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

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 starts by introducing the problem statement and the aims and motivations to solve it. We then give important definitions and concepts that are used throughout the report.

1.1 Problem Statement

The subgraph isomorphism problem involves finding a pattern graph inside a target graph, where a graph is a structure that represents schemaless data such as proteins, chemical compounds, and XML documents. Subgraph isomorphism has a wide variety of applications in many fields. For instance, finding the best treatment to fight a particular cancer involves screening a patient's tumor to search for particular set of biomarkers [25]. Such tasks involve repeatedly examining a large number of graphs, typically stored in a database, comparing each of them with a pattern.

1.2 Aims and Motivations

The remaining of this work is organised as follows. Section 1.3.1 contains terminology, definitions and notation used throughout this work. Chapter 2 presents review and analysis of existing approaches to solve the problem and properties of 4 real datasets used for testing and evaluation. An implementation and empirical analysis of two graph indexing and filtering methods is presented in Chapter 3. Chapter 4 describes a new approach to solve the problem, called Light Filters. The evaluation of Light Filters and analysis of the hardness of satisfiable and unsatisfiable 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.

1.3 Terminology, Definitions and Notations

In this Section, we introduce all preliminary terminology and definition used throughout the document. 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.

1.3.1 Graph Theory

A *graph* $G = (V_G, E_G, L_G)$ consists of set of vertices $V_G = \{u_i\}$, $1 \leq i \leq |V_G|$, set of edges $E_G = \{(u_k, u_m) \mid u_k \in V_G, u_m \in V_G\}$, and function L_G: $V_G \rightarrow \mathcal{L}$ that assigns a label $l \in \mathcal{L}$ to each $v \in V_G$, where $\mathcal{L}$ is the set of all possible labels. A graph is *undirected*, if for every $(u, v) \in E_G \Rightarrow (v, u) \in E_G$. In this work, only undirected graphs are considered. The *size* of G is equal to the number of edges in the graph (i.e the cardinality of E_G denoted as $|E_G|$). The *order* of G is equal to the cardinality of its vertex set $|V_G|$.

A *path* in a graph is a sequence of distinct edges which connect a sequence of distinct vertices. A path from vertex *u* to vertex *v* has *u* as the first and *v* as the last vertex in the sequence. A *cycle* is a path where the first vertex in the sequence is also the last. In a given graph G, there may be zero, one or more than one path from *u* to *v* (and similarly for cycles).

The *degree* of $v \in V_G$ is the number of vertices adjacent to *v*, which are referred to as the neighbours of *v*. By $v \sim_G w$ we mean that *w* is a neighbour of *v* in graph G. The set of neighbours of v forms the *neighbourhood* of v, denoted as N_v.

Example 1 Figure 1.2 shows an undirected labeled graph G_t, where each different color represents a vertex label. For instance, vertex 1 is labeled in yellow (Y) and vertex 2 is labeled in red (R). The degree of vertex 1 is 1, because it has only one vertex as a neighbour, namely $1 \sim_{G_t} 2$. Consequently, the neighbourhood of 1 (N_1) is $\{2\}$.

An example of a path from vertex 2 to vertex 6 is the one that goes through vertices 2, 3, 4 and 6. Graph G_t has multiple cycles. For instance the path 2, 3, 8 is also a cycle.

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*.

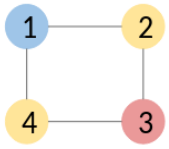


Figure 1.1: graph G_p

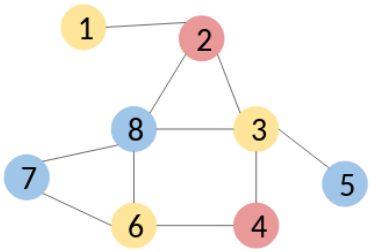


Figure 1.2: graph G_t

Figure 1.3: Instance of subgraph isomorphism problem (SIP)

1.3.2 The Subgraph Isomorphism Problem

A graph $G'(V_{G'}, E_{G'}, L_{G'})$ is a *subgraph* of G if and only if the vertices, edges and labels of G′ are subsets of the vertices, edges and labels of G, that is $V_{G'} \subseteq V_G$ and $E_{G'} \subseteq E_G$ and $L_{G'} \subseteq L_G$.

The *Subgraph Isomorphism Problem (SIP)* between a graph $G_p(V_p, E_p, L_p)$, called pattern and graph $G_t(V_t, E_t, L_t)$, called target, is to find a function f that maps a different vertex of the target to every vertex in the pattern such that for every $v \in V_p$, $v = f(v)$ and $L_p(v) = L_t(f(v))$, where $f(v) \in V_t$; and for all pairs of vertices $u,v \in V_p$, $(u, v) \in E_p \Leftrightarrow (f(u), f(v)) \in E_t$. In other words, a valid mapping of G_p to a subgraph in G_t preserves the labeling and the set of neighbours (and therefore the degree) of each vertex in G_p. If f exists, we say that the SIP

instance is *satisfiable (SAT)*, i.e. it has at least one solution, otherwise it is *unsatisfiable (UNSAT)*, i.e. there are no solutions.

SIP can be *induced* or *non-induced*. The induced version also requires that if $v \nsim_{G_p} w$, then $f(v) \nsim_{G_t} f(w)$. In this work, we discuss only the non-induced SIP that does not have this requirement.

When a valid matching function f exists that maps vertex $v \in G_p$ to vertex $w \in G_t$, we write $v \rightarrow_f w$.

Example 2 Lest us consider the SIP instance (G_p, G_t) displayed in Figure 1.3. This instance is SAT, because a function f exists that produces a valid mapping of each vertex $\in G_p$ to a vertex $\in G_t$. For instance, $1 \rightarrow_f 8$, $2 \rightarrow_f 3$, $3 \rightarrow_f 4$ and $4 \rightarrow_f 6$. However, if $4 \in G_t$ was labeled in Y, then the SIP instance (G_p, G_t) would be UNSAT, because no valid mapping that associates each vertex in G_p to different vertex in G_t would exist.

There are various existing algorithms for the subgraph isomorphism problem [9, 21, 18, 16, 6, 31, 20]. More thorough analysis of the problem and discussion on existing work is presented in Section 2.5.

1.3.3 The Filter-Verification Paradigm

A set of target graphs is called a *database* and it is denoted by *T*. A set of pattern graphs, also known as *query set* or *queries*, is denoted by *P*. A *dataset D* is composed of *T* and *P*.

Subgraph query processing is, when given a set of query graphs, for each query return all graphs from the database that contain the query. These graphs form the *answer set* $\mathcal{A}_q$ of the corresponding query q. A way to perform subgraph query processing is to run a SIP algorithm for every instance (G_p, G_t) $\in$ P$\times$T, where $G_p \in$ P and $G_t \in$ T.

Example 3 Let us consider again Figure 1.3 and assume that it represents a dataset *T*, where Figure 1.1 is the query set, which contains only G_p, and Figure 1.2 represents the graph database, which contains G_t. Then, subgraph query processing would solve the SIP (G_p, G_t), because P $\times$ T = (G_p, G_t), returning $\mathcal{A}_{G_p} = \{G_t\}$, as SIP ($G_p$, G_t) is SAT.

Another way of processing queries is the *filter-verification framework*, which consists of two steps. The *filter* step tries to prune targets that can not be matched to a given query. The set of targets left after this step formes the set of *candidates*, denoted as $\mathcal{C}_p$, where $\mathcal{C}_p \subseteq$ T. The following *verification* step then performs SIP for every pair (G_p,$\mathcal{C}_{G_p}$) $\in$ P $\times$ $\mathcal{C}_p$. The usage of the filter-verification paradigm is motivated by the fact that subgraph isomorphism is NP-complete and reducing the number of SIP calls by discarding targets that are UNSAT for a given pattern during the filtering step and performing SIP using the limited set of candidates would yield to significant performance improvement [14, 15].

There are number of subgraph query processing algorithms based on the filter-verification framework [14, 17, 5, 32, 11]. This approach is discussed in more detail in Chapter 2, where more definitions and annotations are introduced in Section 2.3 followed by analysis of related work in Section 2.4.

Chapter 2

Review of existing work

2.1 The nature of the problem

2.2 Datasets

This Section includes the specifications and the nature of the four real datasets used for performance study of existing work in Section 2.4.3 and for evaluation in Section 5. The datasets are obtained from [2]. Many related research publications use some of them to assess the performance of their developed subgraph query processing methods [14, 17, 5, 32, 11, 30, 15]. Therefore, running our experiments on these datasets makes our results easier to compare to existing work. Description of each dataset and its specifications are explained below.

All four datasets consist of undirected labeled graphs, as defined in 1.3.1, that may contain cycles. Each dataset has a set of target graphs, also called a database, and a set of query graphs. Every graph and every vertex within every graph have a unique id.

Aids is the standard database of the Antiviral Screen dataset of the National Cancer Insitute. The database contains the topological structure of 40,000 chemical compounds, represented as graphs. The number of vertices varies from 4 to 245. The graphs in this dataset are small and barely connected 2.1. Pcms contains 200 contact maps that represent relationships among amino acids. The graphs here are small and dense. Pdbs consists of 600 target graphs, which represent proteins. It contains larger graphs, each of them contains between 1,683 and 7,979 vertices. The last dataset is ppigo and it has a database that consists of 20 protein interaction networks, where networks belong to species.

The specifications of each dataset is shown in Table 2.1. One can see that the pcms dataset is very dense, as the average vertex degree is 23.01. Pdbs and aids are very sparse with average vertex degree of about 2. The dataset with lowest number of edges per graph is aids (46.95) and the dataset with largest number of edges per graph on average is ppigo (4,942).

Table 2.2 shows the number and percent of 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%).

	aids	pcms	pdbs	ppigo
# graphs	40,000	200	600	20
# disconnected graphs	3,157	200	360	20
# unique labels	62	21	10	46
# vertices on average	45	377	2,939	4,942
# edges on average	46.95	4,340	3,064	4,942
average density	0.0475	0.0612	0.0007	0.0022
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: Number of SIP instances for each dataset and how many of them are SAT and UNSAT

2.3 Filter-Verification paradigm

This Section gives an overview of already existing algorithms based on the filter-verification paradigm, defined in Section 1.3. We first start by introducing more preliminary concepts and notations that complement the definitions in Section 1.3.

2.3.1 Filtering

This step consists of building an index of the database and then rejecting for verification with the query targets that can be proved not to contain the query. The process of rejecting/pruning uses the data stored in the index for the particular target and the characteristics of the query. The details follow below.

Database Index

An *index* of a database D, denoted as $\mathcal{I}_D$ is a collection of data, often stored in a file, that contains characteristics of the graphs in D, also known as *features*. Features can be represented by paths, subtrees, cycles, or subgraphs. They are computed for every graph in the database. The index is independent of the queries so that it can be computed once and then reused for multiple queries as long as there is no change in the database. The process of computing the features of the graph to store them in the index is called *feature extraction*.

Example 1 Let us have a database consisting of the graph G_t displayed on Figure 1.2. In this example, we want to compute the index of G_t using paths up to maximum length 2 as features and store each unique path as a string

containing the labels of the vertices in the path. Each vertex in G_t is labeled in yellow(Y), blue(B) or red(R).

An example feature extraction algorithm will derive the following paths: B, Y, R, B-B, B-Y, R-Y, B-R [1]. These paths will then be stored in the index of G_t and used further in the filtering process to check whether G_t could be rejected for verification with a given query.

Computing the database index is the process is very expensive in terms of running time and data storage. However, the cost of building such structure is compensated by the fact that it can be reused and that it is query independent.

Candidate Set

The candidate set of a database D and query graph G_p, denoted as $\mathcal{C}_p$ is a set of graphs that contain all features in G_p. $\mathcal{C}_p$ is derived using $\mathcal{I}_D$ and $\mathcal{F}_{G_p}$, where $\mathcal{F}_{G_p}$ contains all features in G_p, computed using the same algorithm that computed $\mathcal{I}_D$. Note that unlikely $\mathcal{I}_D$, $\mathcal{F}_{G_p}$ is not reused[2].

The purpose of filtering is to derive as small candidate set as possible in order to limit the number calls to a subgraph isomorphism algorithm during verification. Note that $\mathcal{C}_p \subseteq$ D. The worst case scenario is when $\mathcal{C}_p$ = D, which means that filtering did not manage to prune any target graphs in the database.

Example 2 Let us extend Example 1 by introducing a query, which is the graph G_p on Figure 1.1. Let us use the same characteristics as features as the ones used in Example 1. Then, $\mathcal{F}_{G_p}$ = B, Y, R, B-Y, R-Y and G_t. Therefore, $\mathcal{C}_{G_p} = \{G_t\}$, because all features in G_p are contained by G_t.

2.3.2 Verification

Verification is the process of applying a subgraph isomorphism problem (SIP) algorithm on every pair (G_p,t) of graphs, where p is the query and t is the target, t $\in \mathcal{C}_{G_p}$. The aim of verification is to return the number of graphs in D that contain G_p. All targets that do not contain the pattern, but were not rejected during the filtering step, are called *false-positives*. One wants to construct a filtering technique that admits as low number of false-positives as possible. The number of false-positives is often used as a measure of effectiveness of the filtering method[15].

Most of the subgraph query processing methods that apply the filter-verification paradigm are mainly focused on improving the filtering stage, while reusing the same algorithm, known as VF2, for verification [15]. It is claimed that VF2 is "state of the art"[14]. However, this is not the case ([21, 16, 6, 31, 18], Chapter 5). The subgraph isomorphism tests are reported as "too time consuming"[15] and this is explained by the nature of the complexity of the subgraph isomorphism problem[3][14, 15, 28]. One can see that the reason for the reported low verification performance for some instances could be due to the poor choice of a SIP algorithm. This is further investigated and discussed later in this work.

[1] Note that the representation of some of the paths is not unique, as G_t is undirected. For example, path B-Y is equivalent to path Y-B. In such cases, one can impose constraints on the ordering of the paths to avoid storing repetitive features in the index.

[2] It could be useful to store the query features if large proportion the queries received were repetitive. In such cases, one could incrementally build a second index of the features. We do not know whether this has been tried so far. This can be an area of further investigation in the future.

[3] This problem is proved is NP-Complete by Stephen A. Cook in 1971 [8]

2.3.3 Existing Filtering Techniques

This Section discusses some of the existing filtering algorithms with respect to feature extraction and choice of features.

There are two types of feature extraction techniques, known as graph mining and exhaustive feature enumeration. *Graph mining* techniques ([28, 7, 13, 26, 30, 32, 29]) store the frequent features with high enough discriminative ratio in the database. A feature is *frequent* if its support ratio is higher or equal to a certain algorithm-specific threshold value. The support ratio of a feature is defined as the percentage of graphs in the database containing it. The *discriminative ratio* of a feature is a metric that characterizes the filtering power of the particular feature compared to other features. There is no universal formula to calculate the discriminative ratio of a feature, every technique employs a different calculation method.

Exhaustive feature enumeration techniques ([5, 14, 24, 17]) store in the index all features of every graph in the database, regardless of their importance.

Choosing a feature extraction method often depends on the type of datasets one has to work with. For instance, graph mining techniques are inefficient when the data in the database is frequently being changed. When frequently inserting and deleting graphs in the database, the discriminative ratio of the indexes features may change so that it makes the index outdated. Consequently, the index becomes less reusable and thus the total execution time of subgraph query processing is highly increased. Moreover, graph-mining techniques take longer time to built the database index [15].

One of the advantages of graph mining techniques is that they require much less storage space than exhaustive feature enumeration algorithms, as one does not need to store all features of all graphs. This lowers the storage space requirements and makes the process of constructing the candidate set faster, as less number of target features have to be compared against the query features.

There are various types of structures that can be used as features. Paths ([5, 11]), trees ([13, 29, 14]), subgraphs ([7, 24, 26, 30, 32]) of the targets or all of them combined ([14, 32]) can be extracted from the database graphs and stored in the index. Features can be derived based on the neighbours of each graph vertex [17]. The choice of features influences the filtering performance and it is often a trade off in terms of time and filtering strength. This is further investigated in Section 2.4.3, where we report on performance results obtained from an indexing method that uses a combination of paths, trees and graph cycles as features. A detailed review of this method is presented in the next Section.

2.4 CT-Index

CT-Index is an existing subgraph query processing approach that employs the filter-verification paradigm[14]. This method supports undirected graphs with edge and vertex labels and also wild card patterns. Although not explicitly stated in [14], CT-Index addresses the non-induced subgraph isomorphism problem defined in Section 1.3. In this Section, we introduce and discuss the filtering algorithm and analyze its complexity in Section 2.4.1. Section 2.4.2 explains the verification algorithm. Also presented is a complexity analysis of the algorithms used by CT-Index and an empirical study of its performance (using an open-source Java implementation) in Section 2.4.3.

2.4.1 Filtering

During the filtering step, the features of all graphs in the target data set are extracted and saved to a file, i.e. the target index ($\mathcal{I}_T$). $\mathcal{I}_T$ is used to filter out target graphs (G_t) that cannot contain the pattern (G_p). Features are specific subgraphs used to classify graphs, and are stored as hash-key fingerprints. Features may be paths, subtrees or cycles of bounded length. Since vertices and edges may contain labels, these features can be viewed as strings from a specified alphabet (where the alphabet is the labels). In [14] 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 graph G with e_{max} number of edges, 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 $e_{uv} \in G$ with vertices $u \in G$ and $v \in G$, such that one of them (say, *u*) is part of the current tree and the other (say, *v*) is not. If we have *n* number of vertices in the current tree, each with degree *d*, then 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 $e_{uv} \in F$, until the tree reaches size e_{max}.

In the start of the tree extraction 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 initially is 2.(d-1). After every recursive call, one more vertex is added to the tree, which introduces (d-1) new edges. That makes a total of e_{max} + 1 vertices that will be added to the tree and (e_{max} + 1)(d - 1) visited edges. Consequently, the complexity of extracting tree features is $\mathcal{O}(|e|(e_{max}+1)(d-1))$. From this formula one can see that the number of edges in the graph has significant impact on the performance of the algorithm. When increasing the graph density, the algorithm will have slower performance, caused by the degree of each vertex and the total number of edges, which both will increase.

CT-Index computes a unique representation of each distinct feature, its *canonical form*, and stores its string encoding in $\mathcal{I}_T$. 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 are 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 e_{pu} and e_{pv}, 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 [14] 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 trees, 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 [14] it is claimed that step (2) is not linear time but 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. Therefore one could expect poor performance for data sets with less label diversity and with big trees as features. More specifically, as e_{max} increases, or average degree increases, so too does the cost of step (2), and performance suffers (we conduct experiments to test this hypothesis in Section 2.4.3).

Fingerprints

CT-Index uses a storage technique called *hash-key fingerprint* to capture the features in the graph. A separate fingerprint is computed from the canonical labels for each graph in the database. A fingerprint is an array of bits and denotes whether a particular feature occurs in the graph or not. 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[4]. 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 features may map to the same bitset position, introducing false-positives. The [14] 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.

2.4.2 Verification

The verification step checks all candidates computed in the filtering step via a subgraph isomorphism test. A backtracking algorithm [3], similar to VF2 [9] 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 [5], gCode [17] and Tree+Δ [32]. Most papers claim that VF2 is "state of the art". However, this is not the case ([21, 16, 6, 31, 18]. VF2 has been shown to perform erratically and poorly [18]. 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.

2.4.3 Performance

There is an existing work, where several well-established subgraph query processing techniques as well as CT-Index are evaluated [15]. The experiments are conducted on the four datasets described in Section 2.2 using the default input parameters for each of the compared techniques. CT-Index requires five integers as an input, specified in the following order:

[4]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.

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.
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 > [14, 15]. No information on why exactly these parameters should be used is given. For each of the four datasets, the indexing algorithm is run, putting a time limit of 8 hours[5]. The obtained results are presented in four Diagrams, showing filtering time, index size, verification time and false positive ratio (FP ratio). The FP ratio is calculated using formula (2.1) and its purpose is to indicate how many of the UNSAT instances are filtered without using a SIP algorithm. $|A\{p\}|$ denotes the number of SAT instances for a pattern p ∈ P and $|\mathcal{C}\{p\}|$ is the cardinality of the candidate set for p.

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

One can notice that when the value of $|A\{p\}|$ becomes closer to the value of $|\mathcal{C}\{p\}|$ (filtering has managed to discard most of the UNSAT instances), the FP ratio approaches 0. Whenever there is a big difference between $|\mathcal{C}\{p\}|$ and $|A\{p\}|$ (filtering did not discard many UNSAT instances, the size of the candidate set is close to the size of the database) the formula approaches 1. We can deduce that the closer to 1 this formula is, the less successful filtering was. Similarly, when the formula approaches 0, this indicates high filtering performance.

The results obtained by [15] using these parameters for each dataset are as follows.

- For datasets “pcm” and “ppigo”, CT-Index never manages to complete execution of the filtering step. Therefore, no verification is done.
- For the “aids” dataset, CT-Index performs best. Filtering takes about 80 seconds and verification: about 0.7 seconds. The FP ratio is 0.8.
- CT-Index takes about 9,000 seconds for filtering graphs in “pdbs” and about 1 second to execute the verification step. The FP ratio is 0.2.

We decided to conduct experiments to investigate in more depth the performance of the algorithms implemented in CT-Index depending on the input parameters. We used an open source java implementation of CT-Index written by its authors and we ran the software with the “aids” dataset using alternative to the default set of parameters. For our experiments, we chose a fixed size of parameter set[6] the purpose of which is to identify how the change of one parameter changes the performance of the algorithms implemented in CT-Index. As in [15], we measured the filtering and verification time, the FP ratio using formula (2.1) and calculated the total time, that is the sum of the filtering and the verification time. For the size of the fingerprints (2^n for some integer n), we varied n from 0 to 16. 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 (G_p, G_t) ∈ Px. The values of path, subtree and cycle maximum length are a permutation of combination of -1 and 5. Here, -1 tells us how much the effective rate of filtering drops when a feature is switched off, and 5 shows the performance of filtering when the feature is on (i.e -1 and 5 play the role of binary to indicate for each feature that it is either off or on). For every n, we executed the algorithm with all permutations of every combination of assignment of -1 and 5.

[5]When the algorithm did not finish it execution within the time limit, the authors waited for more than 24 hours, but to no avail.
[6]This was done due to the large number of possible input values.

Row #	Input	Filtering T[sec]	Verification T[sec]	Total T[sec]	FP Ratio
1	1 5 5 5	108.934	67.395	176.329	0.913
2	64 5 5 5	110.797	59.622	170.419	0.85
3	128 5 5 5	110.9	31.711	142.611	0.87
4	256 5 5 5	104.586	22.373	126.959	0.81
5	512 5 5 5	104.883	17.008	121.891	0.69
6	1024 5 5 5	112.132	15.477	127.609	0.64
7	2048 5 5 5	107.024	14.693	121.717	0.63
8	4096 5 5 5	106.136	13.006	119.142	0.60
9	8192 5 5 5	107.625	12.337	119.962	0.62
10	16384 5 5 5	107.991	14.083	122.074	0.61
11	4096 5 5 -1	105.441	12.926	118.367	0.62
12	4096 5 -1 5	39.743	32.068	71.811	0.88
13	4096 -1 5 -1	81.826	12.785	94.611	0.62
14	4096 -1 -1 5	6.234	61.346	67.580	0.913
15	4096 -1 5 5	82.376	5.896	88.272	0.62
16	4096 5 -1 -1	37.621	7.929	45.550	0.88

Table 2.3: CT-Index: Running time and results depending on the specified parameters

Table 2.3 shows a selection of our results that aids to illustrate best the change of performance depending on the input parameters. The first 10 rows show the changes in running time and FP-ratio depending on the size of the fingerprints. The values of all parameters are fixed. As expected, the FP ratio goes down with the increase of the fingerprint size. We also notice significant decrease in the verification running time when we allow bigger bitset size. The reason for these results is the fact that the number of collisions when hashing the features to places in the bitset is lower when the bitset has bigger size. Then, more targets can be rejected during filtering and less number of SIP tests have to be computed during verification.

The last 7 rows of Table 2.3 show the performance of CT-Index depending on the other 3 parameters. The input with the worst verification run time is in row 14. No target was rejected during the filtering stage for each of the patterns, as indicated by the high FP ratio value. This data shows that extracting only cycles as features is not effective in pruning 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 don‘t’ have many cycles.

Comparing the results from rows 8 and 15, one can see that extracting paths and subtrees as features for this dataset gives worse performance in both filtering and verification steps. The reason for this could be that subtrees and cycles are descriptive enough to be used as the only types of features.

Using only paths as features is the option that shows best performance (row 16). Subtree extraction for this dataset takes about 60 seconds more, it gives better filtering ratio, however slightly worse verification run time(rows 16 and 11). From this we can see that the algorithm for subtree extraction significantly lowers the filtering run time, as derived during the empirical analysis in 2.4.1. 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.

We tried running similar experiments with the other datasets discussed in 2.2. However, as also observed in [15], most of the experiments finished computation within matter of days.

2.5 Subgraph Isomorphism Algorithms

This Section reports on existing work in the area of the subgraph isomorphism problem (SIP).

- subgraph isomorphism with constraints programming

2.5.1 Partick and Ciaran's paper

2.5.2 Christine's paper

Chapter 3

Framework for graph indexing and filtering

This Chapter describes a subgraph-query filtering and indexing framework designed and implemented in Java. The framework implements the first stage of the filtering-verification paradigm. It takes a set of target and a set of pattern graphs, computes their indices and generates a candidate set for SIP. There are two path based exhaustive feature enumeration techniques that can be used to generate the index of a set of graphs: Path Index and Path-Subtree Index. Their design, implementation and empirical analysis is presented in Section 3.3.

The framework is composed of the two indexing algorithms described previously (PI and PSI) and the filtering method. Subgraph query filtering can be performed using either PI or PSI. To run the framework, the user specifies the filename of the target graphs, the filename of the pattern graphs, l_{max} and a bit which if 1, the framework will index the graphs using PSI, or otherwise use PI. More detailed instructions on how to run the framework can be found in the code repository on github [4].

3.1 Graph representation

Graphs are represented with a Java class *Graph* that has an integer id and a collection of objects of type *Vertex* as fields. A Vertex has an id, a label and a list of edges that connect it to its neighbours. The number of edges a Vertex has equal to its degree. An Edge object has a label and a destination Vertex as fields. As the graphs we are working with are not directed, two Edge objects are created to represent an edge. The first object has one of the vertices as destination Vertex and then added in the list of edges of the other Vertex: the source Vertex. The second edge object will have the source Vertex of the first edge object as destination Vertex and included in the list of edges of the destination Vertex of the first edge object, which will be its source Vertex. The label of both edge objects will be the same.

3.2 Paths Extraction

A path is a sequence of vertices, such that every pair of consecutive vertices in the sequence are neighbours. Path extraction is referred to the process of generating all paths up to a given length l_{max} in a graph G. For every vertex *v* in G, we execute a recursive depth-first search algorithm with a bound on the size of the stack of maximum number of vertices l_{max}. 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 the p-feature and store it in the index.

Algorithm 1 describes our approach. Before generatePath procedure is called, an empty stack *s* is initialized

(line 3). The stack is used for storing the sequence of vertices generated during depth-first search procedure. Procedure generatePath (Algorithm 1) takes all vertices in the target graph and l_{max} as parameters. It calls procedure dfsBounded for every vertex in the list as starting vertex (line 5). The start vertex is also pushed on the stack (line 4), as it is part of the path that is to be generated.

Algorithm 2 performs a depth-first search with bound path length. Given a starting vertex *v*, l_{max} and stack s, Algorithm 2 iterates through all neighbours of *v*, adding each neighbour *n* to the stack (line 10) and calling dfsBounded with *n* as a starting vertex (line 11). At each call of dfsBounded, the current sequence of vertices in the stack is passed to procedure computeFeature (Algorithm 3) until all vertices reachable by the start vertex within distance l_{max} are visited.

Algorithm 1 Paths extraction

```
1:  procedure GENERATEPATH (vertices, l_max)
2:      for v in vertices do
3:          s ← init                                   ▷ initialize the stack
4:          push v on top of s
5:          DFSBOUNDED(v, l_max, s)
6:      end for
7:  end procedure
```

Algorithm 2 Depth First Search of bound length

```
1:  procedure DFSBOUNDED (v, l_max, s)
2:      if s ≤ l_max then
3:          newPath ← s                                ▷ new path of size up to l_max is found
4:          GETPATH(newPath)
5:      end if
6:      if s == l_max then
7:          s.pop()
8:      end if
9:      for neighbour of v do
10:         if neighbour not in s then
11:             s.push(neighbour)
12:             DFSBOUNDED(neighbour, l_max, s)        ▷ recursive call
13:         end if
14:     end for
15:     if s is full then                              ▷ all neighbours of node are on the stack
16:         s.pop()
17:     end if
18: end procedure
```

The complexity of dfsBounded is derived as follows. Algorithm 3 iterates through each vertex in the graph, calling the dfsBounded procedure. The number of calls to dfsBounded from this Algorithm is equal to the order of the graph, say *n*. Let us denote the degree of a vertex as *d*. Then, dfsBounded will generate $d^{l_{max}-1}$ paths. This will result to $d + d^2 + \ldots d^{l_{max}-1}$ calls to dfsBounded in total. Therefore, the overall cost of extracting all paths in a graph of maximum length l_{max} is $\mathcal{O}(n(d + d^2 + \ldots d^{l_{max}-1}))$.

3.3 Indexing and candidates extraction algorithms

The rest of the subgraph query filtering process is carried out by using two algorithms: Path Index (PI) and Path-Subtree Index (PSI). PSI and PI employ different feature representation and candidates extraction algorithms,

which are explained below.

3.3.1 Path Index

This Section presents PI: a technique to compute features of a graph commonly used in many filtering algorithms [5]. Below we describe the features calculation and candidate extraction algorithms employed in PI.

Features

PI computes the features stored in the index using the paths extracted during the path extraction procedure outlined in Section 3.2. To store a path in $\mathcal{I}_D$, we compute its unique string representation, referred to as *p-feature*. The procedure to derive p-features and store them in $\mathcal{I}_D$ is the following.

1. Given a sequence of vertices v_{seq} (i.e. a path), replace each vertex in v_{seq} with its label to obtain its *p-feature*.
2. Reverse the sequence v_{seq} to obtain v'_{seq}.
3. Calculate *p-feature'*, that is the unique string representation of v'_{seq}.
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*).

The complexity of each of the four steps of the p-feature computation and storage is the following. Step 1 involves iterating through the sequence of vertices, that is at most l_{max} elements. Accessing the label of each vertex is a constant time operation, therefore the complexity of Step 1 is $\mathcal{O}(l_{max})$. The complexity of Step 2 and Step 3 is $\mathcal{O}(l_{max})$. Step 4 has $\mathcal{O}(p)$ worst case complexity, where p is the current size of the index of the current graph being indexed. Note that as the index grows larger and larger, the cost of adding new p-features increases.

Example 1 Figure 3.1 represents a graph G with 6 vertices, each with an id: a number from 1 to 6, and a label: either yellow(Y) or blue(B). The sequence of vertices <1, 5, 6> is a valid path, because every pair of consecutive vertices in the sequence are neighbours in G. Executing Step 1 of the procedure explained above results in obtaining the p-feature "Y-B-B". After computing Step 2 and 3, we derive the p-feature of the reversed sequence equal to "B-B-Y". We store the string "B-B-Y" in $\mathcal{I}_D$, because it is lexicographically smaller than "Y-B-B".

The path extraction algorithm will encounter both v'_{seq} and v_{seq} throughout its execution, but only one p-feature to represent both of them will be stored in the index. Storing the p-feature of both of them would be redundant, because the graphs we work with are undirected. Therefore, choosing to store only one of them (in this case the lexicographically smaller one) limits the size of the index by half.

Theorem 3.3.1. *Let us have two paths: v_{seq} and v'_{seq} in an undirected graph G, such that v'_{seq} is the reverse of v_{seq}. We can represent v_{seq} and v'_{seq} using one p-feature without loosing structural information about G.*

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

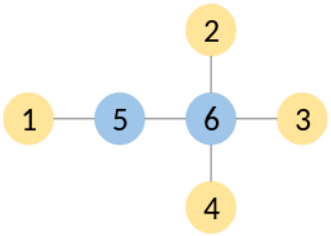


Figure 3.1: Graph G

Implementation

Algorithm 3 describes the implementation of Steps 1-4 described above. Given a sequence of vertices *vseq* 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 (vseq)
2:      pfeature ← vseq.toString()        ▷ returns a string of the labels of the nodes in vseq
3:      rseq ← vseq.reverse()             ▷ reverse the order of nodes in vseq
4:      pfeature' ← rseq.toString()       ▷ returns a string of the labels of the nodes in rseq
5:      if pfeature' < pfeature then
6:          pfeature ← pfeature'          ▷ put to index the lexicographically smaller string
7:      end if
8:      if pfeature not in I_D then
9:          add pfeature to I_D
10:     end if
11: end procedure
```

Candidates Extraction

The process of computing the set of candidates for subgraph isomorphism test with the pattern is referred to as candidate extraction. Algorithm 4 shows the 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 database (line 3) and for each target checks whether it contains all p-features of the pattern (lines 5, 6, 7). Let the number of targets be n, the size of the pattern be p_i and the average size of a target graph index be t_i. Then the time complexity is equal to $\mathcal{O}(n.p_i.t_i)$.

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 [21]. 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_{G_p}, I_{G_t})
2:     candidates ← new ArrayList<>()
3:     for Graph t : targets do
4:         flag ← true
5:         for pfeature : I_{G_p} do
6:             if not CONTAINS ( t.index, pfeature) then ▷ check whether pfeature is contained in the index of t
7:                 flag ← false; break
8:             end if
9:         end for
10:        if flag then candidates.add(t.id)
11:        end if
12:    end for
13:    return candidates
14: end procedure
```

Features

The difference between PSI and PI is the approach of computing the unique string representation (p-feature) of paths. The difference lies in the labeling employed by PSI: it is based on the neighbourhood 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 the graph using the labels of the vertices in its neighbourhood. 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 3.1 represents a graph with 6 vertices with ids from 1 to 6. Each vertex is labeled either in yellow (Y) or in blue (B), as shown in the Figure. The n-label of vertex 6 is "BBYYY", as it is its label is B, it has 4 neighbours, 3 of them with label Y and one with label B. Similarly, the n-label of vertex 5 is "BBY", and the n-label of vertices 1, 2, 3 and 4 is equal to "YB".

1. Given a sequence of vertices v_{seq}, replace each vertex in v_{seq} with its *neighbourhood label (n-label)* to obtain the p-feature of v_{seq}.
2. Reverse the sequence v_{seq} to obtain v'_{seq}.
3. Calculate *p-feature'*, that is the unique string representation of v'_{seq}.
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*).

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.

One can see that the difference between the features representation and computation between PI and PSI lies only in Step 1. The algorithm to implement the features representation and storage 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.

Algorithm 5 Set n-label procedure

```
1: procedure SETNLABEL
2:     nlabel ← ""
3:     for Edge e : this.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 $\mathcal{I}_{G_p}$.

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 *Tindex* that is identical to a given pattern feature *pf*, we check whether the following two conditions are met:

1. The label of a vertex in ith position in the target path is equal to the label of a vertex in ith position in the pattern path.
2. The nlabel of a vertex in ith position in the target path contains the nlabel of a vertex in ith position in the pattern path.

The first condition is equivalent to the filtering procedure employed by PI: we check for compatibility using only the 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 vertex in the target.

Algorithm 6 illustrates the implementation of the contains procedure. For every path in the target index, it calls Algorithm 7 to check that the two conditions discussed above 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 containsLabel takes two nlabels as arguments and returns true if the second nlabel is contained in the first nlabel or false otherwise.

Algorithm 6 Contains procedure

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

Algorithm 7 containsFeature procedure

```
1: procedure CONTAINSFEATURE(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.size()) do   ▷ for i^th nlabel in tarf, check that it contains the i^th nlabel in patPath
8:         if not CONTAINSLABEL(tarPath.get(i), patPath.get(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 first filtering condition implemented by Algorithm 7 puts an upper bound on the size of $\mathcal{C}_{PSI}$ to be at most $|\mathcal{C}_{PI}|$ and condition 2 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 Empirical analysis and suggestions for improvement

In this Section we discuss the performance of the framework using PI and PSI. Both of them were ran with the datasets described in Section 2.2 and their filtering strength and execution time was compared with CT-Index [14], analyzed in Section 2.4. CT-Index results are also used as a benchmark for correctness of the implemented algorithms. 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 after running the framework using PI as $\mathcal{I}_{PI}$ and $\mathcal{C}_{PI}$ respectively, and the index and the candidate set obtained after running the framework using PSI as $\mathcal{I}_{PSI}$ and $\mathcal{C}_{PSI}$.

PSI requires significantly more storage space than PI. 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 three times bigger than the size of $\mathcal{I}_{PI}$.

Theorem 3.3.2 states that the filtering performance of PSI is not worse than the filtering performance of PI. In practice, the size of $\mathcal{C}_{PSI}$ is rarely close to the size of $\mathcal{C}_{PI}$. In particular, when running the framework with the AIDS dataset (Section 2.2) with maximum path length of 2, 3, 4 and 5, the size of $\mathcal{C}_{PSI}$ was several times smaller than the size of $\mathcal{C}_{PI}$. Moreover, setting the maximum path length bound to 5 removes 80% of the unsatisfiable instances on average. This filtering result is not only better than PI, but it also outperforms CT-Index.

Both PI and PSI are much slower than CT-Index. In particular, increasing the maximum path length bound significantly increases the algorithms computation time. This stems from the fact that maximum path length makes great impact on the complexity of the path extraction algorithm and on the size of the index. PSI is slower than PI both during index computation and candidates extraction time. Looking at the difference in their time complexities, this result is not surprising.

The filtering power of PSI and PI does increase linearly when increasing the maximum path length bound. There exists a bound *m* on the maximum path length after which there is almost no filtering gain, but only significantly increased computation and storage overhead. The value of *m* is usually smaller for PSI than for PI. For instance, when PSI is ran with instances from the AIDS dataset, *m* is equal to 5. This is the peak when the algorithm performs best. Any value larger than 5 results in much worse computation speed and almost unchanged filtering performance.

There are several ways in which the framework can be made more efficient. In the framework, the index of a database is 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 [23] 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 [23, 19, 22] 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 and it is based on a modified version of the filter-verification paradigm. Here, filtering and verification steps are applied separately for each SIP instance. This approach uses simple filtering tests that require much less computational effort (i.e. lighter filtering). More importance is placed on the quality of the SIP algorithm than on the filters, because the major computational effort goes for solving the non-filtered SIP instances.

Algorithm 8 is a pseudocode of our method. Given a file with targets T and a file with patterns P, we first read in each graph (lines 2, 3). Filtering is performed for every (G_p, G_t) pair and if the instance is not rejected, a call to SIP algorithm is made (line 7). The filter step consists of 5 naive tests, performed before the call to SIP1. If the conditions of any of the tests are not met, search does not proceed and we consider this to be a *trivial fail*.

Each step of the Light Filters algorithm is explained thoroughly. First, we introduce the theory behind the trivial fails and their implementation in Section 4.1. We then introduce the SIP algorithm called SIP1 and discuss its implementation in Section 4.2. Evaluation of the light filters approach is described in Chapter 5.

Algorithm 8 Light filters algorithm

```
1:  procedure PROCESS (File p, File t)          ▷ file with patterns and file with targets
2:      T_list ← read in all targets from file, initialize objects
3:      P_list ← read in all patterns from file, initialize objects
4:      for G_p in P_list do
5:          for G_t in T_list do
6:              if !FILTER (G_p, G_t) then      ▷ If the instance is not rejected during filtering, perform verification
7:                  SIP1(G_p, G_t)
8:              end if
9:          end for
10:     end for
11: end procedure
```

4.1 Trivial Failures

This Section introduces the five trivial failure tests, implemented as part of the filtering stage of SIP1.

4.1.1 Neighborhood Degree Sequence

The neighbourhood degree sequence of G and the degree sequence of labels in G correspond to four of the trivial tests. Very similar approach of using the degree sequence of vertices to reject UNSAT SIP instances is used in the filtering part of the algorithm presented in [21].

Definition 1 (Label Degree Sequence)**.** *The label degree sequence (lds) of $l \in L_G$, denoted as lds(G, l) is the non-increasingly ordered list of the degrees of all vertices in V_G that have l assigned as their label.*

Definition 2 (Neighborhood Degree Sequence)**.** *The neighbourhood degree sequence (nds) of G, written as nds(G), is the list of tuples (l, nds(l)) for every $l \in L_G$.*

Example 1 Let us have a graph G with two labels: A and B, where lds(A) = {5, 4, 3, 2, 2} and lds(B) = {4, 3, 3, 1}. This shows us that the order of G is 9, where 4 vertices with degrees 4, 3, 3 and 1 are labeled as B and 5 vertices with degrees 5, 4, 3, 2 and 2 are labeled as A. Using lds, we derive that nds(G) = {{A, {5, 4, 3, 2, 2}}, {B, {4, 3, 3, 1}}}.

Definition 3 (G_t subsumes G_p)**.** *We say that G_t subsumes G_p if for each label l both in G_p and G_t, the length of the lds(l$\in G_p$) is smaller or equal to the length of lds(l$\in G_t$), and the degree on i^{th} position in lds(l$\in G_p$) is less than or equal to lds(l$\in G_t$) for each i between 0 and $|lds(l \in G_p)|$.*

The notion of nds lets us define several simple tests for incompatibility between pattern (G_p) and target (G_t), based on checking whether nds(G_p) is a subset of nds(G_t). nds(G_p) $\subseteq$ nds(G_t) if all following conditions are true:

1. the order of G_p is less than or equal to the order of G_t
2. the number of unique labels in G_p is less than or equal to the number of unique labels in G_t
3. every label in G_p is also in G_t
4. G_t subsumes G_p

Example 2 Let us have nds(G_p) = {{A, {3, 2, 1}}, {B, {4}}} and nds(G_t) = {{A, {5, 4, 3, 2, 2}}, {B, {4, 3, 3, 1}}, {C, {5, 4, 2, 1}}}.

In this example, nds(G_p) is a subset of nds(G_t), because each of the four conditions is true. In particular:

1. The order of G_p (4) is less than the order of G_t (13).
2. The number of unique labels in G_p = 2 and the number of unique labels in G_t is 3. 3 is clearly bigger than 2.
3. G_p has labels A and B which are both in G_t.
4. The lds of every label in G_p is clearly contained in the lds of the same label in G_t.

Theorem 4.1.1. *If nds(G_p) $\nsubseteq$ nds(G_t), then SIP(G_p, G_t) is UNSAT.*

Proof. Suppose that SIP (G_p, G_t) is SAT and that nds(G_p) $\nsubseteq$ nds(G_t). Then, at least one of the conditions for nds(G_p) $\subseteq$ nds(G_t) is not true.

1. Suppose that the first condition is false. Then, G_p must have at least one vertex more than G_t, i.e. at least one vertex $\in G_p$ will be unmatched. Therefore, SIP (G_p, G_t) is UNSAT if the order of G_p is bigger than the order of G_t and we reach a contradiction. Therefore, this condition must always hold.

2. Suppose that the second condition is false so that there exists a label $l' \in L_{G_p}$ with $l' \notin L_{G_t}$. Therefore, there exists a vertex $v \in G_p$ that is assigned the label l' and there is no vertex in G_t with label l'. From this follows that v can not be matched to any vertex in G_t and SIP(G_p, G_t) is UNSAT, which leads to a contradiction. Therefore, condition 2 must hold.

3. Suppose that the third condition is false. Then, there exists a label $l' \in L_{G_p}$ and $l' \notin L_{G_t}$. As proved before, this is a contradiction, therefore condition 3 must hold.

4. Suppose that previous three conditions hold and G_t does not subsume G_p.

   First, suppose that for each label l both in G_p and G_t, the length of the lds(l$\in G_p$) is smaller or equal to the length of lds(l$\in G_t$). Then, there exists a degree value $deg_{pi} \in$ lds(l$\in G_p$) at position i in the list for i between 1 and $|lds(l \in G_p)|$ such that $deg_{pi} > deg_{ti}$, where $deg_{ti} \in$ lds(l$\in G_t$). Then, there is a vertex $v \in G_p$ with label l and degree deg_{pi} that will not be matched to any vertex in G_t. Therefore SIP (G_p, G_t) is UNSAT which is a contradiction, therefore the degree of the i^{th} element in lds of each label in G_p has to be smaller or equal to the degree of the i^{th} element for the corresponding label in G_t for every i between 0 and the length of lds of every label in G_p.

   Second, suppose that there exists a label l both in G_p and G_t such that the length of the lds(l$\in G_p$) is bigger than the length of lds(l$\in G_t$). Then, the number of vertices with label l in G_p is bigger than the number of vertices with label l in G_t and it is impossible to match each vertex in G_p to a different vertex in G_t which leads to contradiction.

   Therefore both conditions of subsumption must hold and G_t subsumes G_p. This is a contradiction, therefore condition 4 must be true.

All four conditions must hold. It follows that nds(G_p) $\subseteq$ nds(G_t) which is a contradiction. Therefore, if nds(G_p) $\subseteq$ nds(G_t), then SIP(G_p, G_t) is UNSAT. □

The four conditions for nds(G_p) $\subseteq$ nds(G_t) are the first four of the trivial failures added to SIP1, also displayed on Table 4.1. Their implementation is discussed in Section 4.1.4.

4.1.2 Domain wipe out

This is the fifth of the trivial failures implemented as light filtering on top of the search. Every pattern vertex v has a bitset domain dom_v. The size of dom_v is equal to the order of the target, where every bit corresponds to a vertex in the target. For every vertex $w \in G_t$, if L(w) is equal to L(v) and the degree of v is smaller or equal to the degree of w, we set the bit for w in dom_v to true, i.e. v can be mapped to w. Whenever no bit in dom_v is set to 1, it means that no target vertex exists that can be mapped to v i.e. no valid mapping from all vertices in G_p to a different vertex in G_t can exist and SIP (G_p, G_t) is UNSAT. The algorithm returns false without proceeding to search.

Table 4.1 shows the trivial failures discussed. They are executed as tests in the same hierarchy as shown in the Table. The first four failures are namely the conditions for nds(G_p) $\subseteq$ nds(G_t). If either of these tests fails, then SIP(G_p, G_t) is UNSAT (Theorem 4.1.1).

Trivial Fail	Meaning
1	G_t.order $\geq$ G_p.order
2	G_t unique labels $\geq$ G_p unique labels
3	G_p labels $\subseteq$ G_t labels
4	G_t subsumes G_p
5	dom_v, $v \in G_p$ not empty

Table 4.1: Specification of the measured failure types

4.1.3 Order of Tests

Tests follow a strict hierarchy, where test 1 performs the cheapest and most trivial task and test 5 is the most expensive and its filtering is based on the least trivial test. The cost of each test in terms of complexity is discussed in Section 4.1.4.

4.1.4 Implementation

Algorithm 9 describes the implementation of the 5 trivial failures from Table 4.1 as part of the filtering stage. Procedure "filter" is executed for every SIP instance before the verification stage. If any of the if statements (lines 2, 4, 6, 8 and 17) is false, the procedure returns false and the verification for the corresponding instance is not executed. Otherwise, if all 5 tests are true, the procedure makes a call to SIP1, i.e proceeds to the verification step. Pseudocode for SIP1 is shown in Algorithm $saywhere$ and discussed in the next Section.

Failures 1 and 2 are the fastest: each of them takes time $\mathcal{O}(1)$ to compute. For failure 1, one needs only to return the sizes of the number of vertices in G_p and in G_t. 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 graph is read from the file. To check whether test 2 is true, one needs to compare the size of the labels array d_p of G_p with the size of the labels array d_t of G_t. Finding the values of d_t and d_p takes time $\mathcal{O}(1)$.

Failures 3 and 4 have slower running time. For every label l $\in$ G_p, failure 3 occurs if l is not a label in G_t. Checking whether l is a label in G_t involves iterating over the labels in G_t, which is of time $\mathcal{O}(d_t)$ worst case, if l is the last label in G_t or it is not present. Therefore, the overall complexity of failure 3 is $\mathcal{O}(d_p.d_t)$. Algorithm 10 shows the pseudo code for the "subsumes" procedure. Its complexity is $\mathcal{O}(plds)$, where plds is the lds of label *l* $\in$ G_p. Therefore, the overall complexity of failure 4 is $\mathcal{O}(plds.d_p)$.

Failure 5 is the most expensive one. It takes time $\mathcal{O}(m.n)$, where n is the order of G_p and m is the order of G_t. Algorithm 9 has to visit each vertex $v \in G_p$, initialize dom_v and for every $w \in G_t$, check whether *w* can be mapped to *v*. The two checks (line 13) take time $\mathcal{O}(1)$.

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

- Class Graph
  Creates graph (G) from a file. A Graph object has size, order, id, array of the degree of each vertex, bitset array of the neighbours of each vertex, where the i^{th} element in the array contains the neighbours of vertex i in the graph and array of labels, where similarly, the element in position i contains the label of vertex i.

Algorithm 9 Lights Filters

```
1: procedure FILTER (G_p, G_t)
2:     if not G_t.order ≥ G_p.order then return false                ▷ trivial failure 1
3:     end if
4:     if not G_t unique labels ≥ G_p unique labels then return false   ▷ trivial failure 2
5:     end if
6:     for l in L_{G_p} do
7:         if not l in L_{G_t} then return false                     ▷ trivial failure 3
8:         end if
9:         if not SUBSUMES (G_p, G_t, l) then return false           ▷ trivial failure 4
10:        end if
11:    end for
12:    alldoms ← initialize   ▷ An array of size the order of G_p that contains the domain of each vertex in G_p
13:    for every v ∈ G_p do
14:        dom_v = new BitSet(G_t.order)   ▷ initialize dom_v to bitset of size the order of the target
15:        for every w ∈ G_t do
16:            if v.label == w.label and v.degree ≤ w.degree then
17:                set dom_v[w] to 1
18:            end if
19:        end for
20: alldoms[v] ← dom_v
21:        if empty dom_v then return false                          ▷ trivial failure 5
22:    end for
23: SIP1(alldoms)                          ▷ if no failures occurred, call SIP1 algorithm
24: end procedure
```

Algorithm 10 G_t Subsumes G_p

```
1: procedure SUBSUMES (G_p, G_t, l)
2:     plds ← lds(l) in G_p                    ▷ the label degree sequence of l in G_p
3:     tlds ← lds(l) in G_t                    ▷ the label degree sequence of l in G_t
4:     for i between 0 and pSeq.length - 1 do
5:         if plds[i] > tlds[i] then return false   ▷ exists vertex in G_p that can't be matched to any vertex G_t
6:         end if
7:     end for
8: return true
9: end procedure
```

While reading in the graph, we initialize each field, which takes time $\mathcal{O}(size + order)$. When the array of labels is built, a new object for each unique label *l* is created and the nds of the graph is computed.

- Class Label
  This class represents a label in a Graph object. A Label has a name and lds which is an array of integers, sorted in non-increasing order. The degree sequence array is built using insertion sort algorithm which is of complexity $\mathcal{O}(n^2)$, where n is the size of lds [10].

- 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.

4.2 SIP1 Implementation

SIP1 is based on the simplest of the Glasgow algorithms [18]. Given a G_p and G_t, SIP1 has a variable for each vertex in G_p, each with domain that is the set of compatible vertices in G_t (alldoms, Algorithm 11, initialized in Algorithm 9). Compatible vertices have the same labels and the degree of the target vertex is greater than or equal to the degree of the pattern vertex. Bit sets are used to represent the domains and the adjacency matrices of the graphs. When a pattern variable u is instantiated with a target value i (Algorithm 11, line 7), all uninstatiated (future) variables have i removed from their domains (Algorithm 11, line 12). If a future variable v is adjacent to u in G_p then the domain of v becomes the intersection of the current domain of v with the neighborhood of vertex i in G_t. This constraint is enforced by applying a logical and operation between the two bit sets (Algorithm 11, line 14). SIP1 uses forward checking (FC) with fail first heuristic [12]: for all uninstantiated variables representing pattern vertices, it selects to explore the one that has the smallest domain before the others (Algorithm 11, line 4).

Class SIP1 contains the implementation of the verification step. For every SIP(G_p, G_t) instance, if it is not discarded after filtering, it goes to the SIP1 procedure (line 21, Algorithm 9). Algorithm 11 is a pseudocode of the implementation.

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.

Algorithm 11 SIP1

```
1: procedure SIP1 (alldoms)
2:     if alldoms is empty then return solution                          ▷ true or false
3:     end if
4:     dom_u ← smallest(alldoms)          ▷ select vertex u with the smallest domain first
5:     newAlldoms ← initialize with size=(alldoms.size - 1)
6:     for i in dom_u.getNextSetBit do                  ▷ for each entry in u with bit set to 1
7:         assign i as a value of vertex u
8:         for (dom_v in alldoms) ∧ (dom_v != dom_u) do                ▷ the domain of each vertex
9:             if !consistent then return consistent ∧ SIP1(newAlldoms)
10:            end if
11:            newdom_v ← dom_v
12:            set i^th entry of newdom_v to false                    ▷ cannot take value assigned to u
13:            if u is adjacent to v in G_p then
14:                (newdom_v)AND(neighbours of i in G_t)     ▷ v can only take vertices in G_t adjacent to i
15:            end if
16:            add newdom_v to newAlldoms
17:            consistent ← !(newdom_v == 0)     ▷ if there is a domain wipe out, consistent becomes false
18:        end for
19:        consistent ∧ SIP1(newAlldoms)
20:    end for
21: end procedure
```

Chapter 5

Evaluation

5.1 Light Filters

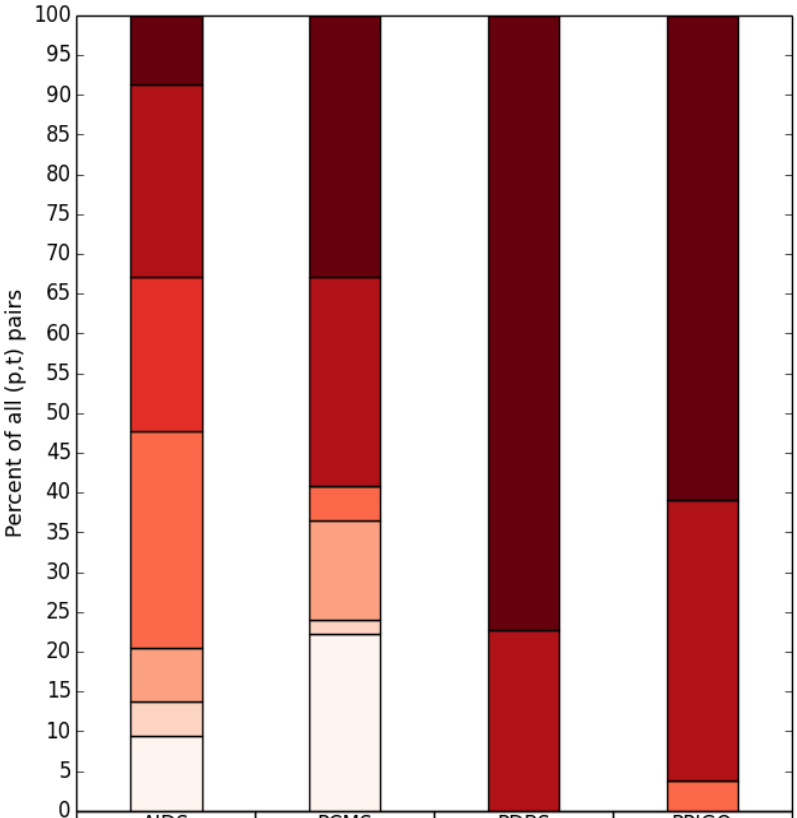
This Section reports on the observed performance of the subgraph query processing algorithm described in Chapter 4. SIP1 was run with each of the datasets discussed in Section 2.2, some of which were also used for empirical study by [15, 5, 11, 5, 14, 17, 32, 30]. The experiments are conducted on a Windows 7 SP1 host with 2 Intel Xeon E5-2660 CPUs (2.20GHz, 20MB Cache, 8 cores/16 threads per CPU) and 128GB of RAM, same machine used by [15]. 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.

For each rejected SIP instance during filtering, we recorded the test that rejected it using the scale on Table 4.1 and calculated the total number of instances that were eliminated by a particular test for each dataset. These numbers were then used to derive the percentage of SIP instances eliminated by each fail test. We also computed the percentage of SAT and UNSAT SIP instances for each dataset. Figure 5.1 shows our results. The brown part of each bar represents the percentage of SIP instances that are SAT, the rest of the bar for UNSAT problems. All satisfiable instances had to go through the verification step, whereas for some of the UNSAT problems were discarded during the filtering. For instance, the leftmost bar represents the aids dataset, where 8.67% of all instances are SAT. Out of the UNSAT problems, 24.211% were discovered during the verification stage and the rest 32.881% were filtered by either of the trivial failures. The number of SAT and UNSAT problems are on Table 2.2. The following observations were made:

- Filtering gives best performance for the instances in the aids dataset. In particular, almost 70% of the targets are rejected before verification. Aids also contains the largest number of UNSAT instances. Also, this dataset tends to be the main one (and sometimes the only one) used for evaluation for some subgraph query processing algorithms like [5, 14, 17, 32].
- Filtering is not successful for any instance in pdbs. Similarly, only 3.75% of the targets were filtered in ppigo. In other words, SIP call was made for every pattern and target graph in the dataset, because they were compatible with respect to every condition on table 4.1. The main reason is that most of the instances of pdbs and ppigo are SAT (77.22% and 61% respectively) and had to go through verification to be solved. Here, filtering can be effective for at most 22.78% and 39% of the instances. In such cases, query processing method that puts low amount of effort (or none) during filtering and implements an efficient verification algorithm will show much higher performance than method that employs heavy filtering approach and naive SIP algorithm for verification. This hypothesis was confirmed by the results of the study presented in [15], where each of the evaluated heavy indexing techniques, evaluated using the

same datasets, demonstrated several times poorer performance than the light filtering technique discussed here.

- There are duplicate target graphs in the pcms and pdbs datasets. The pcms database is supposed to contain 200 targets [2]. In practice, there are only 50 unique graphs and each of them is added 4 times. The pdbs dataset is composed of 600 targets [2], but out of them only 30 are unique, each of them duplicated 20 times.



	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

5.1.1 Hardness of SIP in terms of search nodes

We report on the cost of the verification step in terms of the number of search nodes taken to solve a SIP instance. 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 T, we take all instances that were not rejected during filtering and we compute the number of search nodes taken to solve SIP for each instance. We then compute the number of instances $|T_i|$ solved in a given number of search nodes i for $0 < i < n_{max}$, where n_{max} is search nodes taken to solve the hardest SIP instance in T. Figures 5.2, 5.3, 5.4 and 5.5 present our results. Here, $|T_i|$ 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.

The value on the y-axis for each percentile of T represents the number of search nodes taken to solve the hardest instance that belongs to the percentile. In other words, the graphs below show the hardest instance observed for each percentile of T. For example, if we had 3 graphs that belong to the i^{th} percentile of T 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 using at most 4 search nodes, 28% are solved by using at most 1 node of search effort. The hardest problem (the right-most bar) takes 65 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 is 4 milliseconds and the instance is UNSAT.
- pdbs is harder than ppigo and aids with most varied number of search nodes per instance. It is on average harder than pcms, however, the hardest instance in pcms takes more search effort than the hardest instance in pdbs. Figure 5.4 shows that 20% of the targets in pdbs are solved by using at most 100 nodes, which is significantly higher than ppigo, where even the hardest instance was solved in less than 70 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.
- The dataset with the hardest instance is pcms. The hardest SIP takes 10,470 nodes to solve and it is between pattern "16_1C5G.cm.A" and target "1CY2.cm.A.cmap". It was solved in 12 milliseconds and it is unsatisfiable. Looking at the other 99% of pcms targets, we can see that they are mostly easy. For example, 43% of the SIP instances are solved by using at most 10 nodes of search effort.
- The aids dataset is comparably easy. The maximum nodes taken to solve a SIP instance took 619 nodes of search effort. It is the SIP call between pattern #1 and target #629591, it took 0 milliseconds of time and it is UNSAT.
- Looking at aids, pcms and pdbs, the number of nodes taken to solve SIP grows exponentially with the percentile of the population.
- The hardest instance of each dataset is UNSAT.
- All four datasets are easy.

5.1.2 Hardness of SAT vs UNSAT SIP instances

The observation that the hardest instance of each dataset is unsatisfiable raises the following question: is UNSAT SIP generally harder than SAT SIP? The experiments described in this section are again conducted in terms of number of search nodes and they are intended to further investigate this observation.

The following eight plots below break each of the plots discussed in the previous section (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

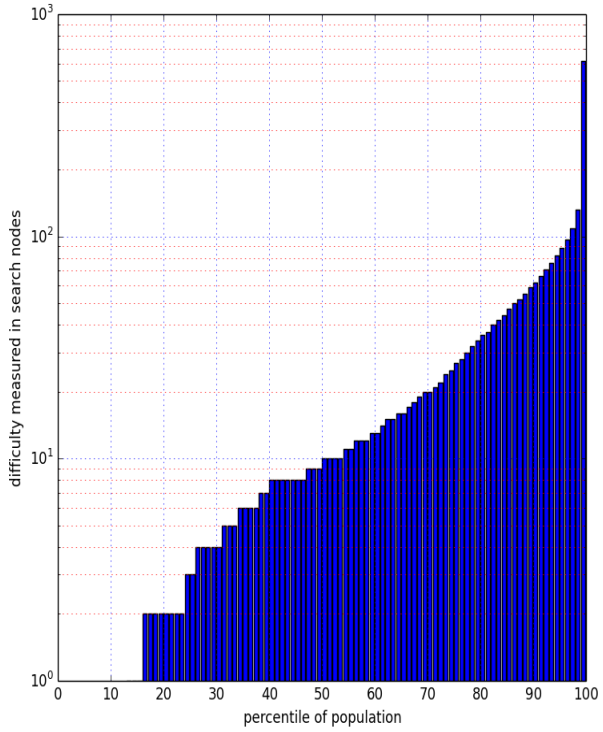


Figure 5.2: SIP on aids dataset

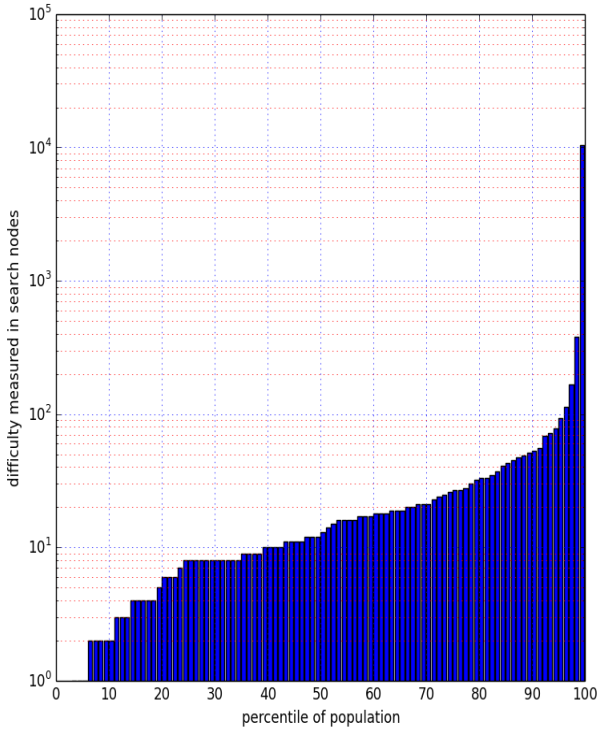


Figure 5.3: SIP on pcms dataset

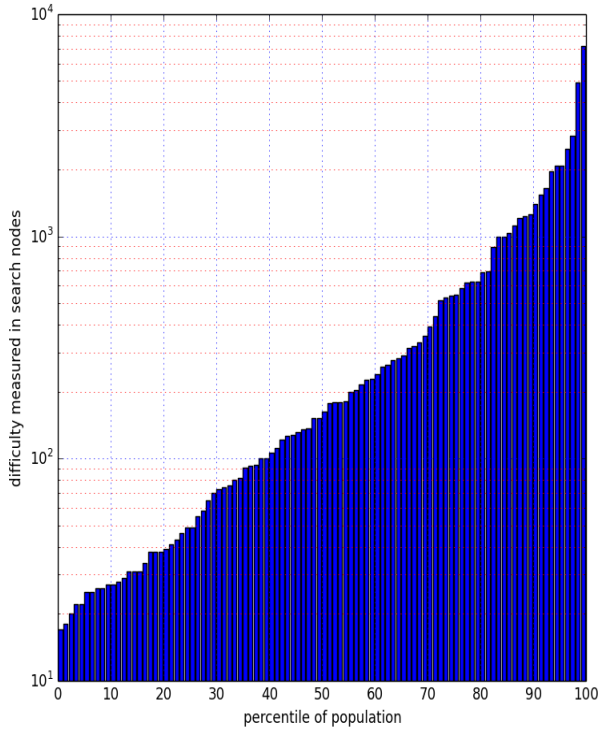


Figure 5.4: SIP on pdbs dataset

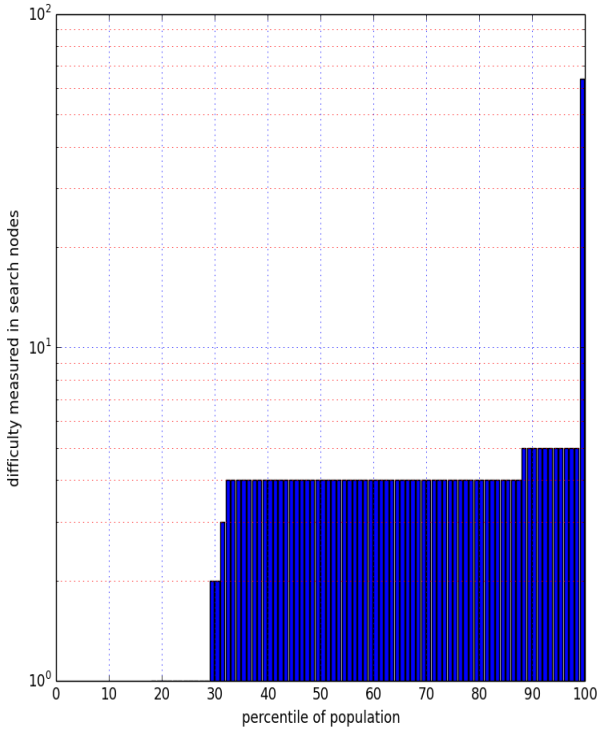


Figure 5.5: SIP on ppigo dataset

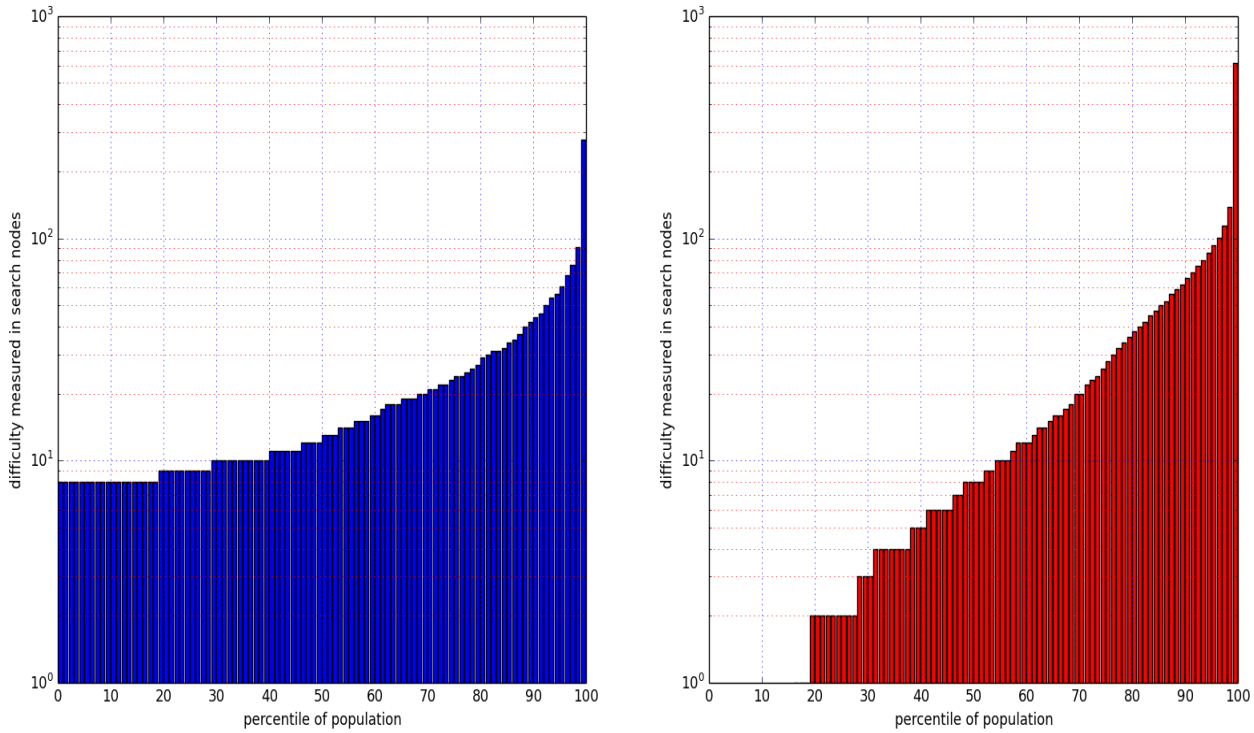


Figure 5.6: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in aids

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 the previous Section.

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 5.1 presents statistics in terms of search effort for SAT (blue columns) and UNSAT (red columns) instances. For instance, the Table shows that the total number of search nodes taken to solve all SAT SIP instances for the aids dataset is 437,108 and the total number of search nodes taken to solve all UNSAT SIP instances in aids is 2,295,724. Using these Figures, we derive that the total number of search nodes taken to solve all SIP instances for the aids dataset is the sum of those two numbers, which is equal to 2,732,832. Figure 2.2 shows the number of 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). Using the Tables and Figures, the following observations can be made:

- For aids, pcms and ppigo, the easiest percentile of UNSAT SIP instances require less number of search nodes to be solved than the easiest SAT SIP instances. The hardest percentile of UNSAT SIP take more search effort than the hardest percentile of SAT SIP instances.
- For 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 5.1 confirm this observation. On average, SAT instances are 3 times easier than UNSAT, the SAT instances

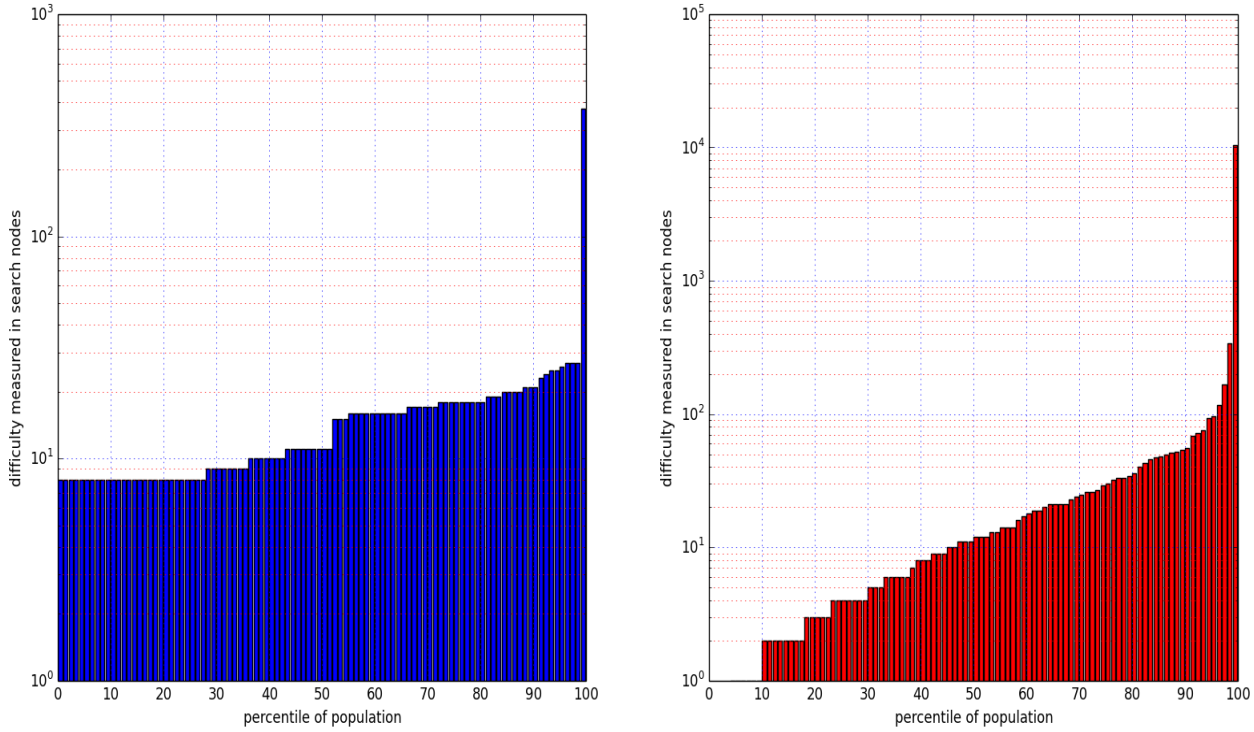


Figure 5.7: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in pcms

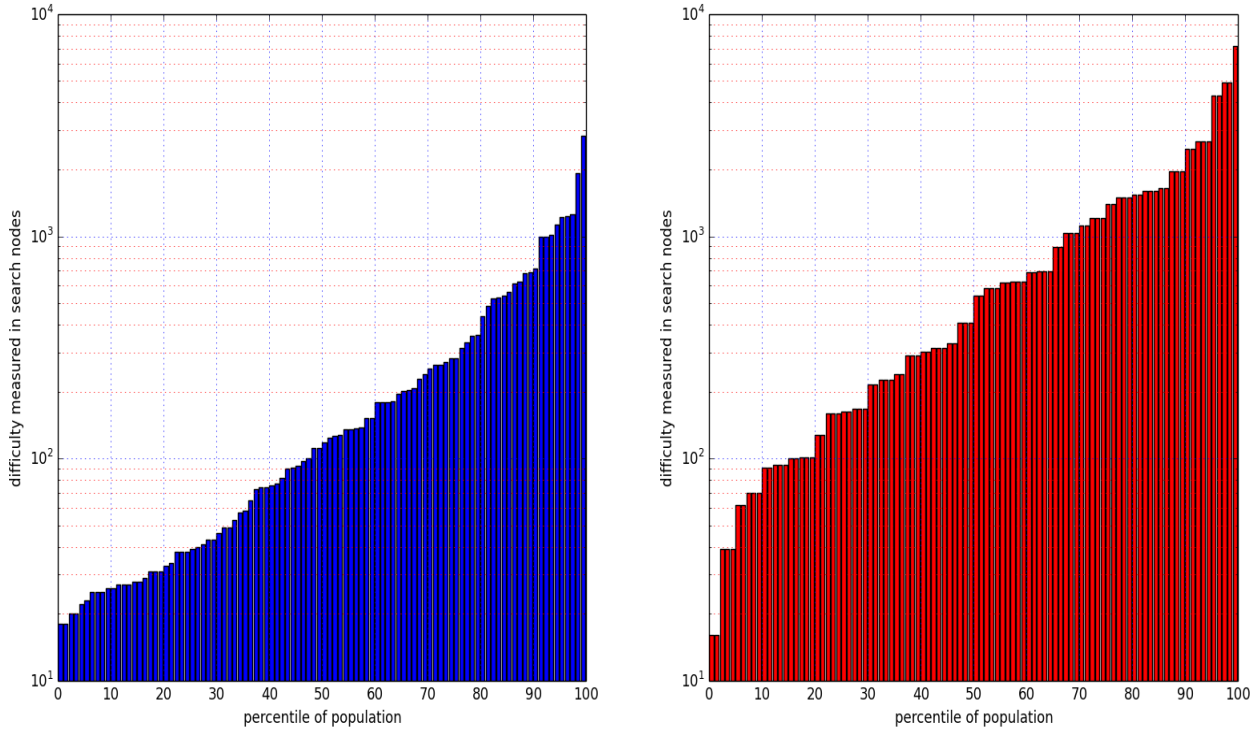


Figure 5.8: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in pdbs

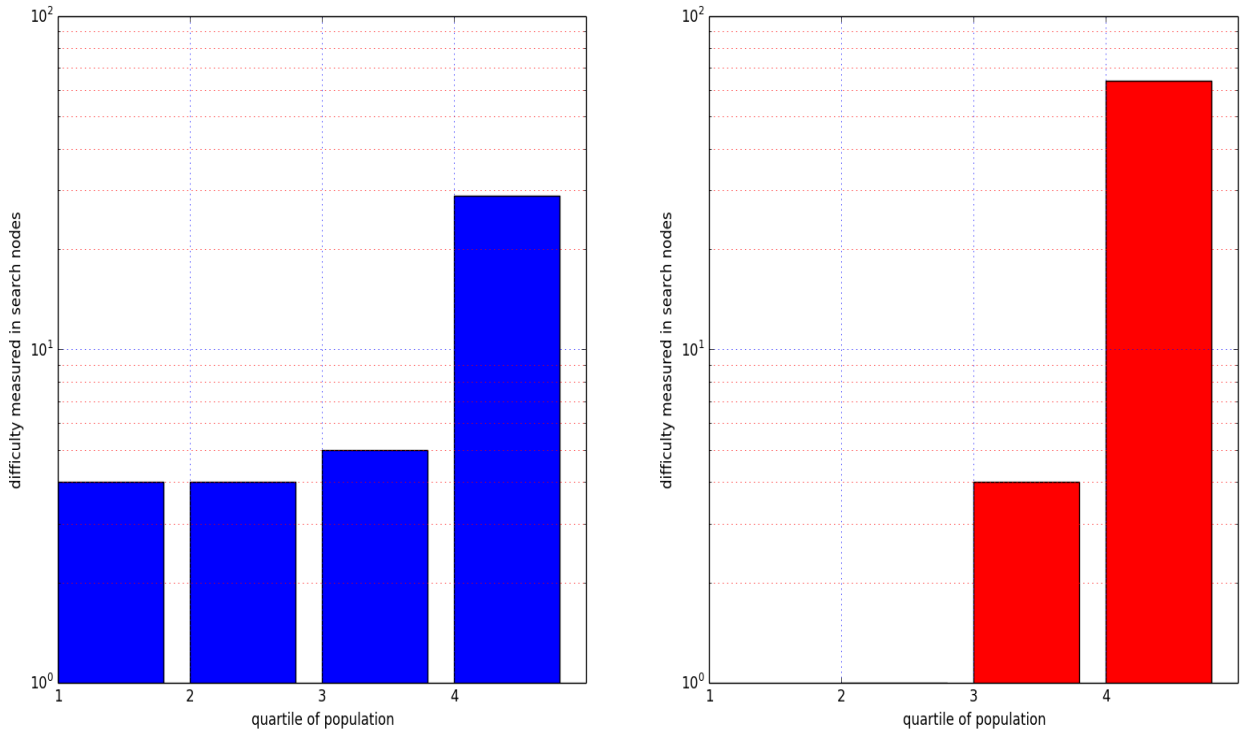


Figure 5.9: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in ppigo

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 nodes taken to solve the hardest UNSAT instance (7,152).

- Table 5.1 shows that for the pdbs dataset, the total search effort taken to solve unsatisfiable problems is bigger (894,260 nodes taken in total for SAT and 854,720 nodes in total taken for UNSAT problems) contrary to what we observed on the Figures. 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 5.1), where a SAT instance is on average 3 times easier to solve than an UNSAT instance.

- The pcms dataset 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 (5.1). 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 (5.1) 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 5.1).

- 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 5.1 shows that SAT and UNSAT SIP instances are similarly hard on average for this dataset.

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

5.1.3 Hardness of SIP in terms of running time

Table 5.2 shows the total time in milliseconds taken to solve all SIP instances of a given dataset. The time on the first row includes file I/O, creating and instantiating objects and domains of variables, the filtering and the verification time. The second row shows the number of milliseconds 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.

It is easy to notice that reading in the graphs from a file and instantiating the required objects and variables takes most of the running time for each dataset. For ppigo and pcms, filtering took more time than verification. These figures are very close for the aids dataset (filtering took 2,569 millis. and verification took 2,687 millis.). 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 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 that the four Big Data datasets are indeed very easy.

Comparison with Big Data algorithms

Evaluation of six "state of the art" subgraph query processing algorithms ([14, 17, 11, 32, 5, 27]) is presented in [15]. The algorithms employ a heavy filtering approach using an index structure and run [9] SIP algorithm during verification over the candidate set ($\mathcal{C}$). We use the results in this work for comparison with the performance of the light filters method with the evaluated approaches.

In the study described in [15], it was observed that for pcms and ppigo, the filtering stage for four of the evaluated algorithms never finished executing, so the instances never underwent verification. Table 5.2 shows that the performance of the light filters method is incomparably faster.

For the aids datasets, the fastest of the evaluated algorithms is GraphGrepSX [5] and it took 9 seconds to perform filtering and about 600 milliseconds for verification. It took us 2,569 milliseconds for filtering (5.2), but verification was slower (2,687 milliseconds). The fastest algorithm evaluated in [15] took about 7 seconds for filtering and 200 milliseconds for verification and it is again GraphGrepSX [5]. The light filters approach has slower verification and slightly faster filtering.

The algorithms evaluated in [15] have an additional overhead that is not present in our approach, which is the size of the index that has to be stored.

TODO: compare with table 2.3 with running times in Section 2.4.3.

	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 5.2: Total running time in millisec for each dataset

5.2 Summary of findings

This Section includes a brief summary of the key points made in this Chapter.

It was discovered that the Big Data datasets are of poor quality. Two of the datasets have targets that are copied multiple times each. All four datasets contain very easy SIP instances. The hardest of the datasets is pdbs. Even with the hardest dataset, a SIP instance took only 4.47 milliseconds to be solved on average. Verification for 50% of the instances in pdbs took much less than 90 search nodes, the most expensive SIP problem costs 10,470 search nodes. Surprising finding was that the datasets can be easily kept in memory. Big Data is much smaller than what we initially expected. Beneficial future work in this area would be to develop better quality, much bigger and harder datasets.

Filtering is bound to work only in the area of UNSAT SIP instances. Consequently, when most of the instances of the dataset are SAT, filters can be more an overhead than help. The SIP algorithm, performed during verification, can both identify SAT and UNSAT problems. Therefore, constructing sophisticated filtering would give little gain, if any, but implementing fast and smart SIP would improve the performance significantly.

We did experiments to find out whether SAT problems are generally easier than UNSAT problems that were not rejected by filtering. What was observed is that for each of the datasets, the hardest and the easiest instances in terms of search nodes were UNSAT. Possible way of improvement is to modify the filters algorithm so that it can prune those hard instances. This will involve further investigation of what makes a problem hard.

Chapter 6

Conclusion and Future work

6.1 What did we do? What does it suggest?

6.2 Suggestions for Future work

Appendices

Appendix A

Implementation

Query Number	Answers Number
0	8 042
1	11 957
2	78
3	461
4	77
5	3

Table A.1: The number of answers for each query for aids dataset

An example of running from the command line is as follows:

```
> java MaxClique BBMC1 brock200_1.clq 14400
```

This will apply $BBMC$ with $style = 1$ to the first brock200 DIMACS instance allowing 14400 seconds of cpu time.

Table A.2: CT-Index: Running time and results

	fingerprint size max path len max subtree len max cycle len	index build T[sec]	query T[sec]	total T [sec]	**query#** #candidates
1	4096 -1 5 5	82.376	5.896	88.272	**0** 11 160 **1** 13 577 **2** 975 **3** 2 950 **4** 2 575 **5** 6
2	4096 5 5 5	108.465	5.948	114.413	**0** 11 168 **1** 13 589 **2** 1058 **3** 2 949 **4** 2 576 **5** 6
3	4096 5 -1 -1	37.621	7.929	45.550	**0** 31 083 **1** 36 458 **2** 4 285 **3** 7 261 **4** 13 316 **5** 252
4	4096 5 1 1	41.482	7.96	49.442	**0** 31 083 **1** 36 458 **2** 4 285 **3** 7 261 **4** 13 316 **5** 252
5	2048 5 1 1	41.269	13.295	54.564	**0** 31 085 **1** 36 458 **2** 4 293 **3** 8 539 **4** 13 319 **5** 252
6	2048 -1 5 5	87.959	8.22	96.179	**0** 11 540 **1** 13 582 **2** 987 **3** 2 983 **4** 2 660 **5** 9

Appendix B

Generating Random Graphs

We generate Erdós-Rënyi random graphs $G(n, p)$ where n is the number of vertices and each edge is included in the graph with probability p independent from every other edge. It produces a random graph in DIMACS format with vertices numbered 1 to n inclusive. It can be run from the command line as follows to produce a clq file

```
> java RandomGraph 100 0.9 > 100-90-00.clq
```

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

[5] 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.

[6] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics*, 14(Suppl 7):S13, 2013.

[7] 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.

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

[9] 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.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

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

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

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

[23] 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.

[24] 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.

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

[26] 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.

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

[28] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach, 2004.

[29] 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.

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

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

[32] 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

candidate set ($\mathcal{C}$) todo. 35

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. 8

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

index todo. 35

search node A search node denotes the number of recursive calls to the SIP algorithm taken to find a solution. 29, 36

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.. 20

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, 20

Acronyms

FC forward checking. 26

G graph. 24

G_p pattern graph aka query. 22, 26

G_t target graph. 22, 26

lds label degree sequence. 22–24, 26

nds neighbourhood degree sequence. 22, 23, 26

SAT Satisfiable. 3, 4, 10, 22, 28–30, 32–34, 36

SIP subgraph isomorphism problem. 2–4, 6, 9–12, 18, 21–24, 26, 28–30, 32–36

UNSAT Unsatisfiable. 3, 4, 10, 22, 23, 28–30, 32–34, 36