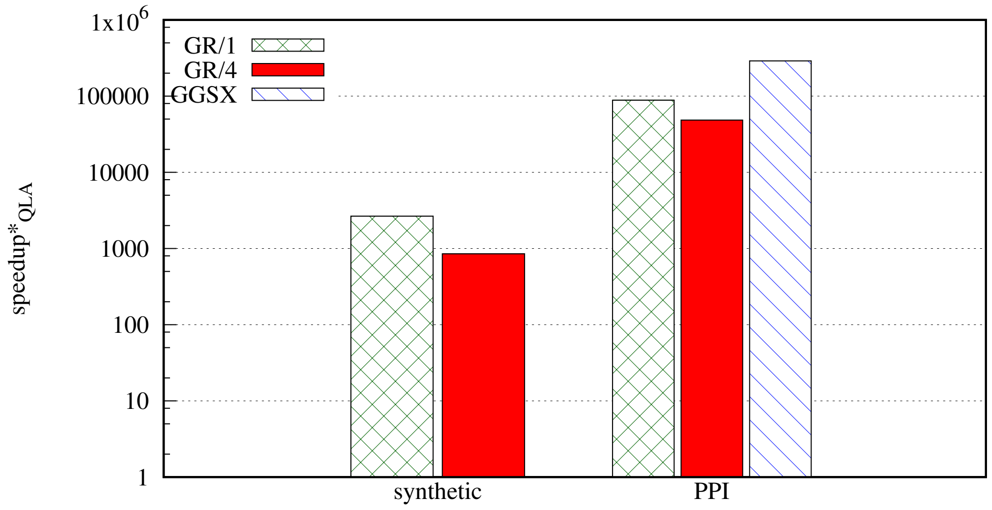


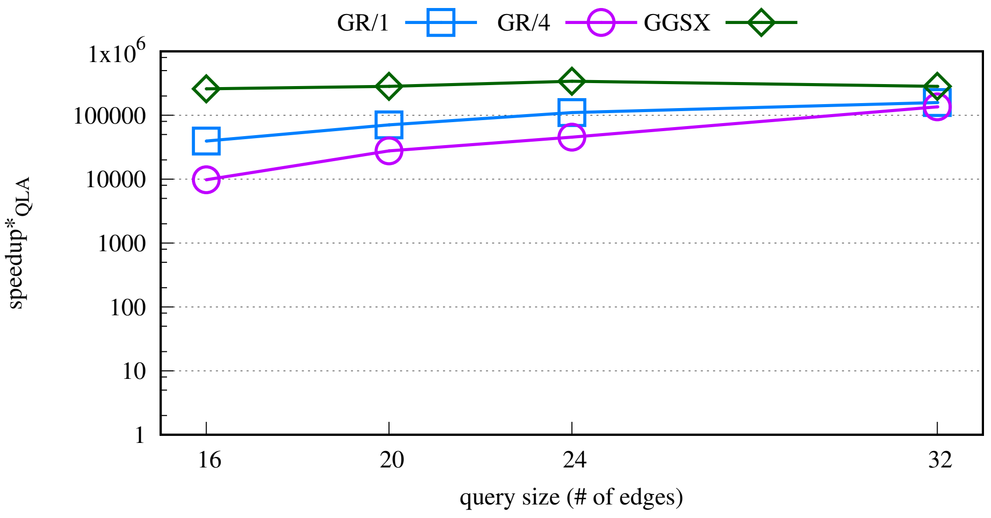
Figure 5.12: Average $speedup^*{}_{QLA}$ for FTV methods across rewritings

FTV methods

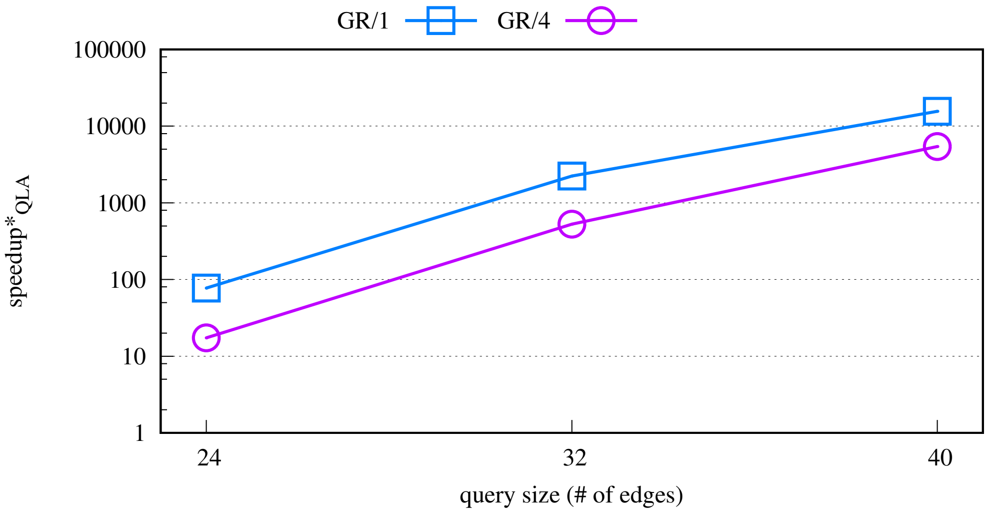
Figure 5.12 presents the average $speedup^*{}_{QLA}$ for the FTV methods for the synthetic and PPI datasets (GGSX/ synthetic results are omitted; see §5.3.4). Additionally, table 5.6 reports the QLA stdDev, min, max and median of $speedup^*{}_{QLA}$. Moreover, as we increased the size of the queries, $speedup^*{}_{QLA}$ increased by up to 5 orders of magnitude, as it can be seen by figure 5.13. Regarding the presented results in table 5.6, median $speedup^*{}_{QLA}$ is close to

		Grapes/1	Grapes/4	GGSX
synthetic	stdDev	53,785.70	24,267.60	-
	min	1.00	1.00	-
	max	3,820,000	2,110,000	-
	median	1.36	1.24	-
PPI	stdDev	302,250	237,573	758,668
	min	1.00	1.00	1.00
	max	3,370,000	2,910,000	9,390,000
	median	3.71	1.67	1,751.22

Table 5.6: $speedup^*_{QLA}$ statistics for FTV methods across rewritings



(a) PPI dataset



(b) Synthetic dataset

Figure 5.13: Average $speedup^*_{QLA}$ for different query sizes on FTV methods

min $speedup^*_{QLA}$, evidencing again a wide variation in the benefits of the isomorphic query rewritings. Keeping in mind that the majority of the queries are *easy* (figure 5.2), we come to the conclusion that large performance gains can come from improving the *hard* queries. Finally, the percentages of the queries that were not helped by any of the rewritings are the same as presented in §5.5.

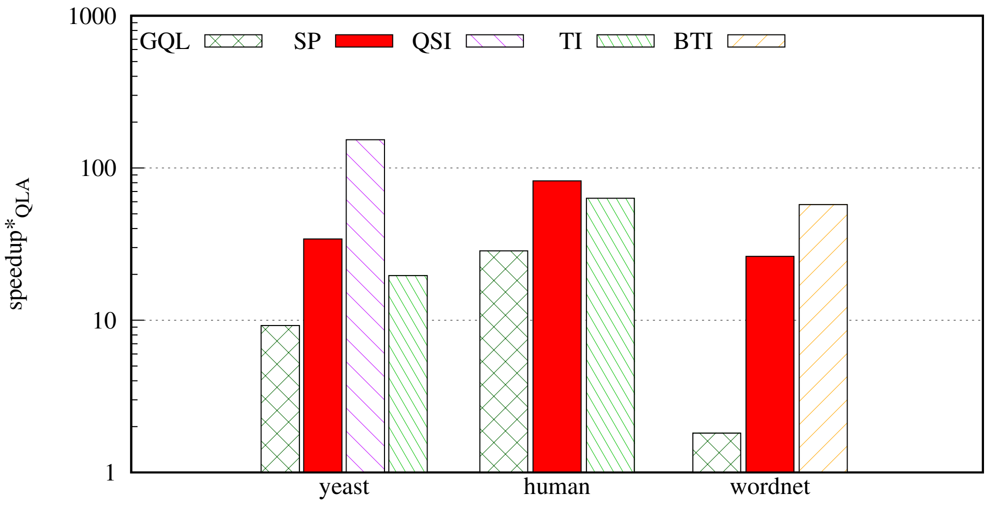


Figure 5.14: Average $speedup^{*}_{QLA}$ for SI methods across rewritings

		GraphQL	sPath	QuickSI	TurboIso	BTI
yeast	stdDev	235.6	422.5	1193.0	498.7	-
	min	1.0	1.0	1.0	1.0	-
	max	7286.3	6695.8	15021.6	14053.3	-
	median	1.1	1.1	1.3	1.7	-
human	stdDev	259.9	492.4	-	695.1	-
	min	1.0	1.0	-	1.0	-
	max	4115.1	4087.8	-	8919.7	-
	median	1.1	1.1	-	1.1	-
wordnet	stdDev	20.5	244.6	-	-	1029.7
	min	1.0	1.0	-	-	1.0
	max	646.4	3081.1	-	-	31121.8
	median	1.1	1.1	-	-	1.0

Table 5.7: $speedup^{*}_{QLA}$ statistics for SI methods across rewritings

SI methods

Figure 5.14 presents the average $speedup^{*}_{QLA}$ for the SI methods for the yeast, human and wordnet datasets (QuickSI-[human/wordnet], TurboIso-wordnet and BTI-[human/ yeast] results are omitted; see §5.3.4). Table 5.7 reports the stdDev, min, max and median of the $speedup^{*}_{QLA}$. The percentages of the queries that were not helped by any rewriting are in accordance with those reported in §5.5. The performance of sPath could seemingly be improved by one to two orders of magnitude across all datasets. The same holds for

QuickSI, TurboIso and BTI on the corresponding datasets. GraphQL could also be improved by more than a factor of $10\times$ on the yeast and human datasets. However, no significant improvement was possible for GraphQL on wordnet. The reason why this is so, is somewhat subtle. Apart from what the algorithms do internally to match the query, other culprits are the characteristics of the actual stored graphs and the generated queries. Looking at the statistics of the graphs (table 3.2), yeast and especially wordnet are very sparse graphs with small average node degree. Thus, the majority of the generated queries are paths, where the rewritings based on node degrees are not effective in this case. Additionally for wordnet, the small number of labels (only 5) and the distribution of the frequencies of the labels being highly skewed leads to the generation of queries that in their majority contain only 1 or 2 labels, with the second label appearing only once. As a result, the rewritings are of little use in these cases.

5.7 Algorithm-specific Stragglers

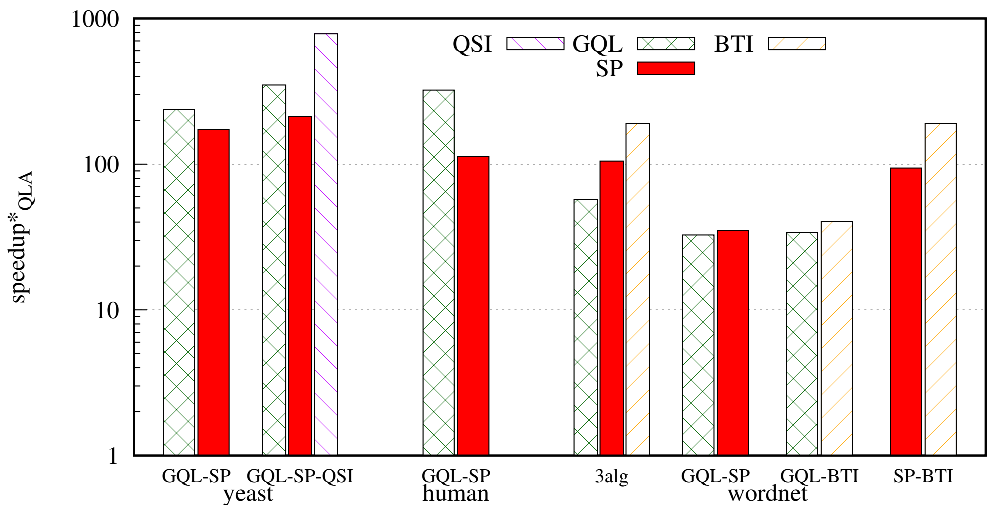
In §5.4, we notice that for the SI methods, different algorithms have different percentages of *hard* queries. With our experiments, we elaborated more on that and we found that different algorithms find different queries *hard*.

Observation 5: "Stragglers" are algorithm-specific; i.e., by evaluating the same query workloads with various algorithms, we have seen that a "straggler"-query for one algorithm can be a typical query for some other algorithm.

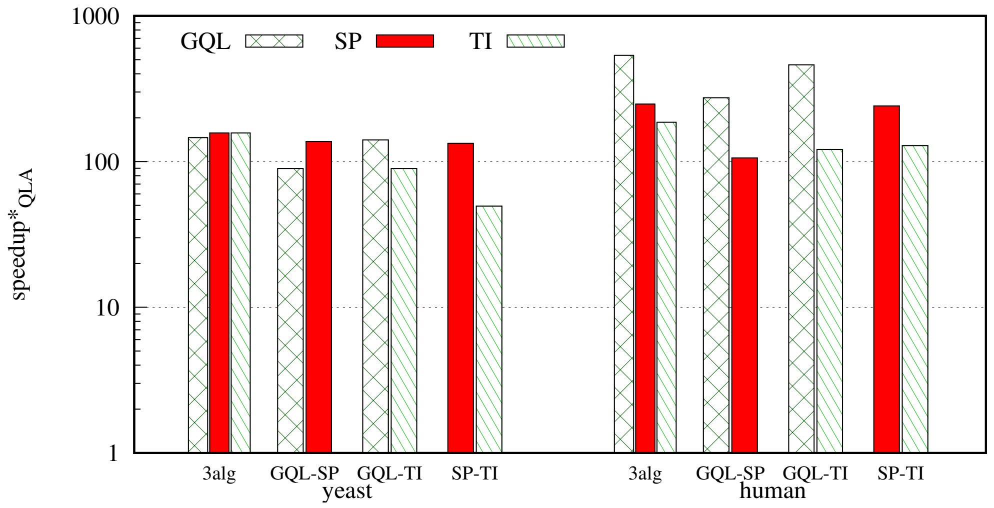
		GraphQL	sPath	QuickSI
yeast_{2alg}	stdDev	1094.57	1051.65	-
	min	1.00	1.00	-
	max	9189.36	9129.60	-
	median	1.00	1.80	-
yeast_{3alg}	stdDev	1596.47	1255.34	2162.97
	min	1.00	1.00	1.00
	max	13060.10	12403.70	12312.70
	median	1.00	1.88	1.32
human	stdDev	1394.34	570.83	-
	min	1.00	1.00	-
	max	30873.80	4341.44	-
	median	1.00	1.04	-

Table 5.8: $speedup^*{}_{QLA}$ statistics when utilizing different algorithms on SI methods for yeast and human

Here, we present the results of our observation for the various SI methods. But before presenting the achieved $speedup^*{}_{QLA}$, we recall the formula of calculating the $speedup^*$



(a) yeast, human, wordnet with all query sizes



(b) yeast & human using GraphQL, sPath and TurboIso, (only queries that qualify for TurboIso, i.e. no queries with >25 vertices)

Figure 5.15: Average $speedup^*_{QLA}$ when utilizing different algorithms on SI methods

metric as presented in §3.3.3. Specifically in our case, let $t_i^{\{\mathcal{M}\}}$ and t_i^M the execution time of q_i over a set of methods $\{\mathcal{M}\}$ and the execution time of q_i over a method M respectively. Then the $speedup^*$ metric is calculated as: $\frac{t_i^{\{\mathcal{M}\}}}{t_i^M}$. In other words, the $speedup^*$ represents what we gain in performance if we choose a set of methods $\{\mathcal{M}\}$ over a single alternative method M.

Figure 5.15 presents the average $speedup^*_{QLA}$ for the yeast, human and wordnet datasets and for the tested algorithms and query workloads. Note that TurboIso cannot execute queries of >25 vertices, as discussed in §5.3.4. Thus, in the presented results, figure 5.15(a) includes all 3 datasets with the algorithms that were able to be executed against all different query sizes and these dataset-algorithms combinations are yeast-[GraphQL/ sPath], yeast-[GraphQL/ sPath/ QuickSI], human-[GraphQL/ sPath], wordnet-[GraphQL/ sPath/ BTI] (ti-

		GraphQL	sPath	BTI
3alg	stdDev	486.72	699.97	668.07
	min	1.00	1.00	1.00
	max	11061.40	12776.40	9376.22
	median	4.51	1.00	4.79
GQL-SP	stdDev	253.56	104.42	-
	min	1.00	1.00	-
	max	3733.78	932.58	-
	median	2.47	1.00	-
GQL-BTI	stdDev	445.85	-	155.64
	min	1.00	-	1.00
	max	11061.40	-	3278.35
	median	1.00	-	1.47
SP-BTI	stdDev	-	714.31	683.56
	min	-	1.00	1.00
	max	-	12776.40	9376.22
	median	-	1.00	3.42

Table 5.9: $speedup^{*}_{QLA}$ statistics when utilizing different algorithms on SI methods for wordnet

tled as 3alg), wordnet-[GraphQL/ sPath], wordnet-[GraphQL/ BTI], wordnet-[sPath/ BTI], whereas figure 5.15(b) includes results for yeast and human and for GraphQL, sPath and TurboIso but only for the graph query sizes that qualify for TurboIso (as discussed in §5.3.4), i.e. the algorithm combinations are [GraphQL/ sPath/ TurboIso] (titled as 3alg), [GraphQL/ sPath], [GraphQL/ TurboIso], [sPath/ TurboIso]. For example, the first bar (from left) on figure 5.15(a) represents the $speedup^{*}_{QLA}$ achieved when using the set of methods $\{\mathcal{M}\}$ [GraphQL/sPath] instead of method M GraphQL on yeast dataset. In tables 5.8 and 5.9, we additionally report the stdDev, min, max and median of $speedup^{*}_{QLA}$ for GraphQL, sPath, QuickSI and BTI and the corresponding datasets; presented results are relevant to figure 5.15(a). We note that for the 5.15(b), where not all query sizes are included because of the implementation restrictions with TurboIso, the corresponding stdDev, min, max and median of $speedup^{*}_{QLA}$ follow similar trends as above.

With the use of multiple algorithms, there were very few query executions that were not helped by any of the employed algorithms. Thus, in figure 5.15(a), for the yeast dataset, we note that all cases were helped by either the use of 2 (GraphQL and sPath) or 3 (GraphQL, sPath, QuickSI) algorithms. In human only 0.8% were not helped by the GraphQL-sPath combination, and in wordnet 0.1% of the queries were not helped by the same algorithms' combination. Including BTI in wordnet, all queries are helped, whereas the GraphQL-BTI combination leads to 0.2% of hard queries and the sPath-BTI combination to 4.7% hard queries. Thus, looking at these results in comparison with figures 5.3 and 5.4, we note that BTI is one of the least performing algorithms, but when combined with other better performing algorithms large performance gains can be achieved.

Similar results are obtained for yeast and human and for the GraphQL, sPath, TurboIso combinations, as presented in figure 5.15(b). Specifically, any combination of algorithms in yeast and the all-3-algorithm combination in human leads to straggler-free query executions. In human, GraphQL-sPath combination leads to 0.35%, GraphQL-TI to 1.51% and sPath-TI to 0.12% *hard* queries.

Finally, the $speedup^*{}_{QLA}$ values for using multiple algorithms are higher compared to the $speedup^*{}_{QLA}$ values achieved with multiple query rewritings (see §5.6). This leads to the conclusion that the use of multiple algorithms could be way more beneficial compared to the rewritings, which are not always effective (§5.6).

5.8 The Ψ-framework

In this subsection we present how we incorporate our findings in a novel framework that exploits parallelism. The proposed framework is called Ψ-framework (**P**arallel **S**ubgraph **I**somorphism framework). Unlike recent related work [30, 23], by having different threads/ machines working on different versions of the problem our Ψ-framework exploits parallelism in a novel way. We utilize Grapes and GGSX as top-performing FTV and GraphQL and sPath as top-performing SI methods that were additionally able to execute against all three considered datasets and query sizes. We do not include TurboIso, BTI and QuickSI as (1) in many cases they showcase worse performance than the incorporated GraphQL and sPath, and (2) because of the implementation restrictions as described in §5.3.1. Within our Ψ-framework we have incorporated the original implementations of Grapes and GGSX as provided by their authors, and of GraphQL and sPath as found in [8].

In the FTV methods we leave the index construction and the filtering stages intact during query processing. In the verification stage, for every graph in the candidate set, we instantiate a number of threads equal to the number of the isomorphic-query rewritings we utilize. These threads run in parallel with each being assigned one rewriting of the initial query, and the first thread to finish is the "winner"; i.e., the rest of the threads are killed and the algorithm proceeds with the verification of the next graph in the candidate set.

Ψ-framework for the SI methods works similarly to the verification stage of the FTV methods. However, we mentioned in observation 5 that stragglers disappear when using an alternative matching algorithm. We incorporate this finding in our Ψ-framework by running simultaneously two threads: one for sPath and one for GraphQL with the original query. Again after the completion of the fastest thread, the rest of them are killed. Additionally, because of the finding that different query rewritings can have widely different execution times, we further enhance Ψ-framework as follows: we run simultaneously multiple threads with each thread being assigned a pair of [algorithm] - [original query and/or rewritings of the original query]. This will be further discussed later in this section.

The recent trends in cpu design go towards multicore/manycore systems, and as such, our proposed approach bears great potential. This is yet another way to harness the parallelism available at the hardware level, not via parallelizing individual algorithms, which might be hard, error prone or outright impractical [119], but via executing algorithms and/or isomorphic instances in parallel.

The aforementioned technique of killing the rest of the threads after the completion of the fastest one is not entirely new. The logic is similar to the branch and bound search where the goal is to find the optimal path between the root node and the goal. In order to achieve this, during space exploration, we can safely prune a path that is more costly than the currently optimal one, as discussed in §2.7.

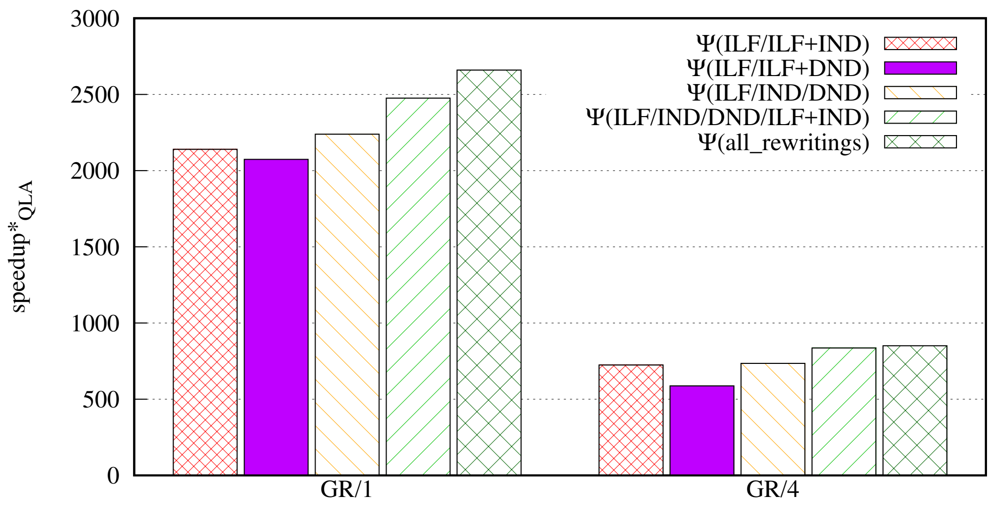
On the one hand we have seen that the more the isomorphic instances we use, the better the speedup we gain in the graph matching process. On the other hand, the instantiation and synchronization of many threads come with a non-trivial overhead, impacting the overall speedup. To this end, in our performance evaluation we report on the speedup achieved by several beneficial combinations of rewritings. We note that our Ψ-framework is of course not the only solution to the straggler-queries' problem. Undoubtedly, it would be preferable to choose the right isomorphic query instance and/or algorithm to use to minimize the query execution time. However, given the complex nature of the sub-iso problem, we leave such design decisions for future work.

The proposed Ψ-framework shares the same concept idea of portfolios of algorithms as discussed in §2.7 and in [116]. This is indeed a common technique in NP-hard problems where (i) the execution times of proposed algorithms can have a wide variance and (ii) there is not a single winning algorithm across all instances of the whole spectrum. We will also see in our experiments that some combinations of algorithms pay-off better than others.

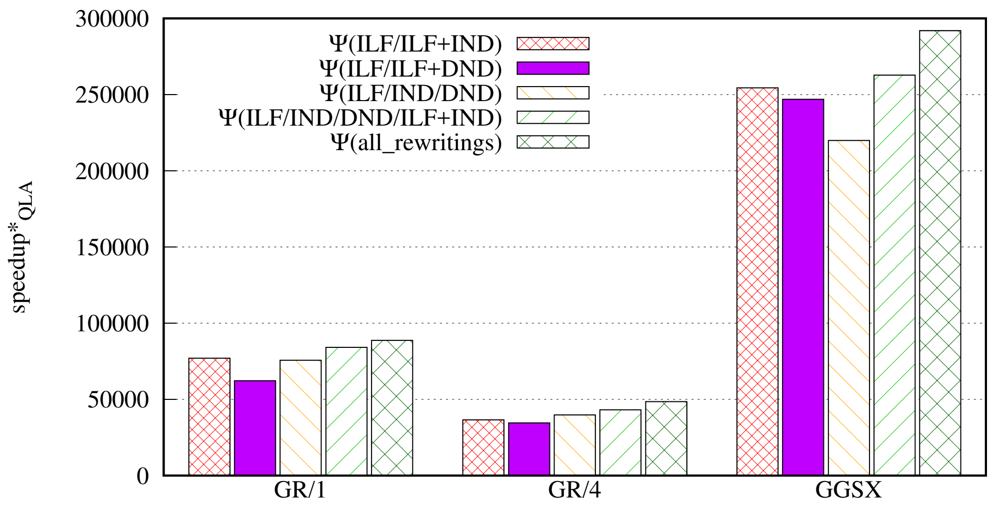
The cost of producing the query rewritings was measured from a few tens (for the smallest query sizes) to a few hundreds (for the biggest query sizes) of μsecs; being a negligible overhead to the overall query processing time, we ignore it in the figures and omit any further discussion of this cost factor.

FTV methods

Figures 5.16 and 5.17 present the average $speedup^*_{QLA}$ and average $speedup^*_{WLA}$ respectively for utilizing different versions of Ψ-framework on the FTV methods. Specifically, we present the average $speedup^*_{QLA}$ and average $speedup^*_{WLA}$ of the following versions of Ψ-framework: (a) ILF/ ILF+IND (2 threads), (b) ILF/ ILF+DND (2 threads), (c) ILF/



(a) Synthetic dataset

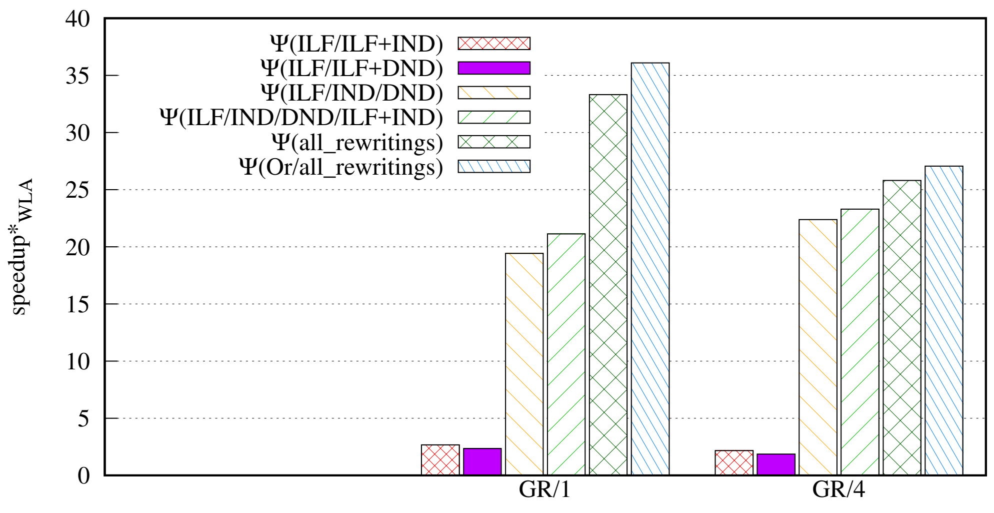


(b) PPI dataset

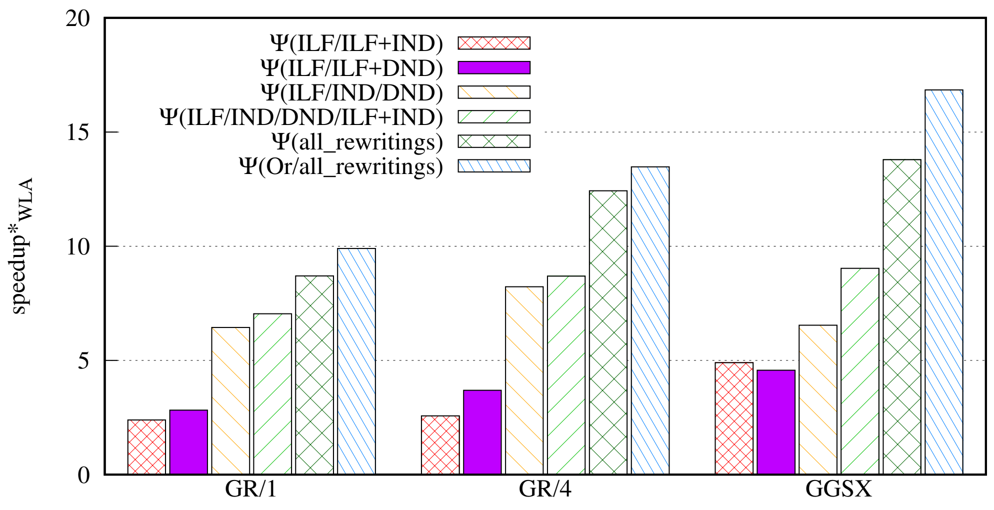
Figure 5.16: Average $speedup^*_{QLA}$ across different versions of our framework on the FTV methods

	Synthetic		PPI		
	GR/1	GR/4	GR/1	GR/4	GGSX
Original Algorithm	0.299%	0.131%	15.888%	6.298%	37.6362%
Ψ(ILF/ ILF+IND)	0.100%	0.061%	6.441%	2.333%	7.168%
Ψ(ILF/ ILF+DND)	0.122%	0.066%	5.353%	1.603%	7.798%
Ψ(ILF/ IND/ DND)	0.007%	0.000%	2.261%	0.715%	5.182%
Ψ(ILF/ IND/ DND/ ILF+IND)	0.007%	0.000%	2.061%	0.658%	3.681%
Ψ(all_rewrtings)	0.003%	0.000%	1.631%	0.458%	2397%
Ψ(Or/ all_rewrtings)	0.003%	0.000%	1.402%	0.372%	1.961%

Table 5.10: Percentage of killed queries of FTV methods and different versions of our Ψ-framework. (Or stands for original query.)



(a) Synthetic dataset



(b) PPI dataset

Figure 5.17: Average $speedup^*{}_{WLA}$ across different versions of our framework on the FTV methods

IND/ DND (3 threads), (d) ILF/ IND/ DND/ ILF+IND (4 threads) and (e) all 5 possible rewritings (5 threads). Table 5.10 reports the percentage of *killed* queries for the original query against the used FTV algorithms and the different used versions of our Ψ-framework. Our framework proves highly beneficial for all algorithms and datasets. As it was expected, by increasing the number of threads running multiple rewritings on the Ψ-framework, not only the average execution time is significantly improved but also the percentage of *hard* queries is decreased, even leading to straggler-free executions. However, note that the Ψ-framework(ILF/ IND/ DND) (3 threads) is only 3-8% worse compared to Ψ-framework(ILF/ IND/ DND/ ILF+IND) (4 threads) for Grapes/1 and Grapes/4.

As Grapes is designed as a multi-threaded application, we additionally compare Grapes/4 against our Ψ-framework running Grapes/1 with the following four rewritings (for a total of

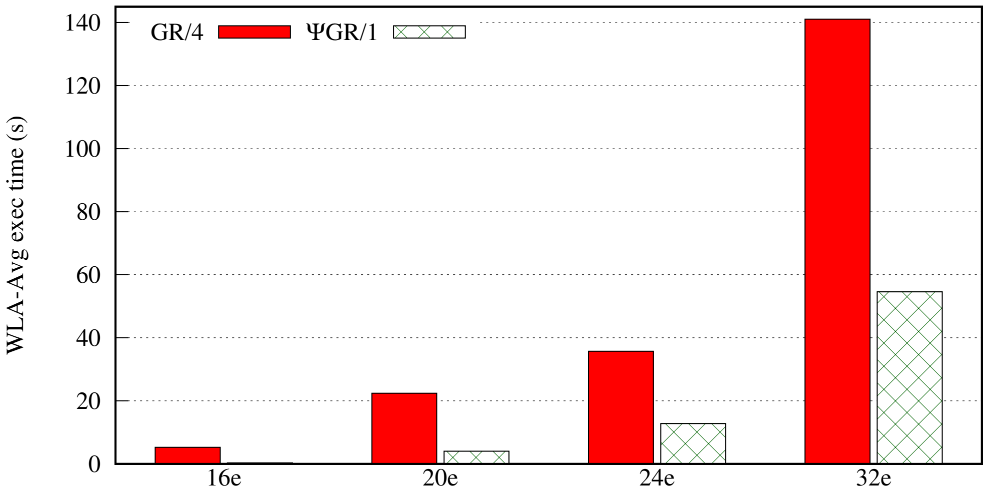


Figure 5.18: Comparison of average execution time over the PPI dataset, for Grapes/4 against the Ψ-framework with 4 rewritings (ILF, IND, DND, ILF+IND) over Grapes/1

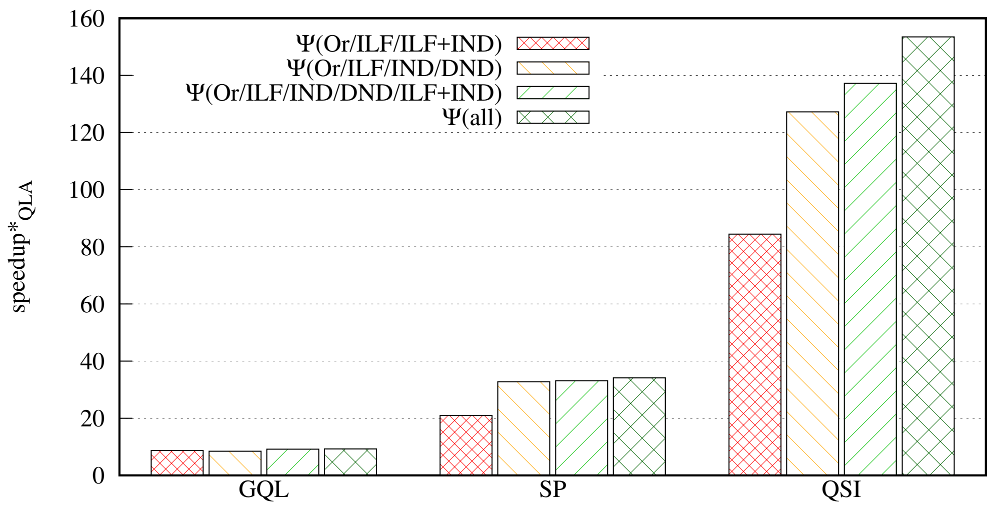
4 threads as well): ILF, IND, DND, ILF+IND. The results are presented in figure 5.18 for the PPI dataset (results for the synthetic dataset were similar). The corresponding percentages of *killed* queries are: 6.298% for Grapes/4 and 2.061% for Ψ(ILF/ IND/ DND/ ILF+IND) (as reported in table 5.10). As it can be conceived, although both contenders have the same level of parallelism, Ψ-framework makes better use of its threads and leads to lower query processing times.

	yeast			human		wordnet	
	GQL	SP	QSI	GQL	SP	GQL	SP
Original Algorithm	4.3%	2.8%	11.3%	10%	4.4%	1.6%	13%
Ψ(Or/ ILF/ ILF+IND)	4.2%	2.4%	9.4%	8.4%	2.2%	1.5%	11.8%
Ψ(Or/ ILF/ IND/ DND)	4.2%	2.1%	8.9%	8.3%	1.4%	1.5%	11.8%
Ψ(Or/ ILF/ IND/ DND/ ILF+IND)	4.2%	2.1%	8.8%	8.2%	1.4%	1.5%	11.8%
Ψ(all)	4.2%	2.1%	8.6%	8.2%	1.4%	1.5%	11.8%

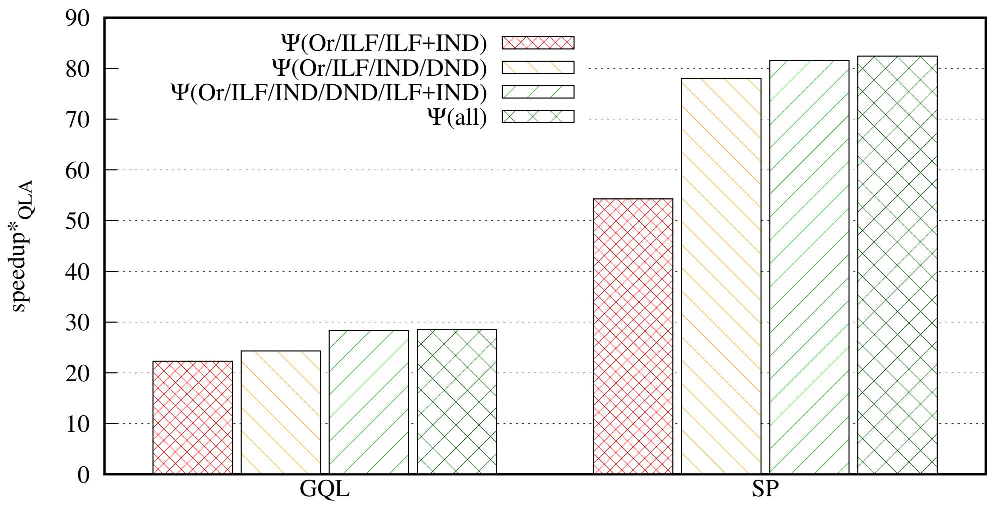
Table 5.11: Percentage of killed queries of SI methods and different versions of our Ψ-framework. (Or stands for original query.)

SI methods

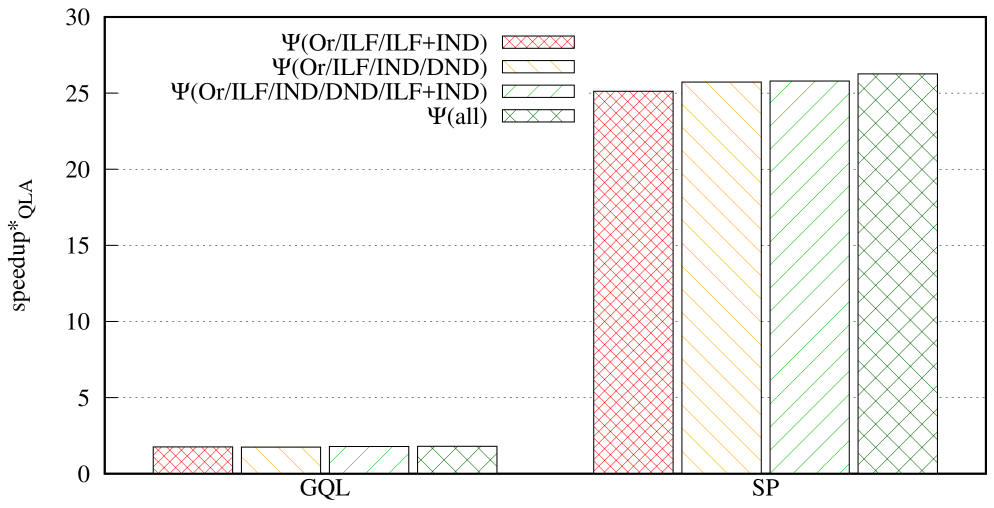
Figure 5.19 presents the average $speedup^*_{QLA}$ for utilizing different versions of Ψ-framework on the SI methods. We utilize the following versions of Ψ-framework and the corresponding number of threads: (a) Orig/ ILF/ ILF+IND (3 threads) (b) Orig/ ILF/ IND/ DND (4 threads), (c) Orig/ ILF/ IND/ DND/ ILF+IND (5 threads), and (d) Orig + all-rewritings (titled as all) (6 threads). Table 5.11 presents the percentage of *killed* queries for the original query against the in use SI algorithm and the different used versions of our Ψ-framework.



(a) yeast dataset



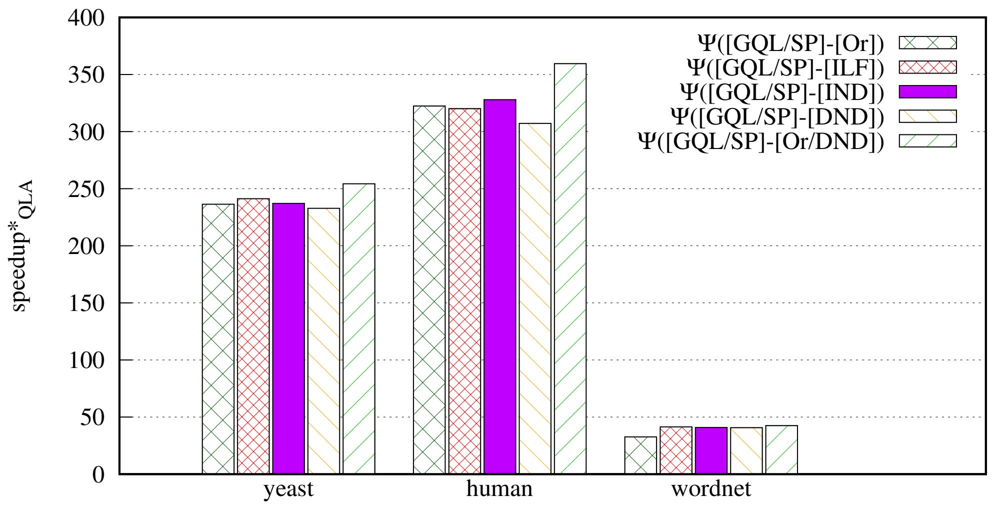
(b) human dataset



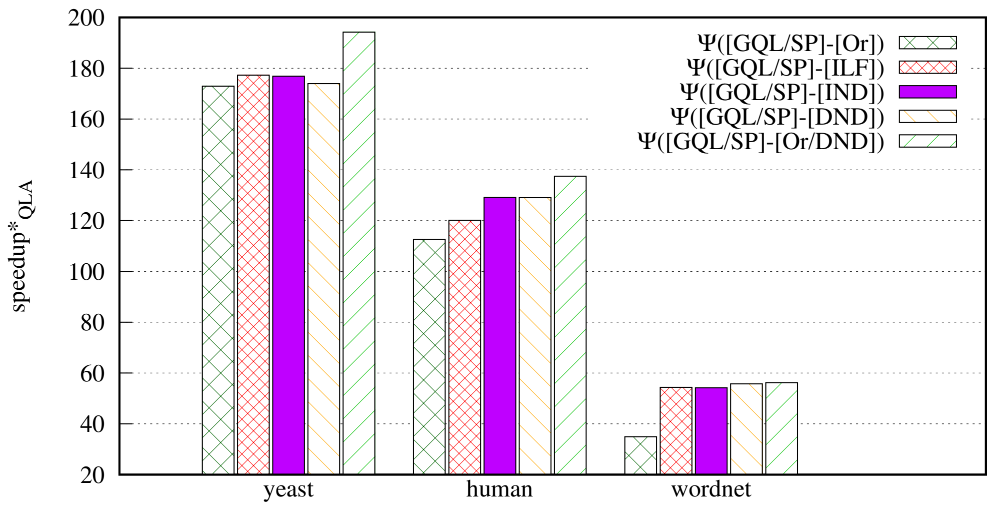
(c) wordnet dataset

Figure 5.19: Average $speedup^*_{QLA}$ across different versions of Ψ-framework on the SI methods

For all tested datasets and workloads, GraphQL was benefited the least from the rewritings. The biggest improvements appear in the human dataset. We attribute this to the fact that this dataset comprises of a denser graph with more labels, thus a larger portion of *hard* queries was benefited from our rewritings and framework. Apart from the case of QuickSI in yeast, the rest of the algorithms do not accomplish significant speedups by increasing the number of threads in most cases. This is particularly evident in Ψ-framework(Orig/ ILF/ IND/ DND/ ILF+IND) with 5 threads and Ψ-framework(all) which operates with 6 threads. Although, Ψ-framework is able to achieve large performance gains, the large number of *killed* queries is still prevalent.



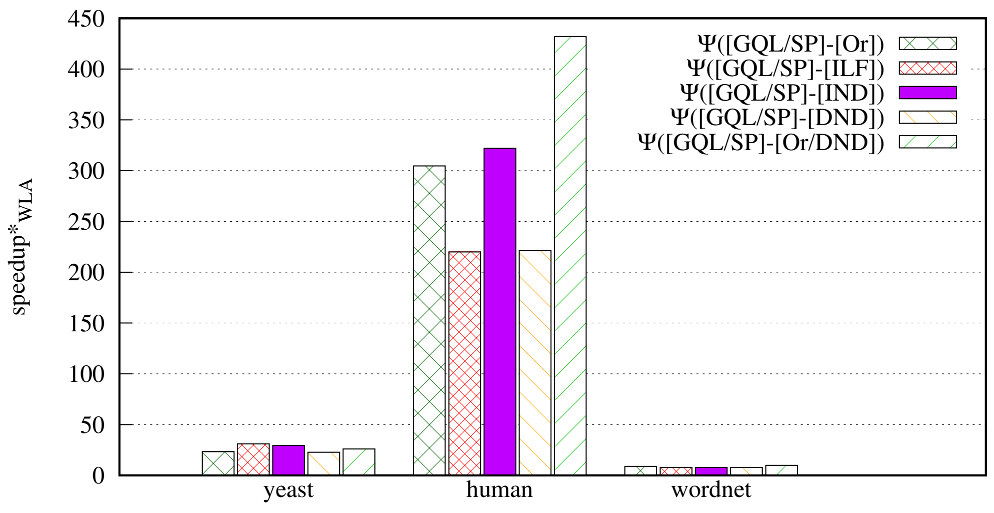
(a) $speedup^*_{QLA}$ for GraphQL



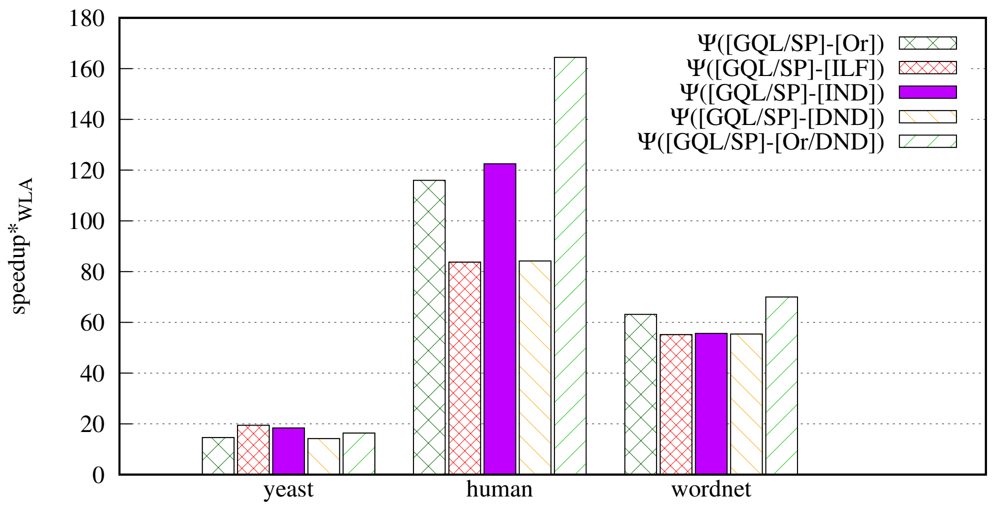
(b) $speedup^*_{QLA}$ for sPath

Figure 5.20: Average $speedup^*_{QLA}$ for running multiple algorithms against SI methods on Ψ-framework

Finally, figures 5.20 and 5.21 depict the average $speedup^*_{QLA}$ and the average $speedup^*_{WLA}$ for utilizing different algorithms and different versions of Ψ-framework on the SI meth-



(a) $speedup^*_{WLA}$ for GraphQL



(b) $speedup^*_{WLA}$ for sPath

Figure 5.21: Average $speedup^*_{WLA}$ for running multiple algorithms against SI methods on Ψ-framework

	yeast	human	wordnet
GraphQL	4.3%	10%	1.6%
sPath	2.8%	4.4%	13%
Ψ([GQL/SP]-[Or])	0%	0.8%	0.1%
Ψ([GQL/SP]-[ILF])	0%	0.8%	0%
Ψ([GQL/SP]-[IND])	0%	0.6%	0%
Ψ([GQL/SP]-[DND])	0%	0.9%	0%
Ψ([GQL/SP]-[Or/DND])	0%	0.7%	0%

Table 5.12: Percentage of killed queries of SI methods and on running multiple SI algorithms on Ψ-framework. (Or stands for original query.)

ods and on yeast, human and wordnet, against vanilla GraphQL and sPath respectively, whereas table 5.12 depicts the corresponding percentages of killed queries. We instantiated the following versions of our Ψ-framework with the corresponding number of threads: (a) GraphQL-Orig/ sPath-Orig (2 threads), (b) GraphQL-ILF/ sPath-ILF (2 threads), (c) GraphQL-IND/ sPath-IND (2 threads), (d) GraphQL-DND/ sPath-DND (2 threads). (e) GraphQL-Orig /sPath-Orig/ GraphQL-DND/ sPath-DND (4 threads). For both GraphQL and sPath, we were able to achieve up to 3 orders of magnitude improvement with our Ψ-framework on both per-query and per-workload metrics. Also, with the Ψ-framework, the percentage of *hard* queries was reduced and, for yeast and wordnet, *hard* queries became extinct – see Table 5.12. Both metrics and the percentage of the *killed* queries reveal that for different workloads and different algorithms, there are different performance gains. This is another manifestation of the algorithm specificity.

5.9 Conclusions

We have studied the subgraph isomorphism problem, in both its decision and matching versions, using well-established FTV and SI methods respectively, and against several different real and synthetic datasets of various characteristics. Our research has revealed and quantified a number of insights, concerning (i) the existence and role of straggler queries in a method's overall performance (§5.4), (ii) the dramatically varying performance of isomorphic queries (§5.5), (iii) the impressive impact that query rewriting can have when used before executing the query with several algorithms ((§5.6)), and (iv) the fact that straggler queries are algorithm-specific (§5.7). We used both WLA and QLA metrics to fully appreciate the performance of algorithms in the presence of stragglers. A number of query rewritings were proposed, and our results showed that in many cases at least one rewriting existed which could offer great performance advantages – with different rewritings being best for different queries. We showcased that, for the SI algorithms, when a query proved to be very expensive with one algorithm, another algorithm would actually manage to compute its answer very efficiently. These findings then naturally culminated into a novel framework, which employs in parallel different threads, each using a different well-known algorithm and/or a specific query rewriting, per query. This introduced dramatic improvements (up to several orders of magnitude) to FTV and SI algorithms. We hope that our findings will open up new research directions, striving to find appropriate, per-query, isomorphic rewritings, in combination with alternate per-query subgraph isomorphism algorithms that can yield large improvements. Using machine learning models to predict which version of our framework (algorithms, rewritings) to employ per query is of high interest.

Limitations We summarize some of the limitations discussed earlier in this chapter along with further steps that we could follow. Experiments in this chapter and especially in §5.6

and §5.8 are limited to the use of 5 different query rewritings. Additional query rewritings could be introduced along with combining existing rewritings in a different sequence, e.g. IND+ILF instead of ILF+IND.

Additionally, in §5.6, we mentioned that in order to perform our query rewriting, we are based on the initial query node IDs of the original query, and we break ties by respecting the order of nodes on the original queries. Undoubtedly, this is not the only way to resolve ties. Thus, it would be preferable to experiment with possible combinations in the case of ties. However, given queries with high number of edges and a large number of ties to break, such an experimentation is not easily feasible.

In §5.8, we discussed that Ψ-framework is not the only solution to the straggler queries problem. Instead, it would be preferable to be able to choose the right algorithm and/or isomorphic query instance to use so as to minimize the query execution time along with the memory overhead that is introduced by the parallel execution. This research direction is left for future work.

Finally, so far, we have considered graph query rewritings for performing the subgraph matching and we leave intact the stored graphs. Thus, similar rewritings could be applied on the stored graph. Undoubtedly, given the size of the stored graphs compared to the size of the query graph, this can be impractical; additionally a much larger number of ties is expected with the proposed rewritings in §5.6 that would have to be resolved with more complex rewritings.

Chapter 6

Hybrid Algorithms for Subgraph Pattern Queries in Graph Databases: An Evaluation

Numerous methods have been proposed over the years for subgraph query processing, as it is central to graph DB analytics. These are fragmented in two major categories; FTV and SI methods. Alas, the current research trend is to totally dismiss FTV methods, because SI methods have been shown to enjoy much shorter query execution times and because of the alleged high costs of managing the DB graph index in FTV methods. As a result, a number of new SI methods is being proposed annually.

In the current work, we initially study the performance of the latest SI algorithms over datasets consisting of a large number of graphs. With our study, we evaluate the algorithms' performance and we provide comparison details with former studies. As a second step, we combine the powerful filtering of a top-performing FTV method, with the various SI methods, which leads to the best practice conclusion that SI and FTV shouldn't be thought of as disjoint types of solutions, as their union achieves better results than any one of them individually. Specifically, we experimentally analyze and quantify the (positive) impact of including the essence of indexed FTV methods within SI methods, showing that query processing times can be significantly improved at modest additional memory costs. To achieve this, we initially quantify the time and space that is required to construct the indexes used by FTV and SI methods. Subsequently, we study the effectiveness of the constructed indexes in the process of filtering away candidate graphs and the time/space trade-offs involved in this process. We show that these results hold over a variety of well-known SI methods and across several real and synthetic datasets. As such, hybrids of the FTV and SI methods reveal a missing opportunity and a blind spot in related literature and trends. With a typical dataset consisting of many large graphs and the query graph existing in a small portion of them, Grapes is capable of filtering out the majority of graphs that do not contain the query graph. Thus, such a combination proves to be beneficial by primarily avoiding the initiation of a large number of redundant subgraph isomorphism tests.

6.1 Introduction

Related work in the subgraph querying problem is segregated in two major categories: the *filter-then-verify (FTV)* and the *subgraph isomorphism (SI)* methods, as discussed in §2.3. Specifically, FTV methods mainly focus on filtering out graphs that do not contain a query graph as an answer and then employ a "standard" SI algorithm for verification, whereas for SI methods indexing/filtering is usually neglected in favor of better/faster SI heuristics. The more recent works, e.g. [8, 55, 23], dismiss the FTV methods with the claim that the fast sub-iso test of the SI methods significantly outperforms the index-based FTV methods. Thus, all recently published methods follow the SI paradigm.

In the current chapter, our goal is to identify the best practices for processing subgraph pattern queries. In turn this rests on two pillars: The first is a head-to-head comparison and evaluation of the state-of-the-art SI methods. Our findings will allow a direct comparison with [8] for the common used algorithms, but will also reveal interesting insights for two notable and high performing SI methods proposed after the publication of [8]. Second, and most importantly, armed with the knowledge of the above conclusions, we investigate best practices for subgraph pattern querying in graph DBs by combining the main assets of the FTV and SI methods to derive hybrid FTV-SI methods. Perhaps surprisingly, for the problem at hand, no prior research has considered to study the impact of hybrid FTV-SI solutions, whereby the benefits of a top-performing graph DB index are combined with the benefits of the faster sub-iso heuristics offered by the SI methods. With the current chapter, we fill this gap by studying the benefits of hybrid FTV-SI solutions and thus putting forward a new point in the solution space. In other words, we investigate the effect of combining (parts of) well known algorithms from different research categories, bringing new insights into the strengths and weaknesses of existing FTV and SI methods, and analyzing the benefits of their hybrids, revealing a blind spot in related research. Overall, the current chapter shows that such approaches can be very beneficial and suggests how to address the key shortcomings of such hybrid solutions.

In total, we provide answers to the following central questions: (1) Does a head-to-head comparison reveal a single winner among the top-performing SI algorithms in both indexing and query processing? (2) Noting that even SI methods utilize a pruning (index-like) structure, how much time/space is required to create the index from the FTV methods and the corresponding SI methods? (3) How effective is the index from the SI methods versus FTV in terms of filtering away candidate graphs? (4) What are the time/space trade-offs involved in this process? (5) Finally, the dominant question is "can we achieve significant speedups by using hybrid solutions and quantify the speedups given memory and time constraints for the index?" With a typical graph dataset consisting of many large graphs and the actual answer set consisting of a small portion of graphs, with this work we show that large performance gains are possible. We employ three real and a synthetic dataset generated with GraphGen[60] to investigate characteristics not present in the real datasets. Finally, we consider five popular and efficient recent SI methods for our evaluation and a top-performing filtering approach from an FTV method to construct our best practice hybrids.

6.2 Related Work and Contributions

The numerous proposed FTV and SI methods have been extensively discussed in §2.3. For the purposes of this chapter and in order to facilitate the discussion in the presented results, we recap here the key-points of both FTV and SI methods. FTV methods were originally proposed for the decision version of the subgraph querying problem, where given a dataset of numerous (typically small) graphs and a query/pattern graph q, the method decides whether q is contained in any graph in the dataset and the IDs of those graphs are returned, whereas SI methods were originally proposed for the matching version to find all the embeddings of q in a typically large, stored graph.

In the index construction phase of FTV methods, stored graphs are decomposed into features which are then indexed. During query processing, query graphs are similarly decomposed into features; graphs from the dataset that do not contain one or more of these features definitely do not contain the query and are thus pruned away, with the remaining graphs forming the candidate set. At the verification stage, the query graph is tested for subgraph isomorphism against each graph in the candidate set to produce the final answer. The target of all these methods is to prune as much as possible the candidate set and thus to reduce the number of subgraph isomorphism tests performed. The underlying isomorphism test of the vast majority of proposed methods is VF2[47], which was widely publicly available. In chapters §4 and §5, we extensively discussed various proposed FTV methods. With our experiments, we concluded that Grapes[11] and GGSX[12] are the best solutions in terms of index construction time, query processing time, and scalability limitations. It was also showed that both Grapes and GGSX enjoy similar filtering power for datasets consisting of relatively small graphs. However, when the graph sizes increase, Grapes outperforms

GGSX in filtering power.

The focus of SI methods is not to filter out graphs in the dataset that definitely do not contain the query as an answer, but for each DB graph (i) to locate the best candidate vertices to expedite the sub-iso test, and (ii) to decide the optimal join plan to follow; i.e., the sequence in which the query vertices will be matched to those of the stored graph. Thus, proposed SI methods, apart from the subgraph isomorphism heuristic algorithm, additionally contain a pre-processing/indexing step where they maintain a feature-based index, along with vertex label lists and additional information to facilitate the sub-iso test. With our experiments on chapter §5, among others, we have shown that all SI proposed methods suffer from "straggler" queries; i.e., queries whose processing time is many orders of magnitude worse compared to the rest. We have also seen that a straggler query on one algorithm can be a typical query on some other algorithm.

There is nothing preventing the SI methods from being applied for the decision problem, as discused in §2.3. FTV methods were originally proposed to work with datasets consisting of numerous, relatively small graphs, and their effectiveness relies on their achieved filtering. In the current research, SI methods gain ground over FTV methods. There are various reasons for that. First reason is the claim that the SI methods enjoy shorter query execution times that can offset the gains of the filtering offered by the FTV methods and thus SI methods outperform FTV methods [8, 55, 23]. Another reason is that FTV methods come bundled with the alleged high costs of managing the constructed DB graph index [7, 9].

<u>Contributions</u>: In the current work, our goal is twofold. We evaluate top-performing SI methods against datasets consisting of a large number of graphs to provide insights about their performance. Our findings compare with [8] for the 3 common algorithms and complement it with inclusion of 2 notable SI methods proposed after the publication of [8]. In parallel, we thoroughly investigate the current community wisdom which tends to totally dismiss FTV methods based on the fact that the fast sub-iso test of the more recent SI methods can significantly outperform the index-based FTV methods [55, 8, 23]. This is indeed a claim we have verified as well: when comparing a fast SI algorithm (even if not the fastest one yet, such as GraphQL) against a top-performing FTV algorithm (such as Grapes) for queries run over a single graph in the DB, SI methods are the winners. However, this fact requires further analysis which has not as of yet been performed. Note that: (1) No analysis for the reasons for this fact has ever been provided. (2) SI methods also essentially develop and utilize indexes for pruning the search space during matching; no one has ever really provided any insights as to how costly in time and in memory space this is. And, combined with (1) above, (3) No evidence exists so far that relates the efficacy of FTV-indexes versus SI-indexes in terms of reducing the search space. Finally, FTV algorithms utilize both a filtering and a verification stage. Hence, if FTV-type indexing is more powerful than SI-type indexing, this implies that the subgraph-isomorphism heuristics of SI methods must significantly outperform the verification algorithms of FTV methods. Therefore, combining the additional power of FTV-type indexing with the great efficiency of SI algorithms appears to be a promising avenue for new performance gains. So, the real issue becomes to (4) Investigate and quantify what the expected performance gains of hybrid FTV-SI solutions are. In the current chapter, we will tackle the above issues and we will show that dismissing completely indexed FTV methods leads to missing an opportunity for significant performance gains and thus reveal a research blind spot. This we hope will motivate new research into hybrid FTV-SI combinations and new indexes and/or new subgraph isomorphism heuristic algorithms for such hybrids.

For our experiments, we employ various real-world and synthetic datasets over the top-performing FTV algorithm (based on the results from [9]) and 5 high-performing SI methods (based on results from [8, 10, 55]). Our results will show that FTV-based indexing can be beneficial despite the added time and space overhead, by offering great pruning in the search space compared to SI methods. As a result, we then study the performance of a novel hybrid method that utilizes the index from Grapes and 5 different top-performing SI methods. We analyze the performance speedup that can be achieved and the related trade-offs.

6.3 Experimental Setup

6.3.1 Algorithms

For the FTV methods, we chose Grapes[11] as top-performing in terms of (i) indexing time, (ii) query processing time, (iii) scalability limitations and (iv) filtering power (§6.2).

For the SI methods, we opted for methods (i) whose code is publicly available or made available to us by the authors upon request, so any conclusions would not be implementation dependent and (ii) that were well recognized as well performing. Thus, we selected GraphQL[24], sPath[29], QuickSI[35], TurboIso[55], and BoostIso[56] over TurboIso. With respect to CFL-Match[57]: we did not employ the algorithm as its authors did not respond to our request for their code.

A more detailed description of the used algorithms can be found in §3.1.

6.3.2 Setup

All the experiments were conducted on a Windows 7 SP1 host, with 2 Intel Xeon E5-2660 CPUs (2.20GHz, 20MB cache) with 8 cores/16 vcores per CPU, 128GB of RAM, and 3.5TB disk. We ran our experiments individually and one at a time to avoid any interference across runs.

For Grapes we used the implementation provided by its authors. For GraphQL, sPath, and QuickSI, we used the implementation provided by [8]. For TurboIso we obtained the binary code from the authors and for BTI we obtained the source code from GitHub[1] where it was publicly available.

We used the default values for the input parameters of the compared algorithms, as they were defined by their respective authors in the relevant publications and/or in their implementation code. Specifically:

- For Grapes, we enumerated paths of up to size of 4. We used 1 and 4 threads; results for executions with 1 (respectively 4) threads are denoted by Grapes/1 (respectively Grapes/4).
- For GraphQL, we used a refined level of iterations of pseudo-subgraph isomorphism $r = 4$.
- For sPath, we used a neighborhood radius of 4 and maximum path length 4.
- TurboIso and BTI do not require any input parameter. However, for TurboIso we were able to execute queries of only up to 25 vertices, due to an inherent limitation in the executable provided to us (and we were unable to amend this because we were only provided with the binary).
- For all SI methods the number of searched embeddings of the pattern graph in the stored graph is capped at 1000; i.e., after finding the first 1000 matches, the algorithms terminate. For TurboIso this option is hardcoded whereas for the rest it is configurable.

6.3.3 Datasets

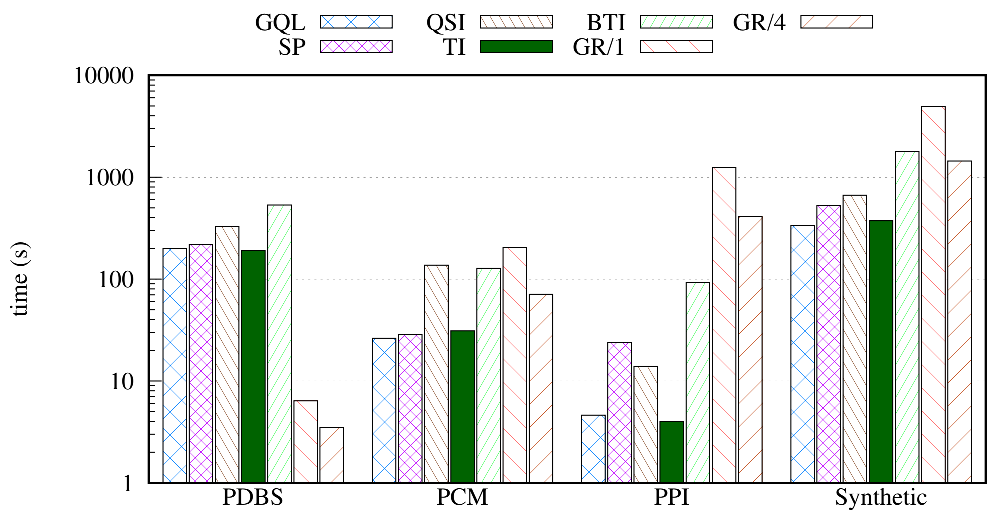
Table 3.1 summarizes the characteristics of the employed datasets. Recall that PDBS, PCM and PPI are 3 real datasets that were previously used in [11, 9]. PDBS and PCM represent chemical compounds comprising of 600 and 200 graphs respectively, whereas PPI represents 20 protein-protein interaction networks. The majority of existing real datasets comprise of relatively small and sparse graphs, and thus in the lack of real datasets publicly available that preserve the required properties (i.e., many large graphs), we additonally employ a synthetic dataset of 1000 graphs generated with GraphGen[60]. GraphGen was presented in detail in §3.2.1.

6.3.4 Query Workloads

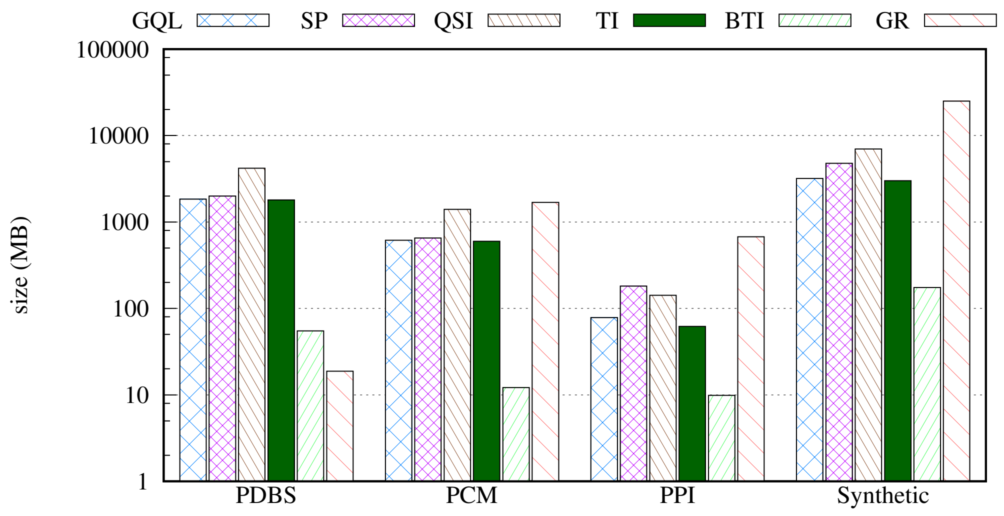
To generate each of the query graphs, we follow the procedure described in §3.2.3.

[1] `https://github.com/UltraHector/BoostIsoGraphAdaptation`

For PDBS and PCM, we use queries of size 20 and 24 edges. For PPI, we use queries of size 16, 20, 24, and 32 edges. For the synthetic dataset, we use queries of size 24, 32 and 40 edges. For every query size we use 200 queries for PDBS and PCM and 100 queries for PPI and the Synthetic dataset. Finally, we were unable to execute queries greater that 25 vertices on TurboIso, as discussed in §6.3.2. Thus, in the presentation of our results in the subsequent figures and because of the small number of queries of ⩾25 edges in PPI and Synthetic dataset that qualify for TurboIso, we omit them completely and we only present results for queries up to 24 edges.



(a) Indexing time



(b) Index size

Figure 6.1: Indexing time and size of Grapes and SI methods

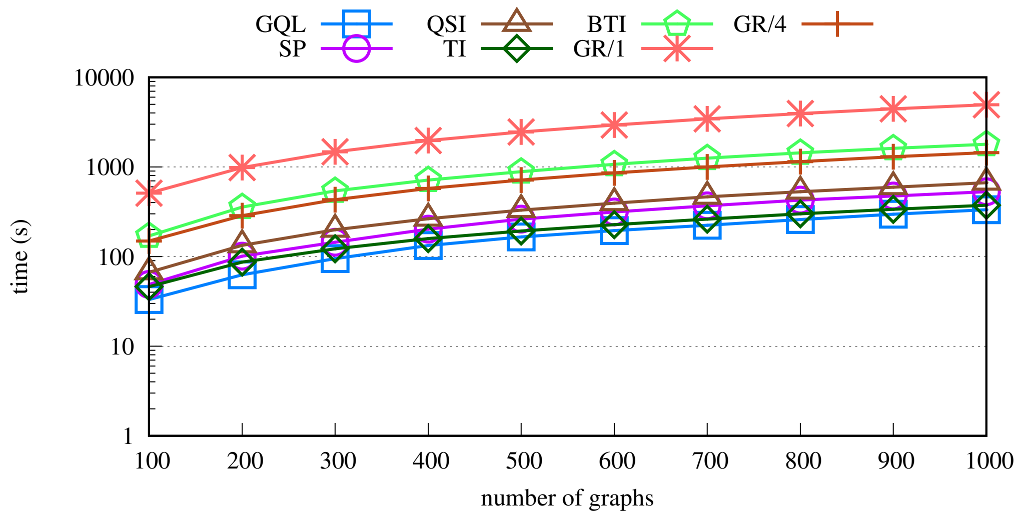
6.4 Index construction

Both FTV and SI methods rely on an index which is used to fulfill different purposes in each case. For the FTV methods, the index construction facilitates the pruning of graphs in the dataset that definitely do not contain the query graph as an answer. For the SI methods the index purpose is to locate the candidate vertices on the stored graph to expedite the underlying subgraph isomorphism test. Thus, SI methods require less time and space to construct and store their index respectively.

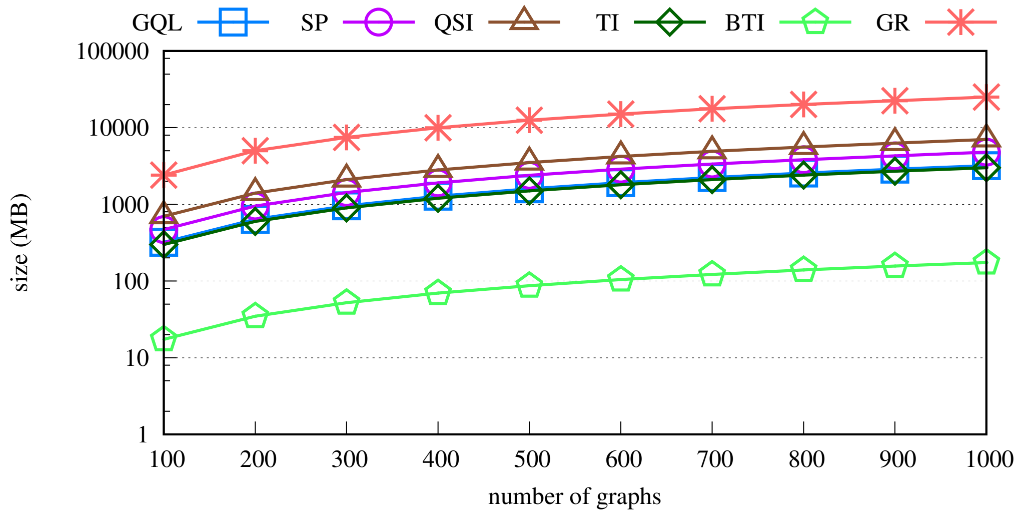
Figure 6.1 presents the results from the index construction phase for all datasets and algorithms. For Grapes, the index size is independent of the number of threads and thus only one bar is used in the corresponding figure 6.1(b) for this algorithm. Among the SI methods, we notice the following trend in the index sizes: $Size_{QuickSI} > Size_{sPath} > Size_{GraphQL} > Size_{TurboIso} > Size_{BTI}$ and this trend is also followed by the indexing time, with the sole exception of BTI, where the indexing time is comparable to that of QuickSI. In the majority of algorithms, this is somewhat expected because of the structures used by each algorithm to maintain the index. Specifically, based on the code we had available, we note that GraphQL and sPath along with the additional information they require to store their index – i.e. labels of neighboring nodes in radius i shorted in lexicographical order for GraphQL, and shortest paths for sPath – they also store the actual graphs in a convenient format as presented in TurboIso (§3.1.2). Our results come in agreement with [8] for GraphQL and sPath but not for QuickSI. Finally, BTI's index consists of 2 distinct files that store the hypergraph and containment graph (as described in §3.1.2) and even though their size is already small enough, it could be diminished even more if the index files were stored in a binary format instead of plain "txt" files.

Comparing sPath, that constructs the second largest in size index, with Grapes, we note that both sPath and Grapes work with paths. However, sPath maintains only the shortest paths, whereas Grapes enumerates all paths up to maximum length and additionally maintains location and frequency information. Thus, as it was expected, Grapes/1 constructs up to 1 order of magnitude bigger indices than sPath and it requires up to 1 order of magnitude more time to achieve this, with the sole exception of PDBS. To justify the results in PDBS, we need to look at the dataset characteristics in table 3.1. PDBS is a very sparse dataset, with only 10 labels totally and an average number of only 6.4 distinct labels per graph. As a result, the enumerated distinct paths are well compressed in the trie utilized by Grapes and less time and space are required to build/store the index.

Finally, to highlight how the number of graphs in the DB affect the indexing phase, we employ the Synthetic dataset, that consists of 1000 graphs and we utilize appropriate subsets of the dataset. Specifically, we utilize 10 different subsets consisting of the first 100, 200,



(a) Indexing time



(b) Index size

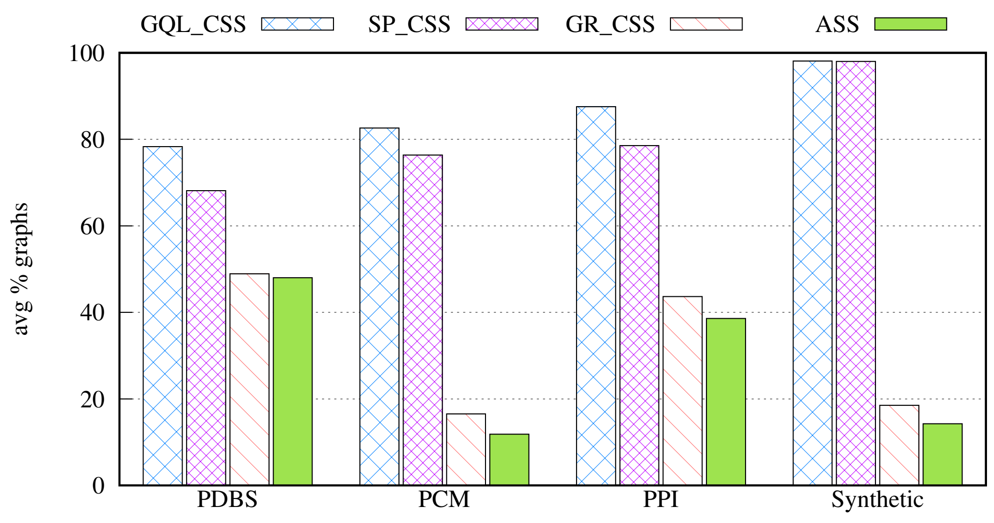
Figure 6.2: Indexing time and size of subsets of the Synthetic dataset

..., 1000 graphs. Figure 6.2 presents the corresponding results. As it was expected, a linear increase in the number of graphs leads to a linear increase in both indexing time and size consistently across all algorithms.

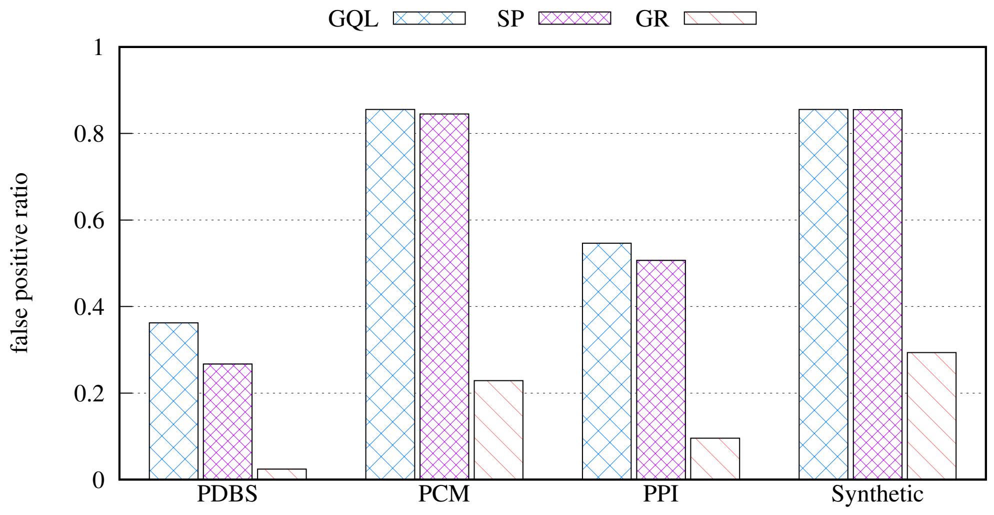
6.5 Filtering power

To quantify the filtering power, we utilize 2 different metrics, as they were presented in §3.3.2. These are: (1) the percentage of graphs that constitute the candidate set for each algorithm, before proceeding with the final subgraph isomorphism test and (2) the FPR as defined in §3.3.2.

Figure 6.3 presents the results for the pruning power of used algorithms. In the presented



(a) Candidate and answer sets for all algorithms



(b) False Positive Ratio

Figure 6.3: Pruning Power of Grapes and SI methods

figures, CSS stands for Candidate Set Size and ASS stands for Answer Set Size. QuickSI, TurboIso, and BTI are not included in the presented results as they do not perform any filtering and they proceed directly with the subgraph isomorphism test. Grapes provides the same filtering power independently of the number of threads that are executed and thus on the corresponding figures we do not distinguish the results. For comparison purposes, we report the percentage of graphs that constitute the average answer set size along with the percentage of graphs that constitute the average candidate set size for each algorithm in figure 6.3(a).

In the aforementioned figures, we observe that different number of graphs were filtered out by the three different methods. Although it is not presented in the above figures, the filtering power of the SI methods is slightly improved as the query size increases and the same effect holds for Grapes. In the majority of cases, Grapes was able to filter out at least double the amount of graphs compared to GraphQL and sPath, leading to candidate sets very close to the actual answer set, and this is also evident in the low FPR. However, Grapes' filtering comes at an extra cost of a much larger index to store and more time to construct (see §6.3.2), with the sole exception of PDBS. sPath, that constructs a slightly more elaborate index compared to GraphQL, was also able to achieve an up to 10% better filtering than GraphQL. A very interesting observation is the fact that in very rare cases the graphs that were filtered out by the SI methods were not always a subset of the graphs filtered out by Grapes. This occurred in $<1\%$ of graph-query pairs and was more evident when increasing the query size. Finally, it is important to observe the FPR in combination with the candidate set and answer set sizes. To showcase this, we note that although PDBS is the only dataset where the average candidate set sizes are among the biggest for all 3 algorithms reported, the corresponding FPR are the lowest for all datasets because of the high answer set size.

6.6 Performance of SI methods

SI methods keep gaining ground over FTV methods, with the current tendency in recent work to totally dismiss FTV methods with the claim that the smart subgraph isomorphism heuristics of the SI methods outperform the index-based FTV methods. Before proceeding with further investigating this claim, we provide in figure 6.4 a head-to-head comparison of the average query execution time of SI algorithms across all datasets. Because of the restriction mentioned in §6.3.2, for TurboIso and for PPI and Synthetic dataset only results with queries ≤ 24 edges are presented.

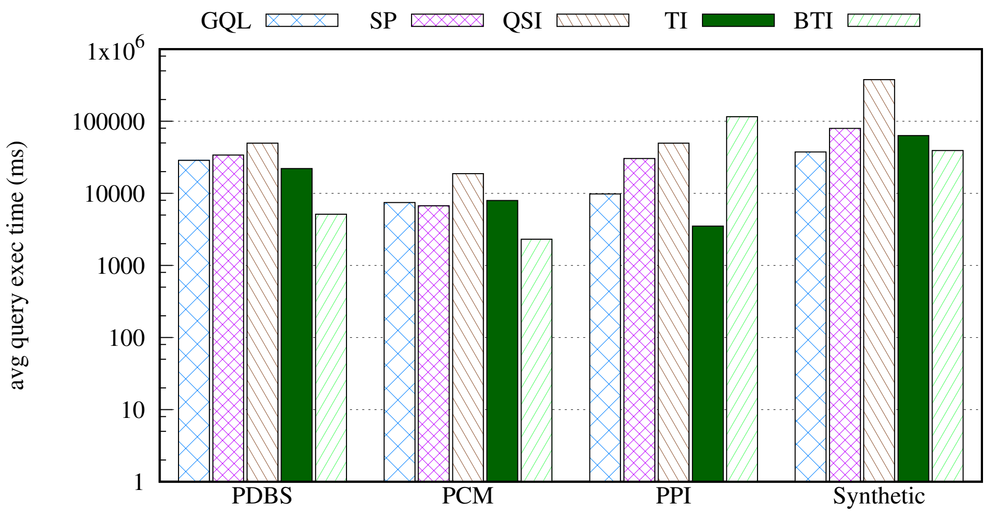
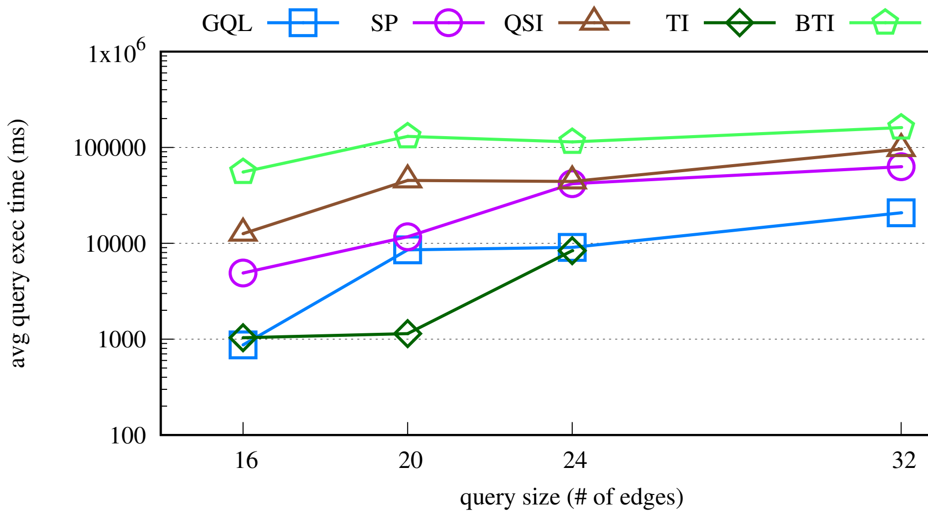
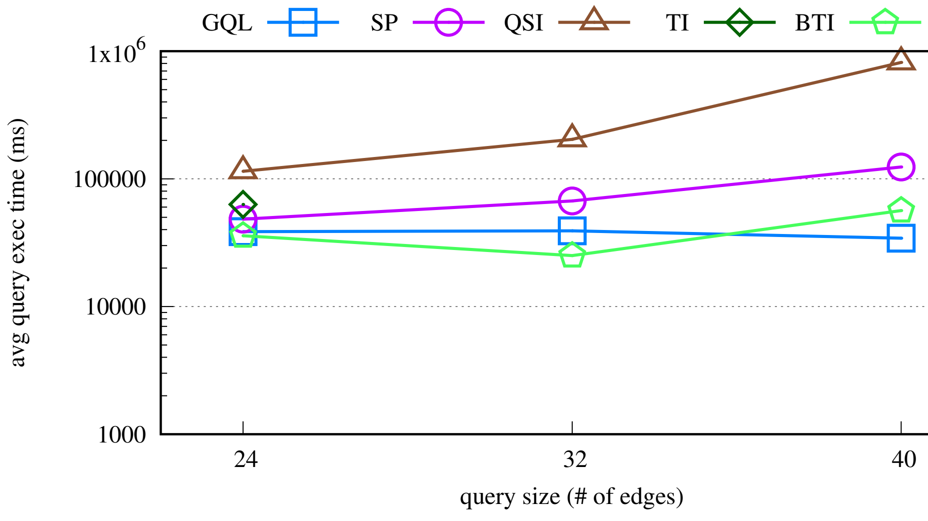


Figure 6.4: Avg query exec time (ms) of SI methods

As it can be seen from figure 6.4, there is no winner algorithm across all datasets. This finding comes in agreement with [8] and our experiments in chapter §5, where for the SI algorithms the tested dataset contained only a single, large graph. TurboIso and BTI, the newest additions in the set of SI algorithms, are favored particularly in datasets consisting of a small number of labels because of the smart rewritings applied on the query graph (and on the stored graph in the case of BTI). The least promising SI algorithm is QuickSI, but outperforms BTI on PPI where the number of distinct labels is more abundant.



(a) PPI



(b) Synthetic

Figure 6.5: Avg query exec time (ms) of SI methods for PPI and Synthetic datasets and for different query sizes

As we increase the query size, query processing becomes harder for all algorithms, with the exception of PDBS and PCM where there are no significant differences for different query sizes. The relevant results for the PPI and the Synthetic dataset can be seen in figure 6.5. It is worth mentioning that there is not a single winning algorithm for different query sizes even for the same dataset; this is consistent with both the findings in [10] and the results presented in §5.4.

6.7 Evaluating the hybrid FTV-SI method

Having discussed that the filtering of Grapes is more powerful than that achieved by the SI methods, we set out to construct a hybrid FTV-SI solution. The proposed hybrid solution works as follows: We construct the index for both Grapes (the in use FTV method) and for the in use SI method. In the query processing, we perform the filtering of Grapes (as discussed in §3.1.1) and till the stage of forming the candidate set. Subsequently, for those graphs that pass the filtering stage we use GraphQL/ sPath/ QuickSI/ TurboIso/ BTI, instead of Grapes's default (and expensive) VF2 subgraph isomorphism test. Since an extra filtering step is introduced (namely, the filtering from Grapes), it is worthwhile evaluating, analyzing, and quantifying the effect of the cost to perform this additional on-line filtering on the overall achievable performance. As Grapes was originally designed to work in parallel, we utilize $(\cdot)/N$ to denote the in use number of threads N for Grapes-[GraphQL/ sPath/ QuickSI/ TurboIso/ BTI] as the FTV-SI combination of algorithms. In this section we utilize only one thread; additional parallelism will be discussed in the following section. Last, for the rest of the discussion, we assume that the indices are already loaded into main memory once at the beginning of the execution for each query workload.

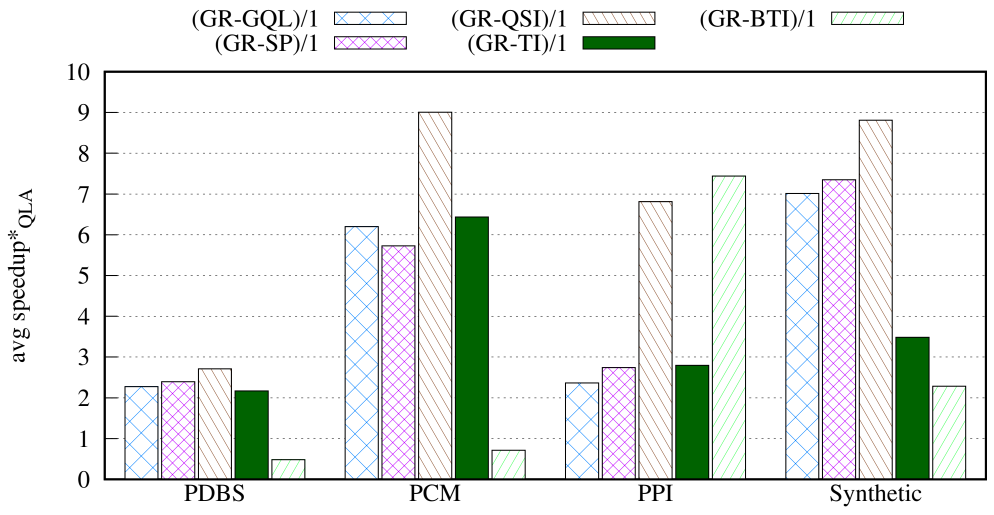
6.7.1 Performance Metrics

For every query against a dataset of graphs, we measure the *execution time*, while *avg exec time* denotes the average execution time. For the SI methods the execution time includes the time for constructing the index of the query, the matching of the query's index to the databases' index (where applicable) and the time required to perform the subgraph isomorphism test. For our proposed hybrid FTV-SI solution the execution time includes (i) the time required to perform the filtering of Grapes as described in §3.1.1 and (ii) the execution time of the in use SI method for the graphs that passed the filtering stage (that substitutes the VF2 verification algorithm).

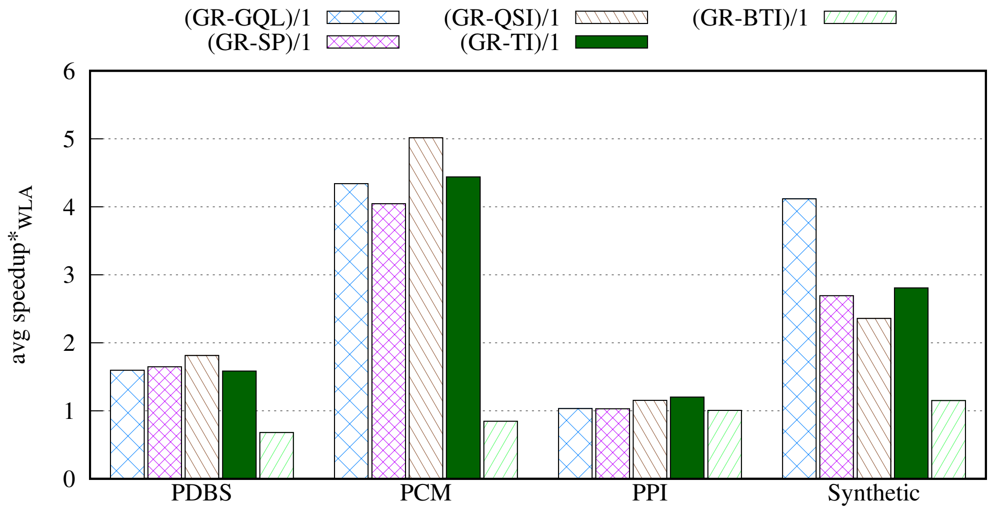
Let q_i be a given query. Let also t_i^M be the execution time of q_i over method M, where M can be one of the SI methods, i.e., GraphQL/ sPath/ QuickSI/ TurboIso/ BTI, and $t_i^{Grapes-M}$ be the execution time of q_i over the hybrid combination of Grapes with method M over all the graphs in the dataset. In order to evaluate the performance of this combination, we utilize the $speedup^*$ metric defined as: $\frac{t_i^M}{t_i^{Grapes-M}}$. $speedup^*$ represents what we lose in performance if we choose the original method over the various alternatives; i.e., $speedup^*$ equals the maximum attainable speedup over the original method, if we chose the best of the examined alternatives. As we described in §3.3.4, the aforementioned $speedup^*$ metric can have a QLA and WLA version, denoted with a matching subscript; e.g. ${speedup^*}_{QLA}$.

6.7.2 Performance Results

Before proceeding with the presentation of the achieved speedups, we initially discuss the indexing costs we need to pay for our hybrid FTV-SI solution. Thus, the size of the constructed index for all datasets for the hybrid FTV-SI solution is the addition of Grapes' index with the index of the in use SI method as presented in figure 6.1(b). The same holds for the corresponding indexing times. The pruning power of the FTV-SI solution is equal to the pruning power of Grapes, as it was presented in figure 6.3 and for the corresponding datasets.



(a) Average $speedup^*_{QLA}$



(b) Average $speedup^*_{WLA}$

Figure 6.6: Average $speedup^*_{QLA}$ & $speedup^*_{WLA}$ of the hybrid FTV-SI method

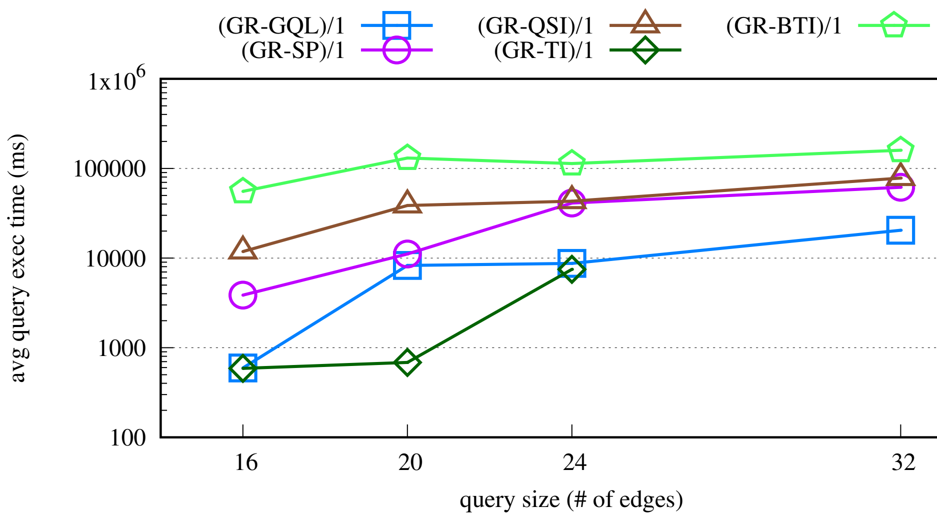
Figure 6.6 presents the average QLA and WLA speedups for all datasets and query sizes. We were not able to execute queries >25 vertices with TurboIso (§6.3.2); as a result in PPI and the Synthetic dataset the presented speedup for the hybrid Grapes-TurboIso combination

		(GR-GQL)/1	(GR-SP)/1	(GR-QSI)/1	(GR-TI)/1	(GR-BTI)/1
PDBS	stdDev	1.783	2.035	2.589	1.556	0.456
	min	0.777	0.798	0.847	0.682	0.022
	max	10.361	12.179	15.185	9.107	3.055
	median	1.937	1.934	2.119	1.915	0.308
PCM	stdDev	3.454	3.093	6.560	3.655	0.218
	min	1.164	1.182	1.250	1.195	0.053
	max	14.912	14.029	26.001	17.099	1.078
	median	5.436	5.153	6.975	5.619	0.776
PPI	stdDev	2.886	7.112	37.603	2.852	119.611
	min	0.857	0.914	0.001	0.877	0.067
	max	24.196	29.198	89.884	18.479	90.6
	median	1.496	1.416	1.522	1.865	0.993
Synthetic	stdDev	5.465	9.444	21.386	2.248	15.599
	min	1.269	1.094	1.168	1.633	0.292
	max	28.554	65.586	29.042	12.920	61.415
	median	5.107	3.683	2.777	2.649	0.964

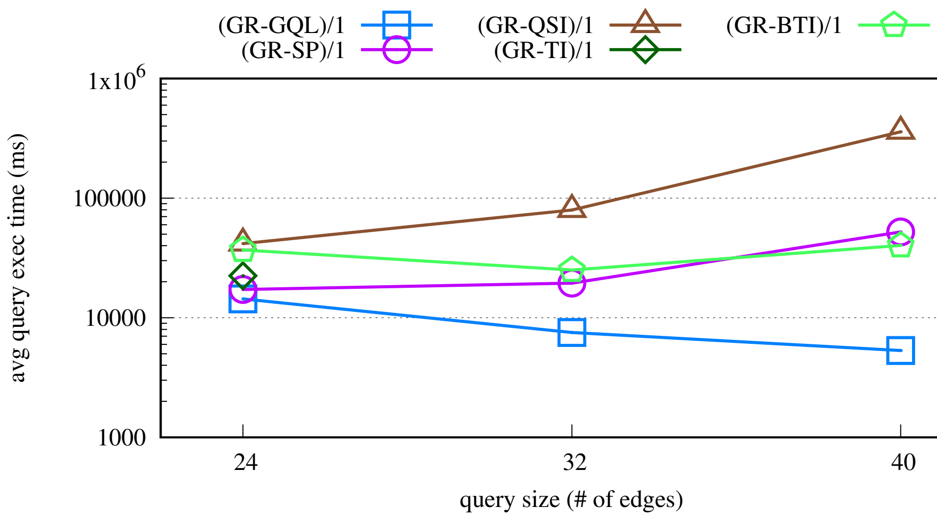
Table 6.1: $speedup^*{}_{QLA}$ statistics for FTV-SI combination with 1 thread

refers to queries $\leq$ 24 edges, and thus results are not directly comparable with the rest of the results for the hybrid Grapes-M combinations. BTI is the sole algorithm that is rather hurt than improved by this hybrid combination in PCM where the size of data graphs is relatively small, and in PDBS where the size of the candidate graphs in the dataset is relatively high. In the majority of cases the achieved speedups are higher as the query size increases and this effect is more profound in the Synthetic dataset, because of the much higher number of graphs that constitute the Synthetic dataset and the higher percentages of graphs that were filtered out (as we can observe in combination with figure 6.3(a)). To showcase this, we provide in figure 6.7 the average query execution times of the hybrid FTV-SI methods for PPI and Synthetic datasets and for different query sizes that can be compared against the average query execution times of the hybrid SI methods for the same datasets (figure 6.5). In other words, the query graphs in the Synthetic dataset provide better selectivity than those in the 3 Real datasets and this is reflected in the achieved speedups. A notable fact, presented in figure 6.8, is that the different achieved speedups for all algorithms bring about significant changes in their average query execution times, as they were presented in figure 6.4. However, not a single winner algorithm across all datasets and different query sizes is yet identified!

Table 6.1 presents additional statistics for min, max, median and stdDev of the achieved $speedup^*{}_{QLA}$. For all algorithms except for BTI, we observe that the min achieved speedup is not always > 1, but the median $speedup^*{}_{QLA}$ is in all cases > 1. In other words, there are some queries that the time gained from the filtered out graphs does not pay off. For the executed queries, this phenomenon occurred when the candidate set size was ≥ 500 graphs



(a) PPI



(b) Synthetic

Figure 6.7: Avg query exec time (ms) of hybrid FTV-SI methods for PPI and Synthetic datasets and for different query sizes

in PDBS and ≥ 15 graphs in PPI.

6.8 Reducing filtering time with parallelism

As Grapes was designed to work in parallel, we studied this effect with additional experiments. Specifically, we used Grapes/4 for the filtering stage, alongside one of the SI algorithms, on the same set of datasets and query workloads. We utilize as many different (parallel) instances of SI algorithms as the number of threads N utilized by Grapes. We maintain the graphs that passed Grapes' filtering test in a queue and the first N graphs are assigned to the N threads. Till the graph queue is empty, the first graph in the queue is assigned

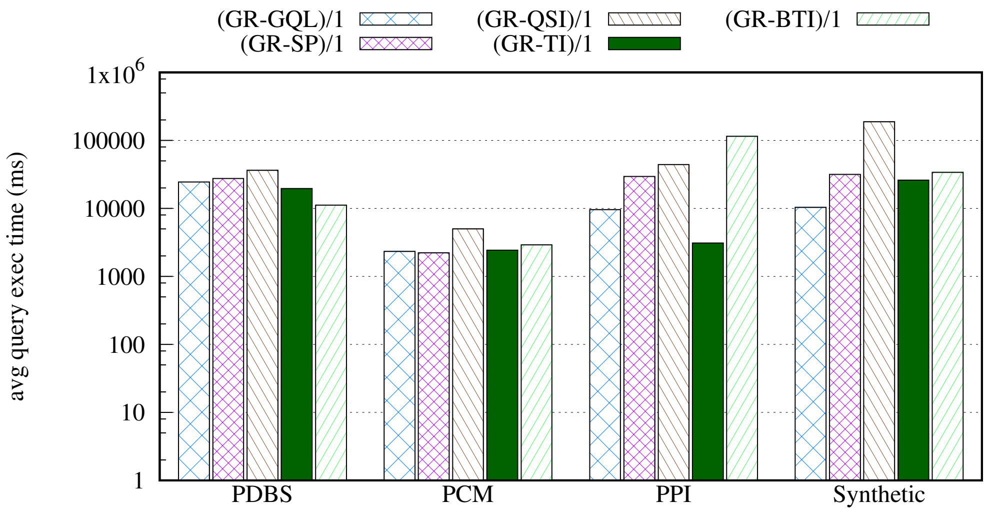
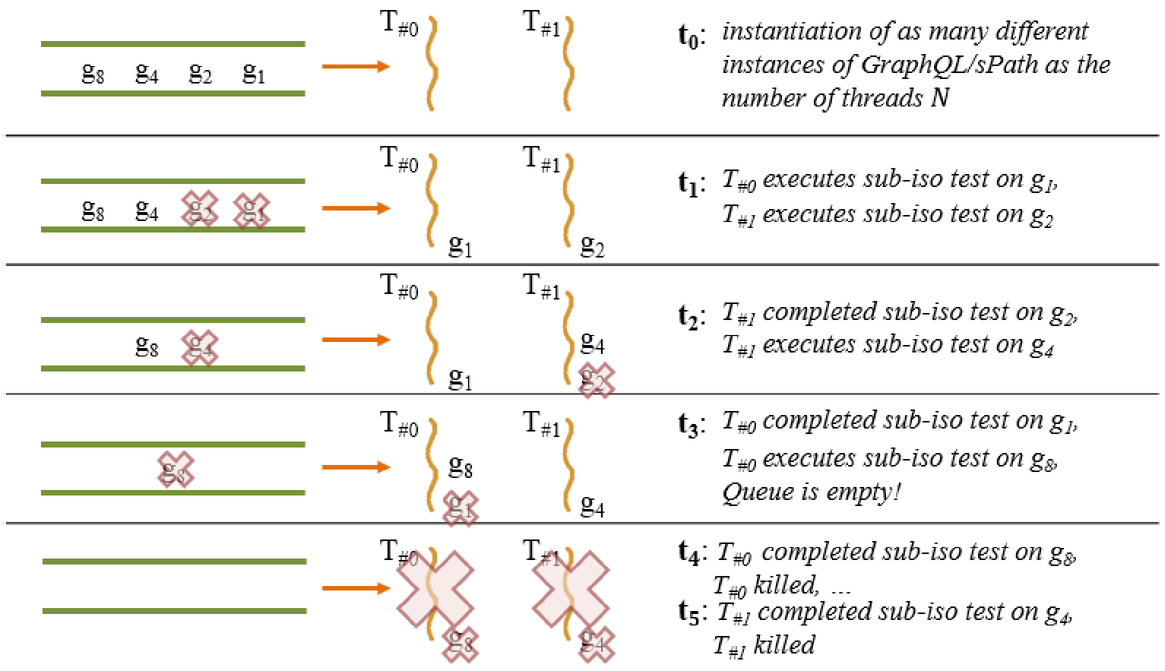


Figure 6.8: Avg query exec time (ms) of the FTV-SI hybrid methods

to the next available thread. This choice brings additional performance improvement.



$\mathbf{t_0}$: *instantiation of as many different instances of GraphQL/sPath as the number of threads N*

$\mathbf{t_1}$: *$T_{\#0}$ executes sub-iso test on g_1, $T_{\#1}$ executes sub-iso test on g_2*

$\mathbf{t_2}$: *$T_{\#1}$ completed sub-iso test on g_2, $T_{\#1}$ executes sub-iso test on g_4*

$\mathbf{t_3}$: *$T_{\#0}$ completed sub-iso test on g_1, $T_{\#0}$ executes sub-iso test on g_8, Queue is empty!*

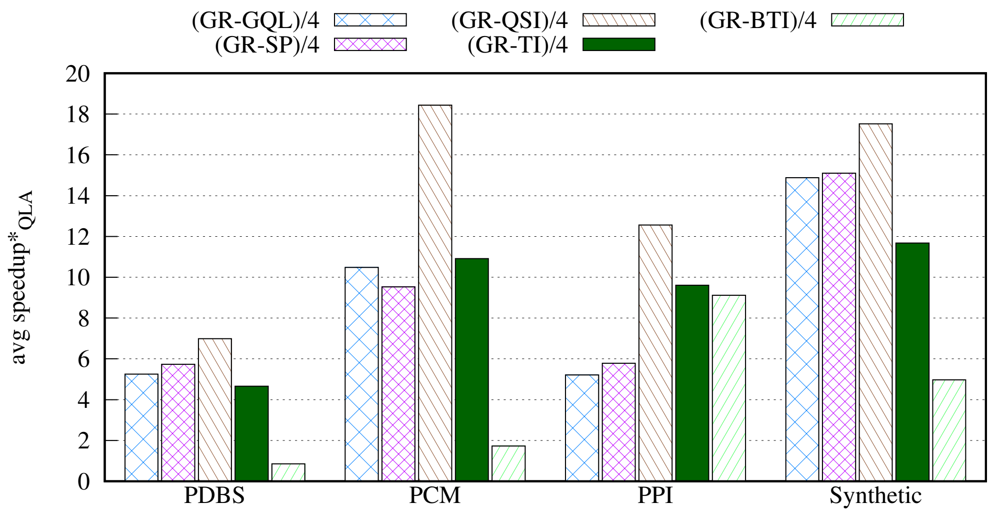
$\mathbf{t_4}$: *$T_{\#0}$ completed sub-iso test on g_8, $T_{\#0}$ killed, ...*

$\mathbf{t_5}$: *$T_{\#1}$ completed sub-iso test on g_4, $T_{\#1}$ killed*

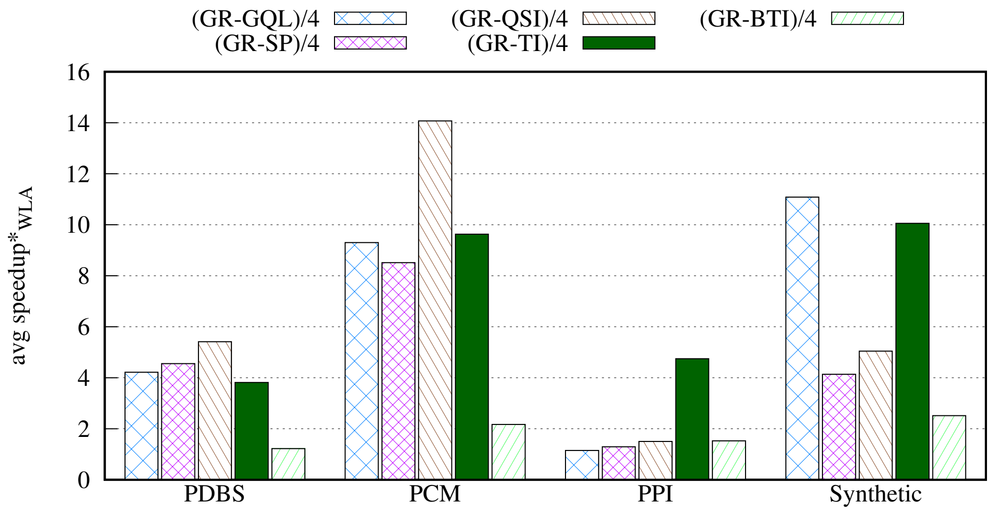
Figure 6.9: Example on parallel execution of the verification stage of the hybrid FTV-SI combination with number of threads $N = 2$. (We assume that graphs g_1, g_2, g_4, and g_8 formed the candidate set after the filtering stage. The red X is used to represent the removal of a grpah from the queue or the completion of a thread execution.)

To showcase this, we will additionally present an example (figure 6.9). We assume that we use $N = 2$ threads and that after the filtering stage of Grapes the candidate set is formed by graphs g_1, g_2, g_4 and g_8. Thus, in the verification stage and at time t_0, g_1, g_2, g_4, and g_8 are placed in a queue. We also instantiate as many different instances of the in use SI method as the number of threads N to run in parallel, denoted $T_{\#i}$, where $i = 0, ..., N-1$.

At t_1, the first graph (g_1) is removed from the queue and is assigned to thread $T_{\#0}$. At the same time, thread $T_{\#1}$ is free, and thus g_2 is also removed from the queue and is assigned to thread $T_{\#1}$. At t_2, thread $T_{\#1}$ completed the subgraph isomorphism on g_2. Thus, g_4 is removed from the queue and thread $T_{\#1}$ starts executing subgraph isomorphism test on g_4. At t_3, thread $T_{\#0}$ completed the sub-iso on g_1. Similar to before, g_8 is removed from the queue and thread $T_{\#0}$ executes subgraph isomorphism test on g_8. The queue is now empty. Finally, at t_4 thread $T_{\#0}$ completed the subgraph isomorphism on g_8. With the queue being empty, thread $T_{\#0}$ is killed. At t_5, thread $T_{\#1}$ is killed accordingly after the completion of the subgraph isomorphism on g_4.



(a) Average $speedup^*{}_{QLA}$, 4 threads



(b) Average $speedup^*{}_{WLA}$, 4 threads

Figure 6.10: Average $speedup^*{}_{QLA}$ & $speedup^*{}_{WLA}$ of the hybrid FTV-SI method, 4 threads

Figure 6.10 presents the corresponding $speedup^*$ result for all datasets. As expected, by increasing the number of threads N from 1 to 4, we were able to achieve up to 4 times

		(GR-GQL)/4	(GR-SP)/4	(GR-QSI)/4	(GR-TI)/4	(GR-BTI)/4
PDBS	stdDev	2.393	2.838	4.067	2.022	0.829
	min	1.794	1.924	2.201	1.353	0.022
	max	13.988	16.847	23.709	11.568	4.278
	median	5.250	5.616	6.585	4.770	0.397
PCM	stdDev	3.397	2.925	8.550	3.603	0.901
	min	3.579	3.521	4.453	3.776	0.054
	max	18.425	17.255	40.317	21.375	3.475
	median	10.298	9.435	17.664	10.838	1.721
PPI	stdDev	3.280	16.967	61.712	7.481	134.922
	min	1.001	1.003	1.003	3.403	0.0893
	max	26.366	5.613	85.818	63.498	297.41
	median	4.511	4.278	4.409	7.001	1.704
Synthetic	stdDev	6.184	12.819	30.828	6.639	22.757
	min	1.318	1.134	1.636	6.153	0.330
	max	31.975	108.596	96.647	30.388	72.763
	median	13.613	10.892	8.913	9.491	3.342

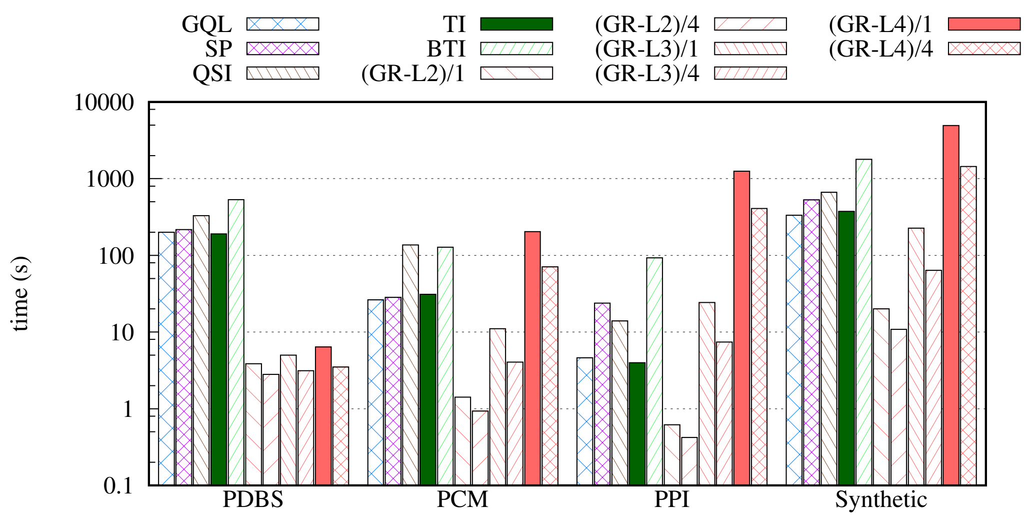
Table 6.2: $speedup^*_{QLA}$ statistics for FTV-SI combination with 4 threads

better speedups compared to the single-threaded executions. Table 6.2 presents additional statistics for min, max, median and stdDev of the achieved $speedup^*_{QLA}$ in the case of 4 threads, where in all datasets and query workloads the achieved speedup is > 1.

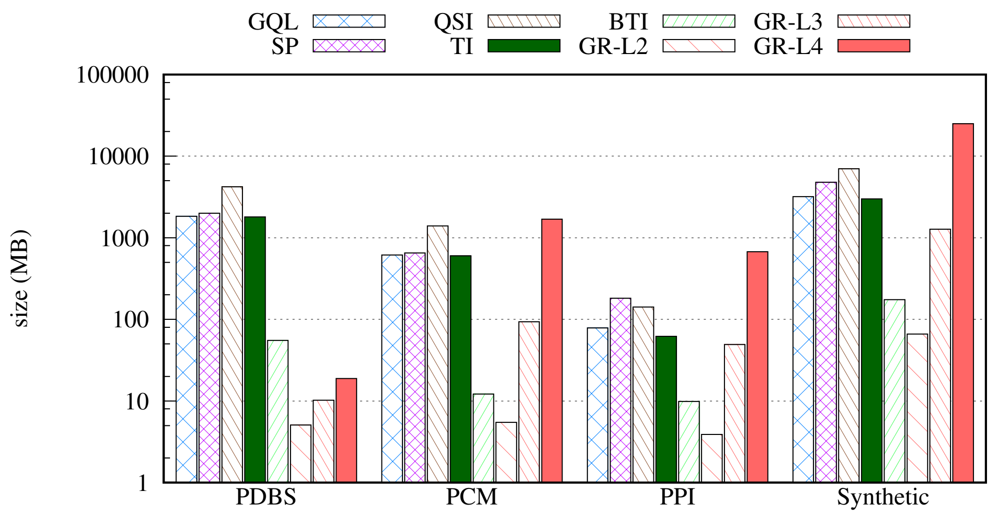
6.9 Index Time/Size - Filtering Power Tradeoff

In our discussion so far we used the default values of the enumerated features for constructing the index for Grapes, as suggested by the respective authors. But as we have seen, the constructed index of FTV methods is costly both in size and in time. In this section we tweak the size of the enumerated features $maxL$ (see §3.1.1) and we observe the filtering that can be achieved and how this affects the gained speedups in our hybrid proposed solution. Thus, for our experiments we have used $maxL = 2, 3, 4, 5$. We report that for $maxL = 5$, the index process was utilizing excessive amount of memory, leading to thrashing even to our 128GB machine and thus no numbers are reported for this case. In the subsequent figures, we utilize Grapes-Li, $i = 2, 3, 4$ (or GR-Li for short) to denote the $maxL$ value used.

Figures 6.11(a) and 6.11(b) report the indexing time and size for Grapes and the different $maxL$ tried for all employed datasets. For comparison, we additionally include the corresponding values for the employed SI methods. For all datasets, except for PDBS, there is a difference of up to 3 orders of magnitude for both indexing time and size and for $maxL$ from 2 to 4, leading to times and sizes much smaller than the SI methods in most cases. For PDBS, the small number of labels and thus the small variation of enumerated paths leads to



(a) Indexing Time

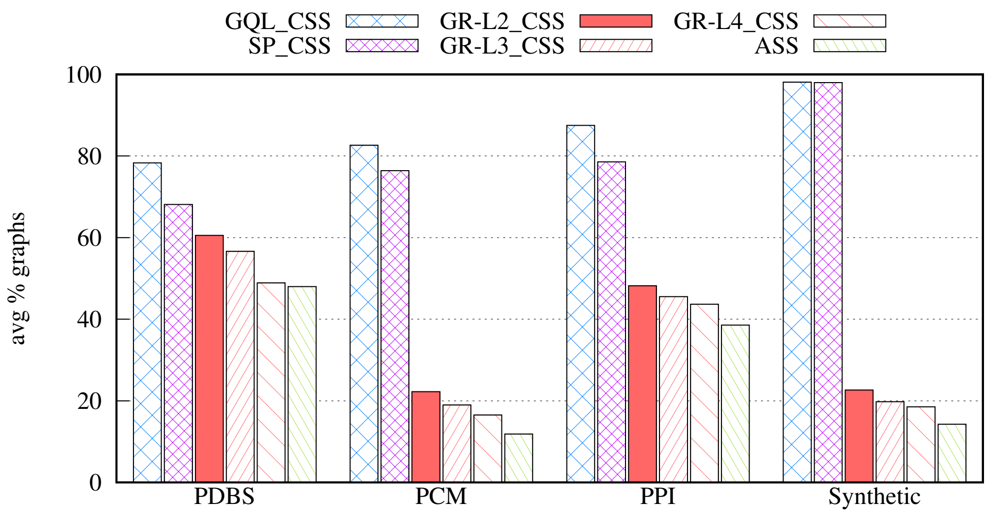


(b) Indexing Size

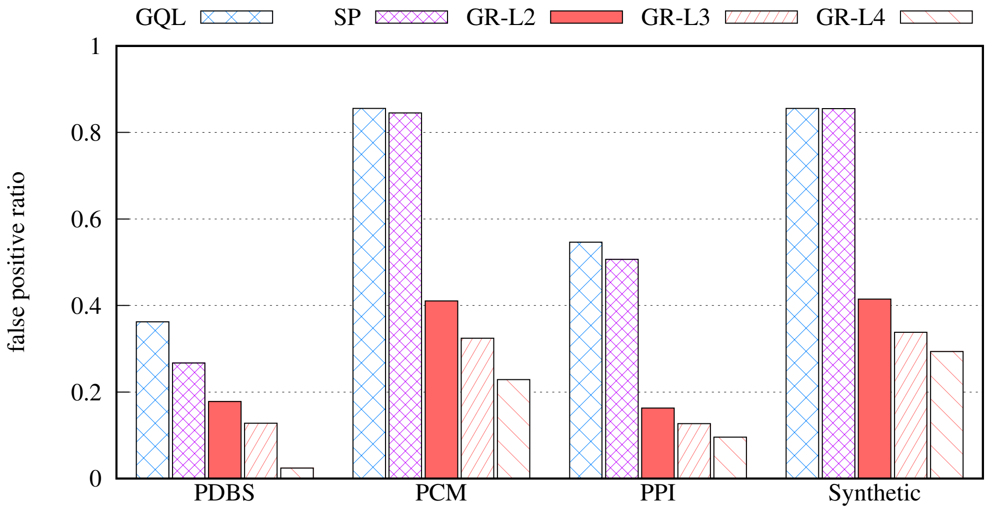
Figure 6.11: Tweaking the maxL parameter, index construction

up to 1 order of magnitude difference of the index size from $maxL = 2$ to 4. Thus, overall in the hybrid FTV-SI method, the use of smaller $maxL$ values results in adding only a trivial time and space overhead in the indexing phase.

Figures 6.12(a) and 6.12(b) present the pruning power of Grapes utilizing different feature sizes on all datasets. As it was expected, as we increase the size of the features, the candidate set size decreases and it affects accordingly the FPR. However, in all occasions the filtering power is better than that achieved by the SI methods. This is particularly evident in PPI, PCM and the Synthetic dataset. For these datasets and for all feature sizes, the candidate set size is very close to the answer set size. Thus, we also obtain relatively small FPR values. PDBS follows the same trends but with less steep divergence from the SI methods because of the high answer set size. This leads to the conclusion that we can still achieve



(a) Candidate and answer sets

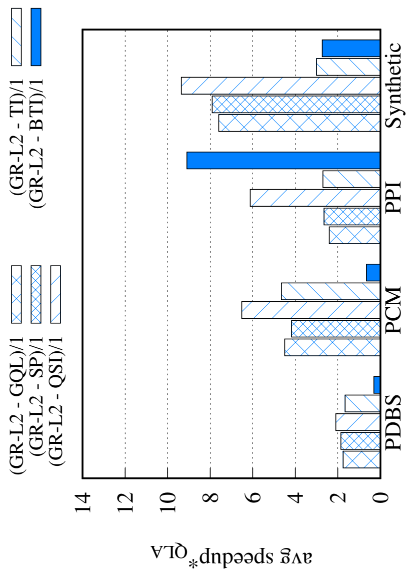


(b) False Positive Ratio

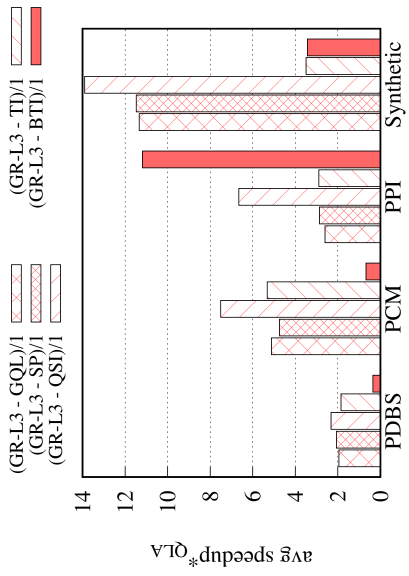
Figure 6.12: Tweaking the maxL parameter, filtering power

high speedups with smaller feature sizes.

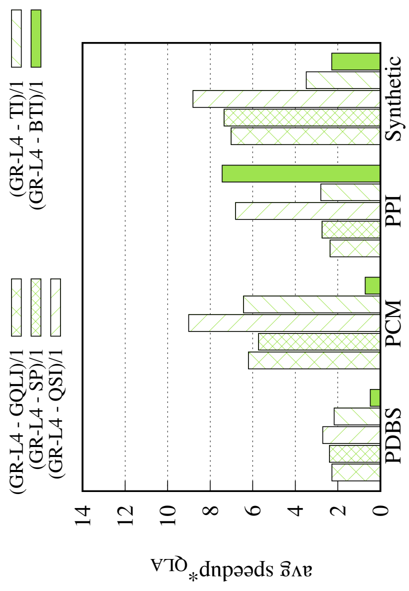
Finally, figures 6.13(c) - 6.14(f) report the achieved QLA and WLA speedups by tweaking the $maxL$ value of Grapes for the used datasets for our hybrid FTV-SI solution when utilizing 1 and 4 threads and for all query sizes. A notable observation here is that in some cases, with the sole exception of PDBS and PCM, the achieved speedups with smaller values of $maxL$ outperform the speedups with larger values of $maxL$. We attribute this to the fact that for smaller $maxL$ values less time is required to construct the index of the query and match it to the dataset's index. Additionally, the fact that the candidate set sizes that are formed after Grapes's filtering are close for the various $maxL$ values, contributes to this. For completeness, tables 6.3 and 6.4 provide additional statistics for min, max, median and stdDev of the achieved $speedup^*_{QLA}$ with different feature sizes and different number of



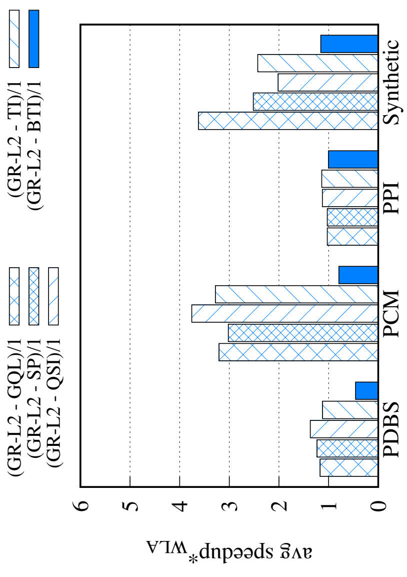
(a) Avg *speedup*$^*_{QLA}$, $maxL = 2$, 1 thread



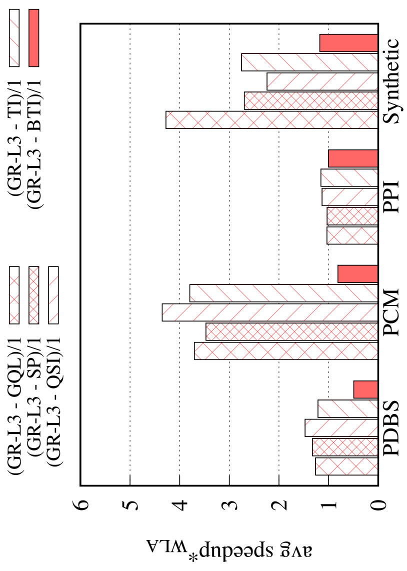
(b) Avg *speedup*$^*_{QLA}$, $maxL = 3$, 1 thread



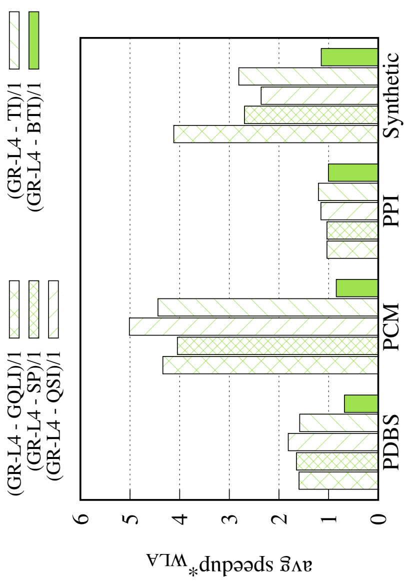
(c) Avg *speedup*$^*_{QLA}$, $maxL = 4$, 1 thread



(d) Avg *speedup*$^*_{WLA}$, $maxL = 2$, 1 thread

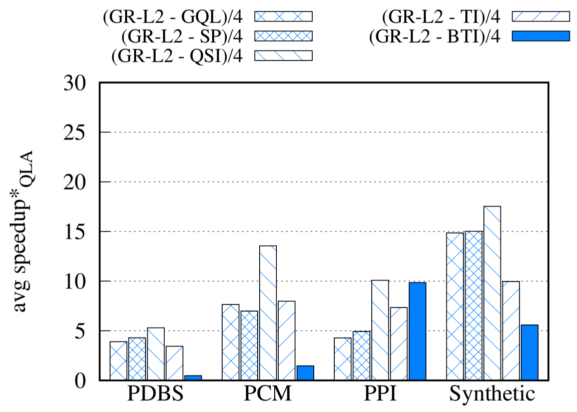


(e) Avg *speedup*$^*_{WLA}$, $maxL = 3$, 1 thread

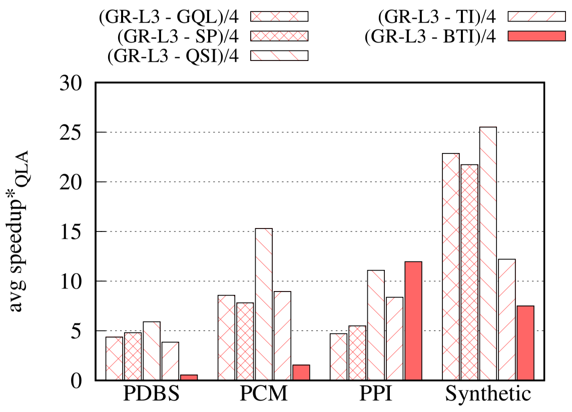


(f) Avg *speedup*$^*_{WLA}$, $maxL = 4$, 1 thread

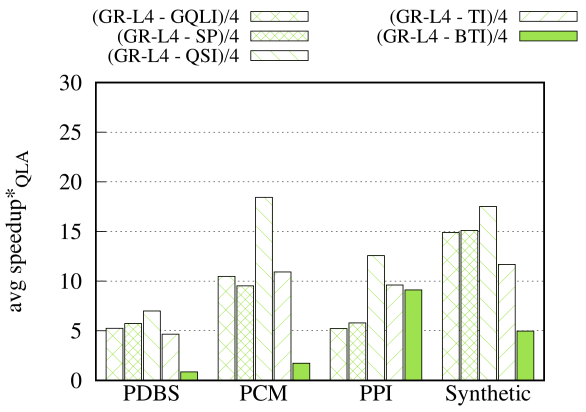
Figure 6.13: Tweaking the maxL parameter, achieved *speedup**, 1 thread



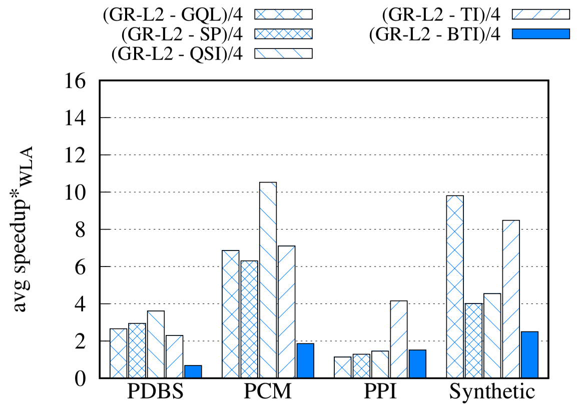
(a) Avg $speedup^*{}_{QLA}$, $maxL = 2$



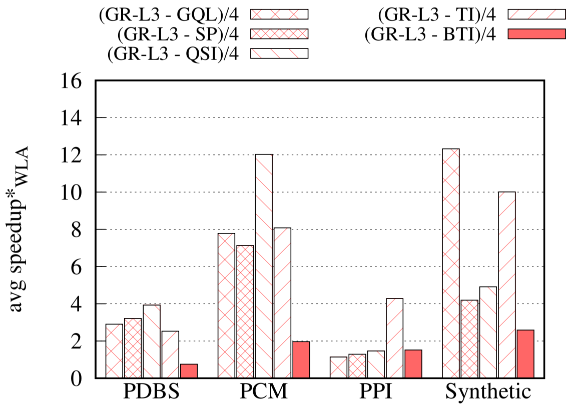
(b) Avg $speedup^*{}_{QLA}$, $maxL = 3$



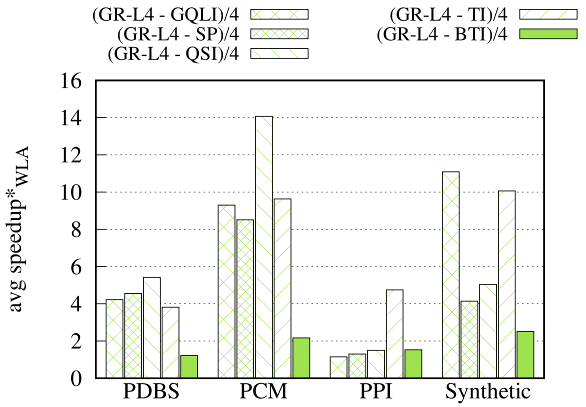
(c) Avg $speedup^*{}_{QLA}$, $maxL = 4$



(d) Avg $speedup^*{}_{WLA}$, $maxL = 2$



(e) Avg $speedup^*{}_{WLA}$, $maxL = 3$



(f) Avg $speedup^*{}_{WLA}$, $maxL = 4$

Figure 6.14: Tweaking the maxL parameter, achieved $speedup^*$, 4 threads

		1 thread					4 threads				
		GR-GQL	GR-SP	GR-QSI	GR-TI	GR-BTI	GR-GQL	GR-SP	GR-QSI	GR-TI	GR-BTI
PDBS	stdDev	1.481	1.689	2.185	1.332	0.215	2.253	2.612	3.669	2.024	0.418
	min	0.581	0.607	0.688	0.485	0.021	1.019	1.104	1.370	0.748	0.021
	max	9.712	11.853	16.612	8.838	0.849	12.758	16.189	22.375	10.904	1.608
	median	1.687	1.781	1.893	1.643	0.251	4.225	4.706	5.587	3.789	0.308
PCM	stdDev	2.509	2.247	4.864	2.661	0.221	2.378	2.047	6.292	2.556	0.822
	min	1.049	1.039	1.101	1.030	0.042	3.173	3.070	3.808	2.981	0.043
	max	11.155	10.022	21.071	12.203	1.050	13.468	11.685	29.372	14.227	3.242
	median	4.008	3.690	4.891	4.119	0.710	7.569	6.896	12.712	7.959	1.440
PPI	stdDev	3.515	6.060	36.828	3.234	151.385	3.563	11.606	54.506	8.728	151.346
	min	0.832	0.889	0.910	0.846	0.081	1.001	1.003	1.002	2.201	0.081
	max	30.759	8.574	98.490	22.104	225.010	30.759	7.716	89.218	71.103	325.01
	median	1.364	1.301	1.385	1.640	0.992	3.298	3.393	3.380	4.560	1.613
Synthetic	stdDev	8.549	13.667	22.341	1.901	21.319	9.238	16.897	38.426	5.134	30.581
	min	1.260	1.092	1.105	1.533	0.474	1.315	1.135	1.636	5.342	0.476
	max	51.635	101.997	37.179	10.595	62.139	54.019	123.504	112.209	29.812	59.015
	median	4.292	3.145	2.325	2.281	0.972	11.796	9.054	7.566	8.043	3.299

Table 6.3: $speedup^*_{QLA}$ statistics for FTV-SI combination with 1 and 4 threads, $maxL = 2$

		1 thread					4 threads				
		GR-GQL	GR-SP	GR-QSI	GR-TI	GR-BTI	GR-GQL	GR-SP	GR-QSI	GR-TI	GR-BTI
PDBS	stdDev	1.645	1.869	2.374	1.470	0.304	2.491	2.890	4.034	2.201	0.517
	min	0.615	0.640	0.712	0.530	0.022	1.140	1.232	1.529	0.863	0.022
	max	10.011	12.068	15.231	9.139	2.509	13.279	16.594	23.319	11.366	3.368
	median	1.793	1.852	2.016	1.762	0.278	4.581	5.105	6.023	4.198	0.350
PCM	stdDev	2.735	2.447	5.419	2.914	0.221	2.252	2.113	6.750	2.671	0.854
	min	1.094	1.114	1.175	1.119	0.040	3.385	3.383	4.128	3.491	0.041
	max	11.553	11.026	22.261	12.935	1.056	13.622	12.483	30.901	15.242	3.313
	median	4.568	4.361	5.859	4.791	0.736	8.691	7.986	14.445	9.093	1.548
PPI	stdDev	4.069	7.699	38.234	3.467	189.837	4.128	17.364	56.909	10.298	189.797
	min	0.841	0.918	0.936	0.857	0.097	1.001	0.003	0.003	2.401	0.098
	max	38.783	3.419	60.949	25.994	92.800	38.783	5.288	76.124	82.486	392.8
	median	1.491	1.388	1.483	1.772	0.994	3.764	3.723	3.584	5.078	1.667
Synthetic	stdDev	16.481	25.057	47.367	2.431	30.414	19.101	30.924	67.676	7.374	52.543
	min	1.273	1.096	1.140	1.590	0.804	1.324	1.136	1.639	5.729	0.809
	max	73.571	78.934	37.949	15.383	50.745	13.631	201.799	78.102	47.372	80.154
	median	5.313	3.671	2.670	2.648	0.990	16.192	10.827	8.558	9.674	3.577

Table 6.4: $speedup^{*}{}_{QLA}$ statistics for FTV-SI combination with 1 and 4 threads, $maxL = 3$

threads and along with tables 6.1 and 6.2 (presented earlier in§6.7 and §6.8) complete the picture. We note that BTI is again the least benefited algorithm, as evident in the retrieved values of average, min and median $speedup^*_{QLA}$ values. In fact it is rather harmed than improved by the use of smaller feature sizes, i.e., $maxL = 2$ or $maxL = 3$ in the vast majority of cases. In PDBS particularly, BTI with the use of smaller feature sizes is not even benefited by the use of 4 threads.

Undoubtedly, the aforementioned speedups bring about additional changes in the average query execution times, as discussed earlier in §6.7. Thus, the overall conclusion, with the sole exception of BTI, is that the use of smaller feature sizes in the hybrid FTV-SI method is not just a good compromise of the achieved speedups. This solution offers additional benefits in the indexing phase by reducing significantly both the index size and time, as discussed earlier in this section.

6.10 Conclusions

The current trend in research for subgraph pattern queries in graph DBs dismisses FTV methods since the fast subgraph isomorphism heuristics of SI methods significantly outperform FTV methods. We analyzed the problem by answering a set of fundamental questions. Specifically, we initially investigated the index time and space requirements from both FTV and SI methods. Subsequently, we showed that the filtering power of a top-performing FTV algorithm (Grapes) is significantly better than that of all SI methods. Having that in mind and knowing that the subgraph isomorphism testing can be very expensive as the graph DB grows large (in number or size of stored graphs), we set out to initially evaluate the performance of well known SI methods. Our experiments reveal no single winner across all datasets. We then evaluate the performance of a hybrid FTV-SI solution, that incorporates the powerful filtering of the FTV methods with the fast subgraph-isomorphism algorithm of the SI methods. This hybrid proves to be a better practice compared to the traditional SI methods over datasets consisting of a large number of graphs by avoiding redundant and possibly expensive sub-iso tests in the whole DB, and thus by bringing about significant speedups and changes in algorithms' relative performance with not yet a single winner. However, gained benefits come at the extra costs of index memory space and indexing time. We then further analyzed this hybrid method in two dimensions: First, to reduce the space-time index costs, we lowered the size of indexed features. Our results revealed that the filtering power of the FTV index is still much higher than that of SI methods and that high speedups can be achieved, even with these smaller indexes, which are in turn even smaller than the SI indexes. Second, as the time to perform index-based filtering is substantial, we studied the positive effects of doing this in parallel. Our results showed that expected speedups can thus be significantly boosted. Parallelizing this filtering step is a much easier task than parallelizing the actual subgraph-isomorphism algorithm. Overall, this work surfaces new promising possibilities for expediting subgraph queries in graph DBs by experimentally revealing a blind spot in current thinking. We hope this will inspire new research targeting new FTV-style indexes and/or SI-style subgraph-isomorphism algorithms for FTV-SI hybrids.

Limitations The current work is limited to the use of three real and a synthetic datasets. A systematic study, as performed in chapter §4, would be useful to (i) perform a systematic evaluation of existing SI algorithms and the hybrid FTV-SI method and (ii) pinpoint the scalability limitations. Regarding the scalability limits, we also need to identify the existence of straggler-queries (as discussed in chapter §5). Then, we could combine the Ψ-framework (§5.8) and the hybrid FTV-SI combination method (§6.7 and §6.8) to extend the scalability limitations of the originally proposed FTV and SI methods. This will be further discussed in §7.2.

Another limitation of our study is related to the values of the input parameters of the employed SI algorithms (see §4.3.2). We would expect that varying the input parameters on the SI methods (see §6.3.2) would have similar effects as varying the input parameters on the FTV methods (see §6.9).

Chapter 7

Conclusion and Future Steps

After numerous experiments and all the knowledge we have gained so far, we are ready to conclude this thesis. In the current chapter, we briefly review our findings and our major contributions. Of course, there are still open questions and space for improvement. Thus, in the end we set the future steps to be conducted.

7.1 Summary of Contributions

Graphs have great representation power in representing complex structures and their interactions. A common problem that is addressed to such graphs is the subgraph pattern matching problem. Over the years, significant work has been conducted in the field of subgraph matching queries, which entails subgraph isomorphism, a well-known NP-Complete problem. Related work, that was extensively discussed in chapter §2, is classified in two major categories: the FTV methods that typically address the decision version, and SI methods that usually address the matching version of the problem. We have seen that a number of such methods is added in the bibliography annually, that tend to totally dismiss older proposed methods and instead present new ideas with the aim to surpass the performance of former work. With our experiments, we show that both FTV and SI methods show significant limitations in their performance as we were increasing the parameters of the problem, i.e. the number of nodes and / or density per graph, the number of graphs in the dataset, and the size of the query. Thus, in the current thesis we propose that instead of devising new algorithms, we should consider rewriting the original query and / or combining existing, top-performing algorithms appropriately in order to achieve large performance gains.

More specifically, in chapter §4, we conducted a set of experiments with well-known and top-performing FTV methods with an emphasis on newer proposed methods. We have employed four real datasets and many synthetic ones, generated with the well-known GraphGen[60]. We have identified a set of key-factor parameters that influence the performance of FTV methods and in general the underlying subgraph isomorphism test and these are the number of nodes and density per graph, the number of distinct labels and number of graphs in the dataset, along with the size of the query. The primary aim of our experiments was to study their performance and sensitivity in the gradual increasing key-factor parameters (§4.4). Additionally, we stress-tested the various FTV methods by pinpointing points where some algorithms continue to operate whereas others break. Overall, from our analysis we gained the following insights:

- The first major lesson is related to the effect of the key dataset characteristics. The intuition is verified; increasing the number of graphs (see figures 4.10 and 4.11) leads to a linear increase in the problem's complexity. The frequent mining techniques are more severely affected, as more features have to be located across more graphs. By complexity, we mean the indexing time and size and the query processing time. By increasing the number of nodes and density, the complexity increases in a super-linear way (see figures 4.3 and 4.4, 4.5 and 4.6). However, increasing the number of labels leads to an easier problem, because there are more distinct features and the filtering works better (see figures 4.8 and 4.9). Finally, by increasing the size of the queries, the query processing becomes harder, with the effect being more profound on dense graphs and specifically on frequent mining techniques (see figures 4.4, 4.6 and 4.11 and §4.5.1).
- The second major lesson is that simplicity wins (see §4.5.2). Many methods advocate that graphs are more expressive than trees and trees are more expressive than paths accordingly, as graphs can maintain more structural information. We agree with this claim. However, considering that all paths are trees but the opposite does not hold, given any graph in the dataset and similarly for graphs, the number of produced subgraphs of the same size is higher than the number of trees of the same size and accordingly for paths. If the methods utilizing more complex structures had to store everything, then the indexing time, index size and filtering time would be much higher. Thus, all methods employing complex structures maintain only a subset or apply encoding (see description of gIndex, Tree+Δ, gCode and CT-Index in §3.1.1). As a result, their coverage is decreased, i.e., some features are not represented in the index. On the other side, methods that rely on simpler features (paths), compared to the frequent mining techniques can index all the features up to a certain size, in the cost of an increased constructed index (Grapes and GGSX in figures 4.3(b), 4.5(b), 4.8(b) and 4.10(b)). In the end, exhaustive enumeration techniques have more or less the

same behavior with frequent mining techniques in the filtering, but because of their simplicity, they win in both indexing and query processing time.

- The choice of the right algorithm relies on optimizing different aspects of the problem and these are: efficiency in terms of indexing time, index size, query processing time and scalability limitations. Based on our experiments, in most cases, Grapes and GGSX, utilizing the simplest features, are the winners, as discussed in §4.5.3. However their index size requirements are the highest among all contestants.
- In the relevant bibliography [7], all methods were tested with small and sparse datasets. With our experiments, we have seen that at datasets consisting of larger and denser graphs, these algorithms do not scale. At larger scales, one should consider other options such as devising new algorithms, varying the input parameters of top-performing methods and/or combining existing top-performing algorithms to form a better one, as discussed in §4.5.4.

In chapter §5, we focused on the various subgraph isomorphism tests. We conducted a set of experiments with top-performing FTV, namely Grapes and GGSX, and SI methods, namely GraphQL, sPath, QuickSI, TurboIso and BTI, and we presented key novel discoveries and observations of the nature of the subgraph isomorphism problem. Specifically, we made the following key observations.

- Related works present the average query execution time, calculated as the query workload time divided by the number of the queries in the workload to showcase the performance of their method compared to others. However, such an approach can conceal the real execution times of individual queries. Specifically, with our work we have seen that as the dataset grows large in terms of number of nodes and/or density, query processing becomes harder (see figures 5.1 and 5.3 and tables 5.1 and 5.2). We conducted experiments with a single query against a single large stored graph and we have seen that both FTV and SI methods suffer from straggler queries, i.e., queries with execution times many orders of magnitude worse compared to the majority (see figures 5.2 and 5.4). Straggler queries appear in a small percentage of the total query workloads, but when execution times are averaged, the most expensive queries dominate the overall processing time.
- Isomorphic queries can have widely and wildly different execution times (see figures 5.5 and 5.7 and discussion in §5.5). We recall that for the generation of an isomorphic query to the original query, one can simply permute the node IDs of the original query, as discussed in §2.2. The reason for that is that all proposed methods do not define an absolutely strict order in which the nodes of the query are matched, because it would be too computationally expensive to compute a globally optimal join plan. Instead, proposed methods rely on heuristics to minimize the search space for the join plan (§5.5). We have seen that FTV methods are more vulnerable to such wild variations

as they propose a less strict order in which the nodes of the query are matched to the nodes of the stored graph compared to the SI methods.

- Finally, we have seen that straggler queries are algorithm specific, i.e., a straggler query on one algorithm could be a typical query on some other algorithm (figures 5.15). For our experiments, we employed the various SI methods. We have seen that the use of alternative algorithms is much more beneficial compared to the use of different isomorphic query rewritings, leading to straggler-free executions (figures 5.14 and 5.15).

Many of the issues observed are encountered with other NP-hard problems, such as *branch and bound*[116], as discussed in §2.7 and in §5.5, §5.8. We have seen that proposed algorithms show a wide variance in their execution times as they employ different heuristics and are significantly affected by the search order they impose. Additionally, the way they resolve ties during their processing may further affect the search order, again resulting in unpredictable overall execution times.

Subsequently, we used our observations to make the following key contributions.

- We generated our own isomorphic query rewritings by permuting the query node IDs in a specific manner, as discussed in §5.6. Thus, we implemented and further experimented with 5 such rewritings, namely ILF, IND, DND, ILF+IND and ILF+DND that permute the node IDs in such a way that they take into consideration the frequencies of labels of the stored graph and/or the query node degrees. With our experiments, we have seen that these rewritings are beneficial across different algorithms and datasets (see figures 5.12 and 5.14).
- We proposed and experimented with the Ψ-framework which stands for Parallel Subgraph Isomorphism Framework, as discussed in §5.8. Specifically, we run in parallel different isomorphic instances of the same query and/or different algorithms by instantiating different threads. After the completion of any first thread, the rest of them are killed. To reduce the memory footprint, we experimented with various combinations and a number of threads. Such an execution leads to a performance improvement of many orders of magnitude compared to the original proposed methods.

Similar techniques to our Ψ-framework are applied to other NP-hard searches. These are known as portfolios of algorithms, as discussed in §2.7 and §5.8, where some portfolios pay-off more than others.

In chapter §6, we investigated the current trend that tends to totally dismiss FTV methods and instead employ SI methods, with the claim that SI methods enjoy much shorter execution times whereas FTV methods additionally suffer from elevated costs in managing their constructed index. Thus, we initially evaluated the performance of the aforementioned top-performing SI methods over graph DBs consisting of a large number of graphs. Our experiments in featuring a single winner were inconclusive (see figure 6.4). In parallel, we investigated the time and size costs (figure 6.1) related to the constructed index of both the SI methods and Grapes – the top-performing FTV method in terms of indexing time, query processing time, scalability limitations and filtering power, as shown in chapter §4 – along with the filtering power (figure 6.3) achieved from all aforementioned methods. Having seen that the filtering achieved by Grapes is much higher than that of the SI methods, we set out to combine the powerful filtering of Grapes, with the fast subgraph isomorphism of the various SI methods. Subsequently, we studied the overall query processing time of this hybrid combination of our in use FTV method and the various SI methods (figures 6.6, 6.8 and 6.10). Additionally, taking into consideration that the constructed index of Grapes can have an elevated cost both in time and size, we study various trade-offs involved in the process by varying the size of the enumerated features, as discussed in §6.9. Thus, we repeated all aforementioned experiments and we quantified the indexing time, size, filtering power and achieved speedups using smaller enumerated features compared to what was initially proposed by the respective authors. Our experiments reveal that the efficiency of this hybrid FTV-SI combination is not compromised by employing smaller features. In fact, not only the overall speedups were significantly boosted even with very small feature size, but also the cost of the constructed index was reduced up to 3 orders of magnitude both in time and size, while the filtering power remained of high quality and still much better compared to that of the SI methods.

In chapter §4 and specifically in §4.5.4, we identified significant scalability limitations on top-performing FTV methods. Specifically, we identified both cases where (i) the constructed index was not adequate to secure the execution of query processing in reasonable time at all times, such as in Grapes in the case of increasing the number of nodes beyond 800 nodes (figures 4.3 and 4.4) and (ii) the index construction was not completed in reasonable time because of either excessive time or of space restrictions, as discussed in §4.4. Although, we have not conducted a similar systematic survey in the SI methods, we have identified similar scalability limitations on the SI methods, especially because of the straggler queries, that were identified in chapter §5 and specifically in §5.4. Thus, apart from the aforementioned contributions, with our work we managed to extend the scalability limitations in various ways. Specifically, our proposed Ψ-Framework, presented in §5.8, can ameliorate the problem in the cases that the index could be constructed, but queries could not be answered in reasonable time by some of the FTV or SI methods. Similar to that, our hybrid FTV-SI combination, as presented in §6.7 can be efficiently used in such cases. In the cases that the index could not be constructed, our hybrid FTV-SI combination with smaller sizes of enumerated features is a good option, because the index size and time are significantly reduced whereas the filtering power remains of good quality, as discussed in §6.9.

7.2 Limitations and Future Work

Despite the large number of experiments performed, there are some limitations on the existing work that leave space for improvement on the above proposed solutions for the subgraph matching problem. We now discuss the limitations and we set the future steps to follow.

With the conducted experiments in chapter §4, the initial target was to be able to answer exactly which algorithm to use given any set of parameters, i.e. average number of nodes and density, number of distinct labels and graphs in the dataset. Instead, in our research we establish a set of the "sane" default parameters as discussed in §4.3.3 where we vary only one parameter at a time to examine its effect on the various metrics and algorithms. Finding the exact algorithm to use for any set of given parameters proved to be impractical because the dataset workload characteristics are not enough to let us select the best algorithm to execute. In more detail, we have seen in chapter §5, even different rewritings of a query – i.e., isomorphic instances of the same query – can lead to different algorithms performing the "best".

In chapter §5 and especially in §5.6 and §5.8, our study is limited to the use of 5 different query rewritings. On top of that, we currently break ties by respecting the order of nodes on the original queries, as discussed in §5.6, instead of generating all possible combinations in the case of ties. Thus, additional query rewritings could be introduced, that would introduce new criteria for breaking ties, along with combining existing rewritings in a different sequence, e.g. IND+ILF instead of ILF+IND that we considered. Finally, in this chapter we have not considered rewritings of the stored graphs, which given the size of the stored graphs can be impractical. Additionally a much larger number of ties is expected compared to the query graphs that would have to be resolved with more complex rewritings.

Experiments in chapter §6 are limited in the use of three real and a synthetic dataset. Thus, a systematic study, as performed in chapter §4, would be useful, so as (i) to perform a systematic evaluation of existing SI algorithms and the hybrid FTV-SI method and (ii) to indicate the scalability limitations.

For the majority of experiments, with few exceptions (see §6.9), we rely on and employ algorithm's input parameters as defined by the respective authors, as discussed in §4.3.2. Taking the experiments we performed in §6.9 into consideration, it is shown that varying these parameters can significantly affect the algorithms' performance. Although the suggested input parameters for the various methods are defined by the respective authors, defining the optimal input parameters, requires deep knowledge of the algorithm, the underlying implementation and the problem to solve. Thus, it is often hard even for the authors of these methods to answer what are the optimal parameters to use. The experimentation for finding the optimal values to use for every case is highly impractical. Thus, devising a framework to use that would be able to auto-adjust the optimal values for answering queries in the most effective way on a specific dataset is left for future work.

Thus, apart from the aforementioned limitations that could be addressed, we propose the following two ideas for future work:

Combining the benefits of both the hybrid FTV-SI solution and the Ψ-framework

We concluded the previous section of contributions (§7.1) by stating that our proposed solutions – the hybrid FTV-SI solution and the Ψ-framework – can extend the scalability limitations of the originally proposed FTV and SI methods. But instead of only applying these solutions separately, we could further combine them for better results. Specifically, we have seen that the hybrid FTV-SI solution aims primarily at filtering out graphs that do not contain the query as an answer rapidly and efficiently. Thus, for the datasets consisting of a large number of graphs, we initially employ the hybrid FTV-SI solution as it was presented in §6.7. However, in the verification stage, instead of executing a single SI method against the original query, we could employ our Ψ-framework as it was presented in §5.8. Such a combination requires a set of additional experiments to define how the scalability limitations have been extended. Furthermore, in these experiments it is essential to quantify the additional space required for the parallel executions.

Identifying the right isomorphic query instance and/or right SI algorithm to employ

As we mentioned earlier in chapter §5, our Ψ-framework runs in parallel different isomorphic instances of the same query and/or alternative algorithms, with the generated isomorphic instances adding only a trivial overhead in the whole process. Although such a solution provides an easy fix to the problem of straggler-queries, it can have a non-negligible memory footprint which is proportional to the number of parallel executions that are instantiated. To tackle this problem, we need to identify the right algorithm to execute and/or the right isomorphic instance of the original query to use. In other words, we need to identify which queries are difficult for which algorithms and/or which instances of the same query are difficult for a specific algorithm. To achieve this, we need to identify the characteristics of the stored graphs and of the queries that lead to difficult instances in the subgraph matching problem. Then, we could feed these characteristics in a machine learning model that could predict the right algorithm and/or right isomorphic query to employ. Such a solution would resolve the problem of the large memory footprint.

Bibliography

[1] National Cancer Institute - DTP AIDS antiviral screen dataset, http://dtp.nci.nih.gov/docs/aids/aids_data.html.

[2] E. E. Bolton, Y. Wang, P. A. Thiessen, and S. H. Bryant, “Pubchem: integrated platform of small molecules and biological activities,” *Annual reports in computational chemistry*, vol. 4, pp. 217–241, 2008.

[3] J. L. et al., “Stanford network analysis project,” https://snap.stanford.edu/data/loc-gowalla.html, 2010.

[4] Neo4j Team, “Neo4j,” http://www.neo4j.org/.

[5] L. Garulli, “Orientdb,” 2012, http://www.orientdb.org/.

[6] R. G. Michael and S. J. David, *Computers and intractability: a guide to the theory of NP-completeness*. San Francisco, LA: Freeman, 1979.

[7] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu, “iGraph: a framework for comparisons of disk-based graph indexing techniques,” *PVLDB*, vol. 3, no. 1-2, pp. 449–459, 2010.

[8] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, “An in-depth comparison of subgraph isomorphism algorithms in graph databases,” *PVLDB*, vol. 6, no. 2, pp. 133–144, 2012.

[9] F. Katsarou, N. Ntarmos, and P. Triantafillou, “Performance and scalability of indexed subgraph query processing methods,” *PVLDB*, vol. 8, no. 12, pp. 1566–1577, 2015.

[10] ——, “Subgraph Querying with Parallel Use of Query Rewritings and Alternative Algorithms,” in *Proc. ACM EDBT*, 2017, pp. 25–36.

[11] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha, "GRAPES: A software for parallel searching on biological graphs targeting multi-core architectures," *PloS One*, vol. 8, no. 10, p. e76911, 2013.

[12] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha, "Enhancing graph database indexing by suffix tree structure," in *Proc. IAPR PRIB*. Springer, 2010, pp. 195–203.

[13] F. Katsarou, N. Ntarmos, and P. Triantafillou, "Towards Hybrid Methods for Graph Pattern Queries," in *Proc. GraphQ Workshop EDBT/ICDT*, 2017.

[14] Facebook Graph API, https://developers.facebook.com/docs/graph-api.

[15] C. Vehlow, H. Stehr, M. Winkelmann, J. M. Duarte, L. Petzold, J. Dinse, and M. Lappe, "Cmview: Interactive contact map visualization and analysis," *Bioinformatics*, 2011.

[16] Y. He, et al., "Structure of decay-accelerating factor bound to echovirus 7: a virus-receptor complex," *Proc. National Academy of Sciences of the United States of America*, 2002.

[17] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne, "The protein data bank," *Nucleic acids research*, 2000.

[18] M. Ashburner, C. A. Ball, J. A. Blake, D. Botstein, H. Butler, J. M. Cherry, A. P. Davis, K. Dolinski, S. S. Dwight, J. T. Eppig *et al.*, "Gene ontology: tool for the unification of biology," *Nature genetics*, vol. 25, no. 1, pp. 25–29, 2000.

[19] C. Zhang, K. Hanspers, A. Kuchinsky, N. Salomonis, D. Xu, and A. R. Pico, "Mosaic: making biological sense of complex networks," *Bioinformatics*, vol. 28, no. 14, pp. 1943–1944, 2012.

[20] Khan, Arijit and Wu, Yinghui, "Graph Pattern Matching Queries – Approximation and User-Friendliness," http://wp.sigmod.org/?p=2202.

[21] S. Kijima, Y. Otachi, T. Saitoh, and T. Uno, "Subgraph isomorphism in graph classes," *Discrete Mathematics*, vol. 312, no. 21, pp. 3164–3173, 2012.

[22] P. Heggernes, P. van't Hof, D. Meister, and Y. Villanger, "Induced subgraph isomorphism on proper interval and bipartite permutation graphs," *Theoretical Computer Science*, vol. 562, pp. 252–269, 2015.

[23] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *PVLDB*, vol. 5, no. 9, pp. 788–799, 2012.

[24] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," in *Proc. SIGMOD*, 2008, pp. 405–418.

[25] H. S. de Andrade and C. L. Sales, "Pattern match query in a large graph database," *Encontros Universitários da UFC*, vol. 2, no. 1, p. 1544, 2009.

[26] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum, "Naga: Searching and ranking knowledge," in *Proc. ICDE*, 2008, pp. 953–962.

[27] W. Wu, H. Li, H. Wang, and K. Q. Zhu, "Probase: A probabilistic taxonomy for text understanding," in *Proc. SIGMOD*, 2012, pp. 481–492.

[28] S. Zhang, J. Yang, and W. Jin, "SAPPER: subgraph indexing and approximate matching in large graphs," *PVLDB*, vol. 3, no. 1-2, pp. 1185–1194, 2010.

[29] P. Zhao and J. Han, "On graph query optimization in large networks," *PVLDB*, vol. 3, no. 1-2, pp. 340–351, 2010.

[30] L. Lai, L. Qin, X. Lin, and L. Chang, "Scalable subgraph enumeration in mapreduce," *PVLDB*, vol. 8, no. 10, pp. 974–985, 2015.

[31] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang, "Scalable distributed subgraph enumeration," *PVLDB*, vol. 10, no. 3, pp. 217–228, 2016.

[32] R. Giugno and D. Shasha, "GraphGrep: A fast and universal method for querying graphs," in *Proc. ICPR*, 2002, pp. 112–115.

[33] R. Di Natale, A. Ferro, R. Giugno, M. Mongiovì, A. Pulvirenti, and D. Shasha, "Sing: Subgraph search in non-homogeneous graphs," *BMC Bioinformatics*, 2010.

[34] L. Zou, L. Chen, J. X. Yu, and Y. Lu, "A novel spectral coding in a large graph database," in *Proc. ACM EDBT*, 2008, pp. 181–192.

[35] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *PVLDB*, vol. 1, no. 1, pp. 364–375, 2008.

[36] H. He and A. K. Singh, "Closure-tree: An index structure for graph queries," in *Proc. ICDE*, 2006, pp. 38–38.

[37] S. Zhang, M. Hu, and J. Yang, "TreePi: A Novel Graph Indexing Method," in *Proc. ICDE*, 2007, pp. 966–975.

[38] J. Cheng, Y. Ke, W. Ng, and A. Lu, "FG-index: towards verification-free query processing on graph databases," in *Proc. SIGMOD*, 2007, pp. 857–872.

[39] D. W. Williams, J. Huan, and W. Wang, “Graph database indexing using structured graph decomposition,” in *Proc. ICDE*, 2007, pp. 976–985.

[40] Y. Xie and P. Yu, “CP-Index: on the efficient indexing of large graphs,” in *Proc. CIKM*, 2011, pp. 1795–1804.

[41] X. Yan, P. S. Yu, and J. Han, “Graph indexing: a frequent structure-based approach,” in *Proc. SIGMOD*, 2004, pp. 335–346.

[42] D. Yuan and P. Mitra, “Lindex: a lattice-based index for graph databases,” *VLDBJ*, vol. 22, no. 2, pp. 229–252, 2013.

[43] P. Zhao, J. X. Yu, and P. S. Yu, “Graph indexing: tree + delta >= graph,” in *PVLDB*, 2007, pp. 938–949.

[44] K. Klein, N. Kriege, and P. Mutzel, “CT-index: Fingerprint-based graph indexing combining cycles and trees,” in *Proc. ICDE*, 2011, pp. 1115–1126.

[45] A. Ferro, et al., “Graphfind: enhancing graph searching by low support data mining techniques,” *BMC Bioinformatics*, vol. 9, no. Suppl 4, p. S10, 2008.

[46] X. Yan and J. Han, “gSpan: Graph-based substructure pattern mining,” in *IEEE ICDM*, 2002, pp. 721–724.

[47] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, “A (sub) graph isomorphism algorithm for matching large graphs,” *IEEE TPAMI*, vol. 26, no. 10, pp. 1367–1372, 2004.

[48] B. D. McKay, “Nauty user’s guide (version 2.4),” *Computer Science Dept., Australian National University*, pp. 225–239, 2007.

[49] J. R. Ullmann, “An algorithm for subgraph isomorphism,” *Journal of the (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[50] B. D. McKay *et al.*, “Practical graph isomorphism,” 1981.

[51] B. D. McKay and A. Piperno, “Practical graph isomorphism, ii,” *Journal of Symbolic Computation*, vol. 60, pp. 94–112, 2014.

[52] S. Zhang, S. Li, and J. Yang, “GADDI: Distance Index Based Subgraph Matching in Biological Networks,” in *Proc. ACM EDBT*, 2009, pp. 192–203.

[53] C. R. Rivero and H. M. Jamil, “Efficient and scalable labeled subgraph matching using sgmatch,” *Knowledge and Information Systems*, pp. 1–27, 2016.

[54] M. Saltz, A. Jain, A. Kothari, A. Fard, J. A. Miller, and L. Ramaswamy, "DualIso: An algorithm for subgraph pattern matching on very large labeled graphs," in *IEEE BigData Congress*, 2014, pp. 498–505.

[55] W.-S. Han, J. Lee, and J.-H. Lee, "Turbo$_{iso}$: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proc. SIGMOD*, 2013, pp. 337–348.

[56] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *PVLDB*, vol. 8, no. 5, pp. 617–628, 2015.

[57] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proc. SIGMOD*, 2016, pp. 1199–1214.

[58] P. Peng, L. Zou, L. Chen, X. Lin, and D. Zhao, "Answering subgraph queries over massive disk resident graphs," *WWW*, vol. 19, no. 3, pp. 417–448, 2016.

[59] C. McCreesh, "Solving hard subgraph problems in parallel," Ph.D. dissertation, School of Computing Science, University of Glasgow, 2017.

[60] J. Cheng, Y. Ke, and W. Ng, "GraphGen," http://www.cse.ust.hk/graphgen/, 2007.

[61] B. Suo, Z. Li, Q. Chen, and W. Pan, "Towards scalable subgraph pattern matching over big graphs on mapreduce," in *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2016, pp. 1118–1126.

[62] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proc. ACM SIGMOD*, 2013, pp. 505–516.

[63] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[64] Y. Tian and J. M. Patel, "Tale: A tool for approximate large graph matching," in *Proc. ICDE*, 2008, pp. 963–972.

[65] X. Yan, F. Zhu, P. S. Yu, and J. Han, "Feature-based similarity search in graph structures," *ACM TODS*, vol. 31, no. 4, pp. 1418–1453, 2006.

[66] D. Pal, P. Rao, V. Slavov, and A. Katib, "Fast processing of graph queries on a large database of small and medium-sized data graphs," *Journal of Computer and System Sciences*, vol. 82, no. 6, pp. 1112–1143, 2016.

[67] Y. Tian, R. C. Mceachin, C. Santos, J. M. Patel *et al.*, "SAGA: a subgraph matching tool for biological graphs," *Bioinformatics*, vol. 23, no. 2, pp. 232–239, 2007.

[68] X. Zhao, C. Xiao, X. Lin, W. Wang, and Y. Ishikawa, "Efficient processing of graph similarity queries with edit distance constraints," *VLDBJ*, vol. 22, no. 6, pp. 727–752, 2013.

[69] Z. Zeng, A. K. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing stars: On approximating graph edit distance," in *Proc. VLDB*, vol. 2, no. 1, 2009, pp. 25–36.

[70] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan, "NeMa: Fast graph search with label similarity," in *Proc. VLDB*, vol. 6, no. 3, 2013, pp. 181–192.

[71] H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang, "Connected substructure similarity search," in *ACM SIGMOD*, 2010, pp. 903–914.

[72] A. Khan and Y. Wu, "Graph pattern matching queries – approximation and user-friendliness," http://wp.sigmod.org/?p=2202.

[73] D. Yuan, P. Mitra, and C. L. Giles, "Mining and indexing graphs for supergraph search," *PVLDB*, vol. 6, no. 10, pp. 829–840, 2013.

[74] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu, "Towards graph containment search and indexing," in *PVLDB*, 2007, pp. 926–937.

[75] G. Zhu, X. Lin, W. Zhang, W. Wang, and H. Shang, "Prefindex: an efficient supergraph containment search technique," in *Scientific and Statistical Database Management*. Springer, 2010, pp. 360–378.

[76] J. Cheng, Y. Ke, A. W.-C. Fu, and J. X. Yu, "Fast graph query processing with a low-cost index," *VLDBJ*, vol. 20, no. 4, pp. 521–539, 2011.

[77] J. Wang, N. Ntarmos, and P. Triantafillou, "Indexing query graphs to speedup graph query processing," in *Proc. ACM EDBT*, 2016, pp. 41–52.

[78] ——, "GraphCache: A Caching System for Graph Queries," in *Proc. ACM EDBT*, 2017, pp. 13–24.

[79] ——, "Ensuring consistency in graph cache for graph-pattern queries," in *Proc. GraphQ Workshop EDBT/ICDT*, 2017.

[80] M. Zhou, J. Yu, Y. Liu, Q. Dai, and L. Guo, "PatternTree$_{ISO}$: A Pattern Graph Correlation Framework for Accelerating Subgraph Isomorphism over Massive Graphs," in *Proc. CIKM*, 2016.

[81] X. Ren and J. Wang, "Multi-query optimization for subgraph isomorphism search," *PVLDB*, vol. 10, no. 3, pp. 121–132, 2016.

[82] W. Lin, X. Xiao, J. Cheng, and S. S. Bhowmick, "Efficient algorithms for generalized subgraph query processing," in *Proc. CIKM*, 2012, pp. 325–334.

[83] K. Semertzidis and E. Pitoura, "Durable graph pattern queries on historical graphs," in *Proc. ICDE*, 2016.

[84] A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," in *Principles of Data Mining and Knowledge Discovery*. Springer, 2000, pp. 13–23.

[85] M. Kuramochi and G. Karypis, "An efficient algorithm for discovering frequent subgraphs," *IEEE TKDE*, vol. 16, no. 9, pp. 1038–1051, 2004.

[86] J. Huan, W. Wang, J. Prins, and J. Yang, "SPIN: mining maximal frequent subgraphs from graph databases," in *ACM SIGKDD*, 2004, pp. 581–586.

[87] S. Nijssen and J. N. Kok, "A Quickstart in frequent structure mining can make a difference," in *ACM SIGKDD*, 2004, pp. 647–652.

[88] C. C. Aggarwal and H. Wang, "Graph data management and mining: A survey of algorithms and applications," in *Managing and mining graph data*. Springer, 2010, pp. 13–68.

[89] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph," *PVLDB*, vol. 7, no. 7, 2014.

[90] Y. Chi, Y. Yang, and R. R. Muntz, "Canonical forms for labelled trees and their applications in frequent subtree mining," *Knowledge and Information Systems*, vol. 8, no. 2, pp. 203–234, 2005.

[91] T. A. Junttila and P. Kaski, "Engineering an efficient canonical labeling tool for large and sparse graphs." in *SIAM ALENEX*, vol. 7, 2007, pp. 135–149.

[92] A. Piperno, "Search space contraction in canonical labeling of graphs," *Elsevier*, 2008.

[93] R. O. Obe and L. S. Hsu, *PostgreSQL: Up and Running: A Practical Introduction to the Advanced Open Source Database*. O'Reilly Media, Inc., 2014.

[94] K. Chodorow and M. Dirolf, *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2010.

[95] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. O' Reilly, 2013.

[96] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database system concepts*. McGraw-Hill New York, 2002, vol. 4.

[97] M. A. Rodriguez and P. Neubauer, "The Graph Traversal Pattern." *chapter in Graph Data Management: Techniques and Applications*, 2011.

[98] S. Jouili and V. Vansteenberghe, “An empirical comparison of graph databases,” in *International Conference on Social Computing (SocialCom).* IEEE, 2013, pp. 708–715.

[99] J. J. Miller, “Graph Database Applications and Concepts with Neo4j,” *Proc. of Southern Association for Information Systems*, 2013.

[100] Chris Gioran, “Digital Stain,” http://digitalstain.blogspot.co.uk/.

[101] M. Broecheler, D. LaRocque, M. A. Rodriguez, S. Mallette, and P. Yaskevich, “Titan,” 2012, http://titan.thinkaurelius.com/.

[102] R. Pointer, N. Kallen, E. Ceaser, and J. Kalucki, “Introducing flockDB,” 2010.

[103] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in *Proc. ACM SIGMOD*, 2010, pp. 135–146.

[104] Sakr, Sherif and Orakzai, Faisal Moeen and Abdelaziz, Ibrahim and Khayyat, Zuhair, “Apache Giraph,” https://giraph.apache.org/.

[105] M. Sarwat, S. Elnikety, Y. He, and G. Kliot, “Horton: Online query execution engine for large distributed graphs,” in *Proc. IEEE ICDE*, 2012, pp. 1289–1292.

[106] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, and C. Guestrin, “Graphlab: A distributed framework for machine learning in the cloud,” *arXiv preprint arXiv:1107.0922*, 2011.

[107] A. Kyrola, G. E. Blelloch, and C. Guestrin, “GraphChi: Large-scale graph computation on just a pc,” USENIX, 2012.

[108] D. Chakrabarti, Y. Zhan, and C. Faloutsos, “R-MAT: A recursive model for graph mining,” in *SIAM International Conference on Data Mining*, 2004, pp. 442–446.

[109] T. P. Peixoto, “The graph-tool python library,” *figshare*, 2014. [Online]. Available: https://graph-tool.skewed.de/

[110] G. van Rossum, “Python,” https://www.python.org/.

[111] Siek, Jeremy and Lee, Lie-Quan and Lumsdaine, Andrew, “boost,” http://www.boost.org/.

[112] V. Batagelj, “Stanford network analysis project,” http://snap.stanford.edu/, 2004.

[113] S. University, “Pajek,” http://vlado.fmf.uni-lj.si/pub/networks/pajek/, 2011.

[114] U. Kang, C. E. Tsourakakis, and C. Faloutsos, “PEGASUS: A peta-scale graph mining system implementation and observations,” in *Proc. IEEE ICDM*, 2009, pp. 229–238.

[115] , “Apache Hadoop,” http://hadoop.apache.org/.

[116] C. P. Gomes and B. Selman, “Algorithm portfolios,” *Artificial Intelligence*, vol. 126, no. 1, pp. 43 – 62, 2001, tradeoffs under Bounded Resources.

[117] B. Archibald, P. Maier, R. Stewart, P. Trinder, and J. De Beule, “Towards generic scalable parallel combinatorial search,” in *Proc. of the International Workshop on Parallel Symbolic Computation*. ACM, 2017, p. 6.

[118] Y. A. Liu and S. D. Stoller, “From recursion to iteration: what are the optimizations?” *ACM Sigplan Notices*, vol. 34, no. 11, pp. 73–82, 1999.

[119] A. Suleman, “Parallel programming: When Amdahl’s law is inapplicable,” *Future chips, June*, 2011.