Investigations of Subgraph Query Processing

Iva Stefanova Babukova

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

Level 4 Project — March 20, 2016

Abstract

Education Use Consent

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

Name:		Signature:	

Contents

Chapter 1

Introduction

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

1.1 Problem Statement

1.2 Aims and Motivations

Many substructure-searching problems call for repeatedly examining a large number of molecules (typically stored in a database), comparing each with a pattern. In such situations, it pays to spend some time ”up front,” storing the answers to specific questions for each structure in the database. Subsequent searches of the database use these pre-computed answers to vastly improve search time; the up-front computation time is paid back quickly as repeated searches are performed.

- graphs are widely used nowadays to represent data
- the graph containment problem is widely addressed in many areas of science: genetics, chemistry, XML documents, images, fraud detection and prevention (there was an article in nature about this)

In the core of many graph-related applications, lies a common and critical problem: *how to efficiently process graph queries and retrieve related graphs*. In some cases, the success of an application directly relies on the efficiency of the query processing system.

Applications:

- genome sequencing: find mutations responsible for rare diseases – nature vol 527 no 7576
- treating diseases like cancer: screen a patient’s tumor for a set of biomarkers to choose the best treatment to fight the particular cancer – nature vol 527 no 7578

1.3 Terminology, Definitions and Notations

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

1.3.1 Graph Theory

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

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

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

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

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

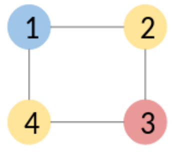


Figure 1.1: graph G_p

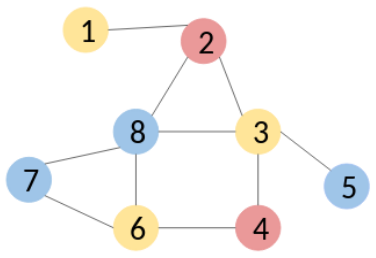


Figure 1.2: graph G_t

Figure 1.3: Instance of subgraph isomorphism problem (SIP)

1.3.2 The Subgraph Isomorphism Problem

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

The *SIP* between a graph $G_p(V_p, E_p, L_p)$, called pattern and graph $G_t(V_t, E_t, L_t)$, called target, is to find a function f that maps a different vertex of the target to every vertex in the pattern such that for every $v \in V_p$, v =

f(v) and L_p(v) = L_t(f(v)), where f(v) $\in$ V_t; and for all pairs of vertices u,v $\in$ V_p, (u, v) $\in$ E_p $\Leftrightarrow$ (f(u), f(v)) $\in$ E_t. In other words, a valid mapping of G_p to a subgraph in G_t preserves the labeling and the set of neighbours (and therefore the degree) of each vertex in G_p. If f exists, we say that the SIP instance is *satisfiable (SAT)*, i.e. it has at least one solution, otherwise it is *unsatisfiable (UNSAT)*, i.e. there are no solutions.

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

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

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

There are various existing algorithms for the subgraph isomorphism problem [6, 15, 14, 12, 5, 17]. More thorough analysis of the problem and discussion on existing work is presented in Section 2.8.

1.3.3 The Filter-Verification Paradigm

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

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

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

Another way of processing queries is the *filter-verification framework*, which consists of two steps. The *filter* step tries to prune targets that can not be matched to a given query. The set of targets left after this step formes the set of *candidates*, denoted as $\mathcal{C}_p$, where $\mathcal{C}_p \subseteq$ T. The following *verification* step then performs SIP for every pair (G_p,$\mathcal{C}_{G_p}$) $\in$ P $\times$ $\mathcal{C}_p$. The usage of the filter-verification paradigm is motivated by the fact that subgraph isomorphism is NP-complete and reducing the number of SIP calls by discarding targets that are UNSAT for a given pattern during the filtering step and performing SIP using the limited set of candidates would yield to significant performance improvement [10, 11].

There are number of subgraph query processing algorithms based on the filter-verification framework [10, 13, 4, 18, 8]. This approach is discussed in more detail in Chapter 2, where more definitions and annotations are introduced in Section 2.3 followed by analysis of related work in Section 2.7.3 and Section 2.7.4.

1.4 Report Organization

Chapter 2

Review of existing work

2.1 The nature of the problem

2.2 Datasets

In this work we consider undirected graphs. We assume that, in each graph, each vertex has a unique identifier and in the graph database each graph has a unique identifier.

This section gives more information about the datasets that were used to check the correctness and performance of the algorithm implementations. All graphs in these datasets are undirected.

AIDS is the standard database of the Antiviral Screen dataset of the National Cancer Insitute [2]. The database has 40 000 molecules, represented as graphs.

∗ A graph G is said to be disconnected if there exist two nodes in G such that no path in G has those nodes as endpoints.

The input we are working with is a file with the following format:

- put a table with the datasets and more information about them
- average, median, etc.
- say we work only with undirected graphs
- say whether we consider induced/ non-induced subgraph isomorphism

2.2.1 Subgraph Isomorphism Answers

Each query in each query set matches at least one graph in the target set. Some queries match more than one graph. Below we give some statistics on number of target graphs matched.

2.3 Filtering-verification paradigm

- smaller C, less time spent on subgraph isomorphism check, but more tie spent on indexing process

	aids	pcms	pdbs	ppigo
# graphs	40 000	200	600	50
# disconnected graphs∗	3 157			
# distinct node labels	62			
# distinct edge labels	0	0	0	0
avg # edges	46.95			
median # edges				
avg # nodes	45			
avg degree	2.09			
median # nodes				
avg # node labels	4.4			
median # node labels				

Table 2.1: Statistics about the datasets

dataset	number of graphs	unique vertex labels	average node degree	min node degree	max node degree
AIDS	40 000	62	2.09	ne znam	ne znam
PCMS	200	21	23.01	ne znam	ne znam
PDBS	600	10	2.06	ne znam	ne znam
PPIGO	20	46	10.87	ne znam	ne znam

Table 2.2: 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 2.3: Number of SIP instances for each dataset and how many of them are SAT and UNSAT

2.4 Motivation for usage

2.5 What makes an index ”good”

2.6 Common techniques

2.6.1 Graph-mining

2.6.2 Non-graph-mining

2.7 Methods with respect to choice of indexing unit

2.7.1 Path-based indexing approach

Follows the general idea: enumerate all the existing paths in a database up to *maxLen* length and index them, where a path is a vertex-edgeProperty-vetex sequence. *example* In order to create an index of a graph *g*, this approach breaks *g* into paths and in this way, the structural information of *g* could be lost. This leads to more false-positive answers returned after the Advantages:

1. Paths are easier to manipulate than trees and graphs.
2. The index space is predefined: all the paths up to *maxLen* length are selected.

Disadvantages:

1. Path is too simple: structural information is lost
2. There are too many paths: the set of paths in a graph database usually is huge.

2.7.2 Tree-based indexing approach

2.7.3 CT-Index

CT-Index [10] is divided into two main parts, filtering and verification, both described below. 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). CT-Index supports data sets with edge and vertex labels and also wild card patterns. Although not explicitly stated in [10], CT-Index addresses the non-induced subgraph isomorphism problem (definition **??**).

Filtering

During the filtering step, the features of all graphs in the target data set are extracted and saved to a file, i.e. the target index. The index is then used to filter out target graphs that cannot contain the pattern. Features are specific subgraphs used to classify graphs, and are stored as hash-key fingerprints. Features may be paths, subtrees or cycles of bounded length. Since vertices and edges may contain labels, these features can be viewed as strings from a specified alphabet (where the alphabet is the labels). In [10] it is stated that the reason for using trees and cycles (as well as paths) is that “trees capture additional structural information” and cycles “represent the distinct characteristic of graphs ... often neglected when using only trees as features”.

Although the time complexity of computing all features of a graph is not reported, it can be derived as follows. To extract a subtree of graph G with $maxT$ 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 (u, v) in G and vertices u and v that belong to G, such that one vertex (say, u) is part of the current tree and the other (say, v) is not. If we have n number of vertices in the current tree, each with degree d, then the size of F is at most $n.(d-1)$. $ExtendTree$ extends the current tree with a specified edge as parameter, generates F and makes a recursive call for every edge in F, until the tree reaches size $maxT$. In the start of the tree extraction when adding the first edge to the empty tree, the vertices on both ends of the edge are also added as part of the tree. Therefore, the size of F initially is $2.(d-1)$. After every recursive call, one more vertex is added to the tree, which introduces $(d-1)$ new edges. That makes a total of $maxT + 1$ vertices that will be added to the tree and $(maxT + 1).(d-1)$ visited edges. Consequently, the complexity of extracting tree features is $\mathcal{O}(|E|.(maxT + 1).(d-1))$. From this formula one can see that the number of edges in the graph has significant impact on the performance of the algorithm. When increasing the graph density, the algorithm will have slower performance, caused by the degree of each vertex and the total number of edges, which both will increase.

CT-Index computes a unique representation of each distinct feature, its *canonical form*, and stores its string encoding in the index file. 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, root 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 vertex 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 [10] the complexity of their canonical labeling can be derived as follows. Step (1) is $\mathcal{O}(n)$, where n is the number of nodes in the tree, as one needs to visit each node before removing it. The complexity of step (2) is as follows. We write $|p|$ for the number of interior nodes in the trees, which is equal to n minus the number of leaf nodes. Step (2) visits a node, then visits its parent, and for every child of the parent node checks whether it should be first or second in the canonical label, using the vertex and edge labels conditions described above. This is repeated for every node in the tree up to the root. Therefore, the complexity of step (2) is $\mathcal{O}(|p|.|c|^2)$, where $|c|$ denotes the number of children of a parent.

In [10] 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 that there exists data sets with less label diversity and with big trees as features that would result in poor performance. More specifically, as $maxT$ increases, or average degree increases, so too does the cost of step (2), and performance suffers (and we conduct experiments to test this hypothesis in section 2.7.3).

Fingerprints

CT-Index uses a storage technique called *hash-key fingerprint* to capture the features in the graph. A separate fingerprint is computed from the canonical labels for each graph in the database. A fingerprint is an array of bits and denotes whether a particular feature occurs in the graph or not. As there is no predefined set of possible features for each graph, reserving one bit for each feature in the feature set is considered infeasible[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.

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 [10] paper briefly discussed some optimization techniques that could be used to minimize the influence of collisions, but it is unclear whether CT-Index employs them. It is stated that "... the loss of information caused by the use of hash-key fingerprints seems to be justifiable

[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.

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. On the other hand, 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.

Verification

The verification step checks all candidates computed in the filtering step via a subgraph isomorphism test. A backtracking algorithm [3], similar to VF2 [6] 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 [4], gCode [13] and Tree+Δ [18]. Most papers claim that VF2 is "state of the art". However, this is not the case ([15, 12, 5, 17, 14]. VF2 has been shown to perform erratically and poorly [14]. Therefore we might summarise CT-Index architecture as using a potentially expensive indexing and filtering stage in order to minimise the computational cost of using an outdated subgraph isomorphism problem (SIP) algorithm.

Performance

This section describes the performance We searched for more information in alternative sources. In [11] the authors compare several well-established indexing techniques, including CT-Index and try to draw conclusions on their performance depending on a set of key-factor parameters. The authors use the four datasets described in section 2.2 as well as a synthetically generated database of targets and queries to conduct their experiments. These are some of the conclusions made after the experiments:

- The size of the query affects the performance of the indexing method
  This conclusion is also very easy to obtain. As the size of the pattern becomes larger, there are more nodes that have to be matched to a target node. Suppose that we have a pattern graph p with $|p|$ number of nodes and a target graph t with $|t|$ number of nodes that is a candidate for subgraph isomorphism. Assume that all techniques to discard t before the search have failed, i.e. $|t| \geq$ to $|p|$, etc. Then, for each node in p, the ct-index algorithm tries to map to a node in t and continues until either it either fails or succeeds. In the first case, the algorithm backtracks and tries to map an alternative node from t and in the second case, the algorithm terminates, as a valid map of all pattern nodes to some of the target nodes is found. One can easily notice that the bigger the size of $|p|$, the more checks the subgraph isomorphism algorithm will do. The same conclusion could be reached for $|t|$. When there are more nodes in t, more different ways exist for matching a node in p to a node in t.
- CT-Index has relatively small index size, compared to other indexing techniques.
  Again, this result could be obtained even without running any experiments. As the CT-Index uses bitset arrays to store the features, it straightforward to reach to the conclusion that CT-Index always has smaller index size, compared to other techniques that do not compress the features, but directly store them in the index file.

The results from [11] have a major limitation: for each indexing methods, the authors use the "default" input parameter values, given by the original authors of each of the indexing methods. They never experiment with alternative values. It is not clear why the corresponding input values are fair and the best ones to use and why the authors do not consider the possibility of obtaining completely different results if they changed the input parameters values. Also, the authors never say whether they have checked how they verified that the indexing methods compared in the paper give correct results.

We conducted a separate evaluation using the implementation and the datasets descried in section 2.2. Table 2.4 shows the running time of CT-Index: time to build the index, time to compute answers out of the set of candidates and the total running time depending on the values of the parameters specified for the AIDS dataset, namely the size of the fingerprint, the maximum bound for path, subtree and cycle size; and the number of candidates for after indexing and filtering for the corresponding query number. Table A.1 shows the actual number of targets that contain the particular query as subgraph. This can be used to check the ratio of false-positives against real results for each query for each of the different input parameter values. We write -1 as the value of maximum path, cycle or subtree length whenever we do not want to use the corresponding structure as feature. We recorded the running time of both indexing and querying, as well as the number of candidates, because in real applications, the index is computed once and then reused multiple times, until the target dataset is changed. This means that for applications that require frequent changes to the database it is desirable to compute the index as fast as possible, whereas for datasets that are rarely changed, the time to compute the index is not that important and one may focus on increasing the quality of the index (i.e. it leads to smaller number of candidates).

To verify the correctness of the results obtained by CT-Index and later on be able to use the implementation as a benchmark, we did the following activities. As we did not have set of answers for each query, we first computed the answers for each query, varying the input parameters values. We verified that all results are the same by executing a script that checks whether every set of answers a query is the same as every set of answers obtained for the same query, but with different parameter values. Finally, we used a subgraph isomorphism solver program that does not employ indexing to compute the answers for each query and we compared these answers with the CT-Index answers to conclude that they are the same and CT-Index is correct at least for the corresponding input parameter values.

Table 2.4: 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

Looking at rows 1 and 2, we can conclude that extracting paths as features as well as subtrees and cycles leads to significantly slower index built time and slightly worse filtering power. Comparing row 1 and row 3, one can see that extracting only paths takes significantly lower amount of time, compared to using only trees as features. However, it leads to worse filtering power. The results from rows 1, 2 and 3 support the claim that trees are much more descriptive features than paths. Also, they show that using paths, subtrees and cycles all together as features leads to slower running time.

As it can be seen that the rows with the smallest number of candidates are the ones that store trees in the index file (rows 1, 2 and 6), these are also the instances with the slowest index built time. The reason could be that trees are more complex than paths and require more time to be extracted and transformed to unique string representations. As explained in the previous section, the authors use algorithm that computes the canonical form the the features of high complexity. Maybe if the authors used more optimal algorithm, it would lead to faster running time.

As it can be seen in column 5 of table 2.4, the total running time of CT-Index varies from 49 seconds to 114 seconds depending on the specified input parameters. If we look at rows 4 and 5, the only difference of the input parameter values is the fingerprint size. As it can be seen from the table, the time difference to build the index in 4 and 5 is only around 200 miliseconds. However, the time taken to compute the answers from the set of candidates is more than 6 seconds slower in row 5 compared to row 4, although the number of candidates for the two rows are almost the same for every query except query 3. Similar difference can be observed when comparing row 1 with row 6. The reason for these results could be that fingerprint of size 2048 is too small for the particular dataset and this leads to more collisions.

To conclude, in CT-Index, when using only paths as features leads to smaller running time for building index, however it significantly bigger number of candidates and therefore bigger amount of time to find all targets containing the query from the set of candidates. Using trees as features results to an index with much better filtering power, but slower to compute. Hashing features constructed from paths and trees does not give any benefit, moreover, it it slower and results to slightly bigger number of candidate graphs.

2.7.4 gIndex, gCoding .... mention them

2.8 Subgraph Isomorphism Algorithms

2.8.1 Partick and Ciaran's paper

2.8.2 Christine's paper

Chapter 3

Path-based query processing algorithms based on filtering-verification

The order of a graph is the cardinality of its vertex set. We write V(G) for the vertex set of a graph G.

3.1 Benchmarks

This section includes more information about the already existing indexing algorithm implementations that were used as a standard point of reference against our graph indexing implementations.

3.2 Index algorithms

All algorithms described in this section generate the candidate set of graphs in the following steps:

1. Compute the index of all graphs in T;
2. Compute the index of all patterns in P;
3. Using the target and pattern indexes computed in the previous two steps, extract all graphs in T that contain all features in the pattern index;
4. All extracted graphs from step 3 form the candidate set C.

In this work, we consider only induced subgraphs (definition **??** in section **??**) and the induced version of the subgraph isomorphism problem (definition **??** in section **??**)

The correctness of the results is checked comparing the answers obtained from CT-Index. We implemented a class called $Verify.java$ that checks whether the resulting candidate set after each query execution contains the answers of subgraph isomorphism. $Verify.java$ was very useful for discovering a lot of bugs during the software development process.

3.3 Path Index

This section gives an overview of the first indexing technique that was designed and implemented. In following subsections we describe the main idea of the algorithm ans the implementation. The performance results that were obtained after running various types of queries on targets of varying size are described along with a summary of the advantages and limitations of the algorithm.

3.3.1 The idea

The algorithm uses an exhaustive path-enumeration approach to build its index. This technique is employed by various algorithms like CT-Index [10], gCode [13] and GraphGrepSX [4]. The main idea behind *exhaustive path-enumeration* is to enumerate all existing paths in a database up to $maxL$ length and index them, where a path is a sequence of vertexes such that each vertex is connected with an edge with the previous and the next vertex in the sequence.

Given a set of target graphs T and a pattern graph p, $Path\text{-}index$ first computes the index of all graphs in T by enumerating exhaustively all paths in every target up to a specified maximum path length $maxL$. The paths are then stored in a file, which is the target$'$s index. The same procedure using the same value of $maxL$ is followed to derive the pattern index. The two files are then used for finding the candidate set C of all graphs in T that have to be checked for subgraph isomorphism with p. Using the pattern index, we check which targets contain all paths in the pattern index and filter out all targets that do not have all paths that the pattern does. The rationale behind this filtering method is that if some features of graph p do not exist in a graph t, t cannot contain p. However, if t contains all features of p, this does not meant that p is subgraph of t: t can be a *false-positive* and a subgraph isomorphism test needs to be performed on t and p to identify whether t is a *false positive* or it indeed contains p. All targets in T that have all paths, extracted from the pattern graph, form the candidate set C. All graphs in C are then checked for subgraph isomorphism with p.

As mentioned before, each path is a sequence of vertices of maximum size $maxL$, such that each vertex is connected with an edge with the previous and the next node in the sequence. Each path, extracted from the graph, is stored in the index file in a string representation, derived from the label of each node in the sequence and the label of the edge (if existent) between the node and its neighbors. In this work, we refer to these path string representations as path-strings. To avoid redundancy index comprises of unique path-strings, computed from the paths taking the following points into consideration:

- Path a is equivalent to its reverse variant $a\text{-}reversed$, because the graphs in the datasets are undirected, as mentioned above. There is no need to store both a and $a\text{-}reversed$ in the index, as this would lead to redundancy.
- Path a is not equivalent to a path $a\text{-}sorted$, where $a\text{-}sorted$ is the sorted variant of a in lexicographical order (either increasing or decreasing). Sorting a can change the places of some of the nodes that comprise the sequence, which means that some nodes may have different neighbors in the sorted path. This introduces new edges in $a\text{-}sorted$ that may not exist in a or in the graph.

To avoid redundancy and make the index search faster, only one unique path-string is written to the file for each path and its equivalent path-string variants, derived using the properties mentioned above. We chose to add to the index the lexicographically smaller path-string between a path and its reverse.

Let the graph on figure 3.2 be a target graph in T and the graph on figure 3.1 be a pattern graph in P, where the number in red next to each node on the two figures denotes the id number of the corresponding vertex. Let $maxL$ be equal to 3. In order to check whether **??** is a candidate for subgraph isomorphism with 3.1, we first compute the indexes of **??** and 3.1 for the specified maximum path length. The target index consists of the paths on figure 3.3 and the pattern index contains the paths on figure 3.4. There is a lexicographically smaller variant of the path-string H-C-H, namely C-H-H. However, no path in the the graph **??** can form such path-string and adding C-H-H to the index file could lead to wrong results.

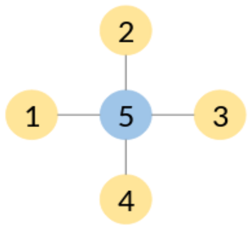


Figure 3.1: Pattern graph P

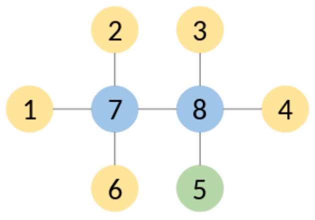


Figure 3.2: Target Graph T

After candidate extraction, Path-Index returns target graph 3.2 as candidate for subgraph isomorphism with pattern graph 3.1, as all paths in 3.1, shown on figure 3.4 are contained in **??**, as it can be seen on figure 3.3.

```
H
C
C - H
C - C
C - C - C
C - C - H
H - C - H
```

Figure 3.3: Target graph path enumeration

```
C
H
C - C
C - H
H - C - H
```

Figure 3.4: Pattern graph path enumeration

Figure 3.5: All paths up to maximum length 3

3.3.2 Implementation

Graph representation

Each graph is represented with a Java class Graph that has an integer id and a collection of node objects as fields, where the integer is a unique identifier. Each node object has an id, a label and a list of edge objects, where the node is a source node for each edge contained in the list. The length of the list of the edge object is equal to the degree of the node. Therefore, each edge object does not need to keep a record of the source node: it only has a label and a destination node 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 nodes as destination node and then added in the list of edges of the other node: the source node. The second edge object will have the source node of the first edge object as destination node and included in the list of edges of the destination node of the first edge object, which will be its source node. The label of both edge objects will be the same.

Path Extraction

A class, called PathExtractor, contains all functionality for computing the paths and path-strings that are stored in the index. To compute the index of a given graph dataset, we use the functionality of PathExtractor to extract all paths of every graph in the dataset. The path extraction algorithm is recursive depth-first-search based and is called on every node of the graph. Algorithm 1 describes our approach. Before we call function generatePath, we initialize an empty stack that is used in the algorithm to store visited nodes. Algorithm 1 generates all paths in the graph of size $maxL$, where $maxL$ is the number of nodes in a path, using two functions: generatePath and generatePathInner. The function generatePath in Algorithm 1 takes a list of all nodes in the graph and path length $maxL$ as parameters and calls generatePathInner for every node in the list as starting node (line 4). The start node is also pushed on the stack (line 3), as it is part of the path that is to be generated.

Algorithm 1 Paths extraction

```
1: procedure GENERATEPATH (nodes, $l_{max}$, s)
2:     for node in nodes do
3:         s ← init                                   ▷ initialize the stack
4:         push node on top of s
5:         GENERATEPATHINNER(node, $l_{max}$, s)
6:     end for
7: end procedure
```

The recursive function generatePathInner generates all paths is length $maxL$, where the length of a path is equal to the number of all nodes that are part of it, from a start node $node$. First, the size of the stack is checked. If it is less or equal to $maxL$, the contents of the stack are copied and passed to a function putToIndex that computes the string representation of the extracted path and puts it to the index. If the stack is equal to $maxL$, the last element from the stack is popped out and

Algorithm 2 Paths extraction

```
 1: procedure GENERATEPATHINNER (node, $l_{max}$, s)
 2:     if s $\leq l_{max}$ then
 3:         new path $\leftarrow$ s                                ▷ new path of size is found
 4:     end if
 5:     if s == $l_{max}$ then
 6:         s.pop()
 7:     end if
 8:     for neighbour of node do
 9:         if neighbour not in s then
10:             s.push(neighbour)
11:             GENERATEPATHINNER(neighbour, $l_{max}$, s)         ▷ recursive call
12:         end if
13:     end for
14:     if s is full then                   ▷ all neighbours of node are on the stack
15:         s.pop()
16:     end if
17: end procedure
```

the function exits. If the size of the stack is less than $maxL$, we traverse the neighborhood of $node$ in the for loop, starting at line 12. Every unvisited neighbor of $node$ is pushed on the stack and then a recursive call to generatePathInner is made with the same neighbor node as $node$(lines 13 and 14).

The stack used in Algorithm 1 follows the principles of the stack datastructure. In this work, we use the Java implementation. The stack has two main functions: to keep track of the visited nodes in the current function call so that no node takes part in a path more than twice, i.e. there are no cycles; and it contains nodes that always form a valid path. The reason why this is the case is that every time a node is pushed on top of the stack, this node is one of the neighbors of the last node on the stack. The only place in generatePathInner function where new node is added in the stack is in line 14 and the node that is pushed on top of the stack belongs to the list of neighbors of $node$, which is always the previous node in the stack. Therefore, the sequence of all nodes on the stack forms a valid path, as every node is connected with an edge with its predecessor and successor in the sequence. The complexity of Algorithm 1 is explained in section 3.3.3.

The function outputPath is called by generatePathInner every time the stack reaches a size equal to $maxL$ (line 8, Algorithm 1). Algorithm 2 derives the lexicographically smallest string representation of a path, following the principles described in section 3.3.1. The resulting path-string is added to the index, if it does not exist there (line 8, Algorithm 2).

Algorithm 2 takes a sequence of nodes as a parameter, which are the contents of the stack when the stack size is $maxL$ (line 8, Algorithm 1). We derive the path-string of the sequence (line 2) that consists of the node label of every node in nodeSequence and the label of the edge it has with the previous node in the path. If the dataset does not have edge labels, a default character is used as label for every edge. In the previous examples, we use the character " $-$ " (figures 3.3 and 3.4). It is important to note that the node labels are ordered in exactly the same way as the nodes in the sequence, for example if a node A is on place *i* in the sequence, the label of A will be at place *i* in the string if the edge labels in the string are not counted (if they are counted, the label of A is at position *i* x *2*).

Next steps are to compute the reversed sequence of the path (line 3), its string representation (line 4) and check whether the reversed string is lexicographically smaller then the path-string (line 5). If yes, we assign the path-string to be equal to the reversed string (line 6) and add it to the index, if it is not present there yet (line 8). We do not sort the path-string lexicographically, due to the reasons explained in section 3.3.1. The complexity of Algorithm 2 is explained in section 3.3.3.

Algorithm 3 Output path procedure

```
1: procedure GETPATH (nseq)
2:     pathStr ← nseq.toString()        ▷ returns a string of the labels of the nodes in nseq
3:     rseq ← nseq.reverse()        ▷ reverse the order of nodes in nseq
4:     pathStrRev ← rseq.toString()        ▷ returns a string of the labels of the nodes in rseq
5:     if pathStrRev < pathStr then
6:         pathStr ← pathStrRev        ▷ put to index the lexicographically smaller string
7:     end if
8:     if pathStr not in index then
9:         add pathStr to index
10:    end if
11: end procedure
```

Candidates extraction

Candidates extraction is the part of the implementation where the set of candidates C from the dataset with target graphs is constructed. Algorithm 3 describes our approach in more detail. Function candidatesExtractor takes the index of one graph from the target dataset and the index of the pattern (or one of the set of pattern graphs, if there are more than one pattern) as parameters (line 1). For every path-string in the pattern, we check whether it is contained in the target index (line 3). If yes, the algorithm continues, because the target might be a candidate (line 6). Otherwise, if the pattern path-string does not exist in the target index, the program returns false and terminates (line 4). The program goes to line 7 iff all path-strings in the pattern are contained in the target index.

The complexity of Algorithm 3 is explained in section 3.3.3.

Algorithm 4 Candidates extraction

```
1: procedure CANDIDATESEXTRACTOR ($\mathcal{I}_{G_p}$, $\mathcal{I}_{G_t}$)
2:     for path$_p$ in $\mathcal{I}_{G_p}$ do
3:         for path$_t$ in $\mathcal{I}_{G_t}$ do
4:             if no path$_t$ is equal to path$_p$ then return false        ▷ path$_p$ is not in $\mathcal{I}_{G_t}$, return false
5:             end if
6:         end for
7:     end for
8: end procedure
```

3.3.3 Performance

This section gives more details about the complexity of each of the algorithms in the implementation, explained in sections 3.3.2 and 3.3.2, continued with statistics about the algorithm running time and performance after running it on various datasets. Following the results obtained, we make a conclusion about the observed performance of the algorithm and analyze its advantages and disadvantages.

Complexity

Algorithm 1 works like depth-first search, but the depth of the search has a maximum limit $maxL$. For each target graph, we conduct depth-first search of depth $maxL$. During the search, each neighbor of $maxL$ number of nodes is visited. The worst case complexity is when the target graph is a clique. If we assume that the number of nodes in the clique is n (equal to the number of edges), then the for loop on line 2 will be executed n times, each time calling generatePathInner function. This function will make recursive call for every unvisited neighbor of the current vertex, until the stack reaches size equal to $maxL$. As after every recursive call, the size of the stack is increased by 1 and the graph is a clique, the number of neighbors that will be visited during the next recursive call decreases by 1. The depth of the recursive calls is at most $maxL$ (the maximum size of the path we want is $maxL$), therefore the function has worst case complexity **O((n - 1).(n - 2) ... (n - maxL - 1))**, which occurs when the graph is a clique.

Algorithm 2 does not have any loops, but it calls the function toString(), which traverses the given path once to change each node with its label and an edge label to the next node in the sequence. The toString() method visits every node in the path, takes the label of the current node, and if the dataset has edge labels, finds the edge label of the edge between the current node and the next node, otherwise adds a default character after the node label. On line 7, there is a traversal through the current index of a graph. The complexity of Algorithm 2 is therefore **O(maxL + index size)**, where index size is the current size of the index for the current graph being indexed. The best case is when the index is empty: then traversing the index takes constant time and the complexity of the toString() method is **O(maxL)**.

Algorithm 3 visits every path stored in the pattern index and for each pattern path, it goes through the target index and checks whether it contains the same path. This procedure is done for every target graph in the dataset. If we assume that the number of paths in the pattern index is *p*, the number of paths in the current target index is *t* and the number of graphs in the target dataset is *n*, the number complexity of the algorithm is **O(ptn)**.

3.3.4 Experimental Results

TODO

3.3.5 Advantages and Limitations

TODO
- structural information is lost, because path is too simple
- it is computationally intense, i.e. very slow especially when computing paths of higher length; there are too many paths
- very easy to implement
- paths are easier to manipulate than trees and graphs — why?
- this indexing technique is good when we have graphs with large number of different labels on their nodes and low density of edges between the graphs' nodes
- the index space is predefined: all the paths up to `maxLen` length are selected

3.4 Path-Subtree Index

$Path$-$Subtree$ index is the second indexing technique that was designed, implemented and evaluated. In this section we explain its main idea, implementation and performance, observed after testing with various types of graph data sets. We write about the problems with the algorithm we discovered after the initial implementation and how we changed $Path$-$Subtree$ index to solve the issues encountered. Lastly, we draw conclusions about the advantages and limitations of this indexing technique.

3.4.1 Initial idea

As mentioned in section 3.3.5, $Path$ index can't extract most of the structural information present in the graphs. It is almost of no use for targets that have small number of different labels, like the AIDS data set for example. $Path$-$Subtree$ index addresses the problem of insufficient structural information extraction by introducing a novel representation of the paths that takes into account the neighborhood of each node. A new version of the label of each node that is present in the path is computed and stored in the index instead of the node$'$s original label. In this work, we refer to this alternate label as *neighborhood label*. The next paragraphs introduce necessary naming conventions and then describe what *neighborhood label* means and why we believe it helps to derive a better index.

Let n be a node with neighbors N = { n_1, n_2 ...n_i }, where i is the size of N. Let us define a labeling function *l* that maps the node n and each node in N to a character, also called the node label. We refer to the label of n as L_n and the label of each node n_j in N, where j is between 1 and i inclusive, as L_j. It is important to note that using this notation we do not mean that each node has unique label, i.e. there may exist nodes n_g and n_h, where g and h are between 1 and i inclusive, and L_g is the same as L_h.

The term neighborhood label is a specific label that is computed for each node in the graph using the label of each node and its neighbors: all nodes, connected with an edge with it. The label derived is then stored in the index file as part of the string representation of the nodes in the paths, similarly to the node labels that are part of the string representations of paths in the previous index algorithm in section 3.3.

The *neighborhood label* of n is derived in the following way. Let the label of n and the labels of all members of N form the set of labels S. The *neighborhood label* of n is constructed using the labels in S, ordered lexicographically from the smallest to the largest label in S. For instance, we take each member of S and derive the sequence S′ = $s_1s_2s_3 \dots s_i$, where i is the number of the neighbors of n and consequently the number of members of the set of labels S. S′ is the resulting *neighborhood label* of n. For example, the *neighborhood label* of each node of the graph on figure 3.1 is shown on figure **??**, where the red number on the left side of each node is its id. From figure 3.1 it can be seen that node with id 4 has neighbors nodes 0, 1, 2, 3 each of them with label H. The label of node 4 is C and after appending the labels of its neighborhood, the resulting *neighborhood label* is CHHHH. Similarly, the label of node 4 is part of the *neighborhood label* of the other nodes.

The resulting path-strings from path extraction of **??** for $maxL$ equal to 3 are shown on figure **??**, where the the character " $-$ " denotes an edge. It is important to note that although the path representation stored in the index file is changed, the original node labels of the graph are not removed so that the structure and labeling remains unchanged. Also, the paths on figure **??** are composed from the same nodes as the paths of figure 3.4. The difference of the representation comes only from the alternative labeling model. The method of extracting the paths remains the same as the one used for $Path$ index and described in section 3.3. Similarly, computing the set of graphs possibly subgraph isomorphic to a pattern graph p, also called the *candidate set*, is done using the same method as $Path$ index.

Like $Path$ index, $Path\text{-}Subtree$ index supports graphs that have labels on edges and nodes. When deriving the representation of each path that is later stored in the index, we include the label of the edge that connects every two nodes a and b that are part of the path. If we take the graph on figure **??**, we notice that there are no labels of its edges. Consequently, the paths stored in the target index, also shown on figure **??** have the character " $-$ " between the *neighborhood label* of each two nodes. If graph **??** had edge labels, we would include the corresponding label instead of " $-$ ".

Theorem 3.4.1. *The set of target graphs filtered by path index is always a subset of the targets filtered by path-subtree index.*

Proof. Prove this

□

- the path extraction is done in the same way only labels are changed

With the following example we show that $Path\text{-}Subtree$ index discriminates graphs that $Path$ index can not. Let us consider the graphs on figure **??** and 3.1. Let **??** be the target graph and 3.1 to be the pattern. Before running the subgraph isomorphism test on the two graphs, we will index them and see whether we can filter out **??**. As we can see, **??** does not contain 3.1 and we are going to check whether Index-2 will return the target graph as a false-positive candidate for subgraph isomorphism with our pattern. After computing the indexes of the two graphs, we get the results for **??** shown on figure **??** and the results for 3.1 on figure **??**. As we can see from the figures, the index of our pattern graph has paths that the target does not contain, for example CHHHH, CH-CHHHH, CH-CHHHH-CH. The target graph is not returned as a candidate for subgraph isomorphism. Computing the index using $Path$ index results in getting the paths, displayed on figure **??** for our target graph **??** and the paths on figure 3.4 for the pattern. As all paths of 3.1 are contained in the index of **??**, $Path$ index returns **??** as candidate for subgraph isomorphism.

3.4.2 Problems with Path-Subtree index

After we implemented *Path-Subtree* index based on the principles described in the previous subsection, we discovered that our algorithm is wrong. Apart from the expected false positives, it had false negatives. False negatives in this case are target graphs that contain the pattern, but are wrongly filtered out after the graph indexing step and are never included in the candidate set. The existence of false negatives is strongly undesired, as it is an indication that the indexing algorithm is wrong and therefore useless. We discovered what the reason for discarding targets wrongly is. In the following paragraphs we explain in more details the problem using a worked example and propose a modification of the algorithm that has the potential to solve the issue.

Let the graph on figure **??** be our target graph and the graph on figure 3.6 be the pattern. As we can see from the two figures, **??** contains 3.6 and we expect that a correct indexing technique will return 3.6 as a candidate for subgraph isomorphism check. After computing the indexes of the target and the pattern graphs, we derive the results for the target, shown on figure **??**, and for the pattern, shown on figure 3.7. As we can see from the two figures, although the target contains the pattern, the paths in their indexes are different. As the target index does not contain paths existent in the pattern index, like `CC, CC-CCHHH` and `CH-CCHHH-CC` for instance. In this case, **??** is a false negative, discarded wrongly because of our incorrect indexing technique.

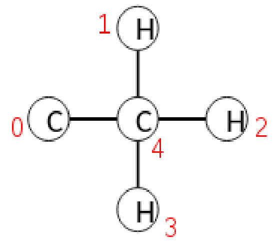


Figure 3.6: Pattern graph

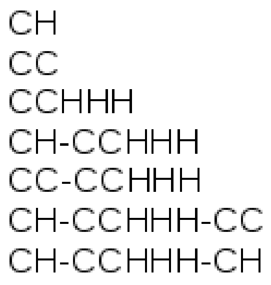

CH
CC
CCHHH
CH-CCHHH
CC-CCHHH
CH-CCHHH-CC
CH-CCHHH-CH

Figure 3.7: Subtree-label paths

3.4.3 Algorithm Refinement

In order to solve the problem with false negatives, we changed the candidate extraction part. For every path pp in the pattern index, we check which target graphs contain pp as a path. Let us have pp = `CH-CCHHH` and we want to check it against the target path tp = `CH-CCCHHH-CHHH`. For every *neighborhood label* in pp, our algorithm will whether it is contained in the corresponding label in tp. We first compare `CH` and `CH`, and as they are equal, we continue with the labels on the next position: `CCHHH` and `CCCHHH`. As `CCCHHH` contains `CCHHH` and this is the last label in pp, we stop and return true, tp contains pp.

Using the refinement described above, we removed the problem with false negatives, described in section 3.4.2. However, this results in weaker filtering power. Now, there are some cases where *path-index* would perform better, although it extracts less structural information about the graphs. We will show how our refined method introduces more false-positives using an example. Let us consider the two graphs on figures 3.8 and 3.9. Let us apply the refined *Path-Subtree* indexing method for the two graphs, where 3.9 is the pattern and 3.8 is the target (clearly, 3.8 does not contain 3.9 as subgraph). The *neighborhood labels* of the target and the pattern for $maxL$=3 are shown on figures 3.10 and 3.11 respectively. Applying the new candidate extraction technique, we will have 3.8 returned as candidate, because it contains all paths in 3.9.

Let us examine in more detail the only path of length 3 in the pattern graph. It is composed from the nodes with ids `0, 1` and `2`. Using *Path-Subtree* index method, this path is represented as `CH-CCH-CH` in the index file. If we used *path-index*, we would store this path as the string `C-H-C`. Although the path representation *neighborhood label* gives us more information about the node neighbors, with us we loose important structural information, which is: the label of the actual node. In the string `CCH`, either of the letters could be the label of the node. As shown in the example above, this confusion leads to more false positives.

TODO - we changed the string representation of the isomer label. First letter is always the original label, the others are from the neighbors, sorted in lexicographic order

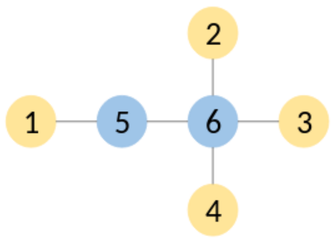


Figure 3.8: Graph A

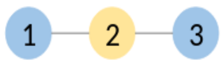
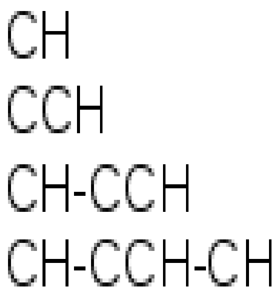
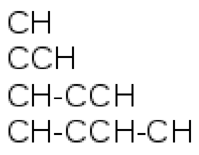


Figure 3.9: Graph B



CH
CCH
CH-CCH
CH-CCH-CH

Figure 3.10: Neighborhood label paths of A



CH
CCH
CH-CCH
CH-CCH-CH

Figure 3.11: Neighborhood label paths of A

3.4.4 Implementation

- extend IB-Index 1 and put an option to extract paths using isomer labels.
- describe the extension of the project to support both types of indexing.
- when parsing the graphs, after parsing each graph, compute the isomer label of every node for this graph
- additional field in Node class for isomer labels
- we have a class Path to represent the paths. Each path is an array of nodes. There is a toString method that returns the desired string that we are to store in the index. Depending on the option, we can return path from the labels (ib-index 1), path from the isomer labels (ib-index 2) or a path from the id of each node, made for testing purposes There is a method to reverse the path and it is used when storing the paths in the index: we always store the lexicographically smaller path, comparing each path with its reversed equivalent.
- re-written the candidates extraction part. We have to check for containment ...

Chapter 4

Light Filters

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

Light Filters is an algorithm for subgraph query processing and it is based on a modified version of the filter-verification paradigm. Here, filtering and verification steps are applied separately for each SIP instance. This approach uses simple filtering tests that require much less computational effort (i.e. lighter filtering). More importance is placed on the quality of the SIP algorithm than on the filters, because the major computational effort goes for solving the non-filtered SIP instances.

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

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

Algorithm 5 Light filters algorithm

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

4.1 Trivial Failures

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

4.1.1 Neighborhood Degree Sequence

The neighbourhood degree sequence of G and the degree sequence of labels in G correspond to four of the trivial tests. Very similar approach of using the degree sequence of vertices to reject UNSAT SIP instances is used in the filtering part of the algorithm presented in [15].

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

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

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

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

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

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

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

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

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

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

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

1. Suppose that the first condition is false. Then, G_p must have at least one vertex more than G_t, i.e. at least one vertex $\in G_p$ will be unmatched. Therefore, SIP (G_p, G_t) is UNSAT if the order of G_p is bigger than the order of G_t and we reach a contradiction. Therefore, this condition must always hold.
2. Suppose that the second condition is false so that there exists a label $l' \in L_{G_p}$ with $l' \notin L_{G_t}$. Therefore, there exists a vertex $v \in G_p$ that is assigned the label l' and there is no vertex in G_t with label l'. From this follows that v can not be matched to any vertex in G_t and SIP(G_p, G_t) is UNSAT, which leads to a contradiction. Therefore, condition 2 must hold.

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

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

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

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

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

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

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

4.1.2 Domain wipe out

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

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

4.1.3 Order of Tests

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

4.1.4 Implementation

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

Failures 1 and 2 are the fastest: each of them takes time $\mathcal{O}(1)$ to compute. For failure 1, one needs only to return the sizes of the number of vertices in G_p and in G_t. For failure 2, for every graph G, we have an array that stores all unique labels that occur in G. The size of this array is known after the graph is read from the file. To check whether test 2 is true,

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

Table 4.1: Specification of the measured failure types

one needs to compare the size of the labels array d_p of G_p with the size of the labels array d_t of G_t. Finding the values of d_t and d_p takes time $\mathcal{O}(1)$.

Failures 3 and 4 have slower running time. For every label $l \in G_p$, failure 3 occurs if l is not a label in G_t. Checking whether l is a label in G_t involves iterating over the labels in G_t, which is of time $\mathcal{O}(d_t)$ worst case, if l is the last label in G_t or it is not present. Therefore, the overall complexity of failure 3 is $\mathcal{O}(d_p.d_t)$. Algorithm 7 shows the pseudo code for the "subsumes" procedure. Its complexity is $\mathcal{O}(plds)$, where plds is the lds of label $l \in G_p$. Therefore, the overall complexity of failure 4 is $\mathcal{O}(plds.d_p)$.

Failure 5 is the most expensive one. It takes time $\mathcal{O}(m.n)$, where n is the order of G_p and m is the order of G_t. Algorithm 6 has to visit each vertex $v \in G_p$, initialize dom_v and for every $w \in G_t$, check whether w can be mapped to v. The two checks (line 13) take time $\mathcal{O}(1)$.

Algorithm 6 Lights Filters

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

Algorithm 7 G_t Subsumes G_p

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

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

- Class Graph
  Creates graph (G) from a file. A Graph object has size, order, id, array of the degree of each vertex, bitset array of the neighbours of each vertex, where the ith element in the array contains the neighbours of vertex i in the graph and array of labels, where similarly, the element in position i contains the label of vertex i. While reading in the graph, we initialize each field, which takes time $\mathcal{O}(size + order)$. When the array of labels is built, a new object for each unique label *l* is created and the nds of the graph is computed.
- Class Label
  This class represents a label in a Graph object. A Label has a name and lds which is an array of integers, sorted in non-increasing order. The degree sequence array is built using insertion sort algorithm which is of complexity $\mathcal{O}(n^2)$, where n is the size of lds [7].
- Class SIP1
  This class implements the light filtering procedure displayed in Algorithm 6 as well as the SIP algorithm, which is explained in more detail in the next Section.

4.2 SIP1 Implementation

SIP1 is based on the simplest of the Glasgow algorithms [14]. Given a G_p and G_t, SIP1 has a variable for each vertex in G_p, each with domain that is the set of compatible vertices in G_t (alldoms, Algorithm 8, initialized in Algorithm 6). Compatible vertices have the same labels and the degree of the target vertex is greater than or equal to the degree of the pattern vertex. Bit sets are used to represent the domains and the adjacency matrices of the graphs. When a pattern variable u is instantiated with a target value i (Algorithm 8, line 7), all uninstatiated (future) variables have i removed from their domains (Algorithm 8, line 12). If a future variable v is adjacent to u in G_p then the domain of v becomes the intersection of the current domain of v with the neighborhood of vertex i in G_t. This constraint is enforced by applying a logical and operation between the two bit sets (Algorithm 8, line 14). SIP1 uses forward checking (FC) with fail first heuristic [9]: for all uninstantiated variables representing pattern vertices, it selects to explore the one that has the smallest domain before the others (Algorithm 8, line 4).

Class SIP1 contains the implementation of the verification step. For every SIP(G_p, G_t) instance, if it is not discarded after filtering, it goes to the SIP1 procedure (line 21, Algorithm 6). Algorithm 8 is a pseudocode of the implementation.

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

Algorithm 8 SIP1

```
procedure SIP1 (alldoms)
    if alldoms is empty then return solution                          ▷ true or false
    end if
    dom$_u$ ← smallest(alldoms)            ▷ select vertex u with the smallest domain first
    newAlldoms ← initialize with size=(alldoms.size - 1)
    for i in dom$_u$.getNextSetBit do             ▷ for each entry in u with bit set to 1
        assign i as a value of vertex u
        for (dom$_v$ in alldoms) $\wedge$ (dom$_v$ != dom$_u$) do          ▷ the domain of each vertex
            if !consistent then return consistent $\wedge$ SIP1(newAlldoms)
            end if
            newdom$_v$ ← dom$_v$
            set i$^{th}$ entry of newdom$_v$ to false                ▷ cannot take value assigned to u
            if u is adjacent to v in G$_p$ then
                (newdom$_v$)AND(neighbours of i in G$_t$)    ▷ v can only take vertices in G$_t$ adjacent to i
            end if
            add newdom$_v$ to newAlldoms
            consistent ← !(newdom$_v$ == 0)  ▷ if there is a domain wipe out, consistent becomes false
        end for
        consistent $\wedge$ SIP1(newAlldoms)
    end for
end procedure
```

Chapter 5

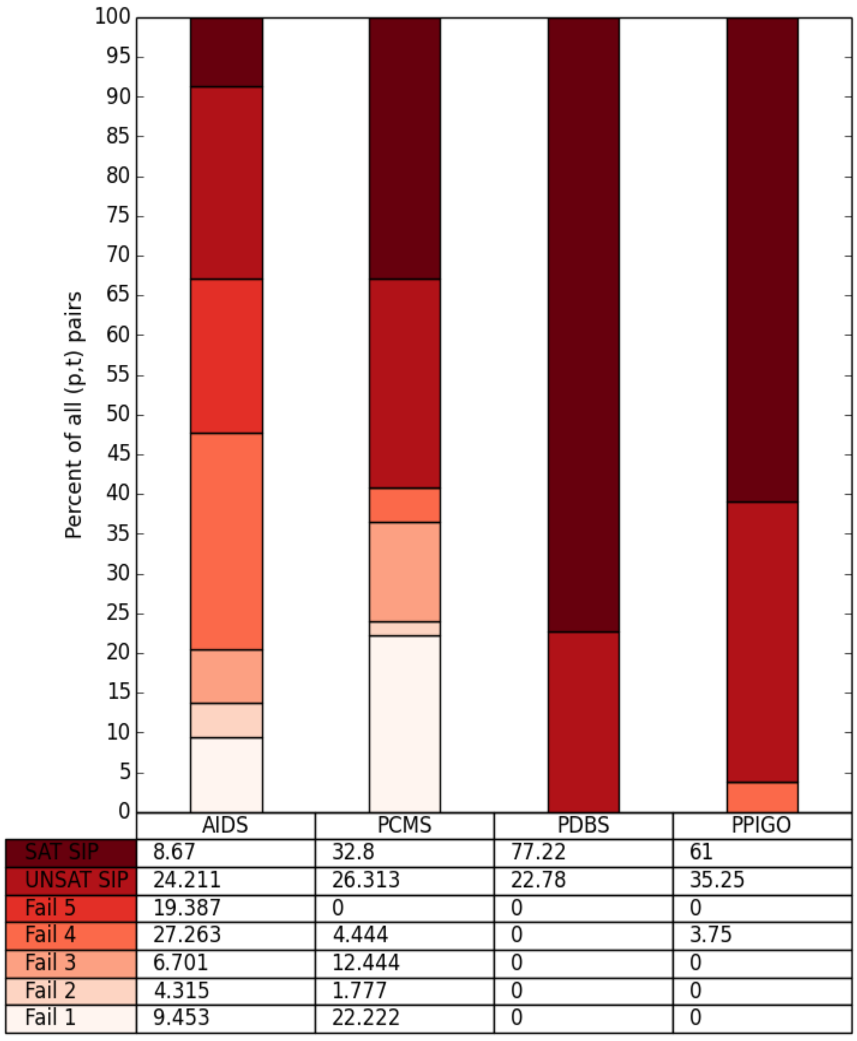
Evaluation

5.1 Light Filters

This Section reports on the observed performance of the subgraph query processing algorithm described in Chapter 4. SIP1 was run with each of the datasets discussed in Section 2.2, some of which were also used for empirical study by [11, 4, 8, 4, 10, 13, 18]. 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 [11]. 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 2.3. 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 [4, 10, 13, 18].
- 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 [11], 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 ith 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

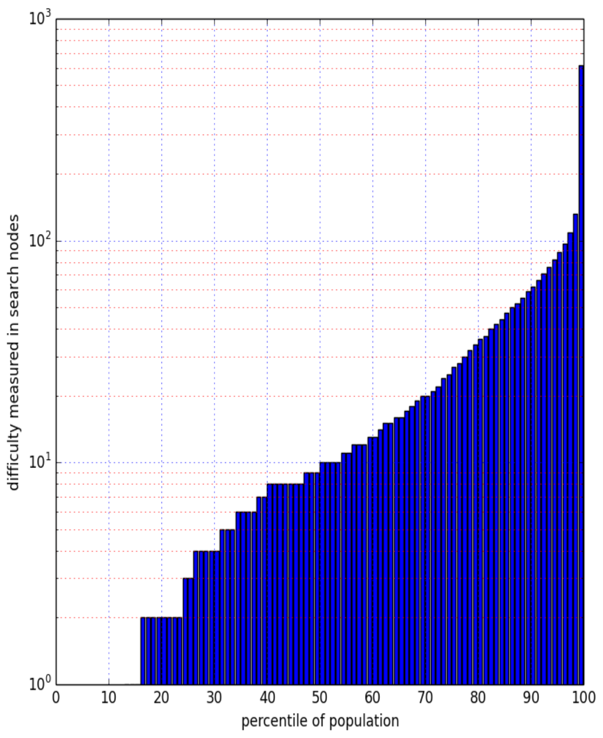


Figure 5.2: SIP on aids dataset

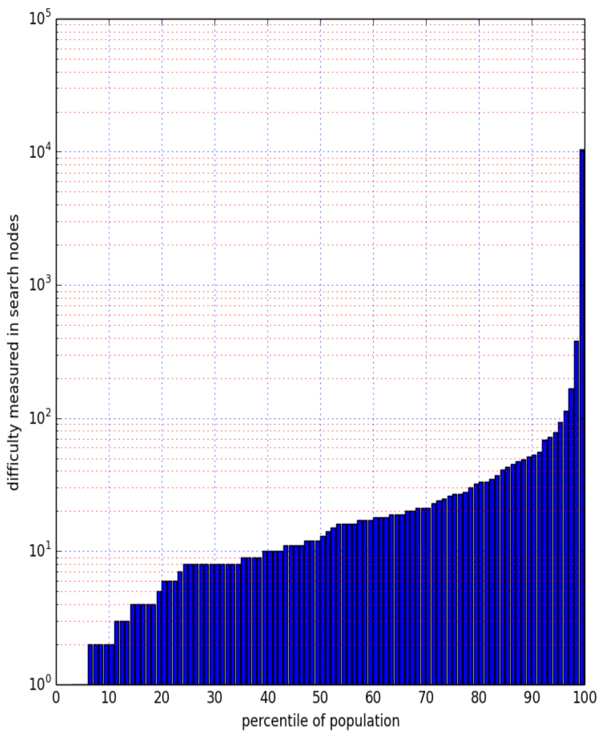


Figure 5.3: SIP on pcms dataset

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.

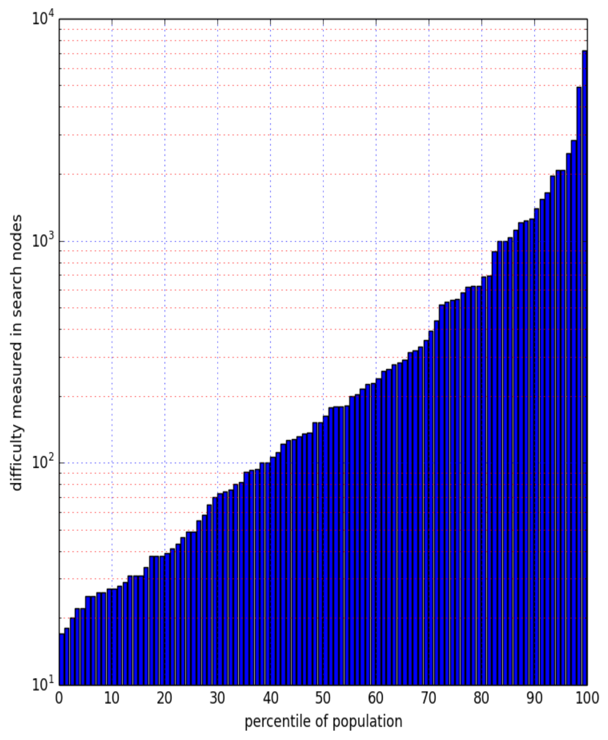


Figure 5.4: SIP on pdbs dataset

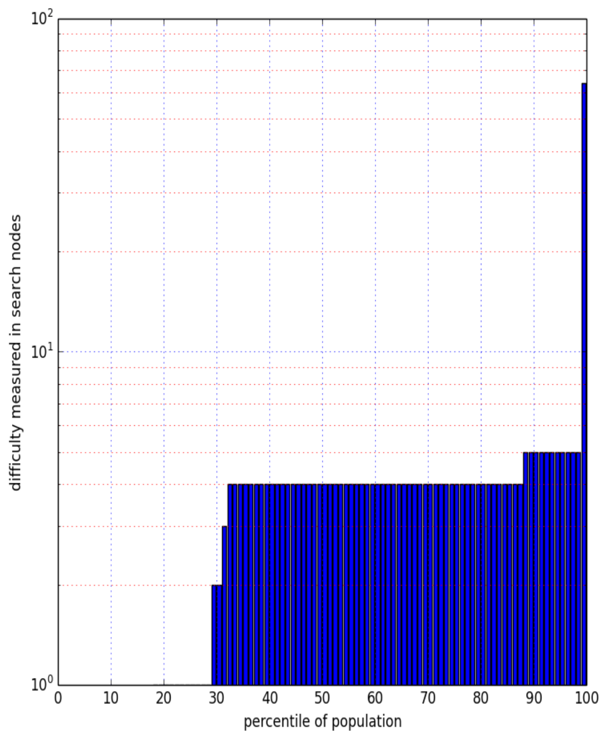


Figure 5.5: SIP on ppigo dataset

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 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 2.3) 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 2.3 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

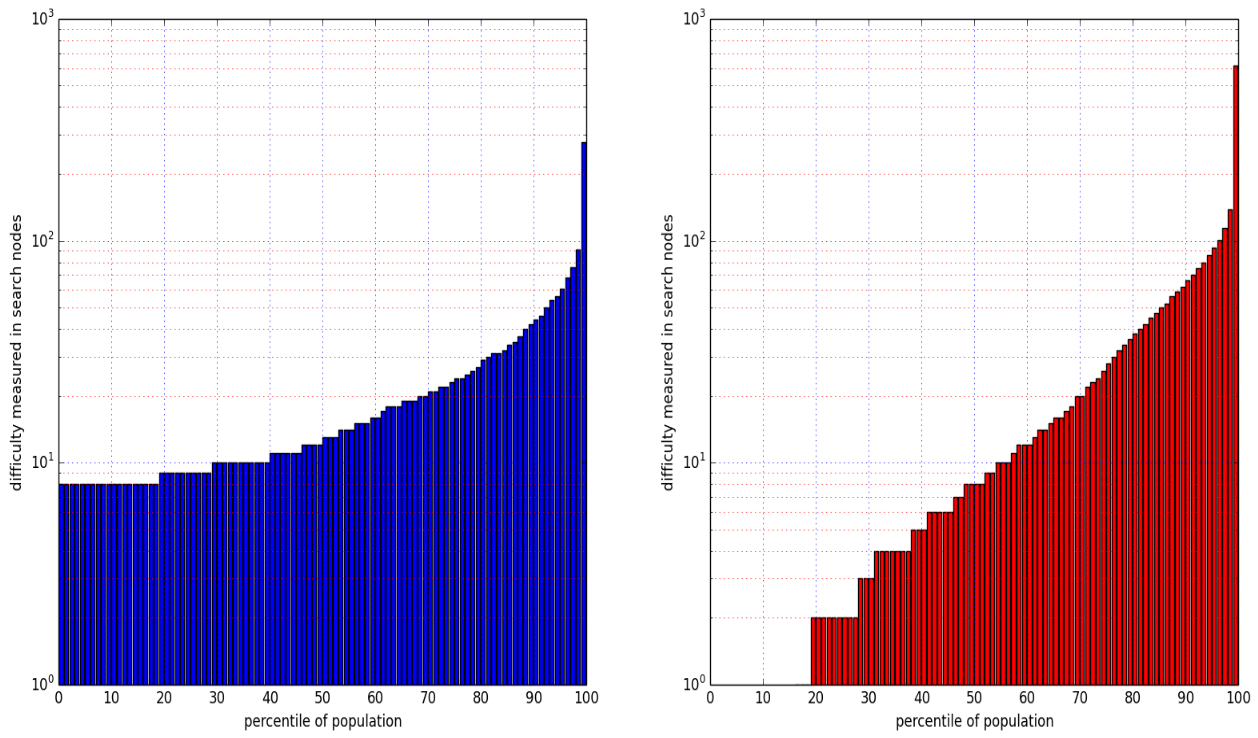


Figure 5.6: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in aids

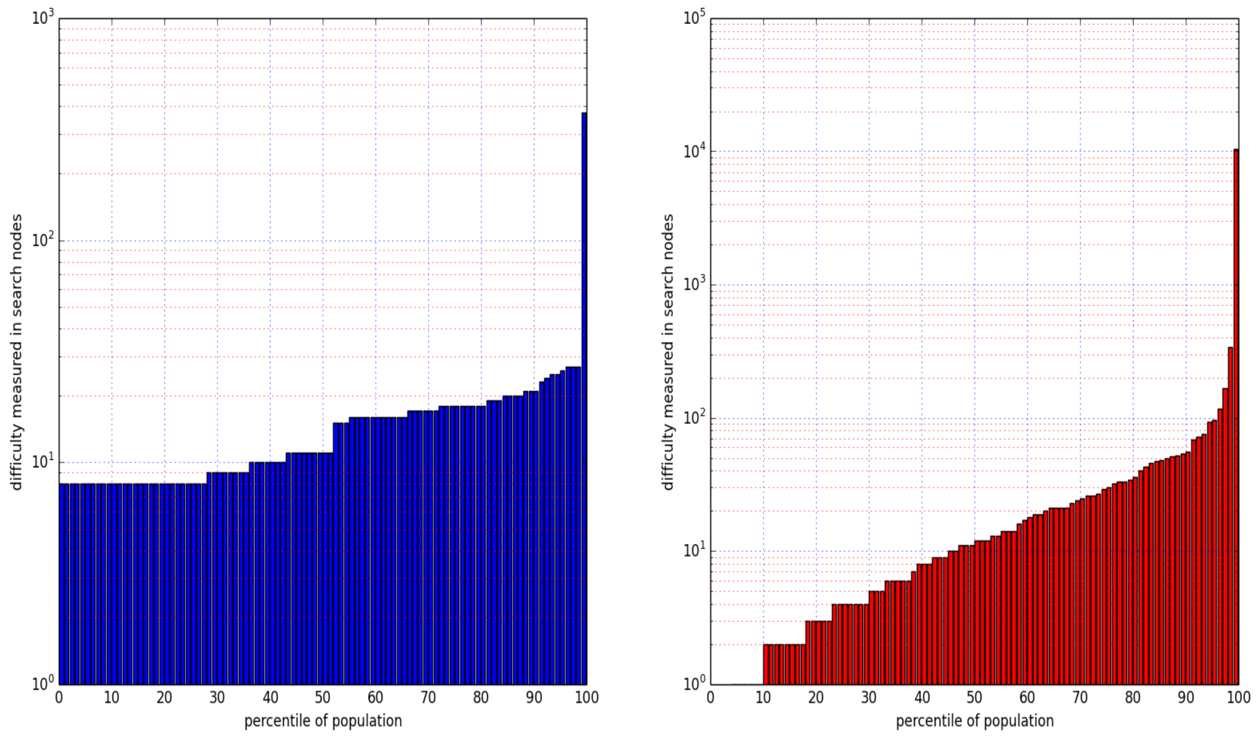


Figure 5.7: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in pcms

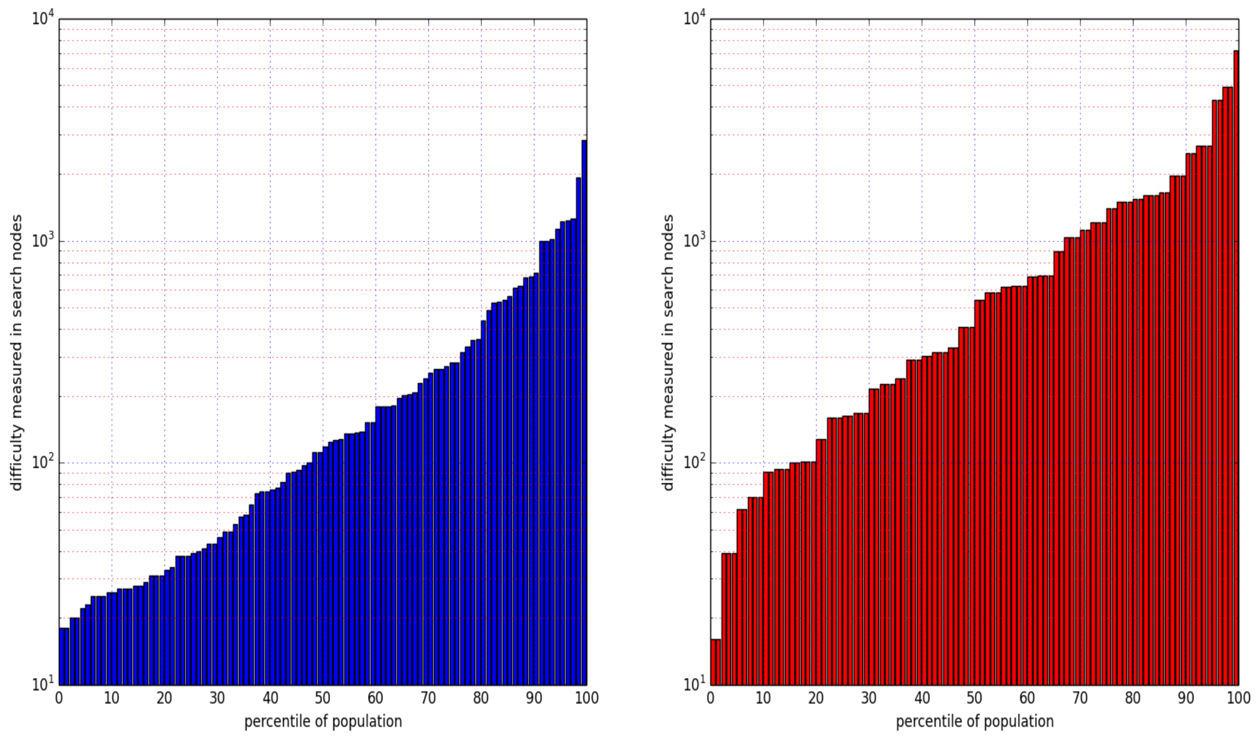


Figure 5.8: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in pdbs

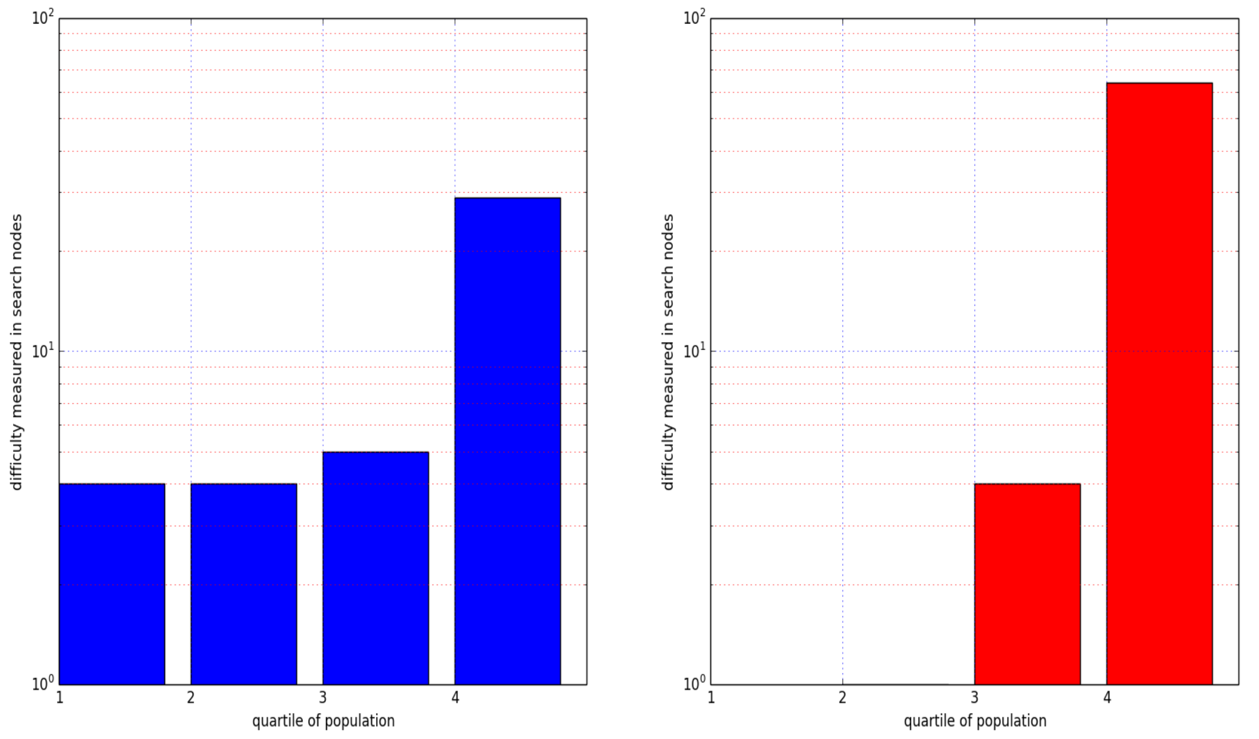


Figure 5.9: Search effort for SAT(blue, left) UNSAT(red, right) SIP instances in ppigo

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

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

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 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 2.3 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 (2.3, 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 (2.3, 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.

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 (2.3) 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.

	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

Comparison with Big Data algorithms

Evaluation of six "state of the art" subgraph query processing algorithms ([10, 13, 8, 18, 4, 16]) is presented in [11]. The algorithms employ a heavy filtering approach using an index structure and run [6] 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 [11], 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 [4] 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 [11] took about 7 seconds for filtering and 200 milliseconds for verification and it is again GraphGrepSX [4]. The light filters approach has slower verification and slightly faster filtering.

The algorithms evaluated in [11] have an additional overhead that is not present in our approach, which is the size of the index that has to be stored.

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.

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] 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.

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

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

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

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

[18] 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. 34

index todo. 34

search node A search node denotes the number of recursive calls to the SIP algorithm taken to find a solution. 28, 34

Acronyms

FC forward checking. 25

G graph. 25

G$_p$ pattern graph aka query. 22, 25

G$_t$ target graph. 22, 25

lds label degree sequence. 22–25

nds neighbourhood degree sequence. 22, 23, 25

SAT Satisfiable. 3, 22, 27, 28, 30–34

SIP subgraph isomorphism problem. ii, 2, 3, 21–23, 25, 27, 28, 30–34

UNSAT Unsatisfiable. 3, 22, 23, 27–34