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

1.1 Terminology, Definitions and Notations

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

1.1.1 Graph Theory

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

Example 1 An example of undirected labeled graph is graph T on Figure 1.1, where the labels of T are the colors of its vertices. L(V(T)) is equal to red (R), yellow (Y) and blue (B).

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

Example 2 An example of a path in graph T on Figure 1.1 is the sequence $\{2, 3, 4, 5\}$, which is a path of length 3. The path consisting of vertices $\{2, 3, 8\}$ is an example of a cycle also of length 3.

The set of neighbours of a vertex v in G consists of all vertices adjacent to v. The *degree* of v is the cardinality of its set of neighbours. By $v \sim_G w$ we mean that vertex w is a neighbour of v in G. The set of neighbours of v

forms the *neighbourhood* of v, denoted as N(G, v). The neighbourhood degree sequence of v, denoted nds(G, v), is a sequence consisting of the degrees of every neighbour of v, taken in non-increasing order.

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

Example 3 Graph T on Figure 1.1 is connected. Let use look at vertex 2. Its degree is equal to 3, because vertex 2 is adjacent to three vertices, namely $2 \sim_T 1$, $2 \sim_T 3$ and $2 \sim_T 8$. Consequently, N(T, 2) = 1, 3, 8.

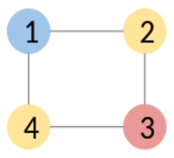


Figure 1.1: graph P

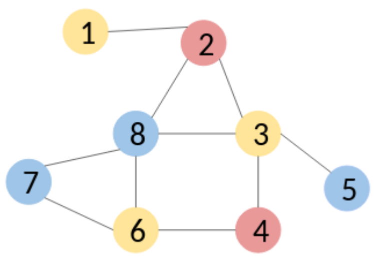


Figure 1.2: graph T

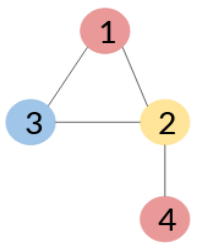


Figure 1.3: graph T1

Figure 1.4: Instances of subgraph isomorphism problem (SIP)

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

$$Density(G) = \frac{2|E(G)|}{|V(G)|(|V(G)| - 1)} \tag{1.1}$$

This formula is derived from the fact that the maximum number of edges that G can contain is $\frac{|V(G)|(|V(G)|-1)}{2}$ when G is a clique. It is easy to see that of the number of edges is increased, while fixing the number of vertices as a constant, the density of G becomes higher.

Example 4 The density of graph T on Figure 1.1 is as follows. T has 10 edges and 8 vertices. Using the formula 1.1, $\frac{20}{8.7}$ is equal to 2.3, which is the density of T.

1.1.2 The Subgraph Isomorphism Problem

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

Example 5 Lets us consider the two graphs in Figure 1.4. The SIP instance between the pattern P (in the left-hand side of the Figure) and the target T (the graph in the middle of the Figure) is SAT- the mapping from P to T maps vertex 1 to vertex 8, vertex 2 to vertex 3, vertex 3 to vertex 4 and vertex 4 to vertex 6.

There are various existing algorithms for the subgraph isomorphism problem [12, 31, 24, 21, 7, 43, 26]. More thorough analysis of the problem and discussion on existing work is presented in Section 2.4.1.

1.1.3 The Filtering-Verification Paradigm

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

Example 6 Let us consider Figure 1.4 and assume that graph P (in the left-hand side) is the query set, and graphs T (in the middle) and T1 (in the right-hand side) represent a database. Then, subgraph query processing would solve the SIP for instances (P, T) and (P, T1), where the former instance will be determined as SAT (Example 5) and the latter as UNSAT (there is no valid mapping from P to T1).

Filtering

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

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

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

There are many existing index computation and feature extraction techniques. In order to illustrate the process of building an index, an example of such in is presented below. Most common approaches are discussed later in this work.

Example 7 Let D be the database and Q the query set defined in Example 6 with the graphs on Figure 1.4. Let the features be paths of length 2 represented as strings of concatenated vertex labels (yellow(Y), blue(B) or red(R)). The features of *T*, *T1* and *P* are B-B, B-Y, R-Y, B-R; B-Y, R-Y, B-R and B-Y, R-Y, B-R respectively. The index of D is then the union of the features of *T* and *T1* and the index of Q consists of the features of *P*. Note that the

[1]Unless it has already been computed.

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

reverse of each of the strings is also a valid feature, as the graphs are undirected. Therefore, storing each feature and its reverse does not give additional structural information (in this case, this is equivalent to storing feature duplicates) for the cost of doubled index size. This is the reason why feature extraction algorithms enforce specific feature ordering requirements. $\mathcal{I}_D$ is then the union of the features of T and $T1$ and $\mathcal{I}_Q$ contains all features of P.

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

Example 8 Taking the indices of D and Q computed in Example 7, the candidate set C consists of both T and T1 as each of them contains all features of P.

1.1.4 Verification

Let S' be the Cartesian product of Q and C, where C is the candidate set computed after the filtering stage. *Verification* is the process of solving the subgraph isomorphism problem (SIP) for every pair of graphs in S'. All targets that do not contain the pattern, but were not rejected during the filtering step (i.e. they were included in the candidate set), are called *false-positives*. The better the filtering technique, the lower number of false-positives it admits. The number of false-positives is often used as a measure of effectiveness of the filtering method[20].

The usage of the filter-verification paradigm is motivated by the fact that the decision version of the subgraph isomorphism problem is NP-complete [11]. Reducing the number of SIP calls by discarding targets that are unsatisfiable (UNSAT) for a given pattern without performing SIP call is believed to give opportunities for significant performance improvement [19, 20]. There are various subgraph query processing algorithms based on this paradigm [19, 22, 6, 44, 15]. They are discussed in more detail in Chapter 2. A thorough analysis of one of them is presented in Section 2.3.

1.2 Aims of the project

The subgraph query processing problem has wide variety of applications in many fields, some of which include bioinformatics [7], chemistry [29], computer vision [14, 33], law enforcement [9], model checking [30] and pattern recognition [10]. It involves repeatedly examining a large database, searching for graphs that contain particular patterns. For instance, in order to apply the best treatment for cancer, one might have to screen a patient's tumor to search for particular set of bio markers to identify the best course of treatment [38].

Filtering-verification paradigm is a new trend in Big Data research motivated by the fact that the decision version of the subgraph isomorphism problem is NP-Complete [20, 19]. The subgraph isomorphism tests are reported as "too time consuming" [20] and this is explained by the nature of the complexity of the subgraph isomorphism problem [19, 20, 40]. Spending more time upfront to build a database index and then reuse it for filtering with subsequent queries is shown in existing literature to "... vastly improve search time and the up-front computation time is paid back quickly as repeated searches are performed" [1].

In this work, we study subgraph query processing methods. We first look at the approach to solving the problem employed by already existing work, i.e. the filtering-verification paradigm. We implement a framework that supports two different filtering techniques and carry out its empirical analysis. We then designed a novel subgraph query processing approach based on the filtering-verification paradigm that does not employ an index structure, but makes use of fast simple filtering before a call to a subgraph isomorphism algorithm. Finally, we compare the performance of the light filters approach with the algorithms that use heavy index-bound filtering, make conclusions on the nature of the problem and the advantages of each method.

1.3 Report organisation

The remaining of this work is organised as follows. Chapter 2 presents review and analysis of existing approaches to solve the subgraph query processing problem and the properties of 4 datasets used for testing and evaluation. An implementation and empirical analysis of our framework consisting of two graph indexing and filtering methods is presented in Chapter 3. Chapter 4 describes a new approach to solve the subgraph query processing problem, which we call Light Filters. An evaluation of Light Filters and an 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.

Chapter 2

Review of existing work

2.1 Datasets

This Section includes the specifications and the nature of the four datasets used for performance study of existing work in Section 2.3.3 and for evaluation in Section 5. The datasets are obtained from [2]. Many related research publications use them to assess the performance of their subgraph query processing methods [19, 22, 6, 44, 15, 42, 20]. Therefore, running our experiments on these datasets makes our results easier to compare with existing work.

All four datasets consist of undirected labeled graphs. Each dataset has a set of target graphs, also called a database, and multiple sets of query graphs. Every graph has an id as well as every vertex in the graph. A description of each dataset is given below and the specifications of each dataset is shown in Table 2.1.

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

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

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

The fourth dataset is Ppigo and it has a database that consists of 20 protein interaction networks, where networks belong to species. This is the smallest dataset which also has the biggest graph order and size on average (4,942 as shown in Table 2.1).

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

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

Table 2.1: Characteristics of the datasets

	AIDS		**PCMS**		**PDBS**		**PPIGO**	
	number	percent	number	percent	number	percent	number	percent
all SIP calls	240,000	100	1,800	100	3,600	100	100	100
SAT SIP calls	20,816	8.67	592	32.8	2,780	77.22	61	61
UNSAT SIP calls	219,184	91.33	1,208	67.2	820	22.78	39	39

Table 2.2: Number of SIP instances for each dataset and how many of them are SAT and UNSAT

2.2 Filter-Verification paradigm

This Section gives an overview of already existing algorithms based on the filter-verification paradigm, defined in Section 1.1. Section 2.2.1 discusses some of the existing filtering algorithms with respect to feature extraction and choice of features and Section 2.2.2 outlines the most commonly used subgraph isomorphism algorithms for verification.

2.2.1 Existing Filtering Techniques

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

To explain what is meant by a graph mining technique, we introduce the following concepts. The *support ratio* of a feature f in a database D is equal to the number of graphs in D that contain f divided by the total number of graphs in D. A feature is *frequent* if its support ratio is higher or equal to a certain algorithm-specific threshold value. *Graph mining* techniques store only features which are considered as frequent in the database. Common graph-mining techniques include [40, 8, 18, 39, 42, 44, 41].

Exhaustive feature enumeration techniques store in the index all features of every graph in the database, regardless of their support ratio. Some well-known exhaustive feature enumeration techniques include [6, 19, 37, 22]

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 [40].

When frequently inserting and deleting graphs in the database, the support ratio of the features may change and make the index outdated. Consequently, the index becomes less reusable and thus the total execution time of subgraph query processing is highly increased. Graph-mining techniques also take longer time to built the database index since they spend additional time on calculating the support ratio of each feature. An advantage of graph mining techniques is that they require much less storage space than exhaustive feature enumeration algorithms, as only frequent features are stored in the index. This lowers the storage space requirements and makes the process of constructing the candidate set faster, since there are less features to be iterated through.

There are various structures that can be used as features. Some of them include paths [6, 15], trees [18, 41, 19], subgraphs [8, 37, 39, 42, 44] of the targets, or all of the aforementioned combined [19, 44]. The choice of features influences the filtering performance. It is often a trade off in terms of time and filtering strength [20]. This is further investigated in Section 2.3.3, where we report on performance results obtained from an indexing method that uses a combination of paths, trees and graph cycles as features.

2.2.2 Verification techniques

Most of the subgraph query processing methods that adopt the filtering-verification paradigm are mainly focused on improving the filtering stage, while reusing the same algorithm for verification [20, 19, 6, 44], which is commonly VF2 [12].

2.3 CT-Index

CT-Index [19] is an existing subgraph query processing approach that adopts the filter-verification paradigm. This method supports undirected graphs with edge and vertex labels and also wild card patterns. Although not explicitly stated in [19], CT-Index addresses the non-induced subgraph isomorphism problem defined in Section 1.1. In this Section, we introduce and discuss the filtering algorithm and analyze its complexity in Section 2.3.1. Section 2.3.2 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 2.3.3.

2.3.1 Filtering

During the filtering step, the features of all graphs in a database D are extracted and saved to a file, i.e. the target index $\mathcal{I}_D$. Features are specific subgraphs used to classify graphs, and are stored as hash-key fingerprints. Features may be paths, subtrees and/or cycles of bounded length. Since vertices may contain labels, these features can be viewed as strings from a specified alphabet (where the alphabet is the labels). In [19] it is stated that the reason for using trees and cycles (as well as paths) is that "trees capture additional structural information" and cycles "represent the distinct characteristic of graphs, often neglected when using only trees as features".

Although the time complexity of computing all features of a graph is not reported, it can be derived as follows. To extract a subtree of a graph G with m 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 in G, such that one of the vertices of e is connected to is part of the current tree and the other 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 edge in F, until the tree reaches size m.

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* is *2(d-1)* initially. After every recursive call, one more vertex is added to the tree, which introduces *(d-1)* new edges. That makes a total of *m + 1* vertices that will be added to the tree and *(m + 1)(d - 1)* visited edges. Consequently, the complexity of extracting tree features is $\mathcal{O}(|E|(m+1)(d-1))$, where $|E|$ is equal to the average size of a graph in *D*. From this formula one can see that the the size of the graphs in *D* has significant impact on the performance of the algorithm. When the graph density is high, the algorithm will be slower because of the high degree of each vertex.

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

Although not stated in [19] the complexity of their canonical labeling can be derived as follows. Step (1) is $\mathcal{O}(n)$, where *n* is the number of nodes in the tree, as one needs to visit each node before removing it. The complexity of step (2) is as follows. We write $|p|$ for the number of interior nodes in the tree, which is equal to *n* minus the number of leaf nodes. Step (2) visits a node, then visits its parent, and for every child of the parent node checks whether it should be first or second in the canonical label, using the vertex and edge labels conditions described above. This is repeated for every node in the tree up to the root. Therefore, the complexity of step (2) is $\mathcal{O}(|p|.|c|^2)$, where $|c|$ denotes the number of children of a parent.

In [19] 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 and one could expect poor performance for data sets with less label diversity and with big trees as features. More specifically, as *m* 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 2.3.3).

Fingerprints

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

Information on the implementation of the hash function is not specified in the paper. Depending on the quality of the hash function, the size of the bitset and the size of the fingerprint, collisions may occur, i.e. different features may map to the same bitset position, introducing false-positives. The [19] 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

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

to be justifiable by the compact nature and convenient processing of bit arrays as long as the amount of false positives does not increase significantly due to collisions".

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

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

2.3.2 Verification

The verification step checks all candidates computed in the filtering step via a subgraph isomorphism test. A backtracking algorithm [3], similar to VF2 [12] with additional heuristics, is used. This test is theoretically NP-Complete, and is avoided as far as possible via the filtering process. CT-Index is not alone in using (essentially) the VF2 algorithm. For example it is used in GraphGrepSX [6], gCode [22] and Tree+Δ [44]. Most papers claim that VF2 is "state of the art". However, this is not the case [31, 21, 7, 43, 24]. VF2 has been shown to perform erratically and poorly [24]. Therefore we might summarize CT-Index architecture as using a potentially expensive indexing and filtering stage in order to minimize the computational cost of using an outdated SIP algorithm.

2.3.3 Performance

This Section presents the evaluation of CT-Index. We look at the CT-Index evaluation made by [19] and [20] and at the performance results we collected when running the CT-Index source code.

The experiments described in [19] use two datasets and run on the same source code used by [20] and us, but with added support for edge labels. The first dataset is Aids, the specifications of which are outlined in Section 2.1. The second dataset is composed of synthetic graphs, the specification of which is not given.

To overcome the fact that some of the indexing methods used in the evaluation do not support labels on edges, the authors split each edge by an additional vertex that encodes the edge label. The resulting graphs are then used as an input to all indexing methods that do not support edge labels. Let us say that the database has n number of edges. According to the Aids properties outlined in Table 2.1, the average vertex degree is approximately 2. Let us say that the number of the vertices in a graph on average is n. Therefore, the number of additional vertices introduced by splitting the edges is n on average (one new vertex for each edge) and the new graphs have $2n$ edges and $2n$ vertices on average, which is twice as much as the original graphs. From the empirical analysis of CT-Index it follows that the size of the graphs and the number of their edges influences the performance of the indexing technique. Therefore, it is unclear how fair it is to compare results obtained by running the same experiments using two types of datasets: one that is modified with graphs size and order twice as much as the size and order of the graphs in the second one, which is the original.

The authors state that they "... removed 40 graphs with more than 255 edges because these graphs tend to cause problems with the implementation of gIndex" [19]. From the specifications of Aids in Table 2.1, one can see that the graphs that were removed have the biggest size in his dataset. According to the empirical analysis of the filtering methods implemented by CT-Index, the size of the graphs in the dataset influences the running time of the algorithms. Not running experiments with graphs of larger size and/or order does not show to what extend the performance of the algorithm suffers.

Following from the previous two paragraphs, we might summarize the experiments conducted by [19] as not complete, leading to inaccurate results. We now look at a second evaluation attempt made by [20], but first introduce the CT-Index input parameters and default settings, which are referred to in the remaining of this Section.

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

1. Fingerprint size. This indicates the number of bits allowed to store the features of each graph in the index. The specified fingerprint size must be equal to 2^n for some integer n. No information on why this is the case is specified in [19]. We were also unable to understand when looking at the source code.
2. Maximum path length. Indicates the maximum length of a path that is allowed to be extracted. If we specify -1, then no paths are extracted.
3. Maximum subtree length. Same as 2, but for subtrees.
4. Maximum cycle length. Same as 2 and 3, but for cycles.

The default input parameters of CT-Index are $<$4096, -1, 4, 4 $>$ [19, 20]. No information on why exactly these parameters should be used is given.

The experiments in [20] are conducted on the four datasets described in Section 2.1 and compare six well established filtering-verification methods, one of which is CT-Index, using the default input parameters of each method. The indexing algorithms are run for each of the four datasets, putting a time limit of 8 hours. The measured performance metrics are filtering time, index size, verification time and false positive ratio (FP ratio). To calculate the FP ratio, the authors propose formula 2.1. The formula is supposed to indicate how many of the unsatisfiable (UNSAT) instances are filtered without using a SIP algorithm. In the formula, $|A\{p\}|$ denotes the number of satisfiable (SAT) instances for a pattern *p* in a query set *Q* and $|\mathcal{C}\{p\}|$ is the cardinality of the candidate set for *p*.

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

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

The previous paragraph shows that the value of the FP ratio computed by formula 2.1 is strongly influenced by the number of SAT instances in the dataset. For instance, the value of FP ratio computed for an indexing algorithm *A* executed for a query set *Q* and a database *D* may be equal to the value of FP ratio computed for another indexing algorithm *B* when executed with a query set *Q′* and *D*. However, from this it does not follow that the filtering power of *A* and *B* is the same. For example, the equality of the FP ratio of *A* and *B* may be caused by the fact that the number of SAT instances for Q and D may be bigger/smaller than the number of SAT instances for Q' and D.

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

Below are some of the main results obtained by the evaluation in [20] the conclusions that can be made from these results.

- For datasets Pcms and Ppigo, CT-Index never manages to complete execution of the filtering step and no verification is performed [20]. Pcms and Ppigo are of largest size among the four compared datasets. Ppigo also has largest order and Pcms has highest graph density (Table 2.1). This suggests that the performance of CT-Index does indeed suffer when the dataset is composed of larger graphs of high density, as conjectured earlier in our empirical evaluation, and backs up our claim that the evaluation in [19] is incomplete.
- For the Aids dataset, CT-Index performs best. Filtering takes about 80 seconds and verification: about 0.7 seconds [20]. The FP ratio is one of the highest for both Aids and Pdbs datasets among the six evaluated techniques. This suggests that the algorithm has relatively poor filtering performance. A reason for this could be that the number of collisions when computing the fingerprints (hash tables) is high.

These results give a basic idea of the performance of CT-Index compared to other techniques considered as “state of the art” in subgraph query processing [20]. However, they do not investigate the performance of CT-Index depending on the type of features stored in the index and the size of the fingerprints (i.e. the hastable size). We report on the results obtained by our experiments, that were conducted in order to find out more about that. We use the open source java implementation of CT-Index written by its authors. Experiments are run on the Aids dataset [2] using input parameters that are different from the default ones. For our experiments, we choose a fixed size of parameter set [3] the purpose of which is to identify how the change of one parameter changes the performance of the algorithms implemented in CT-Index. As in [20], we measure the filtering and verification time, the FP ratio using formula 2.1 [4] and calculate the total time, that is the sum of the filtering and the verification time. The size of the fingerprints (2^n for some integer n) is from 1 to 16384. Note that when n is 0, no index can be created (all features of every graph have to be stored in a bitset of size 1) and verification is computed for every pair (P, T), where P is a pattern in Q and T is a target in D. The values of the maximum length of paths, subtrees and cycles are permutations of combinations of -1 and 5. Here, -1 tells about the influence on the performance of CT-Index when a feature is switched off, and 5 shows the change of performance when the feature is turned on (i.e -1 and 5 play the role of binary 0 and 1 to indicate for each feature that it is either off or on). For every fingerprint size, we executed the algorithm with all permutations of every combination of assignment of -1 and 5.

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

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

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

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

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

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

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

2.4 Subgraph Isomorphism Problem algorithms

This Section reports on analysis of existing methods to solve the subgraph isomorphism problem. A common way to model SIP for a pattern P and a target T is to represent each vertex in P as a variable that can take as value one of the vertices in T. It has to be ensured that every vertex in P is matched to one unique vertex in T. During a recursive search procedure, values are repeatedly tried to be assigned to variables, following an algorithm-specific heuristics until valid mapping from P to T is found or it is proven that no such mapping exists.

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

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

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

There are various algorithms in the literature that employ a variation of this model [31, 35, 29, 21, 32, 24]. We study in more detail one of them, namely [24] in the next section.

2.4.1 Glasgow Subgraph Isomorphism Problem Algorithm

This algorithm, referred to as CP15 discusses the non-induced version of the subgraph isomorphism problem for finite, undirected graphs that do not have multiple edges between pairs of vertices, but may have loops [24]. Although CP15 does not consider labels, it can be easily adjusted to take into an account labeled vertices and/or edges by adding additional constraints. In fact, adding labeling support would make the algorithm faster, as follows from a result proved in [23]. Unlike most recent work in subgraph isomorphism [32, 4], this algorithm replaces strong inference with cheaper techniques and shows that they can be beneficial [24]. The algorithm exploits parallelism for both pre-processing and search and introduces a novel usage of backjumping [27] that does not need maintaining conflict sets [24].

The algorithm uses bitset encodings: graphs are represented as bit sets, each pattern vertex has a domain which is a bit set. This representation allows for fast execution of operations. For instance, the neighbourhood intersections discussed shortly are bitwise-and operations, the unions of domains used during alldifferent propagation, nogood values discovered during search are computed using bitwise-or operations, and cardinality checks involve computing the Hamming weight (also known as population count) of the set [28], which is a single instruction in modern CPUs [24].

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

The initial filtering on top of the search also checks for compatibility of neighbourhood degree sequences of the vertices in P and T, based on the fact that a vertex v can only be mapped to a vertex w if the sequence consisting of the degrees of every neighbour of w is bigger or equal to the sequence consisting of the degrees of every neighbour of v [31]. Neighbourhood degree sequence compatibility was introduced by [31] and is used in the Light filters approach discussed later in this work in Chapter 4.

The algorithm employs a recursive search procedure that repeatedly picks a variable with the smallest domain to branch on, breaking ties on descending static degree in the original pattern graph. The values of the domain of each variable are ordered by descending static degree in the target graph. If the assigned value to a variable is in conflict with the current partial solution or does not obey certain constraints, then the search returns a nogood set of variables. In such case, the algorithm can perform a variation of conflict directed backjumping [27] without explicitly maintaining conflict sets. On success of value assignment of a variable, the algorithm updates the partical solution and makes recursive call until every variable is assigned (success) or no solution is proved to exist.

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

[5]The bound is chosen after an observation that distances greater than 3 rarely give additional filtering power and their construction is computationally very expensive [24]

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

The SIP algorithm achieves SIMD-like parallelism from the bitset encodings. The algorithm also allows for parallel supplemental graphs construction, neighbourhood degree sequence calculation, graph construction and domain initialization. In addition, the subtrees created by recursive calls made during search can be explored in parallel by multiple threads using early diversity work-stealing approach, where a single thread always preserves the sequential search order, finding states of the search space that are to be explored. The rest of the threads take work from the main thread, trying to steal early in the search tree, because value-ordering heuristics are expected to be weakest early in the search [17].

Both the sequential and the threaded versions of the algorithm were compared with an implementation of SND [4], LAD[32] and VF2[12]. The properties of the hardware, implementation-specific details, the nature of the datasets used in the experiments and the results from the evaluation are presented in the paper[24]. Below are highlights of the results obtained after the experiments:

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

CP15 might be summarized as a novel method that is specialized in solving hard instances of the SIP problem. From the experiment results reported in [24], one can see that it is the best solver for non-trivial SIP instances. Furthermore, it can be adjusted for solving trivial instances faster by adjusting the algorithm not to create supplemental graphs. This suggests that current subgraph query processing methods that count entirely on verification without performing filtering could be faster than current methods based on the filtering-verification paradigm even if one compares only the verification stages without counting the cost of filtering. Consequently, we might conclude that the research in the current area of research in filtering-verification algorithms is too focused on improving quality of filtering, while neglecting verification by using an outdated SIP algorithm.

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

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

Chapter 3

Framework for graph indexing and filtering

This Chapter describes a subgraph-query filtering and indexing framework designed and implemented in Java. The framework implements the first stage of the filtering-verification paradigm with a choice of two feature enumeration and candidates extraction techniques: Path Index(PI) and Path-Subtree Index(PSI). Given two sets of graphs: one with patterns and the other one with targets(a.k.a database), the framework computes their indices and generates a candidate set for subgraph isomorphism, which is a subset of the database. In the next sections, we present the design and implementation of every component of the framework and comment on its complexity. Section 3.1 presents the choice of graph representation. Section 3.2 explains the paths extraction algorithm, a procedure based on depth-first search with bound on the path length. Section 3.3 describes PI and PSI and finally Section 3.5 we comment on the performance of the framework and give suggestions to improve its efficiency.

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 distinct vertices, such that each successive pair of vertices are adjacent. A path can also be from a vertex to itself, in which case the first and the last vertices in the sequence are the same. 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 $d + d^2 + \dots d^{l_{max}-1}$ calls to dfsBounded in total. Therefore, the overall cost of extracting all paths in a graph of maximum length l$_{max}$ is $\mathcal{O}(n(d + d^2 + \dots d^{l_{max}-1}))$.

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 [6]. 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. □

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.

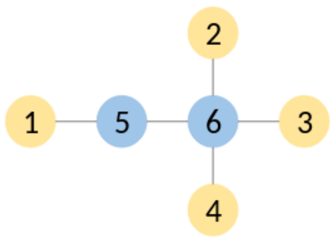


Figure 3.1: Graph G

Algorithm 3 Compute p-features procedure

```
procedure COMPUTEFEATURE (vseq)
    pfeature ← vseq.toString()          ▷ returns a string of the labels of the nodes in vseq
    rseq ← vseq.reverse()               ▷ reverse the order of nodes in vseq
    pfeature′ ← rseq.toString()         ▷ returns a string of the labels of the nodes in rseq
    if pfeature′ < pfeature then
        pfeature ← pfeature′            ▷ put to index the lexicographically smaller string
    end if
    if pfeature not in $\mathcal{I}_D$ then
        add pfeature to $\mathcal{I}_D$
    end if
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 $\mathcal{O}(n.p_i.t_i)$.

Algorithm 4 Candidates Extraction Procedure

```
1:  procedure CANDIDATESEXTRACTOR ($\mathcal{I}_{G_p}$, $\mathcal{I}_{G_t}$)
2:      candidates ← new ArrayList<>()
3:      for Graph t : targets do
4:          flag ← true
5:          for pfeature : $\mathcal{I}_{G_p}$ 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
```

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 [31]. 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.

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 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 $\mathcal{C}_{PSI} \subseteq \mathcal{C}_{PI}$.*

Algorithm 6 Contains procedure

```
procedure CONTAINS (Tindex, patPath)    ▷ Tindex is the target index, patPath is path in the pattern index
    for tarPath : Tindex do
        if CONTAINSFEATURE(tarPath, patPath) then
            return true
        end if
    end for
    return false
end procedure
```

Algorithm 7 containsFeature procedure

```
procedure CONTAINSFEATURE(tarPath, patPath)
    if tarPath.length < patPath.length then return false    ▷ if tarPath is shorter than patPath, then it can‘t contain it
    end if
    if label of each vertex in tarPath not equal label of each vertex in patPath then    ▷ equivalent to PI filter
        return false
    end if
    for i in range(0, tarPath.size()) do    ▷ for $i^{th}$ nlabel in tarf, check that it contains the $i^{th}$ nlabel in patPath
        if not CONTAINSLABEL(tarPath.get(i), patPath.get(i)) then
            return false
        end if
    end for
    return true
end procedure
```

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 Running the framework

To run the framework, the user specifies the names of the files containing all target and all pattern graphs, l_{max} (the bound on the path length allowed to be extracted) and a bit denoting which indexing and candidates extraction technique the user wants to run. If the bit is 1, the framework will index the graphs using PSI, otherwise it will run PI. The source code can be downloaded from Github [5].

3.5 Performance 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 2.1 and their filtering strength and execution time was compared with CT-Index [19], analyzed in Section 2.3. 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 2.1) 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 [36] 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 [36, 25, 34] 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 $\mathcal{O}(m)$ time.

Chapter 4

Light Filters

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

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

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

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

Algorithm 8 Light filters algorithm

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

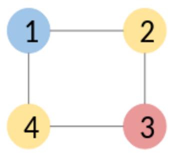


Figure 4.1: graph P

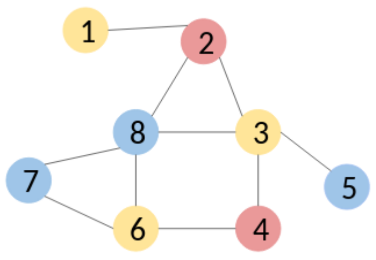


Figure 4.2: graph T

Figure 4.3: Instances of subgraph isomorphism problem (SIP)

4.1 Trivial Failures

This Section introduces the five trivial failure tests, implemented as part of the filtering stage of SIP1. The tests are based on the notion of neighbourhood degree sequence, a concept also used in [31]. First, we introduce definitions and notations that complement the theory section in Chapter 1. We then prove the correctness of our approach and outline its implementation and give an empirical analysis of each of the tests.

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

Let A and B be sequences of integers. We say that B is a *subsequence* of A if B can be derived from A by deleting zero or more elements in A. An element of a sequence A at index i is the i-th element of A, denoted by A_i.

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

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

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

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

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

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

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

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

Table 4.1: Failure tests hierarchy

Theorem 4.1.1. *Let* T *and* P *be graphs. If either of the following failure tests:*

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

is false, then P *is not subgraph-isomorphic to* T.

Proof. By contradiction. Suppose that SIP (*P*, *T*) is SAT. Suppose that test 1 is false. Then, *P* must have at least one vertex more than *T*, which means that this vertex would be unmatched. Therefore, from the definition of subgraph isomorphism, *P* is not isomorphic to *T*. Test 1 must be true.

Suppose that test 2 is false so that there exists a label *l* in L(*P*) that does not belong to L(*T*). Therefore, there exists a vertex *v* in *P* with label *l* that would be left unmatched. Therefore, from the definition of subgraph isomorphism, *P* is not isomorphic to *T*. Test 2 must hold.

Suppose that test 3 is false and there exists a label *l* both in L(*P*) and L(*T*) that is assigned to *m* number of vertices in V(*T*) and to at least *m + 1* vertices in V(*P*). That means that either at least one pattern vertex would be left unmatched, or at least two pattern vertices would be matched to the same target vertex. Either of these cases would violate the necessary conditions for *P* to be subgraph isomorphic to *T*. This is a contradiction, therefore test 3 must hold.

Suppose that *T* does not subsume *P*. Therefore, there exists label *l* both in *P* and *T* such that LNDS (*T*, *l*) $\not\succeq$ LNDS (*P*, *l*). From the proof that the previous three conditions must hold, it follows that LNDS (*T*, *l*) $\not\succeq$ LNDS (*P*, *l*), because there exists at least one vertex *v* whose degree is in LNDS (*P*, *l*) with higher degree than the degree of the corresponding vertex in LNDS (*T*, *l*). A matching, of *v* a vertex in V(*T*) that preserves adjacency does not exists and *P* is not subgraph isomorphic to *T*, which is a contradiction. Test 4 must be true.

We can deduce that if *P* is subgraph isomorphic to *T*, then each of the first four trivial tests must hold. From the rules of classical logic it follows that if either of the tests is false, then *P* is not subgraph isomorphic to *T*. □

The fifth trivial failure is based on the choice of model for our subgraph isomorphism algorithm (SIP1). Given a pattern *P* and a target *T*, we represent each vertex *v* in V(*P*) as a variable that can accept one of the vertices in V(*T*) as a value. The set of all possible values of *v* is called the *domain* of *v*, *domv*. It is represented as a bit array (bitset) of size equal to the order of *T*, where each entry maps to a target vertex. domv[i] is equal to 1 if *v* can be mapped to the i^{th} vertex in V(*T*), or 0 otherwise. If every bit in domv is 0, this means that no valid mapping from *v* to any vertex in V(*T*) exists. In such case *v* has a *domain wipe out*. During search, domain wipe out of *v* indicates that the current partial mapping can not be a subgraph isomorphism from *P* to *T*. During initialization, domain wipe out of *v* indicates that *P* is not subgraph isomorphic to *T*. Test 5 checks whether domv experiences a domain wipe out. If there exists an empty domain, the test fails and the instance does not proceed to search.

4.1.1 Implementation and complexity analysis

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

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

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

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

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

- Class *Graph* It creates graph objects, given a file with graphs represented in a certain format. A graph object *G* has size, denoted as *m*, order, denoted as *n*, *id*, array of the degree of each vertex, called *deg*, bitset array of the neighbours of each vertex, called *N*, and array, called *labels*, that stores all labels assigned to vertices in *G*. N_i contains a bitset of the neighbours of the i-th vertex in *G* and $labels_i$ contains the label of the i-th vertex in *G*, for every i between 0 and *n*. Initialization of each of the aforementioned properties takes $\mathcal{O}$(m + n) time. When *labels* is constructed, a new object for each unique label *l* is created and LNDS (*G*, *l*) is computed.

Algorithm 9 Lights Filters

```
procedure FILTER ($G_p$, $G_t$)
    if $\neg$ $|V(T)| \geq |V(P)|$ then return false                    ▷ trivial failure 1
    end if
    if $\neg$ $|L(V(T))| \geq |L(V(P))|$ then return false              ▷ trivial failure 2
    end if
    for $l \in |L(V(P))|$ do
        if $\lambda_P(l) > \lambda_T(l)$ then return false
        end if                                                      ▷ trivial failure 3
    end for
    if $\neg$SUBSUMES (T, P) then return false
    end if                                                          ▷ trivial failure 4
    alldoms ← initialize    ▷ An array of size the order of $G_p$ that contains the domain of each vertex in P
    for every $v \in$ V($P$) do
        domv ← new BitSet($|V(T)|$)    ▷ initialize domV to bitset of size the order of the target
        for $\forall$ $w \in$ V($T$) do
            if L(P, $v$) = L(T, $w$) $\wedge$ $v$.degree $\leq$ $w$.degree then
                domv[$w$] ← 1
            end if
        end for
        alldoms[v] ← domv
        if domv = $\varnothing$ then return false                       ▷ trivial failure 5
    end for
    SIP1(alldoms)                         ▷ if no failures occurred, call SIP1 algorithm
end procedure
```

Algorithm 10 Graph T subsumes graph P

```
procedure SUBSUMES (T, P)
    for $l \in |L(V(P))|$ do
        B ← LNDS (P, $l$)
        A ← LNDS (T, $l$)
        for i in range (0, $|A|$) do
            if B[i] > A[i] then return false
            end if
        end for
    end for
return true
end procedure
```

- Class *Label* This class represents a label l in G. l has a *name*, and an array of integers, sorted in non-increasing order that represents LNDS (G, l). It is built using insertion sort algorithm which is of complexity $\mathcal{O}(|LNDS(G,l)|^2)$ [13].
- 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 a subgraph isomorphism algorithm, based on the simplest of the Glasgow algorithms [24]. Algorithm 11 shows a pseudocode of SIP1. It takes the domains of all vertices in the pattern P, initialized in Algorithm 9), and repeatedly tries to assign to each variable (pattern vertex) a value (target vertex). If current assignment is compatible with the partial solution, the algorithm makes a recursive call (line 19) otherwise it backtracks. When a pattern variable u is instantiated with a target value i (line 9), all uninstatiated (future) variables have i removed from their domains (line 12). If a future variable v is adjacent to u in P then domv becomes the intersection of domv with the neighborhood of vertex i in T. This constraint is enforced by applying a logical *and* operation between the two bit sets (line 14). SIP1 uses forward checking (FC) with fail first heuristic [16]: for all uninstantiated variables representing pattern vertices, it selects to explore the one that has the smallest domain before the others (line 4).

Algorithm 11 SIP1

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

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

Chapter 5

Evaluation

5.1 Light Filters

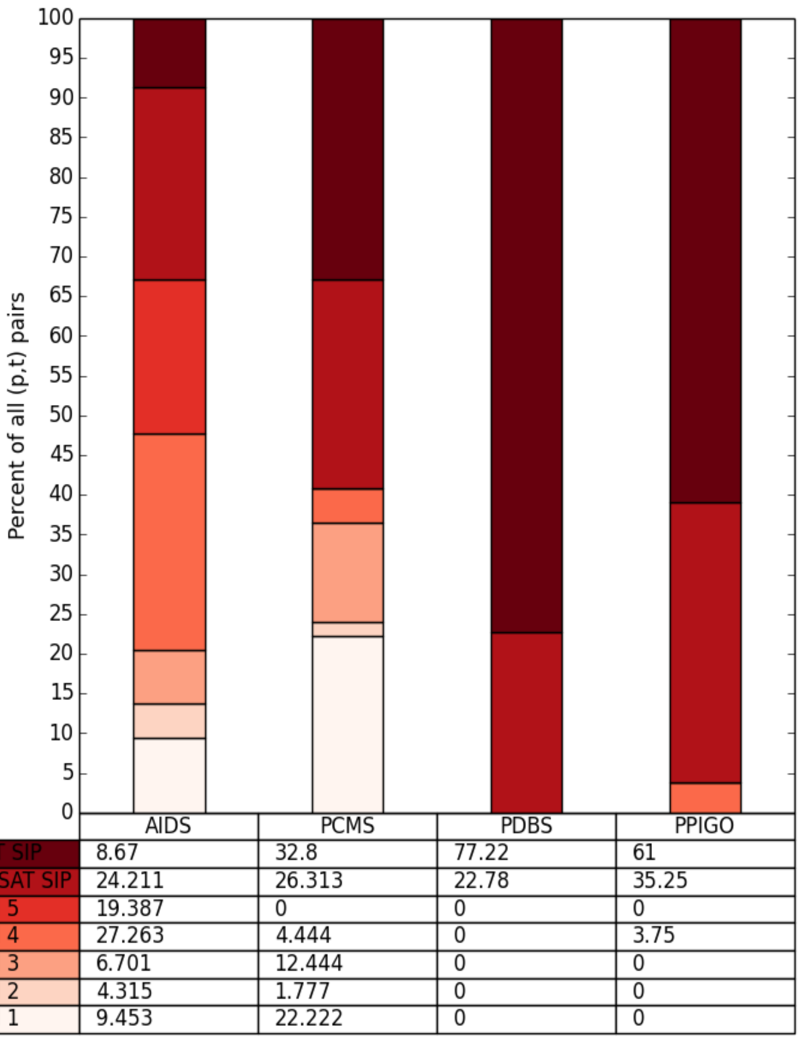
This Section reports on the observed performance of the subgraph query processing algorithm described in Chapter 4. SIP1 was run with each of the datasets discussed in Section 2.1, some of which were also used for empirical study by [20, 6, 15, 6, 19, 22, 44, 42]. The experiments are conducted on a Windows 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 [20]. 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 [6, 19, 22, 44].
- 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 [20], 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

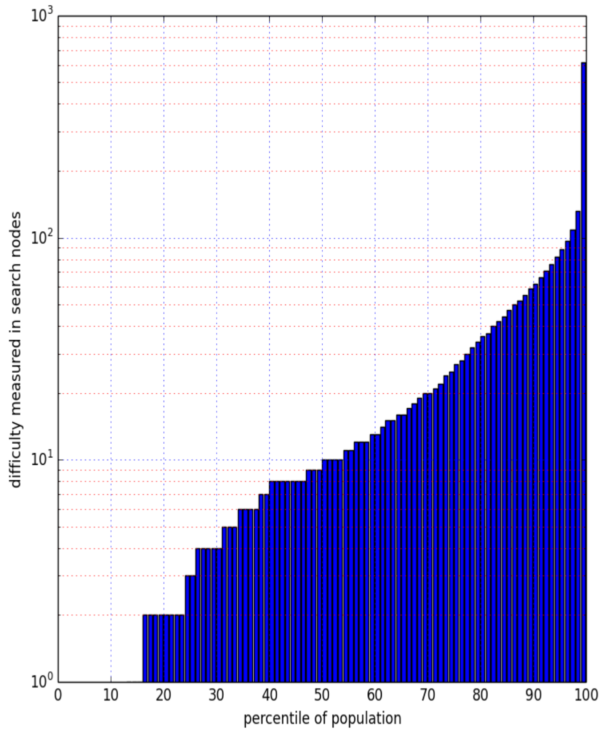


Figure 5.2: SIP on aids dataset

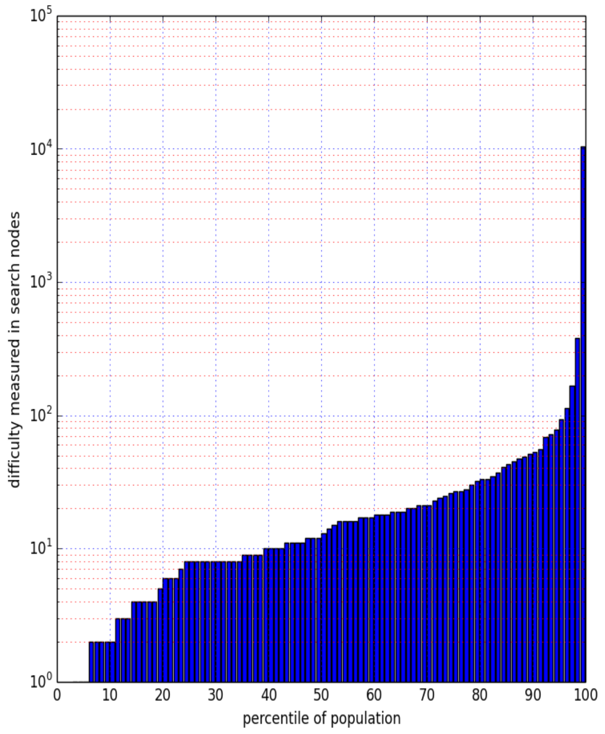


Figure 5.3: SIP on pcms dataset

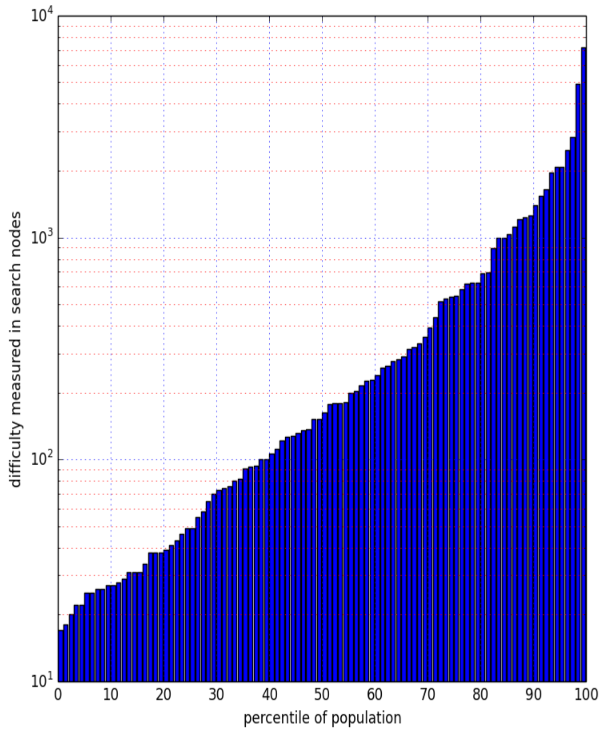


Figure 5.4: SIP on pdbs dataset

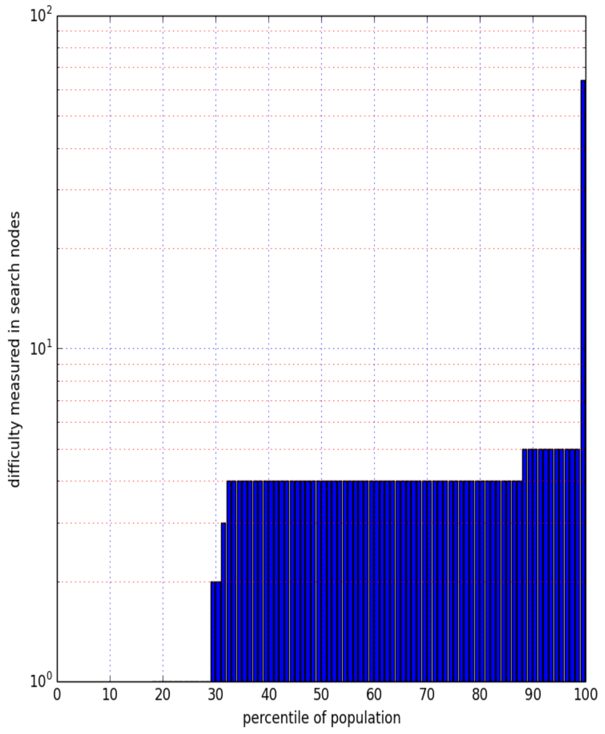
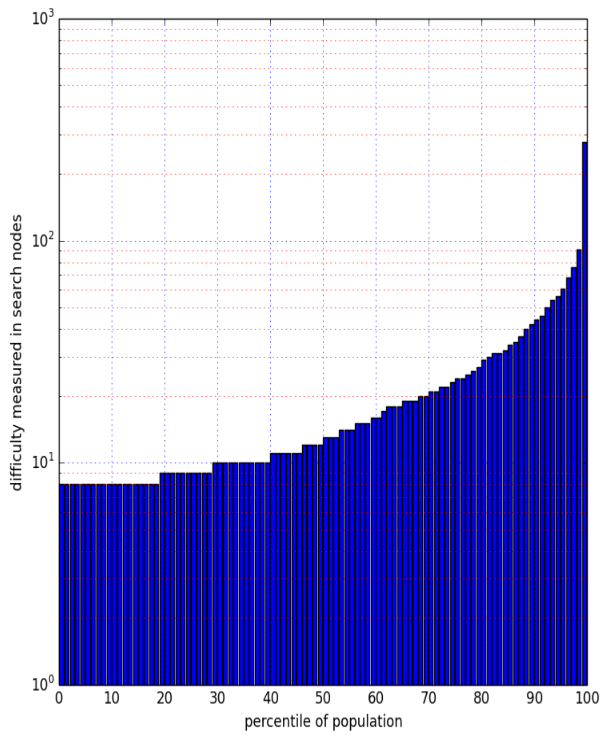


Figure 5.5: SIP on ppigo dataset



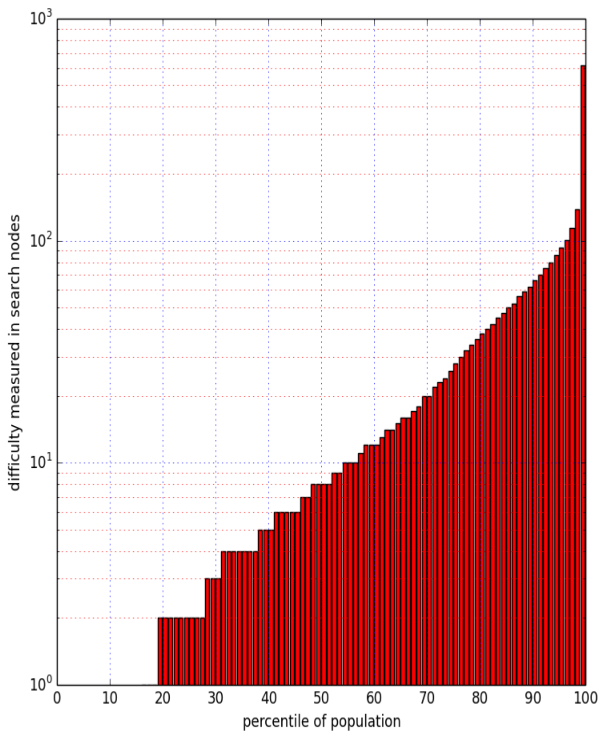


Figure 5.6: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in aids

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

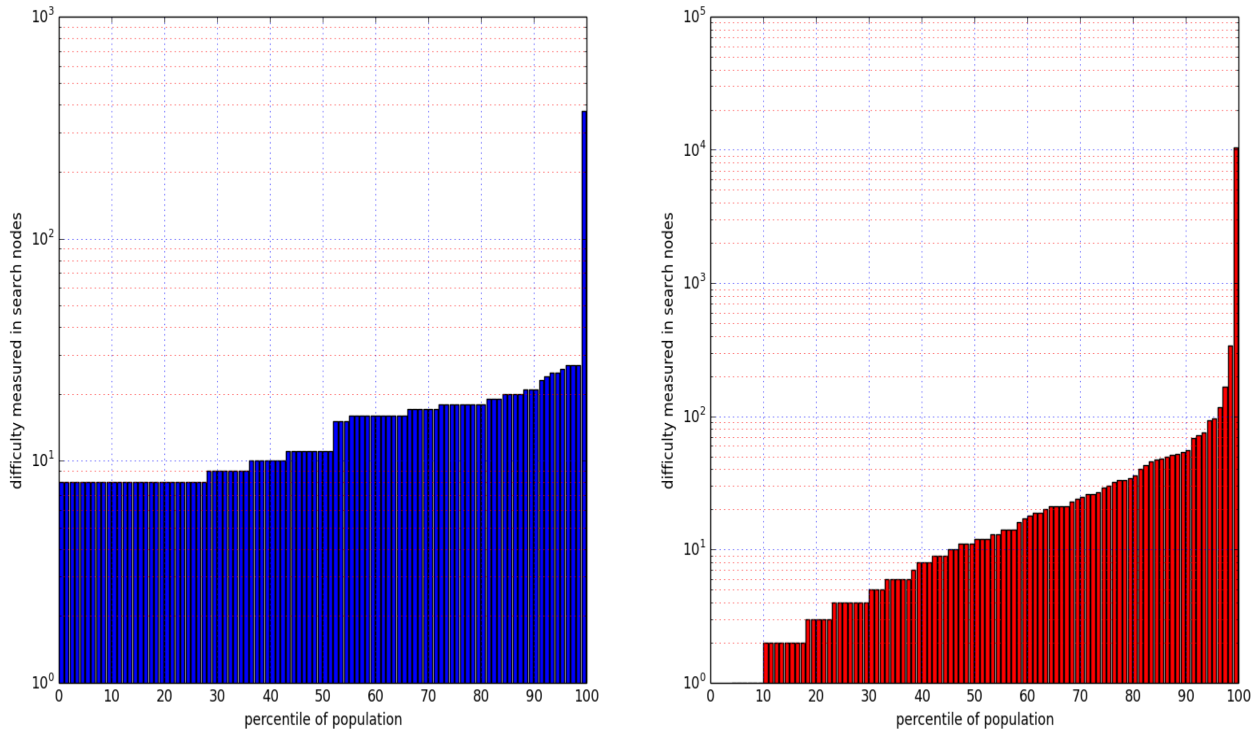


Figure 5.7: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in pcms

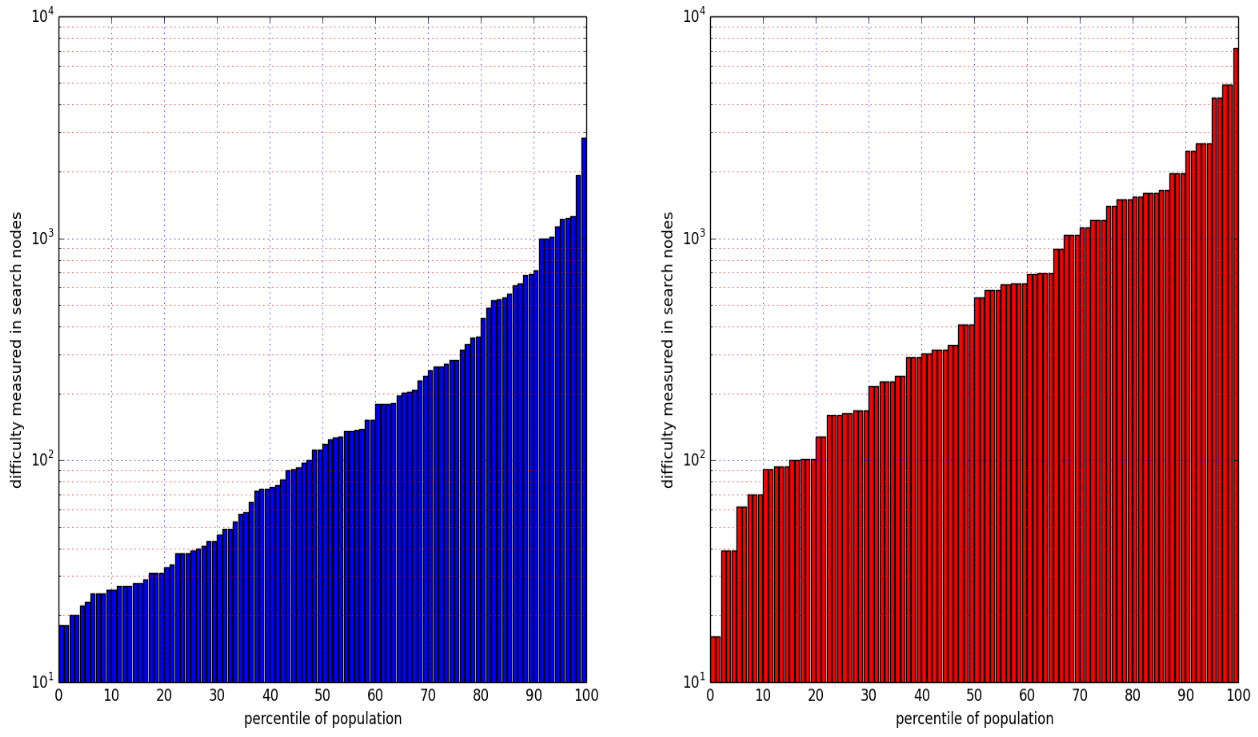


Figure 5.8: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in pdbs

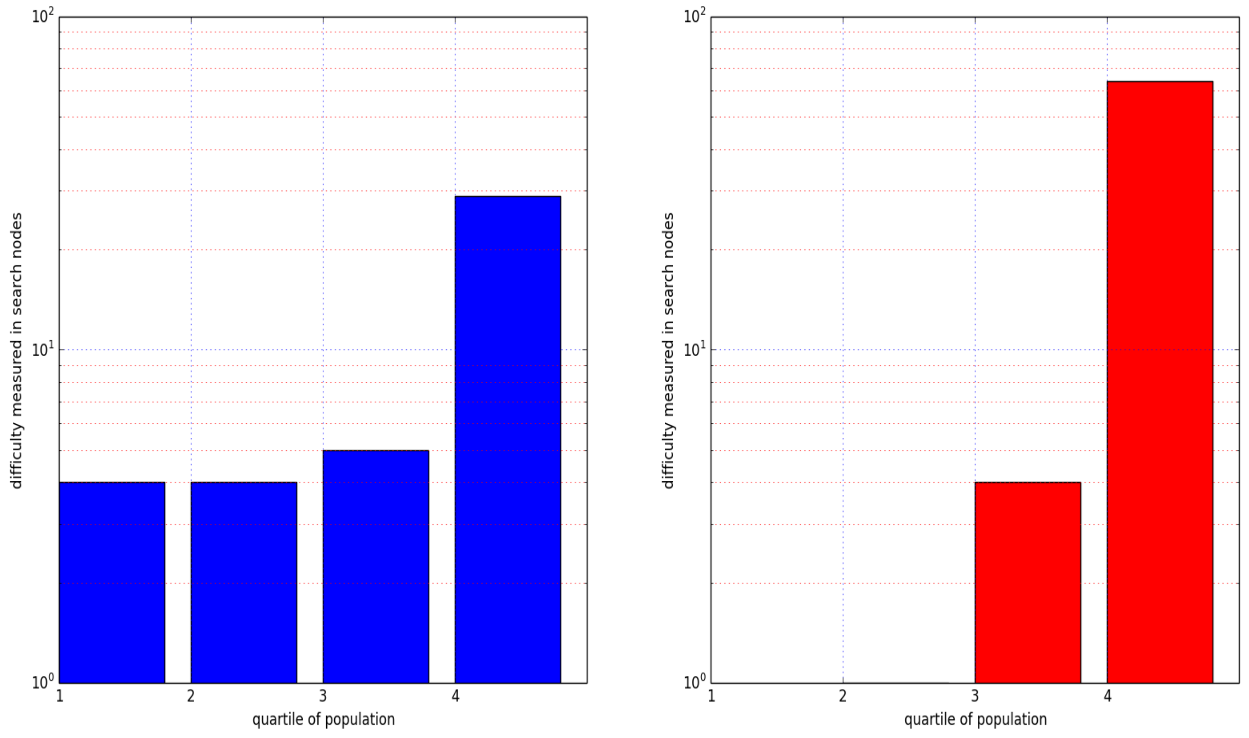


Figure 5.9: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in ppigo

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 ([19, 22, 15, 44, 6, 40]) is presented in [20]. The algorithms employ a heavy filtering approach using an index structure and run [12] 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 [20], 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 [6] 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 [20] took about 7 seconds for filtering and 200 milliseconds for verification and it is again GraphGrepSX [6]. The light filters approach has slower verification and slightly faster filtering.

The algorithms evaluated in [20] 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 2.3.3.

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

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

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

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

[8] 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.

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

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

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

[12] 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.

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

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

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] Ciaran McCreesh. Labels in randomly generated subgraph isomorphism problems, 2016.

[24] 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.

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

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

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

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

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

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

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

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

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

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

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

[36] 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.

[37] 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.

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

[39] 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.

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

[41] 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.

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

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

[44] 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. 37

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

depth-first search An algorithm for traversing or searching tree or graph data structures reported to be introduced by Charles Pierre Trémaux, a 19^{th} century French mathematician. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. 16, 17

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

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

index todo. 37

search node A search node denotes the number of recursive calls to the SIP algorithm taken to find a solution. 31, 38

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

suffix tree A suffix tree S is a compressed trie containing all the suffixes of the given text as their keys and positions in the text as their values. It has the following properties: the tree has exactly n leaves numbered from 1 to n; except for the root, every internal node has at least two children; each edge is labeled with a non-empty substring of S; no two edges starting out of a node can have string-labels beginning with the same character; the string obtained by concatenating all the string-labels found on the path from the root to leaf i spells out suffix S[i..n], for i from 1 to n.. 23

tree A tree is an undirected graph such that any two vertices are connected by exactly one path. In this work, we refer to the vertices of the tree as *nodes*.. 8, 9, 23

Acronyms

FC forward checking. 29

SAT Satisfiable. 6, 11, 26, 30–32, 34–36, 38

SIP subgraph isomorphism problem. 3, 4, 6, 10–12, 21, 24, 26, 29–32, 34–38

UNSAT Unsatisfiable. 4, 6, 11, 24, 30–32, 34–36, 38