University of Glasgow | School of Computing Science

Investigations of Subgraph Query Processing

Iva Stefanova Babukova

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 20, 2016

Abstract

Subgraph query processing is an important algorithmic problem in Big Data research with numerous applications in natural sciences and beyond. A comprehensive review of existing work is carried out, a choice of existing algorithms is implemented and evaluated and new algorithms “Light Filters” and “Path-Subtree Index” are proposed. Light Filters consist of 5 lightweight graph filtering tests and a simple and efficient subgraph isomorphism algorithm. The filtering tests are argued to be a good alternative to existing expensive filtering strategies, because of reasonably strong filtering power provided at significantly reduced memory and speed cost. Theoretical analysis of the proposed approach is given, and an empirical study based on existing benchmark datasets is carried out. The benchmark datasets are in themselves a subject of critical analysis.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

This Chapter gives definitions and concepts used in this work and states the aims and motivations of the project.

1.1 Terminology, Definitions and Notations

In this Section, we introduce all preliminary terminology and definitions used in this work. We start with basic introduction to graph theory, explaining the main problem that is discussed in this work, namely the subgraph isomorphism problem. Then, other concepts and notations are introduced, which are referred to later in this work.

1.1.1 Graph Theory

A graph G consists of a set of vertices V, a set of edges E, where each edge is a pair of vertices in V, and a *labeling function* L: V $\rightarrow \mathcal{L}$ that assigns a label l $\in \mathcal{L}$ to each vertex in V, where $\mathcal{L}$ is the set of all possible labels. Therefore, the set of all labels in G is L(V(G)). We write V(G) for the vertex set of G and E(G) for the set of edges in G. By L(G, v) = x we mean that vertex v in G has label x. We say that G is *undirected*, if every edge in E(G) is an unordered pair of elements of V(G). In this work, only undirected graphs are considered. The *size* of G is equal to the cardinality of E(G), denoted as m. The *order* of G, denoted as n, is equal to the cardinality of V(G).

Example 1 An example of undirected labeled graph is graph T on Figure 1.2, where the labels of T are the colors of its vertices. L(V(T)) is equal to red (R), yellow (Y) and blue (B).

A *sequence* is an ordered collection of objects in which repetitions are allowed. The *length* of a sequence is equal to the number of its objects. The position i of an element in a sequence A is called its index, denoted by A[i].

Example 2 An example of a sequence of integers, ordered in non-decreasing order, is $\{1, 2, 3, 4, 4\}$.

A *string* is a sequence of characters. Characters can be letters, numerical digits, punctuation marks, and whitespace.

Let A and B be sequences of integers. We say that B is a *subsequence* of A if B can be derived from A by deleting zero or more elements in A. A subsequence of a string is called a *substring*.

Example 3 Let A and B be two sequences of integers, where $A = \{1, 2, 5, 3, 8\}$ and $B = \{2, 5, 3\}$. Then B is a subsequence of A.

A *path* in a graph is a sequence of distinct vertices, such that each successive pair of vertices are adjacent (i.e. connected by an edge). A *cycle* is a path such that the first and the last vertices are adjacent. There may be zero, one or more distinct paths from vertex u to vertex v in G. The *length* of a path is equal to the number of its edges and the *length* of a cycle is the number of its edges incremented by 1.

Example 4 An example of a path in graph T on Figure 1.2 is the sequence $\{2, 3, 4, 5\}$, which is a path of length 3. The path consisting of vertices $\{2, 3, 8\}$ is an example of a cycle also of length 3.

The set of neighbours of a vertex v in G consists of all vertices adjacent to v. The *degree* of v is the cardinality of its set of neighbours. By $v \sim_G w$ we mean that vertex w is a neighbour of v in G. The set of neighbours of v forms the *neighbourhood* of v, denoted as N(G, v). The neighbourhood degree sequence of v, denoted nds(G, v), is a sequence consisting of the degrees of every neighbour of v, taken in non-increasing order.

A graph G is *connected* if there exists a path between any pair of vertices in G. An *acyclic* graph has no cycles. A *tree* is an acyclic connected graph. In this work, we refer to the vertices of the tree as *nodes*.

Example 5 Graph T on Figure 1.2 is connected. Let use look at vertex 2. Its degree is equal to 3, because vertex 2 is adjacent to three vertices, namely $2 \sim_T 1$, $2 \sim_T 3$ and $2 \sim_T 8$. Consequently, N(T, 2) = 1, 3, 8. nds(T, 2) = $\{8, 3, 1\}$

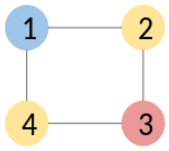


Figure 1.1: graph P

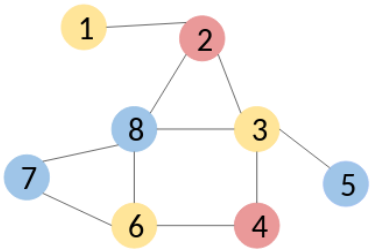


Figure 1.2: graph T

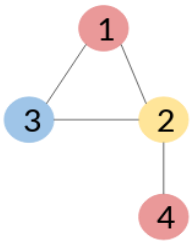


Figure 1.3: graph T1

Figure 1.4: Instances of subgraph isomorphism problem (SIP)

The *density* of a graph G measures what is the size of E(G) compared to the maximum possible number of edges between the vertices in V(G). Density for undirected graphs is calculated as follows:

$$Density(G) = \frac{2m}{n(n-1)} \tag{1.1}$$

This formula is derived from the fact that the maximum number of edges that G can contain is $\frac{n(|n-1)}{2}$, when G is a clique. If the m is increased, while fixing the number of vertices as a constant, the density of G becomes higher. Similarly, when n is increased and m is a constant, the density of G becomes lower.

Example 6 The density of graph T, with 10 edges and 8 vertices, on Figure 1.2 is 2.3.

1.1.2 The Subgraph Isomorphism Problem

A *subgraph* of G is a graph H whose vertices and edges are subsets of the vertices and edges of G and the labeling on the vertices is preserved. A *non-induced subgraph isomorphism* is an injective mapping $i : P \to T$ from a graph P to a graph T that preserves adjacency and labeling on vertices. That is, if $v \sim_P w$, then $i(v) \sim_T i(w)$ and if $L(P, v) = l$, then $L(T, i(v)) = l$. The *non-induced subgraph isomorphism problem* (SIP) is to find such a mapping from a given graph P, called a pattern, to a given graph T referred to as a target. The *induced* SIP additionally requires that if $v \nsim_P w$ then $i(v) \nsim_T i(w)$. In this work, we discuss only the non-induced version of SIP. We say that an instance of SIP is *satisfiable* (SAT) if such a mapping i exists and P is said to be *subgraph-isomorphic* to T. Otherwise, the instance is *unsatisfiable* (UNSAT).

Example 7 Consider graphs P and T in Figure 1.4. The SIP instance between the pattern P (on the left-hand side of the Figure) and the target T (the graph on the middle of the Figure) is SAT- the mapping from P to T maps vertex 1 to vertex 8, vertex 2 to vertex 3, vertex 3 to vertex 4 and vertex 4 to vertex 6.

There are various existing algorithms for the subgraph isomorphism problem [13, 34, 26, 23, 7, 46, 28]. More thorough analysis of the problem and discussion on existing work is presented in Section 2.4.1.

1.1.3 The Filtering-Verification Paradigm

Let S be the Cartesian product of a *query set* Q and a *database* D, where Q contains patterns and D contains targets. The process of solving the Subgraph Isomorphism Problem (SIP) for every pair in S is referred to as the *subgraph query processing* problem. The *filtering-verification paradigm* is a subgraph query processing technique, executed for D and Q, that applies heavy *filtering* procedures before the subgraph isomorphism algorithm execution in order to prune UNSAT instances.

Example 8 Consider Figure 1.4 and assume that graph P (in the left-hand side) is the query set, and graphs T (in the middle) and T1 (in the right-hand side) represent a database. Then, subgraph query processing would solve the SIP for instances (P, T) and (P, T1), where the former instance will be determined as SAT (Example 7) and the latter as UNSAT (there is no valid mapping from P to T1).

Filtering

The first step of the filtering-verification paradigm is *filtering*. During the filtering procedure, pruning algorithms are applied on the graphs in D with respect to Q in order to obtain the *candidate set* C. C is a subset of D and it contains all graphs that were not pruned by the applied filters. The second step, referred to as *verification*, involves executing a SIP algorithm for every instance in the Cartesian product of Q and C.

In order to apply pruning algorithms on D, a specific data structure has to be computed for both D and Q. This structure is known as *index* and its definition is as follows: an *index* of a set of graphs H, denoted as $\mathcal{I}_H$ is a collection of data, often stored in a file, that contains characteristics of the graphs in H. The characteristics are called *features*. Features can be represented in various ways. For instance, they could be paths, subtrees, cycles, or subgraphs. They are computed for every graph in H. The first algorithm executed during the filtering stage is to compute such index for Q denoted as $\mathcal{I}_Q$ and D, written as $\mathcal{I}_D$[1]. The process of computing the features of the graph to store them in the index is called *feature extraction*.

[1] Unless it has already been computed.

The main reason why indices were introduced is *reusability*. For instance, if $\mathcal{I}_D$ has been computed during a previous execution of subgraph query processing with a different query set Q' and D has not been changed, then one can reuse $\mathcal{I}_D$. This would reduce the running time significantly[2]. In theory, one could also reuse $\mathcal{I}_Q$, but in practice this is rarely exploited. The set of queries Q usually consists of much smaller number of graphs of lower order and smaller size than D. This makes index computation much less expensive. Secondly, subgraph query processing with repeated query set rarely happens.

There are many existing index computation and feature extraction techniques. Most common approaches are discussed later in this work. Example 9 illustrates an example approach.

Example 9 Let D be the database and Q the query set defined in Example 8 with the graphs on Figure 1.4. Let the features be paths of length 2 represented as strings of concatenated vertex labels (yellow(Y), blue(B) or red(R)). The features of T, $T1$ and P are {B-B, B-Y, R-Y, B-R}, {B-Y, R-Y, B-R} and {B-Y, R-Y} respectively. The index of D, $\mathcal{I}_D$, is the union of the features of T and $T1$ and the index of Q, $\mathcal{I}_Q$, consists of the features of P. Note that the reverse of each of the strings is also a valid feature, as the graphs are undirected. Therefore, storing each feature and its reverse does not give additional structural information (in this case, this is equivalent to storing feature duplicates) for the cost of doubled index size. This is the reason why feature extraction algorithms enforce specific feature ordering requirements.

The procedure that follows after computing indices is referred to as *candidates extraction*. A *candidate set* $\mathcal{C}$ of a database D and a query set Q is a subset of D that contains all graphs that were not pruned by the applied filters. The purpose of filtering is to derive as small candidate set as possible in order to limit the of number calls to a subgraph isomorphism algorithm during verification. Note that the candidate set is a subset of D. The worst case scenario is when $\mathcal{C}$ and D are equal, which means that filtering did not manage to prune any target graphs in D.

Example 10 Taking the indices of D and Q computed in Example 9, the candidate set $\mathcal{C}$ consists of both T and T1 as each of them contains all features of P.

Verification

Let S' be the Cartesian product of Q and $\mathcal{C}$, where $\mathcal{C}$. *Verification* is the process of solving the SIP for every pair of graphs in S'. All targets that do not contain the pattern, but were not rejected during the filtering step (i.e. they were included in the candidate set), are called *false-positives*. The better the filtering technique, the lower number of false-positives it admits. The number of false-positives is often used as a measure of effectiveness of the filtering method [22].

The usage of the filter-verification paradigm is motivated by the fact that the decision version of the SIP is NP-complete [12]. Reducing the number of SIP calls by discarding targets that are unsatisfiable (UNSAT) for a given pattern without performing SIP call is believed to give opportunities for significant performance improvement [21, 22]. There are various subgraph query processing algorithms based on this paradigm [21, 24, 6, 47, 17], which are discussed in more detail in Chapter 2. A thorough analysis of one of them is presented in Section 2.3.

1.2 Aims of the project

The subgraph query processing problem has a wide variety of applications in many fields, some of which include bioinformatics [7], chemistry [32], computer vision [15, 36], law enforcement [10], model checking [33] and

[2]Results in Section 2.3.3 show that the bottleneck of the filtering-verification paradigm is the database index construction

pattern recognition [11]. It involves repeatedly examining a large database, searching for graphs that contain particular patterns. For instance, in order to apply the best treatment for cancer, one might have to screen a patient's tumor to search for particular set of bio markers to identify the best course of treatment [41].

The filtering-verification paradigm is a new trend in Big Data research, motivated by the fact that the decision version of the subgraph isomorphism problem is NP-Complete [22, 21], which makes it "too time consuming" to solve [21, 22, 43]. Spending more time to build a database index and then reuse it for filtering with subsequent queries is shown in existing literature to "... vastly improve search time", and the index computation time is "... paid back quickly as repeated searches are performed" [1].

In this work, we study subgraph query processing methods. We first look at the approach to solving the problem employed by already existing work. We implement a framework that supports two filtering and candidates extraction techniques, and carry out their empirical analysis. We then design a novel subgraph query processing approach based on the filtering-verification paradigm that does not employ an index structure, but uses fast filtering before a call to a SIP algorithm. The performance of Light Filters is evaluated in terms of filtering strength, filtering and verification running time, and it is compared with the algorithms that use heavy index-bound filtering. A thorough analysis of the instances in the datasets is carried out, and finally conclusions are made on the nature of the problem and the advantages of each method.

1.3 Report organisation

The remaining of this work is organised as follows. Chapter 2 presents review and analysis of existing approaches to solve the subgraph query processing problem, and the properties of the 4 datasets, that are used for testing and evaluation. Chapter 3 presents the implementation and analysis of the indexing framework. Chapter 4 describes our new approach to solve the subgraph query processing problem, called Light Filters. The evaluation of Light Filters and the analysis of the subgraph isomorphism instances in the datasets are presented in Chapter 5. Chapter 6 provides a summary of this work and suggestions how to extend it in the future.

Chapter 2

Review of existing work

2.1 Datasets

This Section presents the specifications of the four datasets used for performance study of existing work in Section 2.3.3 and for empirical study in Section 5 and Section 3.5. The datasets are obtained from [2]. They are used as benchmarks in the evaluation of many subgraph query processing methods, some of which include [21, 24, 6, 47, 17, 45, 22]. All four datasets consist of undirected labeled graphs. Each dataset has a set of target graphs, also called a database, and multiple sets of query graphs. A description of each dataset is given below and the specifications of each dataset is shown in Table 2.1.

Aids is the standard database of the Antiviral Screen dataset of the National Cancer Institute. The database contains the topological structure of 40,000 chemical compounds, represented as graphs. The order of each graph is between 4 and 245. The graphs in this dataset are small and barely connected (Table 2.1 shows that the average vertex degree is 2.1 and the average graph order is 45). This dataset has the largest number of unique labels, namely 62, and the largest database size.

Pcms contains 200 contact maps that represent relationships among amino acids. The graphs here are with much bigger order and higher density than the graphs in Aids. Table 2.1 shows that the graphs in Pcms have the highest density, which is 0.06, calculated using formula 1.1, defined in Chapter 1.

Pdbs consists of 600 target graphs, which represent proteins. The order of the graphs is 2,939 on average. The graph with the smallest order has 1,683 vertices and the graph with the biggest order has 7,979 vertices.

The fourth dataset is Ppigo. Its database consists of 20 protein interaction networks, where networks belong to species. This is the smallest dataset with the biggest graph order and size, both equal to 4,942 on average (Table 2.1).

Table 2.2 shows the number and percent of satisfiable (SAT) SIP instances for each dataset. For instance, Pdbs consists of large number of SAT problems (77.22%), whereas Aids is the dataset with highest percent of UNSAT problems (91.33%).

2.2 Filtering-Verification paradigm

This Section gives an overview of some already existing algorithms based on the filter-verification paradigm. Section 2.2.1 discusses some of the existing filtering algorithms with respect to feature extraction and choice of

	Aids	Pcms	Pdbs	Ppigo
# graphs	40,000	200	600	20
# disconnected graphs	3,157	200	360	20
# unique labels	62	21	10	46
average graph order	45	377	2,939	4,942
average graph size	46.95	4,340	3,064	4,942
average density	0.05	0.06	0.01	0.01
average vertex degree	2.09	23.01	2.06	10.87
# labels on average	4.4	18.9	6.4	28.5

Table 2.1: Characteristics of the datasets

	Aids		Pcms		Pdbs		Ppigo	
	number	percent	number	percent	number	percent	number	percent
all SIP calls	240,000	100	1,800	100	3,600	100	100	100
SAT SIP calls	20,816	8.67	592	32.8	2,780	77.22	61	61
UNSAT SIP calls	219,184	91.33	1,208	67.2	820	22.78	39	39

Table 2.2: The number of SAT/UNSAT SIP instances for each dataset

features, and Section 2.2.2 outlines the most commonly used subgraph isomorphism algorithms for verification.

2.2.1 Existing Filtering Techniques

There are two main types of feature extraction techniques, known as graph mining and exhaustive feature enumeration.

To explain what is meant by a graph mining technique, we introduce the following concepts. The *support ratio* of a feature f in a database D is equal to the number of graphs in D that contain f divided by the total number of graphs in D. A feature is *frequent* if its support ratio is higher than or equal to a certain algorithm-specific threshold value. *Graph mining* techniques store only features which are considered as frequent in the database. Common graph-mining techniques include gIndex [43], fgIndex [9], closure-tree [20], cpIndex [42], L-Index [45], tree+Δ [47] and TreePi [44].

Exhaustive feature enumeration techniques store in the index all features of every graph in the database, regardless of their support ratio. Some well-known exhaustive feature enumeration techniques include GGSX [6], CT-Index [21], GDIndex [40] and gCode [24].

Choosing a feature extraction method often depends on the dataset. For instance, graph mining techniques are slower in building the database index than exhaustive feature enumeration techniques, since they spend additional time on calculating the support ratio of each feature [43]. This makes them inefficient for datasets that are being frequently changed. When graphs are frequently inserted and/or deleted in the database, the support ratio of the features may change and make the index outdated, and the indexing algorithms have to be re-run again.

When it is highly desirable to obtain smaller index, graph mining techniques are better, because they require

much less storage space than exhaustive feature enumeration algorithms, as only frequent features are stored in the index. Another benefit of having index with small size is that the process of constructing the candidate set is faster, since there are less features to be iterated through.

There are various structures that can be used as features. The most commonly used features are paths, trees and subgraphs, derived from the targets in the database, or all of the aforementioned combined. Examples of features based on paths are GGSX [6] and Grapes [17]. Both of them are based on the exhaustive feature enumeration approach. There is a wide variety of indexing techniques that represent features using subgraphs of the targets. Some of them include fgIndex [9], GDIndex [40], cpIndex [42] and L-Index [45]. Examples of indexing algorithms that use trees as features are closure-tree [20] and TreePi [44]. Two approaches were found in existing literature, that implement a combination of the aforementioned features. These are CT-Index [21] and tree+Δ [47].

The choice of features influences the filtering performance and it is often a trade off in terms of time and filtering strength [22]. This is further investigated in Section 2.3.3, where the results of three empirical studies of CT-Index are discussed.

2.2.2 Verification techniques

Most of the subgraph query processing methods that adopt the filtering-verification paradigm are mainly focused on improving the filtering stage, while reusing the same algorithm for verification, which is commonly VF2 [13] [22, 21, 6, 47].

2.3 CT-Index

CT-Index [21] is an existing subgraph query processing approach that adopts the filter-verification paradigm. This method supports undirected graphs with edge and vertex labels and also wild card patterns. Although not explicitly stated in [21], CT-Index addresses the non-induced subgraph isomorphism problem defined in Section 1.1. This Section presents an extensive theoretical and empirical analysis of the algorithms implemented in CT-Index. The filtering procedures and their complexity are discussed in Section 2.3.1. Section 2.3.2 presents the verification algorithm. Section 2.3.3 presents the results of three independent empirical studies, performed using some of the benchmark datasets, described in Section 2.1.

2.3.1 Filtering

During the filtering step, the features of all graphs in a database D are extracted and saved to a file, i.e. the database index $\mathcal{I}_D$. Features are specific subgraphs used to classify graphs, and are stored as hash-key fingerprints. Features may be paths, subtrees and/or cycles of bounded length. Since vertices may contain labels, these features can be viewed as strings from a specified alphabet (where the alphabet is the labels). In [21] it is stated that the reason for using trees and cycles (as well as paths) is that "trees capture additional structural information" and cycles "represent the distinct characteristic of graphs, often neglected when using only trees as features".

Although the time complexity of computing all features of a graph is not reported, it can be derived as follows. To extract a subtree of a graph G of size m and order n and average vertex degree d, one starts with initially empty tree and repeatedly adds edges to extend the vertices that are in the current tree via the recursive function *ExtendTree*. We write F for the set of every edge e in G, such that one of the vertices of e is connected to is part of the current tree and the other is not. The size of F is at most $n(d - 1)$. *ExtendTree* adds an edge specified

as parameter to the current tree, generates *F* and makes a recursive call for every edge in *F*, until the tree reaches size *m*.

At the start of the tree extraction procedure, when adding the first edge to the empty tree, the vertices on both ends of the edge are also added as part of the tree. Therefore, the size of *F* is *2(d-1)* initially. After every recursive call, one more vertex is added to the tree, which introduces *(d-1)* new edges. That makes a total of *m + 1* vertices that will be added to the tree and *(m + 1)(d - 1)* visited edges. Consequently, the complexity of extracting tree features is $\mathcal{O}(m(m+1)(d-1))$. From this formula one can see that the the size of the graphs in *D* has significant impact on the performance of the algorithm. Also, when the graph density is high, the algorithm will be slower because of the high degree of each vertex.

CT-Index computes a unique representation of each distinct feature, referred to as its *canonical form*, and stores its string encoding in $\mathcal{I}_D$. Thus, the equality of two features can be checked by testing the equality of their canonical forms. The canonical label of a tree feature is computed as follows: (1) find the root node *r* of the tree, (2) impose a unique ordering of the children of each node. Step (1) is computed by repeatedly removing all leaf nodes of a tree until a single node or two adjacent nodes remain. In the first case, *r* is the last node left. In the second case the edge connecting the two remaining nodes is removed to obtain two trees, each with one of the remaining nodes as a root. Step (2) is based on the ordering of edge and node labels. For each node *p* that is a parent of nodes *u* and *v*, deciding whether *u* is before *v* depends first on the labels of the edges (p, u) and (p, v), then on the labels of *u* and *v* and finally on the subtrees of *u* and *v*. A bottom-up approach is used (i.e. start with the nodes in the lowest level and move up towards the root) to compute this.

Although not stated in [21] the complexity of their canonical labeling can be derived as follows. Step (1) is $\mathcal{O}(n)$, where *n* is the number of nodes in the tree, as one needs to visit each node before removing it. The complexity of step (2) is as follows. We write $|p|$ for the number of interior nodes in the tree, which is equal to *n* minus the number of leaf nodes. Step (2) visits a node, then visits its parent, and for every child of the parent node checks whether it should be first or second in the canonical label, using the vertex and edge labels conditions described above. This is repeated for every node in the tree up to the root. Therefore, the complexity of step (2) is $\mathcal{O}(|p|.|c|^2)$, where $|c|$ denotes the number of children of a parent.

In [21] it is claimed that step (2) is not linear time, but it is tolerable because "... the trees occurring as features usually are small and vertex and edge labels are diverse and hence the order can be solved quickly". Therefore, we might assume that CT-Index is designed to support only specific types of data sets and one could expect poor performance for data sets with less label diversity and with big trees as features. More specifically, as the size of the target graphs increases, or the average vertex degree in the target graphs increases, so too does the cost of step (2), and performance suffers (we conduct experiments to test this hypothesis in Section 2.3.3).

Fingerprints

CT-Index uses a storage technique called *hash-key fingerprint* to capture the features in the graph. A *fingerprint* is an array of bits, also called a bitset, that denotes whether a particular feature occurs in the graph or not. A separate fingerprint is computed from the canonical labels for each graph in the database. As there is no predefined set of possible features for each graph, reserving one bit for each feature in the feature set is considered infeasible[1]. A hash function maps extracted features to bit positions. CT-Index is not the first indexing algorithm to employ fingerprints as a storage technique. The chemical information system called Thor and developed by Daylight [1] is an example of an information processing system that uses bit arrays to store the features of the graphs.

Information on the implementation of the hash function is not specified in the paper. Depending on the quality of the hash function, the size of the bitset and the size of the fingerprint, collisions may occur, i.e. different

[1]However, due to the restricted alphabet of labels it may be possible to enumerate all possible features thus avoiding some of the pitfalls of hashing, such as collisions and sensitivity to hash table size.

features may map to the same bitset position, introducing false-positives. The [21] paper briefly discusses some optimization techniques that could be used to minimize the influence of collisions, but it is unclear whether CT-Index employs them. It is stated that "... the loss of information caused by the use of hash-key fingerprints seems to be justifiable by the compact nature and convenient processing of bit arrays as long as the amount of false positives does not increase significantly due to collisions".

Collisions can occur also if the size of the fingerprint is too small for the particular data, i.e. there is bigger number of features than the number of spaces in the array to store them. However, making the fingerprint size too big introduces additional overhead by requiring more memory storage space that is not used. The paper does not specify the hash function used or how to decide on the size of bitsets (feature hash tables).

The main advantage of hashing the features and storing them in arrays is that this makes certain operations much cheaper. For example, checking whether a pattern fingerprint is included in a target fingerprint involves inexpensive bit operations. In particular, one only needs to compute a bitwise AND-operation with the two fingerprints to determine if features in the pattern exist within the target. If this test returns false then the target cannot be a candidate for that pattern. However, if it returns true then the target *may* be a candidate and subgraph isomorphism must be verified.

2.3.2 Verification

The verification step checks all candidates computed in the filtering step via a subgraph isomorphism test. A backtracking algorithm [3], similar to VF2 [13] with additional heuristics, is used. This test is theoretically NP-Complete, and is avoided as far as possible via the filtering process. CT-Index is not alone in using (essentially) the VF2 algorithm. For example it is used in GraphGrepSX [6], gCode [24] and Tree+Δ [47]. Most papers claim that VF2 is "state of the art". However, this is not the case [34, 23, 7, 46, 26]. VF2 has been shown to perform erratically and poorly [26]. Therefore we might summarize CT-Index architecture as using a potentially expensive indexing and filtering stage in order to minimize the computational cost of using an outdated SIP algorithm.

2.3.3 Performance

This Section presents the evaluation of CT-Index, which consists of an analysis of the empirical study of CT-Index by its authors [21], the results published in [22] and the performance results we collected when running the CT-Index source code.

The experiments, described in [21], use two datasets and are performed on the same source code that is used by [22] and us, but with added support for edge labels. The first dataset is Aids, the specifications of which are outlined in Section 2.1. The second dataset is composed of synthetic graphs, generated by the authors.

To overcome the fact that some of the indexing methods used in the evaluation do not support labels on edges, the authors split each edge by an additional vertex that encodes the edge label. The resulting graphs are then used as an input to all indexing methods that do not support edge labels [21]. Consider a database D, where graphs have size m, order n and vertex degree d on average, and both vertices and edges are labeled. Let D' be the database derived from D after performing the procedure of splitting edges. Therefore, the size of a graph in D' is $2m$, the order is $m + n$ and the degree is $d + 1$. From the empirical analysis of CT-Index it follows that the size of the graphs and the number of their edges influences the performance of the indexing technique. Therefore, it is unclear how fair it is to compare results obtained by running the same experiments using two types of datasets: one that is modified with graphs size and order twice as much as the size and order of the graphs in the second one, which is the original.

The authors state that they "... removed 40 graphs with more than 255 edges because these graphs tend to cause problems with the implementation of gIndex" [21]. From the specifications of Aids in Table 2.1, one can see that the graphs that were removed have the biggest size in his dataset. The complexity analysis of the filtering algorithms implemented in CT-Index in Section 2.3.1 shows that the size of the graphs influences their complexity. It is expected that when the size of the graphs in *D* grows, filtering becomes slower. Therefore, the experiments in [21] are incomplete, which leads to dubious performance results.

The experiments conducted by [21] can be summarized as incomplete, and the results- questionable. We now look at a second evaluation attempt, published in [22]. Let us first introduce the CT-Index input parameters and default settings, which are referred to in the remaining of this Section.

CT-Index requires five integers as an input, specified in the following order:

1. Fingerprint size. This indicates the number of bits allowed to store the features of each graph in the index. The specified fingerprint size must be equal to 2^n for some integer n. No information on why this is the case is specified in [21]. We were also unable to understand when looking at the source code.
2. Maximum path length. Indicates the maximum length of a path that is allowed to be extracted. If we specify -1, then no paths are extracted.
3. Maximum subtree length. Same as 2, but for subtrees.
4. Maximum cycle length. Same as 2 and 3, but for cycles.

The default input parameters of CT-Index are <4096, -1, 4, 4 > [21, 22]. No information on why exactly these parameters should be used is given.

The experiments in [22] are conducted on the four datasets, described in Section 2.1, and compare six well established filtering-verification methods, one of which is CT-Index, using the default input parameters of each method. The indexing algorithms are run for each of the four datasets, putting a time limit of 8 hours. The measured performance metrics are filtering time, index size, verification time and false positive ratio (FP ratio), which is a metric that indicates how many of the unsatisfiable (UNSAT) instances are filtered without using a SIP algorithm. To calculate the FP ratio, the authors propose formula 2.1. In the formula, $|A\{p\}|$ denotes the number of satisfiable (SAT) instances for a pattern *p* in a query set *Q* and $|\mathcal{C}\{p\}|$ is the cardinality of the candidate set for *p*.

$$FPRatio = \frac{1}{|Q|} \sum_{p \in Q} \frac{|\mathcal{C}\{p\} \setminus A\{p\}|}{|\mathcal{C}\{p\}|} \tag{2.1}$$

If a perfect indexing technique, that manages to prune all UNSAT instances, is achieved, $|\mathcal{C}\{p\} \setminus A\{p\}|$ equals 0 and the value of FP ratio is 0. If the indexing technique does not filter any graph, the size of the candidate set is equal to the size of the database. In this case, the FP ratio depends on the number of SAT instances in the database and the number of queries in *Q*. For instance, if no subgraph isomorphism exists from any pattern to any target in the dataset (i.e. $|A\{p\}|$ is 0), $\frac{|\mathcal{C}\{p\} \setminus A\{p\}|}{|\mathcal{C}\{p\}|}$ is equal to 1 and the FP ratio is then equal to $\frac{1}{|Q|}$. If all instances in the dataset are SAT, then $|\mathcal{C}\{p\} \setminus A\{p\}|$ is equal to 0, therefore the value of FP ratio becomes 0. If one assumes a constant value of $|A\{p\}|$, the value of the FP ratio grows when the size of the candidate set is increased. It converges to 1 when the size of $\mathcal{C}\{p\}$ becomes closer to the size of the database. Similarly, when the size of $\mathcal{C}\{p\}$ is decreased, the FP ratio converges to 0. Therefore, there is an upper and a lower bound on the value of the FP ratio, which also depends on the value of $|A\{p\}|$.

The previous paragraph shows that the value of the FP ratio computed by formula 2.1 is strongly influenced by the number of SAT instances in the dataset. For instance, the value of FP ratio computed for an indexing algorithm *A* executed for a query set *Q* and a database *D* may be equal to the value of FP ratio computed by *A*

when executed with a query set Q' and D. However, from this it does not follow that A achieved the same filtering performance for both Q and Q'.

The conclusion is that formula 2.1 should not be used for comparing FP ratios obtained after running filtering techniques on different datasets and that a certain value of FP ratio might in some cases mean decent filtering performance and in others - poor, depending on the number of SAT instances for a given query set and a database. This formula can give accurate understanding of the performance of a given filtering algorithm only when different values of FP obtained when executing different filtering techniques, using the same dataset, are compared.

Below are some of the main results, obtained during the evaluation presented in [22], and the conclusions that can be made from these results.

- For datasets Pcms and Ppigo, CT-Index never manages to complete execution of the filtering step and no verification is performed [22]. Pcms and Ppigo are of largest size among the four compared datasets. Ppigo also has largest order and Pcms has highest graph density (Table 2.1). This suggests that the performance of CT-Index does indeed suffer when the dataset is composed of larger graphs of high density, as conjectured earlier, and backs up the claim that the evaluation in [21] is incomplete.
- For the Aids dataset, CT-Index performs best. Filtering takes about 80 seconds and verification: about 0.7 seconds [22]. The FP ratio is one of the highest for both Aids and Pdbs among the six evaluated techniques. This suggests that the algorithm has relatively poor filtering performance. A reason for this could be that the number of collisions when computing the fingerprints (hash tables) is high, and more appropriate default fingerprint size should be chosen.

These results give a basic idea of the performance of CT-Index compared to other techniques considered as "state of the art" in subgraph query processing [22]. However, they do not investigate the performance of CT-Index depending on the type of features stored in the index and the size of the fingerprints (i.e. the hastable size). We use the open source java implementation of CT-Index, written by its authors, to investigate this. Experiments are run on the Aids dataset, [2] using input parameters that are different from the default ones. For our experiments, we choose a fixed size of parameter set, [3] the purpose of which is to identify how the change of one parameter influences the performance of the algorithms implemented in CT-Index. As in [22], we measure the filtering and verification time, the FP ratio using formula 2.1 [4] and calculate the total time, that is the sum of the filtering and the verification time. The size of the fingerprints (2^n for some integer n) is from 1 to 16384. Note that when n is 0, no index can be created (all features of every graph have to be stored in a bitset of size 1) and verification is computed for every pair (P, T), where P is a pattern in Q and T is a target in D. The values of the maximum length of paths, subtrees and cycles are permutations of combinations of -1 and 5. Here, -1 shows the influence on the performance of CT-Index when a feature is switched off, and 5 shows the change of performance when the feature is on (i.e -1 and 5 play the role of binary 0 and 1, each feature is either off or on). For every fingerprint size, we execute the algorithm with all permutations of every combination of assignment of -1 and 5.

Table 2.3 shows a selection of our results. The first 10 rows show the changes in running time and FP-ratio depending on the size of the fingerprints. The values of all other parameters are fixed. As expected, the FP ratio goes down with the increase of the fingerprint size, i.e. the filtering performance becomes better. A significant decrease in the verification running time is observed when the fingerprint size is increased. The two results follow from the fact that the number of collisions decreases with the increased fingerprint size, as we allow for each fingerprint to denote the existence/absence of more features. Consequently, more targets can be rejected during filtering and less number of SIP tests have to be computed during verification.

The last 6 rows of Table 2.3 show the performance of CT-Index depending on the other 3 parameters. The input with the worst verification running time is in row 14. Few targets were rejected during the filtering stage, as

[2] As the purpose of our experiments is to test the change of performance of CT-Index depending on the input parameters (fingerprint

Row #	Input	Filtering T	Verification T	Total T	FP Ratio
1	1 5 5 5	108.9	67.4	176.3	0.91
2	64 5 5 5	110.8	59.6	170.4	0.85
3	128 5 5 5	110.9	31.7	142.6	0.87
4	256 5 5 5	104.5	22.4	126.9	0.81
5	512 5 5 5	104.9	17	121.9	0.69
6	1024 5 5 5	112.1	15.5	127.6	0.64
7	2048 5 5 5	107	14.7	121.7	0.63
8	4096 5 5 5	106.1	13	119.1	0.60
9	8192 5 5 5	107.6	12.3	119.9	0.62
10	16384 5 5 5	108	14	122	0.61
11	4096 5 5 -1	105.4	12.9	118.3	0.62
12	4096 5 -1 5	39.7	32.1	71.8	0.88
13	4096 -1 5 -1	81.8	12.8	94.6	0.62
14	4096 -1 -1 5	6.2	61.3	67.5	0.91
15	4096 -1 5 5	82.4	5.9	88.3	0.62
16	4096 5 -1 -1	37.6	7.9	45.5	0.88

Table 2.3: CT-Index: Running time in seconds for filtering and verification and the FP ratio depending on the specified parameters

indicated by the high FP ratio. The data shows that extracting only cycles as features is not effective in proving SIP instances as UNSAT and it is almost equivalent to not having an index structure at all. A reason for this can be that the graphs in the datasets do not have many cycles. The verification time in row 14 also shows that the SIP algorithm used by CT-Index is indeed very inefficient. We will see later in this work that 61.3 seconds for running a SIP algorithm on the instances of the Aids dataset is an extremely poor result (Chapter 5).

Using only paths as features shows best performance in terms of running time (row 16). Adding subtree extraction (row 11) takes about 60 seconds more and gives better filtering ratio and slightly worse verification running time. Therefore, the algorithm for subtree extraction significantly lowers the filtering run time, as derived during the complexity analysis in 2.3.1. One can reach the same conclusion when comparing rows 15 and 16. The options in row 15 give better performance during verification time, but they result in more than twice slower filtering time.

2.4 Subgraph Isomorphism Problem algorithms

This Section presents an analysis of existing methods to solve the subgraph isomorphism problem (SIP). A common way to model SIP for a pattern P and a target T is to represent each vertex in P as a variable that has a set of possible value assignments, referred to as its *domain*, where the values are vertices in T. It has to be ensured that every vertex in P is matched to one unique vertex in T. During a recursive search procedure, values

size, types of extracted features), that there is no need to experiment on more than one dataset.

[3]This was done due to the large number of possible input values.

[4]Note that the FP ratio obtained by this formula can be used as a measurement for filtering performance, because we use the same database and query set for all experiments.

are repeatedly tried to be assigned to variables, following an algorithm-specific heuristics until valid mapping from P to T is found or it is proven that no valid subgraph isomorphism mapping exists.

There are various algorithms in the literature that employ a variation of this model [34, 38, 32, 23, 35, 26]. We study in more detail one of them, namely [26] in the next section.

2.4.1 Glasgow Subgraph Isomorphism Problem Algorithm

This algorithm, referred to as CP15 discusses, the non-induced version of the subgraph isomorphism problem for finite undirected graphs that do not have multiple edges between pairs of vertices, but may have loops [26]. Although CP15 does not consider labels, it can be easily adjusted to take into an account labeled vertices and/or edges by adding additional constraints. In fact, adding labeling support would make the algorithm faster, as follows from a result proved in [25]. Unlike most recent work in subgraph isomorphism [35, 4], this algorithm replaces strong inference with cheaper techniques and shows that they can be beneficial [26]. The algorithm exploits parallelism for both pre-processing and search and introduces a novel usage of backjumping [30] that does not need maintaining conflict sets [26].

The algorithm uses bitset encodings: graphs are represented as bit arrays and each pattern vertex has a domain which is a bit array. This representation allows for fast execution of operations. For instance, the neighbourhood intersections discussed shortly are bitwise-and operations, the unions of domains used during alldifferent propagation, nogood values discovered during search are computed using bitwise-or operations, and cardinality checks involve computing the Hamming weight (also known as population count) of the set [31], which is a single instruction in modern CPUs [26].

The supplemental graph G[c,l] is a graph that has at least c number of paths of length exactly l between two vertices v and w in G. Supplemental graphs are introduced in CP15 and they are based on the idea that if i is a valid mapping from a pattern P to a target T, then F(i) is also a valid mapping from P to T for certain functions F. Supplemental graph pairs are constructed from the pattern and the target with bound on the path distance 3 [5]. The intuition behind their usage is to put a restriction that vertices of distance x apart must be mapped to vertices that are within distance x. This also implies that adjacent vertices in the pattern must be mapped to adjacent vertices in the target. These restrictions are put on top of the search during the initialization of domains in order to perform initial filtering at the top of the search.

The initial filtering on top of the search also checks for compatibility of neighbourhood degree sequences of the vertices in P and T, based on the fact that a vertex v can only be mapped to a vertex w if nds(P, v) is a subsequence of nds(T, w) [34]. This was introduced by [34] and is used in the Light filters approach discussed later in this work in Chapter 4.

The algorithm employs a recursive search procedure that repeatedly picks a variable with the smallest domain to branch on, breaking ties on descending static degree in the original pattern graph. The values of the domain of each variable are ordered by descending static degree in the target graph. If the assigned value to a variable is in conflict with the current partial solution or does not obey certain constraints, then the search returns a nogood set of variables [26]. In such case, the algorithm can perform a variation of conflict directed backjumping [30] without explicitly maintaining conflict sets. On success of value assignment of a variable, the algorithm updates the partical solution and makes recursive call until every variable is assigned (success) or no solution is proved to exist.

A variable assignment algorithm is called every time a variable v is to be given a value e from its domain D_v. The algorithm assigns e to v and then infers which values may be eliminated from the remaining domains.

[5] The bound is chosen after an observation that distances greater than 3 rarely give additional filtering power and their construction is computationally very expensive [26]

It removes e from the domains of all other variables and eliminates any detected Hall sets [6] from future variables [7], thus detecting that an assignment is impossible even if values remain in each variable's domain.

The SIP algorithm achieves SIMD-like parallelism from the bitset encodings [26]. The algorithm also allows for parallel supplemental graphs construction, neighbourhood degree sequence calculation, graph construction and domain initialization. In addition, the subtrees created by recursive calls made during search can be explored in parallel by multiple threads using early diversity work-stealing approach, where a single thread always preserves the sequential search order, finding states of the search space that are to be explored. The rest of the threads take work from the main thread, trying to steal early in the search tree, because value-ordering heuristics are expected to be weakest early in the search [19].

Both the sequential and the threaded versions of the algorithm were compared with an implementation of SND [4], LAD[35] and VF2[13]. The properties of the hardware, implementation-specific details, the nature of the datasets used in the experiments and the results from the evaluation are presented in the paper[26]. Below are highlights of the results obtained after the experiments:

- The algorithm is the single best among the evaluated approaches for non-trivial instances. VF2 [13] is stronger on trivial instances. The reason for this is that CP15 needs more time to instantiate domains, generate supplemental graphs and perform the inference on top of the search than VF2. These results show that there is no single algorithm that performs best on all instances and suggests for taking more flexible future approach that allows for instance-specific configuration. For example, for trivial problems one may use simpler version of CP15 by adjusting it not to create supplemental graphs.
- Except at very low sequential runtimes, parallelism results in general speed improvement. For exceptionally hard satisfiable instances, parallelism results to superlinear speedup. This shows that the early work stealing approach is able to recover from early mistakes of value ordering heuristic choices by avoiding strong commitment to such choices.
- Backjumping always either pays for itself or gives slight improvement. For some instances that constitute of highly structured data, it makes an improvement of several orders of magnitude.

CP15 might be summarized as a novel method that is specialized in solving hard instances of the SIP problem, but it could be adjusted to be fast for easy instances as well. From the experiment results reported in [26], one can see that it is the best solver for non-trivial SIP instances. This suggests that current subgraph query processing methods that count entirely on verification could be faster than current methods based on the filtering-verification paradigm even if one compares only the verification stages without counting the cost of filtering. Consequently, it can be concluded that the research focused on the filtering-verification algorithms is too committed on improving the quality of filtering, while neglecting verification by using an outdated SIP algorithm.

[6] A set of n whose domains include only n values between them. Finding a Hall set allows for removing the values part of the hall set from the domains of variables that are not part of the Hall set.

[7] The algorithm may fail to identify some Hall sets if the initial ordering of domains is imperfect. However, it gives substantial pay off in terms of running time, as validated experimentally in the paper

Chapter 3

Framework for graph indexing and filtering

This Chapter describes a new subgraph-query filtering and indexing framework, implemented in Java. The framework implements two alternative feature enumeration and candidates extraction techniques: Path Index (PI) and Path-Subtree Index (PSI). In this Chapter, we present the design and the implementation and the theoretical analysis of every component of the framework. Section 3.1 describes the choice of graph representation. Section 3.2 explains the paths extraction algorithm. Section 3.3 describes PI and PSI and the specific feature representations and candidates extraction methods employed by each of them. The last Section discusses the performance of the framework and gives suggestions for improvement.

3.1 Graph representation

A graph G is represented by a Java class *Graph* that has an integer *id* and a collection of objects of type *Vertex* as fields. A Vertex v has an *id*, a *label* and a list of *edges* that connect it to its neighbours. The number of the edges of v is equal to its degree. An Edge object has a *label* and a *destination vertex* of type Vertex as fields. Each edge that connects two vertices v and w in G is represented by two objects of type Edge. The first object has v as destination vertex and is added in the list of edges of w, which is the source Vertex. The second object has w as destination vertex and is added in the list of edges of v, which is the source Vertex. The label of both objects stays the same.

3.2 Paths Extraction

The features in the framework are calculated using paths with bound on their length k. *Path extraction* is the process of generating all paths up to k in a target graph T. A stack data structure stores partial paths. For every vertex v in T, we execute a recursive depth-first search algorithm with a bound on the size of the stack, equal to k. We output the sequence of vertices currently stored in the stack after each recursive call. Every generated sequence is then fed into the 3 procedure to derive its feature representation and store it in the index.

Algorithm 1 describes the path extraction approach. Before generatePath procedure is called, an empty stack s is initialized (line 3). s stores the vertices generated during the depth-first search procedure. Procedure generatePath (Algorithm 1) takes all vertices in the target graph T and k as parameters. It calls procedure dfsBounded for every vertex in T as the starting vertex (line 5), pushed on s (line 4).

Algorithm 2 performs a depth-first search with bound on the maximum path length equal to k. Given a

starting vertex *v*, *k* and stack *s*, Algorithm 2 iterates through all neighbours of *v*, while adding each neighbour *n* to *s* (line 10) and calling dfsBounded with *n* as a starting vertex (line 11). At each call of dfsBounded, the current sequence of vertices in *s* is passed to procedure computeFeature (Algorithm 3) until all vertices reachable by the start vertex within distance *k* are visited.

Algorithm 1 Paths extraction

```
1:  procedure GENERATEPATH (vertices, k)
2:      for v ∈ vertices do
3:          s ← initialize                          ▷ initialize a stack s
4:          s ← s + v                               ▷ push v on top of s
5:          DFSBOUNDED(v, k, s)
6:      end for
7:  end procedure
```

Algorithm 2 Depth First Search of bound length

```
1:  procedure DFSBOUNDED (v, k, s)
2:      if s ≤ k then
3:          newPath ← s                             ▷ new path of size up to l_max is found
4:          GETPATH(newPath)
5:      end if
6:      if s = k then
7:          s ← s - 1                               ▷ remove the top element from s
8:      end if
9:      for neighbour n of v do
10:         if n ∉ s then
11:             s ← s + n
12:             DFSBOUNDED(n, k, s)                 ▷ recursive call. Starting vertex is now n
13:         end if
14:     end for
15:     if s is full then                           ▷ all neighbours of node are on the stack
16:         s ← s - 1
17:     end if
18: end procedure
```

We derive a big O estimate on the number of paths visited by the extraction algorithm. Assume that the path extraction algorithm is executed on a clique of size *n*. This is the worst case, because any other graph of size n will have fewer paths. Let the maximum path length be *k*. Let a path length be $l \leq k$. Then, the algorithm extracts

$$P = \sum_{l=1}^{l=k} \binom{n}{l} l!$$

paths. Since the first term in the sum dominates remaining $(k-1)$ terms in the sum we can write

$$P \leq \binom{n}{k} k! k = \mathcal{O}(n^k),$$

since k is a constant. Therefore $\mathcal{O}(n^k)$ is the worst-case complexity. As one can see, the complexity grows exponentially with *k*. In practice, the complexity will depend on the degree of each vertex.

3.3 Indexing and candidates extraction algorithms

The rest of the subgraph query filtering process can be carried out in two alternative ways by the framework, depending on whether Path Index (PI) or Path-Subtree Index (PSI) is executed. PSI and PI employ different feature representation and candidates extraction algorithms, which are explained below.

3.3.1 Path Index

PI is based on commonly used filtering techniques [6]. Its implementation is described below.

Features

PI derives the features from the paths, extracted during the path extraction procedure, outlined in Section 3.2. To store a path in the index of a database *D*, denoted as $\mathcal{I}_D$, we compute its unique string representation, referred to as *p-feature*, which is done as follows:

1. Given a path *p*, replace each vertex in *p* with its label to obtain its *p-feature*.
2. Reverse the sequence *p* to obtain p'.
3. Calculate *p-feature*$'$, that is the unique string representation of p'.
4. If not added previously, store in $\mathcal{I}_D$ the lexicographically smaller of the two string representations (*p-feature*$'$ or *p-feature*).

The complexity of each of the steps outlined above is the following. Step 1 involves iterating through the sequence of vertices, that is at most *k* elements. Accessing the label of each vertex is a constant time operation, and the complexity of Step 1 is $\mathcal{O}(k)$. The complexity of Step 2 and Step 3 is $\mathcal{O}(k)$. Step 4 has $\mathcal{O}(\mathcal{I}_T)$ worst case complexity, where $\mathcal{I}_T$ is the current size of the index of a graph *T*. Note that as the index grows larger, the cost of adding new p-features increases.

Example 1 Figure 3.1 represents a graph *G* with 4 vertices that have label red (R), yellow (Y), or blue (B) and Figure 3.2 shows the paths of *G* of length less or equal to 1. Their p-features are strings composed of their labels. The index of *G* will consists of the p-features {B, R, Y, B-Y, R-Y}.

The path extraction algorithm will encounter both p' and *p* throughout its execution, but only one of them will be stored $\mathcal{I}_D$. Storing both of them would be redundant, because the graphs we work with are undirected. Moreover, storing both p' and *p* would double the size of $\mathcal{I}_D$.

Theorem 3.3.1. *Let* G *be an undirected graph with two paths* p$'$ *and* p*, such that* p$'$ *is the reverse of* p*. Then, both* p$'$ *and* p *can be represented by one p-feature without loosing structural information about* G*.*

Proof. The correctness of this statement follows from the fact that *G* is undirected. □

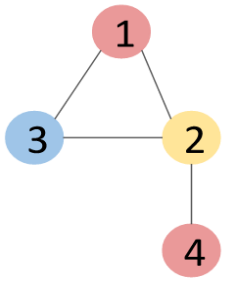


Figure 3.1: Graph G

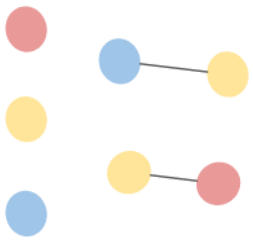


Figure 3.2: Paths of G of length less or equal to 1

Implementation

Algorithm 3 describes the implementation of Steps 1-4 described above. Given a sequence of vertices p as parameter, the procedure computes the *p-feature* (line 2) and *p-feature*$'$ (lines 3, 4) and stores the lexicographically smaller variant in the index.

Algorithm 3 Compute p-features procedure

```
1:  procedure COMPUTEFEATURE (p)
2:      pfeature ← p.toString()          ▷ returns a string of the labels of the nodes in vseq
3:      revp ← p.reverse()               ▷ reverse the order of nodes in p
4:      pfeature' ← revp.toString()      ▷ returns a string of the labels of the nodes in revp
5:      if pfeature' < pfeature then
6:          pfeature ← pfeature'         ▷ put to index the lexicographically smaller string
7:      end if
8:      if pfeature ∉ I_D then
9:          I_D ← I_D + pfeature
10:     end if
11: end procedure
```

Candidates Extraction

Algorithm 4 shows the candidates extraction procedure. It returns a list of the ids of all graphs in the database that contain all p-features of the pattern graph p. Algorithm 4 iterates once through all graphs in the D (line 3) and for each target, checks whether its index contains all p-features of the pattern P (lines 5, 6, 7). Let the size of the index of P be $|\mathcal{I}_P|$ on average and the average size of a target graph index be $|\mathcal{I}_T|$. Then the time complexity is equal to $\mathcal{O}(|D|.|\mathcal{I}_P|.|\mathcal{I}_T|)$.

3.3.2 Path-Subtree Index

Path-Subtree Index (PSI) is a novel indexing technique based on the notion of vertex neighbourhood label, somewhat similar to the labeling approach used for solving the subgraph isomorphism problem, employed by [34]. This Section presents the algorithm for computing the p-features of paths and the candidates extraction approach employed in PSI. We prove that PSI has greater filtering power than PI.

Algorithm 4 Candidates Extraction Procedure

```
1:  procedure CANDIDATESEXTRACTOR (I_P, I_D)
2:      candidates ← new ArrayList<>()
3:      for T ∈ D do
4:          flag ← true
5:          for pfeature ∈ I_P do
6:              if ¬CONTAINS ( I_T, pfeature) then        ▷ check whether pfeature is contained in I_T
7:                  flag ← false; break
8:              end if
9:          end for
10:         if flag then candidates ← candidates + T.id
11:         end if
12:     end for
13:     return candidates
14: end procedure
```

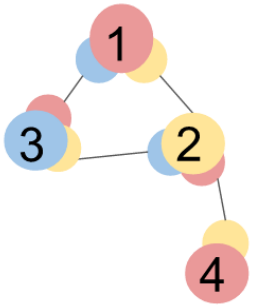


Figure 3.3: Graph G with n-labels

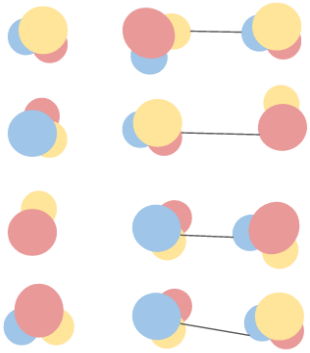


Figure 3.4: Paths of length less or equal to 1 with n-labels

Features

The main difference between PSI and PI is in the algorithms to compute the unique string representation of paths. PSI uses an alternative labeling procedure, based on the *neighbourhood label* (n-label) of a vertex. Below we discuss the notion of vertex neighbourhood label and the procedure of computing p-features.

Vertex Neighbourhood Label

The term *neighborhood label (n-label)* refers to a specific label that is computed for each vertex in a graph G. The n-label of a vertex v with label l is a string composed of the labels of its neighbours and l, ordered in lexicographically increasing order, starting always with l.

Example 2 Figure 3.3 shows the n-labels of the vertices in a graph G in Figure 3.1. The n-label of each vertex starts with its own label, which is denoted in the picture as the biggest circle. For instance, the n-label of vertex 1 is composed of red (R), blue (B) and yellow (Y), starting with its own label, which is R, and then adding the labels of its neighbours (vertex 2 and 3). Note that although the n-labels of vertices 1, 2 and 3 contain the same colors, they are not the same. We write the n-label of vertex 1, 2 and 3 as "RBY", "YBR" and "BRY" respectively.

1. Given a sequence of vertices *p*, replace each vertex in *p* with its *neighbourhood label (n-label)* to obtain the p-feature of *p*.
2. Reverse *p* to obtain p'.
3. Calculate *p-feature*$'$, that is the unique string representation of p'.
4. If not added previously, store in $\mathcal{I}_D$ the lexicographically smaller of the two string representations (i.e. *p-feature*$'$ or *p-feature*).

Example 3 Figure 3.4 shows all paths of length less or equal to 1, composed from *G* with n-labels on its vertices on Figure 3.3. The p-features stored in the index of *G* consist of the lexicographically smaller variant of their string representation. For instance, the p-feature of the top-most right-most path is the string "RBY-YBR".

Implementation

The toString method used in Algorithm 3 is modified to take a bit as an input, which constructs a p-feature using the n-labels of the vertices of the bit is set or returns a p-feature composed of the labels of the vertices otherwise.

The features representation and storage algorithm of PSI is almost the same as the one for PI with (Algorithm 3). The only difference is the modification of the toString() method (line 2,4) to output the n-labels or the labels of the vertices in the sequence depending on the value of a bit given as an input parameter. The p-features are constructed using labels of vertices if the bit is set to false or using the n-labels otherwise. The n-labels of the vertices in each graph are computed prior to the invocation of Algorithm 3, as explained below. Thus there is no additional time complexity to the features representation and storage algorithm used in PI.

Computing the n-label of a vertex is done after all the graphs from the input files are read and initialized. Additional field called n-label of type String and methods to compute it are added to the Vertex class. Algorithm 5 shows our approach. It is executed for every vertex in every target. For every vertex, we compute a sequence of labels of its neighbours ordered lexicographically. The complexity of insertion sort is $\mathcal{O}(n^2)$ [3] and therefore the overall complexity of running Algorithm 5 is $\mathcal{O}(d.n^2)$, where *d* denotes the average degree of a vertex and *n* is the size of the n-label, that is *d+1*.

Algorithm 5 Set n-label procedure

```
1: procedure SETNLABEL
2:     nlabel ← initialize
3:     for Edge e ∈ edges do
4:         u ← e.dstVertex
5:         INSERTIONSORT (nlabel, u.label) ▷ Insert u's' label in n-label in lexicographically increasing order
6:     end for
7: end procedure
8: nlabel ← label + nlabel                    ▷ append the label of the vertex to its n-label
```

Candidates extraction

This Section describes how the set of candidate graphs for a SIP test with the pattern is formed, using the database index $\mathcal{I}_D$ and the set of features of the pattern $\mathcal{I}_P$.

We say that a p-feature *A*, composed by the n-labels of the vertices of a path *a*, *includes* a p-feature *B*, composed by the n-labels of the vertices of a path *b*, if the following conditions are met:

1. Every vertex *v* in position *i* in *b*, has label equal to the label of a vertex *w* in position *i* in a.

2. The n-label of every vertex *v* in position *i* in *b* is a substring of the n-label in position *i* in *a*.

The method to extract the candidate set is the same as the one shown in Algorithm 4. The only change is in the implementation of the contains procedure (Algorithm 4 line 6). Instead of checking whether there exists a feature in the target index $\mathcal{I}_T$ that is identical to a given pattern feature *pf*, we check whether there exists a p-feature in $\mathcal{I}_T$ that includes *pf*.

The first condition is equivalent to the filtering procedure employed by PI: we check for compatibility using only the vertex labels. The purpose of the second condition is to verify that the neighbourhood of each pattern vertex can be matched to a neighbourhood of a target vertex.

Algorithm 6 illustrates the implementation of the includes procedure. For every path in the target index, it calls Algorithm 7 to check that the two conditions for the target p-feature to include the pattern p-feature are met (Algorithm 6 line 3). If condition 1 (Algorithm 7 line 2) is met, then procedure containsLabel checks whether condition 2 is satisfied (Algorithm 7 line 8). Procedure isSubstring takes two nlabels as arguments and returns true if the second nlabel is a substring of the first nlabel or false otherwise.

Algorithm 6 Includes procedure

```
1: procedure INCLUDES (Tindex, patPath)      ▷ Tindex is the target index, patPath is path in the pattern index
2:     for tarPath ∈ Tindex do
3:         if INCLUDESFEATURE(tarPath, patPath) then
4:             return true
5:         end if
6:     end for
7:     return false
8: end procedure
```

Algorithm 7 includesFeature procedure

```
1: procedure INCLUDESFEATURE(tarPath, patPath)
2:     if tarPath.length < patPath.length then return false      ▷ if tarPath is shorter than patPath, then it can't
   contain it
3:     end if
4:     if label of each vertex in tarPath not equal label of each vertex in patPath then   ▷ equivalent to PI filter
5:         return false
6:     end if
7:     for i in range(0, tarPath.length) do ▷ for i^th nlabel in tarf, check that it includes the i^th nlabel in patPath
8:         if ¬ISSUBSTRING(tarPath[i], patPath[i]) then
9:             return false
10:        end if
11:    end for
12:    return true
13: end procedure
```

Algorithm 6 involves visiting each path in the index of a single target, calling Algorithm 7 (line 3), which then visits at most all characters that form a given nlabel. Therefore, the overall complexity of Algorithm 6 is linear with the size of the input.

Theorem 3.3.2. *Let $\mathcal{C}_{PI}$ be the candidate set retrieved by PI and let $\mathcal{C}_{PSI}$ be the candidate set obtained after running the framework using PSI. Then $\mathcal{C}_{PSI} \subseteq \mathcal{C}_{PI}$.*

Proof. The necessary conditions for a p-feature A to include a p-feature B are the proof of the Theorem. The first condition puts an upper bound on the size of $\mathcal{C}_{PSI}$ to be at most $|\mathcal{C}_{PI}|$. The second condition gives PSI additional filtering strength by requiring an existence of matching of the neighbourhood of each pattern vertex to a neighbourhood of a vertex in the target. Therefore $\mathcal{C}_{PSI} \subseteq \mathcal{C}_{PI}$. □

3.4 Running the framework

To run the framework, the user specifies the names of the files containing the database and the query set, k (the bound on the path length allowed to be extracted) and a bit denoting which indexing and candidates extraction technique the user wants to run. If the bit is 1, the framework will use the PSI-specific algorithms, otherwise it will run the algorithms for PI. The source code can be downloaded from Github [5].

3.5 Performance analysis and suggestions for improvement

This Section discusses the performance of the framework with PI and PSI. Both techniques were ran with the Aids dataset, described in Section 2.1. We outline the strengths and weaknesses of PI and PSI and give suggestions for improvement.

Let us denote the index and the candidate set obtained with PI as $\mathcal{I}_{PI}$ and $\mathcal{C}_{PI}$ respectively, and the index and the candidate set obtained with PSI as $\mathcal{I}_{PSI}$ and $\mathcal{C}_{PSI}$.

PSI requires significantly more storage space than PI, due to the fact that it uses n-labels to encode features. The length of the n-label of a vertex is equal to its degree, incremented by 1. In particular, if we assume that the average vertex degree in the target database is d, the p-feature of each path computed using PSI will be d times bigger than the p-feature of the same path computed using PI due to the size of the nlabel of each vertex. Therefore, the size of $\mathcal{I}_{PSI}$ is d times bigger than the size of $\mathcal{I}_{PI}$.

Consider Table 3.3 that shows the performance of the framework when run with PI (Table 3.1) and PSI (Table 3.2). The data on the two tables is obtained after running the database D with query #0 from the query set Q, where D and Q are from the Aids dataset. The size of D is equal to 40,000 and the number of SAT SIP instances between pattern #0 and D is 8,042 [2]. The first column of each table shows the maximum bound on the path length, k, the second column denotes the number of candidates obtained for each k, and the third column shows the framework running time. The fourth column shows the FP ratio for each k, where the FP ratio is calculated using formula 2.1, discussed in Section 2.3.3.

Note that the maximum value of k is equal to 5. This is due to the fact that when k is bigger than 5, the framework for PSI does not finish execution for hours. This follows from the complexity analysis of the algorithms that are implemented in the framework, which was carried out earlier.

Theorem 3.3.2 states that the filtering performance of PSI is not worse than the filtering performance of PI. Our experiments show that for the Aids dataset, the size of $\mathcal{C}_{PSI}$ is rarely close to the size of $\mathcal{C}_{PI}$. Comparing the second column of Table 3.1 with the second column of Table 3.2, one can see that $\mathcal{C}_{PI}$ is more than twice bigger than $\mathcal{C}_{PSI}$ for each k. The increase of k increases the difference between the size of $\mathcal{C}_{PI}$ and $\mathcal{C}_{PSI}$, and when k equals 5, PSI prunes 31,760 targets and $\mathcal{C}_{PSI}$ contains only 198 false-positives. This shows that n-labels manage to capture substantially more information about the structure of the graphs and lead to several times better filtering performance.

It is interesting to observe that the difference of the size of $\mathcal{C}_{PSI}$ is very small for k equal to 4 and 5 (Table 3.2). However, the running time of PSI for k = 5 is 3 times bigger than the running time of PSI for k = 4. This

k	# candidates	running time	FP ratio
2	39,368	16,509	0.79
3	33,995	25,653	0.76
4	31,831	58,559	0.74
5	31,106	149,091	0.74

Table 3.1: PI

k	# candidates	running time	FP ratio
2	17,863	25,662	0.54
3	9,363	71,099	0.14
4	8,336	228,992	0.03
5	8,240	731,450	0.02

Table 3.2: PSI

Table 3.3: Performance of PI and PSI depending on the maximum path length bound (k). The number of SAT SIP instances of D and Q is 8,042

suggests that the filtering power does not increase linearly with the running time with increasing k and there exists a maximum value of $k = \lambda$, when the trade off between filtering power and running time is good. Then, for $k > \lambda$, there is almost no filtering gain, but only significantly increased computation and storage overhead. Similar observation can be made for PI (Table 3.1).

PSI is several times slower than PI. Comparing the third column of Table 3.1 with the third column of Table 3.2, one can see that for each k, the running time of PI is much better than the running time of PSI. Their difference increases with increasing k. There are several reasons for these results. One of them is that the size of $\mathcal{I}_{PI}$ is several times smaller than the size of $\mathcal{I}_{PSI}$. The size of the index plays major role in the complexity of the candidates extraction procedure for both PI and PSI, as shown previously. A second reason is that computing the n-labels of the vertices of each graph requires additional running time.

The performance results show that the framework has slow running time that makes it hard to be used in practice. However, they also show that PSI has very strong filtering capability. Below, we suggest possible techniques for running time improvement of the framework.

The index of the database is represented as the union of the indices of all targets. Naively, it does not take into an account the fact that a feature can be present in more than one graph. When working with datasets where the graphs are similarly structured like Aids, removing repetitive features results in significant decrease of the index size. The following strategies can be employed to decrease the size of the index without lowering its filtering capability.

We can represent the index using a tree data structure similar to suffix tree [39] that stores all extracted features from the database as strings and number of leaf nodes of the tree denotes the number of features. The representation of strings in the tree is the same as the representation employed by suffix trees except from the construction of leaf nodes and the feature suffixes insertion. Each leaf node is a list of the ids of all graphs that contain the corresponding feature. We insert the full feature without inserting its suffixes. This is because each label/nlabel part of a feature is a feature on its own and it will be extracted from the path extraction algorithm. Therefore, there is no need to insert unique termination character at the end of a feature, as it is done with suffix trees. The tree can be built incrementally during features extraction in $\mathcal{O}(n.log(n))$ time on average, where n is the length of the string that results when appending all features in the database, and worst-case time complexity $\mathcal{O}(n^2)$. More efficient suffix tree construction algorithms exist [39, 27, 37] that could be adjusted to work for the tree. Searching for a feature F of length m in the suffix tree requires following a path from the root matching characters until reaching the leaf node and can be done in $\prime(m)$ time.

Chapter 4

Light Filters

This section describes the study of a simple subgraph isomorphism problem(SIP) algorithm, called SIP1, that implements a fast filter that does not employ an index structure.

Light Filters is an algorithm for subgraph query processing that is based on a modified version of the filter-verification paradigm. This approach uses simple filtering tests that require much less computational effort. Given a database D and a query set Q, subgraph query processing for D and Q takes every instance, executes filtering procedures, and if the instance is not proved as (unsatisfiable) UNSAT, a subgraph isomorphism problem (SIP) test is called to solve it. Unlike classical filtering-verification model, no candidate set is computed, because if a target T is candidate for a pattern P, SIP is carried out immediately. Additionally, here more importance is placed on the quality of the SIP algorithm.

Algorithm 8 gives an outline of our approach. We first read in each graph in D and Q and initialize graph objects (lines 2, 3). Filtering is performed for every (P, T) pair, and if the instance is not pruned, a call to a SIP algorithm is made (line 7). The filtering step consists of 5 simple tests, performed before the call to SIP1. If the conditions of any of the tests are not met, search does not proceed, we call this a *trivial fail* and carry on with the next instance.

The remaining of this Chapter gives an explanation of each step of the Light Filters algorithm. First, we introduce the theory behind the trivial failures and their implementation in Section 4.1. We then introduce the subgraphs isomorphism problem algorithm called SIP1 and discuss its implementation in Section 4.2. We give an empirical analysis of each of the algorithms implemented in our subgraph query processing approach. Evaluation of Light Filters and discussion of our experimental results is described in the next Chapter.

Algorithm 8 Light filters algorithm

```
1:  procedure COMPUTE (Q, D)
2:      targets ← read in all targets from D, initialize objects
3:      patterns ← read in all patterns from Q, initialize objects
4:      for P ∈ patterns do
5:          for T ∈ targets do
6:              if !FILTER (P, T) then        ▷ If the instance is not rejected during filtering, perform verification
7:                  SIP1(P, T)
8:              end if
9:          end for
10:     end for
11: end procedure
```

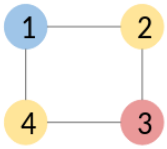


Figure 4.1: graph P

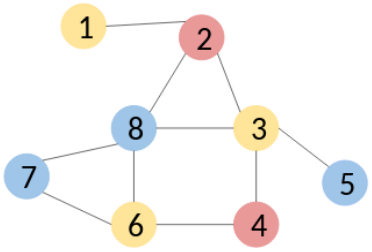


Figure 4.2: graph T

Figure 4.3: Instances of subgraph isomorphism problem (SIP)

4.1 Trivial Failures

This Section introduces the five trivial failure tests, implemented as part of the filtering stage of SIP1. The tests are based on the notion of neighbourhood degree sequence, a concept also used in [34]. First, we introduce definitions and notations that complement the theory section in Chapter 1. We then prove the correctness of our approach and outline its implementation and give an empirical analysis of each of the tests.

The *label neighbourhood degree sequence* (LNDS) of label l in L(G) for a graph G, denoted as LNDS (G, l) is the sequence of the degrees of all vertices in G that have l assigned as their label, taken in non-increasing order.

We say that A *subsumes* B, $A \succeq B$, if there exists a subsequence A' of A with length equal to the length of B, such that B_i is less than or equal to A'_i for all indices i for A', B.

Example 1 Let A be the sequence of integers $\{4, 3, 2\}$ and let B be the sequence of integers $\{2, 2\}$. Then, $A \succeq B$ and an example of sequence A' from the definition is $\{4, 3\}$.

By abuse of notation, we say that a graph T *subsumes* a graph P, $T \succeq P$, if for every label l in P, LNDS (T, l) $\succeq$ LNDS (P, l).

Example 2 Let us look at graphs T and P on Figure 4.3. Both of them consist of three labels: yellow (Y), red (R) and blue (B). Let us compute LNDS of every label in T and P. LNDS (T, Y) = $\{4, 3, 1\}$, LNDS (T, R) = $\{3, 2\}$ and LNDS (T, B) = $\{4, 2, 1\}$. Similarly, LNDS (P, Y) = $\{2, 2\}$, LNDS (P, R) = $\{2\}$ and LNDS (P, B) = $\{2\}$. Clearly, T subsumes P, as for every label l in P, LNDS (T, l) $\succeq$ LNDS (P, l).

Let $\mathcal{L}$ be the labeling function in G and let us define a label counting function for G, $\lambda_G : \mathcal{L} \to \mathbb{N}$ as follows: for any $l \in \mathcal{L}$ if there is no vertex in G with label l, $\lambda_G(l) = 0$. Otherwise $\lambda_G(l)$ is equal to $|L^{-1}(l)|$, that is the number of vertices in G which are assigned l as their label.

Example 3 Let us consider graph P in Figure 4.1 that is assigned three labels: red (R), yellow (Y) and blue (B). The counting function of each of them is the following: λ_P(R) = 1, λ_P(Y) = 2 and λ_P(B) = 1.

We can now introduce several simple filtering tests for incompatibility between graphs P and T. If either of the tests is not true, P is not subgraph isomorphic to T and we say that we have encountered a *trivial fail*. If an instance causes a trivial fail, we do not have to test for subgraph isomorphism, because in such case this instance is UNSAT. The filtering tests are shown on Figure 4.1. Their execution follows a strict hierarchy based on the expected cost of their execution, starting with the test which is shown to require least computational effort (test 1). Analysis of the complexity of each test is given later in this Section. Every test is executed only if all previous

tests succeeded. Below follows a proof of the correctness of first four filtering tests. The fifth test is discussed shortly.

Trivial Fail	Meaning
1	$\|V(T)\| \geq \|V(P)\|$
2	$\|L(V(T))\| \geq \|L(V(P))\|$
3	$\forall\, l \in \mathcal{L}$: $\lambda_P(l) \leq \lambda_T(l)$
4	$T \succeq P$
5	$\forall\, v \in$ V(P), domv $\neg\, \varnothing$

Table 4.1: Failure tests hierarchy

Theorem 4.1.1. *Let* T *and* P *be graphs. Let* P *be subgraph isomorphic to* T. *Then we have:*

1. $|V(T)| \leq |V(P)|$
2. $|L(V(T))| \leq |L(V(P))|$
3. $\forall$ l $\in \mathcal{L}$: $\lambda_P(l) \leq \lambda_T(l)$
4. $T \succeq P$

Proof. By contradiction.

Suppose that 1 is false. Then, P must have at least one vertex more than T and no injective function from P to T exists. From definition of subgraph isomorphism, P is not subgraph isomorphic to T, which is a contradiction.

Suppose that 2 is false so that there exists a label l in L(P) that does not belong to L(T). Therefore, there exists a vertex v in P with label l that cannot be matched by any label preserving function. Therefore, from the definition of subgraph isomorphism, P is not isomorphic to T, which is a contradiction.

Suppose that 3 is false and there exists a label l both in L(P) and L(T) that is assigned to m number of vertices in V(T) and to at least $m + 1$ vertices in V(P). There two possibilities: either at least one vertex in P is unmatched by an injective mapping from P to T, or at least two vertices in P are matched to the same vertex in T. In either case the mapping is not subgraph isomorphism, which is a contradiction.

Suppose that T does not subsume P. Therefore, there exists label l both in P and T such that LNDS (T, l) $\not\preceq$ LNDS (P, l). From the proof of 1, 2, 3, it follows that LNDS (T, l) $\not\preceq$ LNDS (P, l), because there exists at least one vertex v whose degree is in LNDS (P, l) with higher degree than the degree of the corresponding vertex in LNDS (T, l). A matching, of v a vertex in V(T) that preserves adjacency does not exists and P is not subgraph isomorphic to T, which is a contradiction. □

The fifth trivial failure test is based on the choice of model for our subgraph isomorphism algorithm (SIP1). Given a pattern P and a target T, we represent each vertex v in V(P) as a variable that can accept one of the vertices in V(T) as a value. The set of all possible values of v is called the *domain* of v, *domv*. It is represented as a bit array (bitset) of size equal to the order of T, where each entry maps to a target vertex. domv[i] is equal to 1 if v can be mapped to the i^{th} vertex in V(T), or 0 otherwise. If every bit in domv is 0, this means that no valid mapping from v to any vertex in V(T) exists. In such case v has a *domain wipeout*. During search, domain

wipeout of *v* indicates that the current partial mapping can not be a subgraph isomorphism from *P* to *T*. During initialization, domain wipeout of *v* indicates that *P* is not subgraph isomorphic to *T*. Test 5 checks whether domv experiences a domain wipeout. If there exists an empty domain, the test fails and the instance does not proceed to search.

4.1.1 Implementation and complexity analysis

Algorithm 9 describes the implementation of the 5 trivial failures from Table 4.1 as part of the filtering stage. If any of the if statements (lines 2, 4, 7, 10 and 21) is false, the procedure returns false and verification is not executed. Otherwise, if all 5 tests are true, the procedure makes a call to SIP1 (Algorithm 11), which is discussed in the next Section.

Now follows a discussion about the cost of each failure test. Failures 1 and 2 are the fastest: each of them takes $\mathcal{O}(1)$ time to compute. For failure 1, one needs only to return the sizes of the number of vertices in *P* and in *T*, which is computed while initializing the graphs. For failure 2, for every graph *G*, we have an array that stores all unique labels that occur in *G*. The size of this array is known after the initialization of *G*. To check whether test 2 is true, one needs to compare the size of the labels array of *P* with the size of the labels array of *T*, which takes $\mathcal{O}(1)$ time.

Failures 3 and 4 have slower running time. Checking whether $\lambda_P(l)$ is bigger than $\lambda_T(l)$ involves iterating over all labels in L(V(*P*)), and for each of them checking $\lambda_T(l)$, which can be found in constant time by taking the size of the pre-computed LNDS(*T*, *l*). Therefore, the overall complexity of failure 3 is $\mathcal{O}(|L(V(P))|)$. Algorithm 10 shows the pseudo code of the fourth failure test, called in line 10 in Algorithm 9. Test 4 iterates over the LNDS of every label *l* in L(V(*P*)) and checks whether LNDS (*T*, *l*) $\succeq$ LNDS (*P*, *l*) (lines 5, 6). The correctness of the implementation follows from the fact that the elements in LNDS are ordered in non-increasing order. The complexity of test 4 is therefore $\mathcal{O}(|\text{L(V(}P\text{))}|.|\text{LNDS (}T\text{, }l\text{)}|)$.

Failure 5 visits each vertex *v* in V(*P*) (line 13), initializes domv (line 14) and for every *w* in V(*T*), checks whether *w* can be mapped to *v* (lines 15, 16). The complexity of this test is therefore equal to $\mathcal{O}(|V(P)|.|V(T)|)$.

For the implementation of filtering, the following classes are introduced.

- Class *Graph* It creates graph objects, given a file with graphs represented in a certain format. A graph object *G* has size, denoted as *m*, order, denoted as *n*, *id*, array of the degree of each vertex, called *deg*, bitset array of the neighbours of each vertex, called *N*, and array, called *labels*, that stores all labels assigned to vertices in *G*. N_i contains a bitset of the neighbours of the i-th vertex in *G* and $labels_i$ contains the label of the i-th vertex in *G*, for every i between 0 and *n*. Initialization of each of the aforementioned properties takes $\mathcal{O}$(m + n) time. When *labels* is constructed, a new object for each unique label *l* is created and LNDS (*G*, *l*) is computed.
- Class *Label* This class represents a label *l* in *G*. *l* has a *name*, and an array of integers, sorted in non-increasing order that represents LNDS (*G*, *l*). It is built using insertion sort algorithm which is of complexity $\mathcal{O}(|LNDS(G, l)|^2)$ [14].
- Class *SIP1* This class implements the light filtering procedure displayed in Algorithm 9 as well as the SIP algorithm, which is explained in more detail in the next Section.

Algorithm 9 Lights Filters

```
1: procedure FILTER (G_p, G_t)
2:     if ¬ |V(T)| ≥ |V(P)| then return false                          ▷ trivial failure 1
3:     end if
4:     if ¬ |L(V(T))| ≥ |L(V(P))| then return false                    ▷ trivial failure 2
5:     end if
6:     for l ∈ |L(V(P))| do
7:         if λ_P(l) > λ_T(l) then return false
8:         end if                                                      ▷ trivial failure 3
9:     end for
10:    if ¬SUBSUMES (T, P) then return false
11:    end if                                                          ▷ trivial failure 4
12:    alldoms ← initialize   ▷ An array of size the order of G_p that contains the domain of each vertex in P
13:    for every v ∈ V(P) do
14:        domv ← new BitSet(|V(T)|)   ▷ initialize domV to bitset of size the order of the target
15:        for ∀ w ∈ V(T) do
16:            if L(P, v) = L(T, w) ∧ v.degree ≤ w.degree then
17:                domv[w] ← 1
18:            end if
19:        end for
20:        alldoms[v] ← domv
21:        if domv = ∅ then return false                               ▷ trivial failure 5
22:    end for
23: SIP1(alldoms)                          ▷ if no failures occurred, call SIP1 algorithm
24: end procedure
```

Algorithm 10 Graph T subsumes graph P

```
1: procedure SUBSUMES (T, P)
2:     for l ∈ |L(V(P))| do
3:         B ← LNDS (P, l)
4:         A ← LNDS (T, l)
5:         for i in range (0, |A|) do
6:             if B[i] > A[i] then return false
7:             end if
8:         end for
9:     end for
10: return true
11: end procedure
```

4.2 SIP1 Implementation

SIP1 is a subgraph isomorphism algorithm, based on the simplest of the Glasgow algorithms [26]. Algorithm 11 shows a pseudocode of SIP1. It takes the domains of all vertices in the pattern P, initialized in Algorithm 9), and repeatedly tries to assign to each variable (pattern vertex) a value (target vertex). If current assignment is compatible with the partial solution, the algorithm makes a recursive call (line 19) otherwise it backtracks. When a pattern variable u is instantiated with a target value i (line 9), all uninstatiated (future) variables have i removed from their domains (line 12). If a future variable v is adjacent to u in P then domv becomes the intersection of domv with the neighborhood of vertex i in T. This constraint is enforced by applying a logical *and* operation between the two bit sets (line 14). SIP1 uses forward checking (FC) with fail first heuristic [18]: for all uninstantiated variables representing pattern vertices, it selects to explore the one that has the smallest domain before the others (line 4).

Algorithm 11 SIP1

```
1:  procedure SIP1 (alldoms)
2:      if alldoms = ∅ then return solution                ▷ solution is either true or false
3:      end if
4:      domu ← smallest(alldoms)                           ▷ select vertex u with the smallest domain first
5:      consistent ← false
6:      newAlldoms ← initialize with size = (|alldoms| - 1)
7:      for i ∈ domu.nextSetBit ∧ ¬consistent do           ▷ for each entry in position i that could be assigned to u
8:          consistent ← true
9:          u ← i                                          ▷ assign i as a value of vertex u
10:         for (domv ∈ alldoms) ∧ (domv ¬= domu) ∧ consistent do   ▷ iterate through the domain of each
    vertex while the current assignment is consistent
11:             newdomv ← domv
12:             newdomv[i] ← 0                             ▷ cannot take value assigned to u
13:             if (u, v) ∈ E(P) then                      ▷ If u is adjacent to v in P
14:                 newdomv ∧ neighbours of i ∈ T          ▷ v can only take vertices in G_t adjacent to i
15:             end if
16:             newAlldoms ← newAlldoms + newdomv          ▷ add newdomv to newAlldoms
17:             consistent ← (newdomv = 1)                 ▷ if there is a domain wipeout, consistent becomes false
18:         end for
19:         consistent ← consistent ∧ SIP1(newAlldoms)     ▷ call SIP1 if current assignment is consistent
20:     end for
21:     return consistent
22: end procedure
```

In the worst case, SIP1 will assign all values from the domain of each pattern vertex, making recursive calls to SIP1 and failing late, therefore exploring very deep in the search tree before finding that there is no solution. In practice, due to the fail first heuristic used, the algorithm very rarely fails deep in the search tree, because the value that is most likely to fail is first explored, therefore failures occur mostly near the top of the search tree.

Chapter 5

Evaluation

This Section presents the results of the empirical analysis of Light Filters, a subgraph query processing technique described in Chapter 4. Light Filters was run with each of the Big Data datasets, discussed in Section 2.1. The experiments are conducted on a Windows 7 SP1 host with 2 Intel Xeon E5-2660 CPUs (2.20GHz, 20MB Cache, 8 cores/16 threads per CPU) and 128GB of RAM, which is the same machine used by [22]. Run time is measured in milliseconds from when the process starts until it completes, including the time to read in all the graphs, to perform filtering and verification for each instance and to write out all results to a file.

This Chapter is organised as follows. First, we evaluate the filtering performance of Light Filters in Section 5.1. Section 5.2 analyses the hardness of the problems in the four datasets for the SIP algorithm implemented in Light Filters. Section 5.3 investigates the hardness of SAT and UNSAT SIP instances and attempts to find out which of them is more challenging for the Light Filters SIP solver. Section 5.4 compares Light Filters with six well-established subgraph query processing techniques, based on the filtering-verification paradigm.

5.1 Filtering performance

The plot on Figure 5.1 shows the following 7 metrics: the percentage of SAT instances (SAT SIP), the percentage of UNSAT instances that were discovered after verification (UNSAT SIP), and the percentage of instances that were discovered as UNSAT by failing one of the filtering tests (Fail 1, Fail2, etc.), following the same order as the one indicated in Table 4.1 for each dataset. Figure 5.1 shows four separate bars and a table with 5 columns and 7 rows. Each bar represents the 7 aforementioned metrics for one of the datasets. For instance, the leftmost bar represents results obtained from Aids and the rightmost- the results obtained from Ppigo. The color notation of each metric is specified in the first column of the table. For example, the dark-most color shows SAT SIP and the white color represents Fail 1 for a given dataset.

Each table column under a given bar contains the same figures for the same dataset as the ones shown on the bar. They are supplied in order to help the reader see the exact value of each metric. For instance, the leftmost bar represents Aids, where 8.67% of all instances are SAT. Out of the UNSAT problems, 24.211% were discovered during the verification stage and the rest were filtered by either of the trivial failures. The number of SAT and UNSAT problems is also given on Table 2.2.

Note that all SAT SIP instances had to go through verification as well as all UNSAT SIP instances that were not pruned during filtering (UNSAT SIP). For instance, for Aids, the total percentage of instances that were solved by call to SIP1 is 32.881% and the rest of them were pruned by the failure tests. Looking at Figure 5.1, we make the following observations:

- Filtering gives best performance for the instances of Aids. In particular, 67.119% of the targets are rejected before verification. A perfect filtering technique would prune additional 24.211% of all instances in the dataset (UNSAT SIP).
- Aids contains the largest number of UNSAT instances (91.33%), which means that a filtering algorithm, performed on this dataset, has the chance to influence the performance of subgraph query processing for up to 91.33% of all instances. During analysis of existing work we observed that Aids tends to be the most commonly used dataset (and sometimes the only one) for evaluation for subgraph query processing algorithms [6, 21, 24, 47].
- 100% of the instances in Pdbs underwent verification. Similarly, only 3.75% of the targets were filtered in Ppigo. Most of the instances of Pdbs and Ppigo are SAT (77.22% and 61% respectively) and they had to go through verification to be solved. Here, a perfect filtering technique would filter no more than 22.78% and 39% of the instances Pdbs and Ppigo respectively. In such cases, a subgraph query processing method that puts low effort in filtering and implements an efficient verification algorithm will have much higher performance than a method that employs heavy filtering approach and naive SIP algorithm. This hypothesis is confirmed by the results of the study presented in [22], where some of the evaluated indexing techniques, considered as state of the art, never terminate for Pcms and Ppigo. This was also noted during the performance analysis of CT-Index in Section 2.3.3.
- There are duplicate target graphs in the Pcms and Pdbs. For instance, Pcms is supposed to contain 200 targets [2]. In practice, there are only 50 unique graphs and each of them is added 4 times. Pdbs is composed of 600 targets [2], but out of them only 30 are unique, each of them duplicated 20 times.

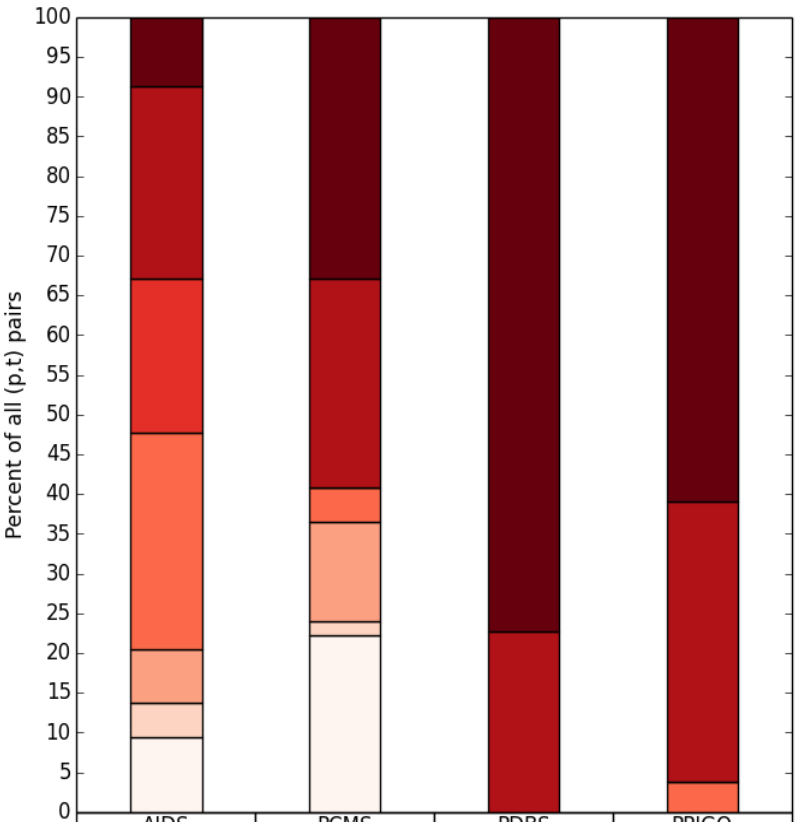
One of the main points made above identifies the existence of a maximum bound on the effectiveness of all filtering techniques, that is the number of UNSAT instances for the given dataset. Unlike filtering, verification does not have such effectiveness limits.

5.2 Hardness of verification

In [22] it is said that subgraph isomorphism tests are "too time consuming", due to the nature of complexity of the subgraph isomorphism problem. We conduct experiments using the four Big Data datasets and the implementation of Light Filters to check the correctness of this hypothesis. We use two measurements for difficulty of SIP- in terms of search nodes, discussed in Section 5.2.1, and in terms of running time, discussed in Section 5.2.2.

5.2.1 Hardness of verification in terms of search nodes

A search node denotes the number of recursive SIP calls taken to find a solution if the problem is SAT or prove that the problem is UNSAT. For every dataset D, we take all instances that were not rejected during filtering and we compute the number of search nodes taken for verification of each instance. We then compute the number of SIP instances n_i solved for a given number of search nodes i. Figures 5.2, 5.3, 5.4 and 5.5 present our results. Here, n_i is represented as percentile of all targets (the x-axis). The search effort is plotted, starting from the easiest percentile (the leftmost part of the x-axis) and finishing with the last percentile representing the hardest instances in terms of search effort (on the rightmost part of the x-axis). The y-axis shows the cumulative difficulty of SIP calls in terms of search nodes for each percentile of the targets in a log scale. For example, looking at Figure 5.2, 24% of the targets are solved by using at most 2 nodes of search and 50% of all targets are solved in less than 10 nodes. The hardest instances take at most 600 nodes.



	AIDS	PCMS	PDBS	PPIGO
SAT SIP	8.67	32.8	77.22	61
UNSAT SIP	24.211	26.313	22.78	35.25
Fail 5	19.387	0	0	0
Fail 4	27.263	4.444	0	3.75
Fail 3	6.701	12.444	0	0
Fail 2	4.315	1.777	0	0
Fail 1	9.453	22.222	0	0

Figure 5.1: Satisifability and average filtering percentage for each method in Table 4.1 for each of the datasets

The graphs below show the hardest instance observed for each percentile of D. For example, if we had 3 graphs that belong to the i^{th} percentile of D and they were solved in 1, 2 and 10 nodes respectively, the y-axis value of i would be 10. Therefore, the datasets are in practice easier than what is shown on Figures 5.2, 5.3, 5.4 and 5.5, which present the hardest instance for each percentile in the dataset. These Figures help us to make the following observations:

- The easiest dataset is Ppigo. Looking at Figure 5.5, 88% of all targets are solved by at most 4 search nodes and 28% are solved by at most 1 node of search effort. The hardest problem (the right-most bar) takes 65 search nodes to solve and it is between pattern "8_1.6" and target "#MUS/Mus_musculus.sif>0.5.sif". The time taken to solve this instance is 4 milliseconds, and the instance is UNSAT.
- Pdbs is harder than Ppigo and Aids and it has instances of most varied difficulty (in terms of search nodes). Figure 5.4 shows that 20% of the targets in Pdbs are solved by using at most 100 search nodes, which is significantly higher than Ppigo, where even the hardest instance was solved in less than 70 search nodes. The hardest instance here is between pattern "32_1ARO" and target "#g" and it is solved in 7,152 nodes for 95 milliseconds. This instance is UNSAT.
- Pcms is the dataset with the hardest instance, which takes 10,470 search nodes to be solved and it is between pattern "16_1C5G.cm.A" and target "1CY2.cm.A.cmap". It is solved in 12 milliseconds and it is UNSAT. Looking at the other 99% of the targets in Pcms, we can see that they are mostly easy. For example, 43% of the SIP instances are solved in 10 search nodes at most.
- Aids is comparably easy. The maximum number of search nodes taken to solve a SIP instance is 619. The instance is between the pattern #1 and the target #629591, it was solved in 0 milliseconds and it is UNSAT.
- Looking at Aids, Pcms and Pdbs, the number of search nodes taken for verification grows exponentially with each percentile.
- The hardest instance of each dataset is UNSAT.

All aforementioned points show that most of the instances of the four datasets require low number of recursive calls of the verification algorithm, implemented in Light Filters.

5.2.2 Hardness of verification in terms of running time

Table 5.1 shows the total time taken to solve all SIP instances of a given dataset, measured in milliseconds. The first row shows the total running time of Light Filters, which includes file I/O, creating and instantiating objects and domains of variables, filtering and the verification time. The second row shows the time taken to perform the filtering step and the third: the SIP algorithm. Note that the filtering step is performed for every SIP instance, whereas verification is applied only on instances that were not rejected during filtering. The percentage of calls to SIP for each dataset can be seen on Figure 4.1.

The table shows that reading in the graphs from a file and instantiating the required objects and variables takes significantly more running time than filtering and verification for each dataset. This shows that further improvement of filtering and verification algorithms would not have significant influence on the total performance of Light Filters. Looking at the results from Pdbs (the dataset where verification was performed for every instance), we can see that even here verification is several times cheaper than file I/O and objects instantiation. Solving the subgraph isomorphism problem for all 4 datasets can not be described as "too time consuming" for the verification algorithm implemented in Light Filters.

The 5,006 milliseconds spent on filtering for SIP problems in Pdbs was wasteful, because no instance was rejected (5.1). Performing SIP algorithm on all 3,600 instances (2.2) took 16,102 milliseconds, which makes

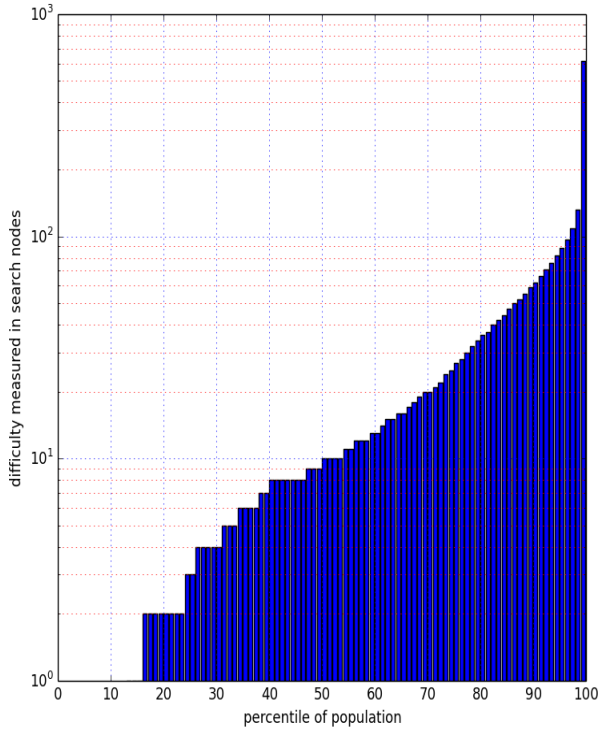


Figure 5.2: SIP on Aids

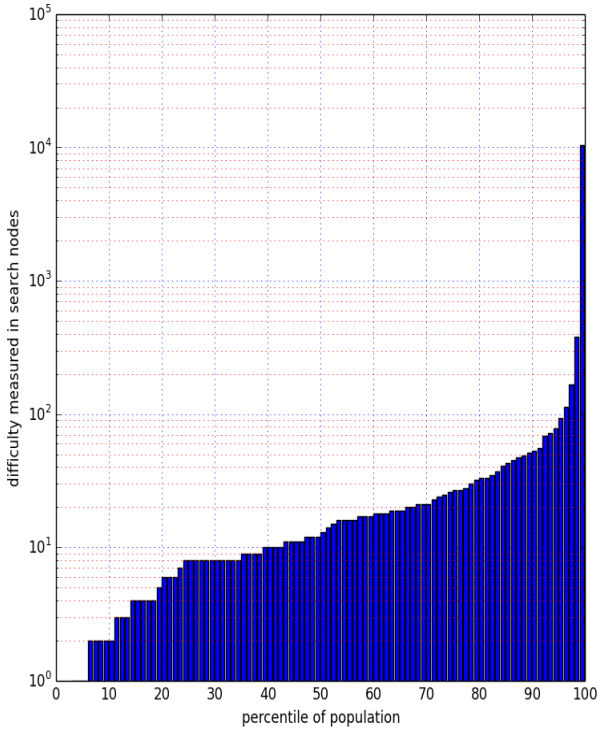


Figure 5.3: SIP on Pcms

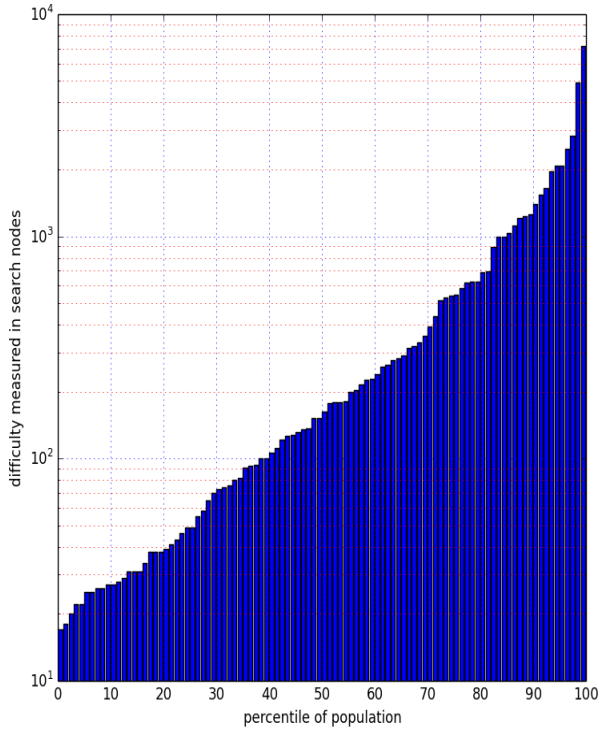


Figure 5.4: SIP on Pdbs

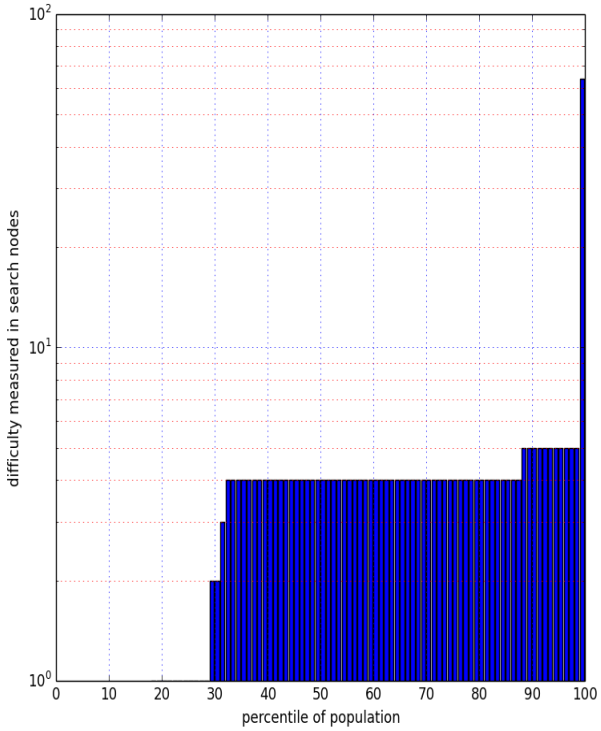


Figure 5.5: SIP on Ppigo

	Aids	Pcms	Pdbs	Ppigo
total cpu T	15,770	26,855	133,451	11,886
total filtering T	2,569	1,500	5,006	379
total verification T	2,687	1,013	16,102	51

Table 5.1: Total running time in milliseconds for each dataset

4.47 milliseconds per instance on average. During the analysis of the search effort, it was noticed that Pdbs is the hardest dataset. Achieving so fast verification time shows again that the four Big Data datasets are indeed very easy.

The investigation of the hardness of the four Big Data datasets shows that all of them are easy and some of them (Ppigo) are particularly easy. The data obtained after the experiments shows that for Light Filters, the bottleneck of the performance of subgraph query processing is mainly I/O and objects initialization even for the dataset that involved executing verification for each instance (Pdbs). This suggests that the claim that solving the subgraph isomorphism problem for every instance takes substantial amount of time is not valid for the studied four datasets and SIP algorithm.

So far we discovered that although Aids, Pcms, Pdbs and Ppigo are commonly used by Big Data research for evaluation, they are of dubious quality. Each of the datasets claims that is contains substantial number of complex graphs, but in practice all of them are composed of easy instances. Moreover, two of the datasets contain large number of duplicates. This puts into question the performance of current filtering-verification techniques, which are evaluated only with these datasets, on larger and more complex datasets. As Light Filters method is evaluated only with the Big Data datasets, we can make conclusions on its performance mainly in comparison with the performance of other filtering-verification techniques. This is done in Section 5.4.

5.3 Are UNSAT SIP instances generally harder to solve than SAT SIP instances?

The observation that the hardest instance of each dataset is UNSAT raises the following question: are UNSAT SIP instances generally harder to solve than SAT SIP instances? The experiments described in this Section intend to investigate this.

The following eight plots below break each of the plots discussed in Section 5.2.1 (namely 5.2, 5.3, 5.4 and 5.5) further down in terms of whether the SIP instances are SAT or UNSAT. The blue plots represent all satisfiable SIP pairs for a dataset D. Similarly, the red plots represent all unsatisfiable SIP instances of D. For each D (namely, for Aids, Pcms, Pdbs and Ppigo), the union of the blue plot (SAT SIP, left-hand side) and the red plot (UNSAT SIP, right-hand side) gives the plot for the corresponding dataset discussed in Section 5.2.1.

Note that the plots on Figure 5.9 contain only 4 bars each, i.e. the data is divided into quartiles instead of percentile. Here, each bar represents 25% of all instances of a category (SAT/UNSAT). For example, the left plot shows that the lowest quartile of the SAT SIP calls takes no more than 4 nodes to solve, as it is also true for the second quartile. We changed the percentile representation for this dataset, because the number of SAT and UNSAT SIP (61 and 39 respectively, Table 2.2) instances is too small to be scaled to percentiles.

Table 5.2 presents statistics in terms of number of search nodes for SAT (blue columns) and UNSAT (red columns) instances. For instance, the Table shows that for Aids, the total number of search nodes taken to solve all SAT SIP instances is 437,108, and the total number of search nodes taken to solve all UNSAT SIP instances is 2,295,724. Using these Figures, we derive that the total number of search nodes taken to solve all SIP instances in Aids is the sum of those two numbers, which is equal to 2,732,832. Table 2.2 also shows the number of instances

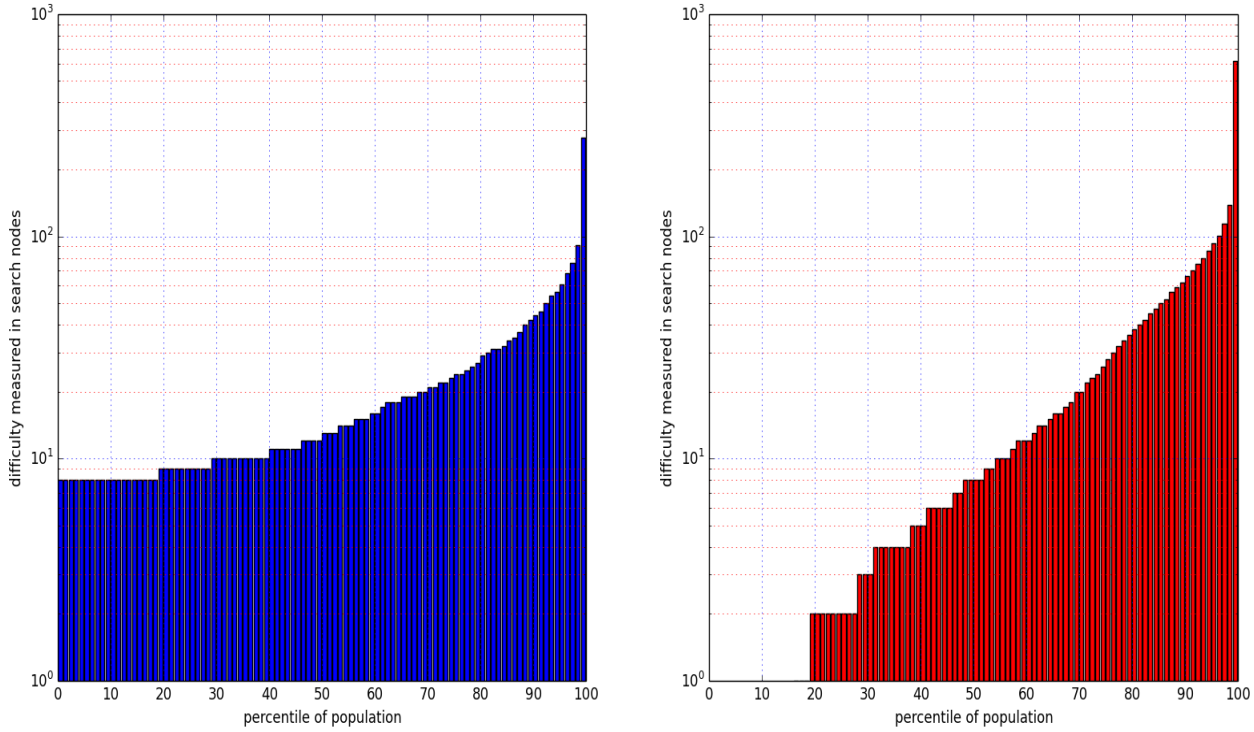


Figure 5.6: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in Aids

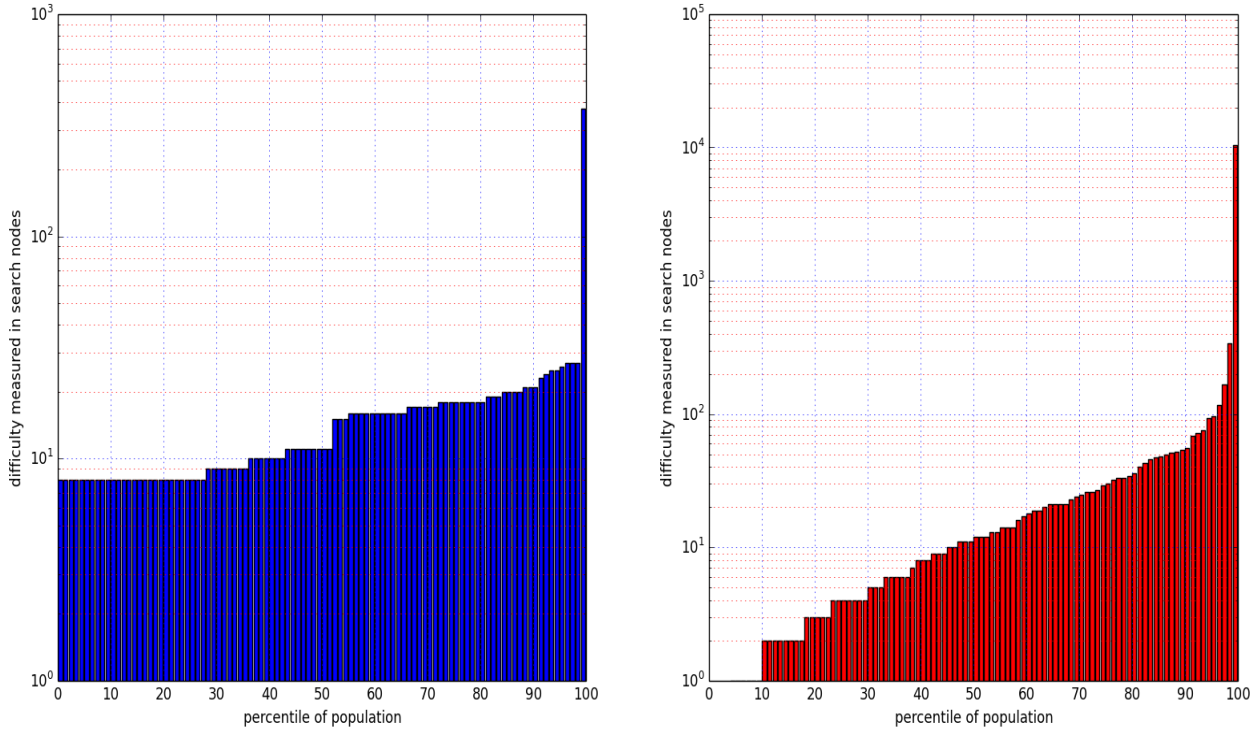


Figure 5.7: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in Pcms

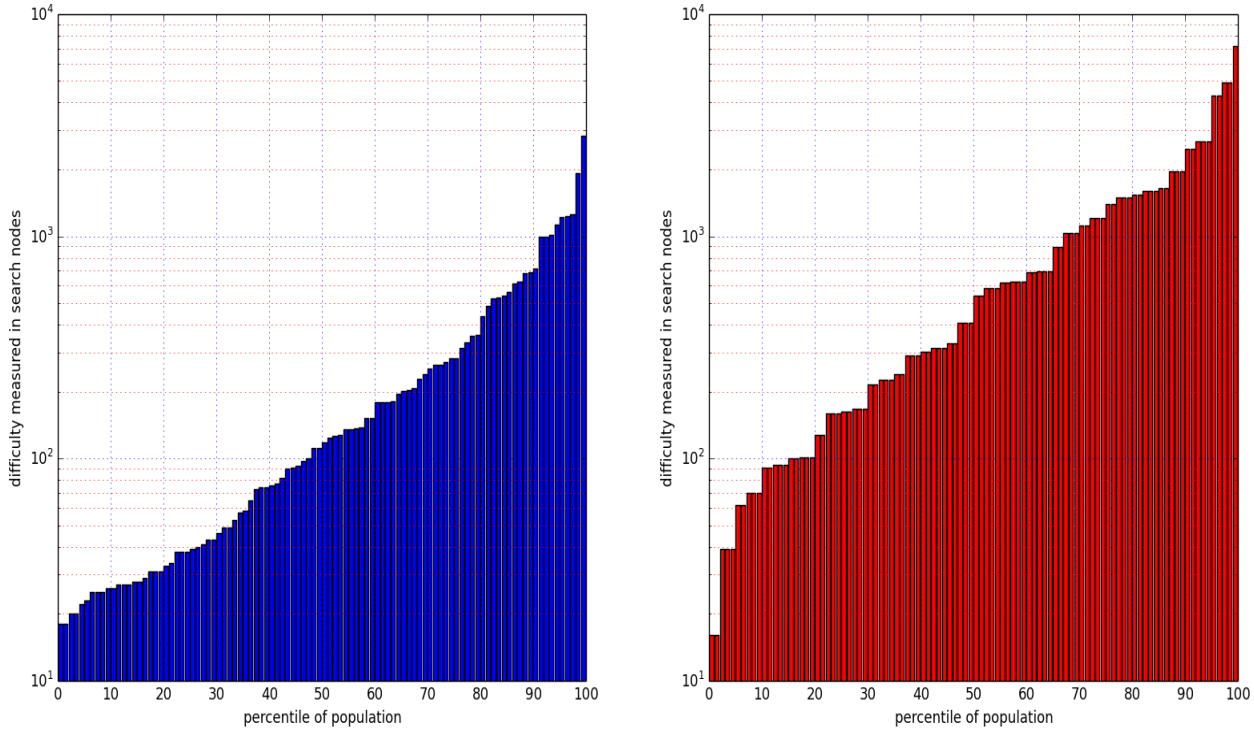


Figure 5.8: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in Pdbs

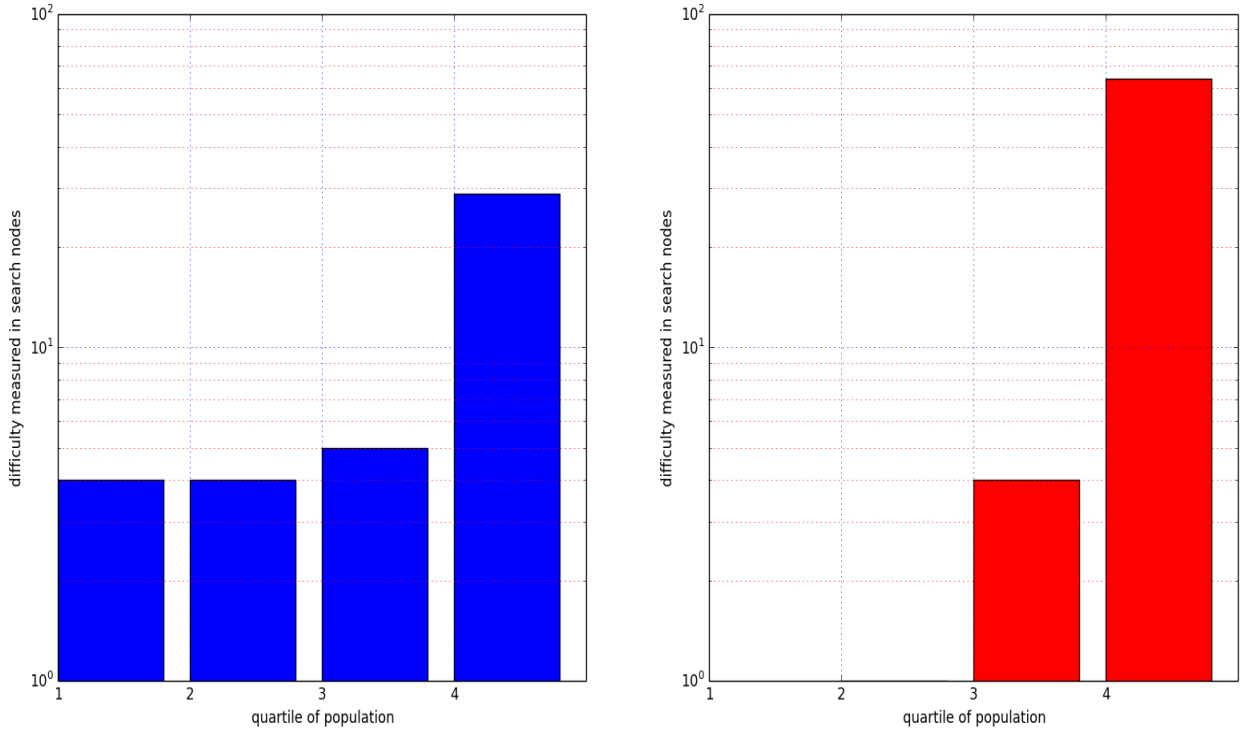


Figure 5.9: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in Ppigo

	Total		Average		Median		Minimum		Maximum	
Aids	437,108	2,295,724	21	10.4	13	0	9	0	279	619
Pcms	13,644	133,276	23	110.3	17	9	9	0	378	10,470
Pdbs	894,260	854,720	322	1,042.3	123	544	18	17	2,845	7,152
Ppigo	714	312	6.932	5.8	6	2	5	0	30	65

Table 5.2: Number of nodes of search effort for each dataset. Blue for solvable and red for unsolvable SIP instances

and percent from each category(SAT/UNSAT). The Table tells us that the reason for the large difference in terms of search effort between SAT and UNSAT instances is that 91% of all instances are UNSAT (almost ten times more than SAT). The two tables and the 8 figures show that:

- In Aids, Pcms and Ppigo, the easiest percentile of UNSAT SIP instances requires less number of search nodes than the easiest percentile of SAT SIP instances. The hardest percentile of UNSAT SIP takes bigger number of search nodes than the hardest percentile of SAT SIP instances.
- In Pdbs, there is a big difference in terms of search effort between SAT and UNSAT problems. For example, SAT instances are easier for every percentile of the targets (5.8). The tabulated results on Figure 5.2 confirm this observation. On average, SAT instances are 3 times easier than UNSAT, the SAT instances median is more than 4 times smaller than the UNSAT instances median and the number of search nodes taken to solve the hardest SAT instance (2,845) is much less than the number of search nodes taken to solve the hardest UNSAT instance (7,152).
- Table 5.2 shows that in Pdbs, the total search effort taken to solve UNSAT problems is bigger (894,260 search nodes taken in total for SAT and 854,720 search nodes in total taken for UNSAT problems). However, Table 2.2 shows that SAT SIP consists of 77.22% of all instances in Pdbs. Therefore, the large search effort of SAT SIP problems for pdbs is due to their substantially larger number compared to UNSAT problems and in practice, UNSAT SIP was much more difficult to solve than SAT SIP for this dataset. This is confirmed by the average search nodes figures (the second row in Table 5.2), where a SAT instance is on average 3 times easier to solve than an UNSAT instance.
- Pcms is composed of mostly UNSAT SIP instances (2.2, 5.1). Similarly to Aids, this is the reason why the total number of search nodes for all UNSAT problems is considerably larger than the number of search nodes for all SAT problems (5.2). However, the hardest SAT problem is substantially easier than the hardest UNSAT. The difference is 10,092 search nodes, where the SAT problem takes 378 search nodes to be solved (5.2) and it is SIP ("#32_1CY1.cm.A.out", "#1CY0.cm.A.cmap"), solved for 4 milliseconds. Average search nodes figures also show that a SAT instance is on average 5 times easier to solve than an UNSAT instance(the second row in Table 5.2).
- 61% of all instances in Ppigo are SAT (2.2, 5.1) and this is the main reason why the total SAT SIP search effort is larger than the UNSAT SIP search effort. The average search effort displayed in Table 5.2 shows that SAT and UNSAT SIP instances are similarly hard on average for this dataset.

5.4 Comparison with Big Data algorithms

We make a comparison of Light Filters with a selection of "well-established" subgraph query processing algorithms, namely CT-Index [21], gCode [24], Grapes [17], tree+Δ [47], GGSX [6] and gIndex [43]. Their

evaluation is described in [22]. All of these algorithms implement the filtering-verification framework with heavy filtering approach, using an index structure, and run SIP algorithm during verification, that is VF2 [13] or a modification of it. The work in [22] uses the same datasets and the same machine as us for its experiments. The comparison is made first with respect to running time and then with respect to filtering strength.

For Aids, the fastest of the evaluated algorithms is Grapes [17]. Filtering took 8 seconds and verification-about 600 milliseconds. It took us 2,569 milliseconds for filtering (5.1), but verification was four times slower (2,687 milliseconds). Light Filters approach has slower verification than CT-Index, but much faster filtering. The algorithms with slowest filtering and verification time, evaluated in [22], are gIndex and tree+Δ (15000 and 1500 seconds for filtering, 8 and 40 seconds for verification respectively [22]). Light Filters is several times better than them on each stage.

In the study described in [22], it was observed that for Pcms and Ppigo, for four of the evaluated algorithms filtering never finished executing. Interestingly, Pcms and Ppigo are the only two datasets that are composed by mainly SAT instances (Figure 5.1). Table 5.1 shows that the performance of the Light Filters is incomparably faster, where filtering of all instances took only 1,500 milliseconds for Pcms and 5,006 milliseconds for Pdbs. The results in [22] show that the only two algorithms, that can solve these datasets, are Grapes and GGSX [6]. The running time of both of them for both filtering and verification is slower than the running time of Light Filters. We can deduce that for solving Pcms and Ppigo, Light Filters is the single best technique.

We get similar results as for the Aids dataset, when we compare running time of Light Filters and the other six techniques for Pdbs.

Below we give a summary of the main discoveries while investigating performance running time.

- With respect to running time, for the two of the datasets, that have mainly SAT instances, Light Filters is the single best technique. In terms of running time of filtering, Light Filters is the fastest. This is not surprising. This result is not surprising. The filtering part of our method constitutes of five simple tests (Section 4.1), whereas the filtering of each of the other methods follows the classical principles of the filtering-verification paradigm. In terms of verification time, Light Filters is neither the fastest, nor one of the slowest. The algorithms evaluated in [22] have an additional overhead that is not present in our approach, which is the size of the index that has to be stored. However, the cost of the filtering procedures of Light Filters is paid every time no matter of the dataset, whereas the index built during filtering of classical filtering-verification methods can be built upfront and reused as long as the database does not change.
- It was previously shown that the maximum bound on effectiveness of filtering is defined by the number of UNSAT SIP instances in the dataset. Four of the indexing algorithms, evaluated in [22], never finish execution for Pcms and Ppigo, which are mainly composed of SAT SIP instances. Therefore, executing expensive filtering algorithms for such datasets is difficult to justify both in terms of effectiveness and efficiency.

We now compare the filtering performance of Light Filters with the six techniques.

To make the filtering performance of Light Filters comparable with the results in the work described in [22], we used formula 2.1, discussed in Section 2.3.3 to calculate the FP ratio of Light Filters for every of the four datasets. Table 5.3 shows our results.

First, we can deduce that formula 2.1 behaves as we discussed in Section 2.3.3. For datasets where filtering removed 0 or close to 0 instances, like Pdbs and Ppigo, the value of FP ratio depends only on the number of SAT instances in the dataset and it does not show that the filtering performed poorly. Figure 5.1 shows that filtering removed 0 instances from the Pdbs dataset, however, the FP ratio for Pdbs shows great filtering performance

	FP Ratio
Aids	0.73
Pcms	0.44
Pdbs	0.23
Ppigo	0.36

Table 5.3: FP ratio of filtering of Light Filters for each of the datasets

of 0.23 (Table 5.3). Filtering was most successful for Aids (Figure 5.1), but according to Table 5.3, it is worst for Aids. This data further supports our observations in Section 2.3.3 that formula 2.1, used to evaluate the filtering power of some filtering-verification algorithms [22], can be used only in order to compare performance of different filtering methods, executed on the same datasets, and the FP ratio values on their own do not suggest anything accurate about the effectiveness of a given filtering algorithm.

According to the results, published in [22], the five simple failure tests do not perform worse than the studied filtering-verification techniques in that publication (which are considered to be "well established indexing methods" [22]). For instance, the best FP ratio for Aids was obtained by tree+Δ with value 0.2 [22] and the worst FP ratio is achieved by 3 algorithms (Grapes, CT-index, gCode) with value 0.8 [22]. The FP ratio of Light Filters for Aids is 0.73 (Table 5.3). The best FP ratio for Pdbs in [22] is obtained by Grapes and has the value 0.04 [22], the worst value is 0.3 [22], which is slightly worse than the FP ratio obtained by Light Filters of value 0.23 (Table 5.3). For Pcms and Ppigo, we can compare only two algorithms with Light Filters, because only they finished execution. Here, Light Filters is significantly better than the worst of the algorithms (Table 5.3), that has FP ratio equal to 0.7 [22], but worse compared to the best algorithm that has FP ratio equal to 0.2 [22]. The results are similar when comparing results from the Ppigo results of [22] and the figures in Table 5.3.

- The comparison of the filtering methods implemented in Light Filters with six well-established filtering-verification methods [22] using the same four datasets, shows that small, easy to compute filtering mechanisms (Light Filters) can in some cases give better filtering performance than subgraph query processing methods that employ heavy index structures.

Chapter 6

Conclusion and Future work

This Chapter presents a summary of this work. It starts by outlining the major steps carried out and what each of them suggests. The second part of the Chapter, discusses opportunities for future work, which could follow on from the project.

6.1 Project Summary

We investigated the subgraph query processing problem and we looked at two currently existing methods to solve it: the filtering-verification paradigm, considered very effective among the area of Big Data research, and directly solving the subgraph isomorphism problem for every instance without the use of computationally expensive filtering methods before search.

Well-established filtering-verification technique, called CT-Index [21], and CP15 [26]- a new subgraph isomorphism problem algorithm that is shown to be one of the fastest, were analysed. After review of the literature, we observed that all filtering-verification based techniques are mainly focused in developing new filtering methods, while reusing the same subgraph isomorphism algorithm for verification, which was proved to be of substandard performance.

We investigated the area of developing filtering methods by designing and implementing a filtering framework with two methods for filtering and candidates extraction, namely Path Index and Path-Subtree Index. Path Index follows a common feature construction algorithm and Path-Subtree Index is a new technique. Theoretical and empirical analysis showed that Path-Subtree Index prunes at least as many instances as Path Index, but has higher complexity and storage requirements than Path Index. Both techniques were shown to be effective, but too slow to be used in practice. Possible ways to lower their complexity were suggested.

A new subgraph query processing technique, called Light Filters, was introduced. It implements a modification of the filtering-verification paradigm that does not employ any index structure. Filtering consists of 5 simple failure tests and verification is performed by a subgraph isomorphism algorithm, based on the simplest of the Glasgow algorithms [26]. The empirical study of Light Filters was carried out using the same datasets and setup as in [22]. This allowed for a comparison between Light Filters and a selection of the best existing filtering-verification techniques, evaluated in [22]. With respect to filtering performance, the results showed that some of these techniques were outperformed by Light Filters. In terms of both filtering and verification running time, Light Filters is the fastest. Although Light Filters uses very simple filtering tests, it has better filtering strength than some of the indexing techniques it was compared with. This shows that Light Filters can be a good alternative to existing expensive filtering strategies.

The 4 Big Data benchmark datasets were discovered to be of dubious quality. They contain easy instances, which puts into question the performance of all filtering-verification based algorithms, that are evaluated only with these datasets, on hard subgraph isomorphism problem instances. The datasets can also be easily kept in memory of a standard laptop without the usage of additional hardware. This shows that Aids, Pcms, Pdbs and Ppigo are not adequate representatives of Big Data datasets, nonetheless they are treated as such. It also raises the question of how Big Data filtering-verification algorithms, would behave when evaluated with bigger and more complex datasets.

We observed that for datasets that are composed of mainly SAT SIP instances, indexing methods are not only highly inefficient (typically never finish execution), but also bound to be of less effectiveness. For such datasets, it is better to use approaches like Light Filters that emphasize on verification and use fast and simple filtering techniques.

We did experiments to find out whether SAT problems are generally easier than UNSAT problems that were not rejected by filtering. The results vary depending on the dataset and no definite conclusion can be made. What was observed for each of the datasets is that the hardest and the easiest instances in terms of search nodes are both UNSAT.

6.2 Future Work

This work puts into question the performance of subgraph query processing methods based on the filtering-verification paradigm, mainly because of the dubious quality of the benchmarks datasets. This leads to several possible extensions of this work. One could develop new datasets that consist of harder subgraph isomorphism problem instances. Existing work shows how to generate "really hard" random instances for subgraph isomorphism problem [29]. A future project could focus on creating large datasets with this methodology and then conduct experimental evaluation of filtering-verification based algorithms (including Light Filters) with these datasets. This could give better understanding of the performance of subgraph query processing methods with datasets that are possibly better representatives of the real demand of Big Data in terms of size and complexity.

Light Filters techniques were shown to be as good as, or outperform classical filtering-verification techniques in several aspects. Future extension of this work could investigate what makes the current filtering tests so effective and how to extend the filtering by adding more tests.

The study of the hardness of SAT and UNSAT SIP instances can be extended to investigate whether filters manage to prune easy or hard instances. Future work in this direction might be able to come up with new heuristics for filtering hard UNSAT SIP instances cheaply.

It was shown that the effectiveness of subgraph query processing algorithms depends on the nature of the datasets [26]. Future extension of this work could focus on creating a large framework that predicts the difficulty of an instance before solving it and depending on its difficulty, chooses an appropriate filtering and verification methods. There is an existing work that gives a formula how to calculate hard instances, given in [16], and an existing work that investigates where the hard problems are [8].

Bibliography

[1] Daylight theory manual: Fingerprints - screening and similarity. http://www.daylight.com/dayhtml/doc/theory/theory.finger.html#RTFToC77. Accessed: 2016-01-27.

[2] Grapes documentation page. http://ferrolab.dmi.unict.it/GRAPES/grapes.html#formats. Accessed: 2016-01-24.

[3] The ohio state university, data structures, backtracking algorithms. http://web.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch19.html. Accessed: 2016-01-30.

[4] Gilles Audemard, Christophe Lecoutre, Mouny Samy-Modeliar, Gilles Goncalves, and Daniel Porumbel. *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, chapter Scoring-Based Neighborhood Dominance for the Subgraph Isomorphism Problem, pages 125–141. Springer International Publishing, Cham, 2014.

[5] Iva Babukova. Subgraph filtering framework source code. https://github.com/ivababukova/graphIndexing. Accessed: 2016-05-14.

[6] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha. Enhancing graph database indexing by suffix tree structure. *In Proc. IAPR PRIB*, pages 195 – 203, 2010.

[7] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics*, 14(7):1–13, 2013.

[8] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'91, pages 331–337, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[9] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. Fg-index: towards verification-free query processing on graph databases. In *in SIGMOD, 2007*, pages 857–872.

[10] Thayne Coffman, Seth Greenblatt, and Sherry Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47(3):45–47, March 2004.

[11] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition, 2004.

[12] Stephen A. Cook. The complexity of theorem-proving procedures. In *In STOC*, pages 151–158. ACM, 1971.

[13] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Match. Intell.*, pages 1367 – 1372, 2004.

[14] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[15] Guillaume Damiand, Christine Solnon, Colin De La Higuera, Jean-Christophe Janodet, and Émilie Samuel. Polynomial Algorithms for Subisomorphism of nD Open Combinatorial Maps. *Computer Vision and Image Understanding*, 115(7):996–1010, July 2011.

[16] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, and Toby Walsh. The constrainedness of search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*, AAAI'96, pages 246–252. AAAI Press, 1996.

[17] Rosalba Giugno, Vincenzo Bonnici, Nicola Bombieri, Alfredo Pulvirenti, Alfredo Ferro, and Dennis Shasha. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. *PLoS ONE*, 8(10):e76911, 10 2013.

[18] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3):263–313, 1980.

[19] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. pages 607–613. Morgan Kaufmann, 1995.

[20] Huahai He and A. K. Singh. Closure-tree: An index structure for graph queries. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, pages 38–38, April 2006.

[21] P. Mutzel K. Klein, N. Kriege. Ct-index: Fingerprint-based graph indexing combining cycles and trees. *Data Engineering (ICDE), 2011 IEEE 27th International Conference, Hannover*, pages 1115 – 1126, 11-16 April 2011.

[22] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. Performance and scalability of indexed subgraph query processing methods. *Proceedings of the VLDB Endowment, Vol. 8, No. 12*, September 2015.

[23] Javier Larrosa and Gabriel Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Computer Science*, 12(4):403–422, 2002.

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

[25] Ciaran McCreesh. Labels in randomly generated subgraph isomorphism problems. personal communication, 2016.

[26] Ciaran McCreesh and Patrick Prosser. A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, pages 295–312, 2015.

[27] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, April 1976.

[28] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 60(0):94 – 112, 2014.

[29] Ciaran McCreesh Patrick Prosser. Heuristics and really hard instances for subgraph isomorphism problems. *25th International Joint Conference on Artificial Intelligence*, page to appear, 2016.

[30] Patrick Prosser. Domain filtering can degrade intelligent backtracking search. In *Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'93, pages 262–267, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[31] I. Reed. A class of multiple-error-correcting codes and the decoding scheme. *Transactions of the IRE Professional Group on Information Theory*, 4(4):38–49, September 1954.

[32] J C Régin. Développement d'outils algorithmiques pour l'intelligence artificielle. application á la chimie organique. Ph.D. thesis, Université Montpellier, 1995.

[33] Michele Sevegnani and Muffy Calder. Bigraphs with sharing. *Theor. Comput. Sci.*, 577(C):43–73, April 2015.

[34] Christine Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artif. Intell.*, 174(12-13):850–864, 2010.

[35] Christine Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artificial Intelligence*, pages 850–864, 2010.

[36] Christine Solnon, Guillaume Damiand, Colin de la Higuera, and Jean-Christophe Janodet. On the complexity of submap isomorphism and maximum common submap problems. *Pattern Recogn.*, 48(2):302–316, February 2015.

[37] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[38] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 28(1):31–42, 1976.

[39] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*, SWAT '73, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

[40] David W. Williams, Jun Huan, and Wei Wang 0010. Graph database indexing using structured graph decomposition. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *ICDE*, pages 976–985. IEEE, 2007.

[41] Chris Woolston. Breast cancer. *Nature*, 527(7578), 2015.

[42] Yan Xie and Philip S. Yu. Cp-index: on the efficient indexing of large graphs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 1795–1804, New York, NY, USA, 2011. ACM.

[43] Philip S. Yu Xifeng Yan and Jiawei Han. Graph indexing: A frequent structure-based approach. In *SIGMOD '04 Proceedings*, pages 335–346, June 2004.

[44] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 335–346, New York, NY, USA, 2004. ACM.

[45] Dayu Yuan and Prasenjit Mitra. Lindex: A lattice-based index for graph databases. *The VLDB Journal*, 22(2):229–252, April 2013.

[46] Stéphane Zampelli, Yves Deville, and Christine Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15(3):327–353, 2010.

[47] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + $\delta \geq$ graph. *In Proc. VLDB*, pages 938 – 949, 2007.

Glossary

API Application Programming Interface. *Glossary:* API

canonical form A canonical form of a graph G is a labeled graph Canon(G) that is isomorphic to G, such that every graph that is isomorphic to G has the same canonical form as G. 9

depth-first search An algorithm for traversing or searching tree or graph data structures reported to be introduced by Charles Pierre Trémaux, a 19^{th} century French mathematician. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. 16

Hall set A set of *n* whose domains include only *n* values between them. Finding a Hall set allows for removing the values part of the hall set from the domains of variables that are not part of the Hall set.. 15

hash function A function that can be used to map data of arbitrary size to data of fixed size. The values returned by a hash function are called hash values. 9

search node A search node denotes the number of recursive calls to the SIP algorithm taken to find a solution. 43

SIMD Single Instruction Multiple Data (SIMD) is a class of computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment.. 15

suffix tree A suffix tree S is a compressed trie containing all the suffixes of the given text as their keys and positions in the text as their values. It has the following properties: the tree has exactly n leaves numbered from 1 to n; except for the root, every internal node has at least two children; each edge is labeled with a non-empty substring of S; no two edges starting out of a node can have string-labels beginning with the same character; the string obtained by concatenating all the string-labels found on the path from the root to leaf i spells out suffix S[i..n], for i from 1 to n.. 24

tree A tree is an undirected graph such that any two vertices are connected by exactly one path. In this work, we refer to the vertices of the tree as *nodes*.. 8, 9, 24

Acronyms

FC forward checking. 30

SAT Satisfiable. 6, 11, 31, 32, 36–39, 43

SIP subgraph isomorphism problem. 3, 4, 6, 10–12, 21, 25, 28, 32, 34, 36–40

UNSAT Unsatisfiable. 4, 6, 11, 25, 31, 32, 34, 36–39, 43