University of Glasgow | School of Computing Science

Investigations of Subgraph Query Processing

Iva Stefanova Babukova

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

Level 4 Project — March 20, 2016

Abstract

Education Use Consent

Contents

Chapter 1

Introduction

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 Graph Theory, terminology and definitions

In this section, we introduce preliminary concepts and outline the main concepts and problems addressed in the document. In definition 4 we explain the format of all graphs used in the document.

1.3.1 Graph Theory

1.3.2 Terminology

- The set of target graphs: T - Each graph in t: t_i, for i from 1 to the number of graphs in T - The set of pattern graphs: P - Each pattern graph in P: p_j, for i from 1 to the number of graphs in P - The candidate set of graphs: C. It is important to not that C is contained in T - There is a trade off between the size of C and the time it takes to be computed.

1.3.3 Definitions

Definition 1 (Graph)**.**

Definition 2 (Sub-graph)**.**

Definition 3 (Induced Sub-graph)**.**

Definition 4 (Graph Format)**.** *A graph G = (V, E, L, λ) is defined as an undirected labeled graph where V is the set of vertices, E is the set of edges(unordered pair of vertices), L is the set of labels, and λ is a labeling function, $\lambda : V \cup E \rightarrow L$, that assigns labels to vertices and edges.*

Definition 5 (Graph Isomorphism)**.**

Definition 6 (Induced Graph Isomorphism)**.**

Definition 7 (Subgraph Isomorphism)**.** *Given a graph database T of target graphs t_0, t_1, $t_2 \ldots t_i$, where i is the number of target graphs in T, and a pattern graph P, find all targets in T that have P as a subgraph.*

Definition 8 (Induced Subgraph Isomorphism)**.**

Definition 9 (Graph Density)**.**

Definition 10 (Subgraph)**.** *A graph whose vertices and edges are a subset of another graph.*

Definition 11 (Graph Query Processing)**.** *Given a graph database D = $\{ g_0, g_1, g_2 \ldots g_n \}$ and a pattern graph p, it returns the query answer set $D_p = \{ g_i | p \subseteq g_i, g_i \in D \}$*

Definition 12 (In-memory computing)**.** *The storage of information in the main random access memory (RAM) of dedicated servers rather than in relational databases operating on comparatively slow disk drives.In-memory computing gives ability to cache countless amounts of data constantly. This ensures extremely fast response times for searches.*

Definition 13 (Graph Index)**.**

1.4 Subgraph Isomorphism

1.5 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 [1]. The database has 40 000 molecules, represented as graphs.

	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

$*$ 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.

Table 2.3: Solvable instances per dataset

satisfiable instances	AIDS	PCM	PDBS	PPIGO
average %	8.590	26.5	69.42	24.4
median %	0.673	19	83.3	26
minimum %	0.0075	2	22.67	14
maximum %	29.89	72	96.7	34

2.3 Filtering-verification paradigm

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

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 [4] 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 [4], CT-Index addresses the non-induced subgraph isomorphism problem (definition 7).

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 [4] 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 [4] 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 [4] 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 [**?**] 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 [4] 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 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".

[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.

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 [**?**], similar to VF2 [**?**] 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 [2], gCode [6] and Tree+Δ [8]. Most papers claim that VF2 is "state of the art". However, this is not the case ([**?**, **?**, **?**, **?**, 7]. VF2 has been shown to perform erratically and poorly [7]. 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 [5] 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 [5] 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.

A non-induced subgraph isomorphism from a graph P (called the pattern) to a graph T (called the target) is an injective mapping from V(P)to V(T) which preserves adjacency that is, for every adjacent v and w in V(P), the vertices i(v) and i(w) are adjacent in T. An induced subgraph isomorphism additionally preserves non-adjacencythat is, if v and w are not adjacent in P, then i(v) and i(w) must not be adjacent in T. We use the notation i : P T for a non-induced isomorphism, and i : P , T for an induced isomorphism.

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 3 in section 1.3.3) and the induced version of the subgraph isomorphism problem (definition 8 in section 1.3.3)

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 [4], gCode [6] and GraphGrepSX [2]. 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 $\text{target}'\text{s}$ 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.

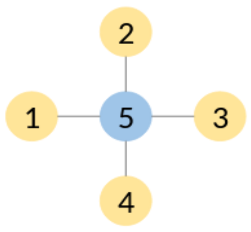


Figure 3.1: Pattern graph P

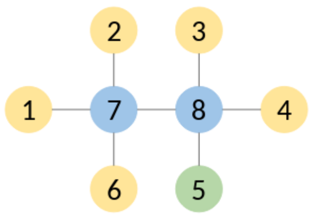


Figure 3.2: Target Graph T

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.

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

Algorithm 1. Paths extraction

```
1  generatePath(nodes, maxL) → void
2      for every start-node in nodes do:
3          push start-node on top of stack
4          generatePathInner(node, maxL)
5
6  generatePathInner(node, maxL) → void
7      if stack is less or equal to maxL then:
8          outputPath(stack)
9      if stack is equal to maxL then:
10         remove the topmost element from the stack
11         exit
12     for every neighbor of node:
13         if neighbor not on stack then:
14             push neighbor on top of stack
15             generatePathInner(neighbor, maxL)
16     if all neighbors are on stack and stack is not empty then:
17         remove the topmost element from the stack
```

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 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 2. Output path

```
1 outputPath(nodeSequence) → void
2   pathStr = nodeSequence.toString()
3   reversedSequence = nodeSequence.reverse()
4   pathStrReversed = reversedSequence.toString()
5   if pathStrReversed < pathStr then:
6     pathStr = pathStrReversed
7   if pathStr not in index then:
8     put pathStr to index
9  exit
```

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 3. Candidates extraction

```
1 candidatesExtractor(target-index, pattern-index) → true or false
2  for every pattern-path in pattern-index:
3    if no target-path equals pattern-path then:
4      return false
5    else:
6      continue
7  return true
```

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

$Path\text{-}Subtree$ 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 $Path\text{-}Subtree$ 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. $Path\text{-}Subtree$ 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, $n_2 \ldots n_i$ }, 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, $Path$-$Subtree$ 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 $Path$-$Subtree$ 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.

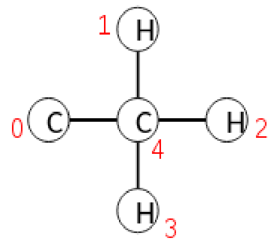
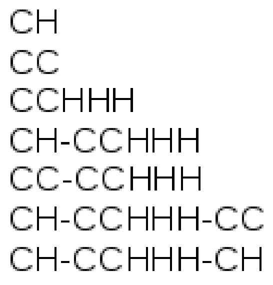


Figure 3.6: Pattern graph



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 $path\text{-}index$ 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 $Path\text{-}Subtree$ 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 $Path\text{-}Subtree$ index method, this path is represented as `CH-CCH-CH` in the index file. If we used $path\text{-}index$, 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

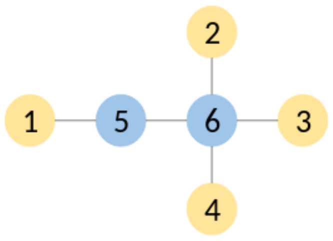


Figure 3.8: Graph A

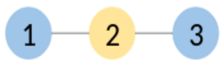
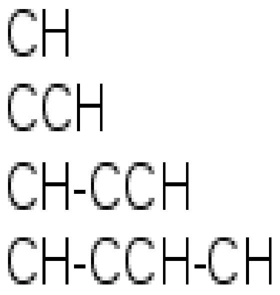
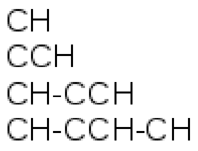


Figure 3.9: Graph B



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 ...

3.4.5 Performance

TODO
- the index file is bigger, but the candidate set is smaller
- put the table with results
- explain the results and why they are like this
- complexity?
- what could be done better and how?

3.4.6 Variants of Path-Subtree index

TODO

3.4.7 Advantages and Limitations

TODO
- very easy to implement, especially when we have index-1 implementation ready

Chapter 4

Light Filters

This section describes the study of a simple subgraph isomorphism problem algorithm with a fast filter implemented on top of it. The algorithm is called SIP1 and it is explained below. We present the experimental evaluation conducted to find out more about the effectiveness of light filtering, the notion of search effort and how it is affected depending on the nature of the datasets and the yes/no answer of the SIP. Finally, the results of the experiments are reported.

4.0.8 SIP1

SIP1 is based on the simplest of the Glasgow algorithms [7]. Given a pattern and a target graph P and T respectively, SIP1 has a variable for each vertex in P, each with domain that is the set of compatible vertices in T. 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, all variables have i removed from their domains. Also, if a future variable v is adjacent to u in P then the domain of v is the intersection of the current domain of v with the neighborhood of vertex i in T. This constraint is enforced by applying a logical and operation between the two bit sets. SIP1 uses fail-first heuristic. For all instantiated variables representing pattern vertices, it selects to explore the one that has the smallest domain before the others.

At the top of the search, 5 naive tests are performed to determine whether the two graphs are compatible. If the conditions of the tests are not met, search does not proceed and we consider this to be a $trivial\ fail$. These tests are presented on table 4.1. The neighborhood degree sequence (NDS) of a graph and a label degree sequence (LDS) is used for most of the tests. In the next section, we define and explain the notion of NDS and LDS and their implementation.

Neighborhood Degree Sequence

NDS and LDS are defined as follows.

Definition 14 (Label Degree Sequence)**.** *Given a graph $G(V, E, L)$ with V the set of vertices, E the set of edges and L the set of labels, the LDS of $l \in L$ is a non-increasingly ordered list of the degrees of all vertices in V that have l assigned as their label.*

Definition 15 (Graph Neighborhood Degree Sequence)**.** *Given a graph $G(V, E, L)$ with V the set of vertices, E the set of edges and L the set of labels, the GNDS of G, also written as NDS(G), is the list of tuples (l,LDS(l)) for every label $l \in L$.*

Example 1 Let us consider again the graph on figure 3.2. There are three distinct labels: yellow (Y) blue (B) and green(Gr). Nodes 1, 2, 3, 4 and 6 are Y and each of them has deg 1. Nodes 7 and 8 are B, both of degree 4. 5 is the only Gr node. Therefore, the LDS of the labels are as follows.

- LDS(Y) = <1, 1, 1, 1, 1>

- LDS(B) = <4, 4>
- LDS(Gr) = <1>

The NDS(graph 3.2) consists of the tuples <Y, LDS(Y)>, <B, LDS(B)> and <Gr, LDS(Gr)>.

The notion of graph neighborhood degree sequence is used to filter out targets that are incompatible with the pattern. A target graph T is compatible with a pattern graph P if the following two conditions are true:

- The set of labels in P is a subset of the set of labels in T
- NDS(P) is a subset of NDS(T)

The second condition can be formally defined as follows.

Definition 16 (NDS(P) is a subset of NDS(T))**.** *Let us write DS(l_P) for the DS(l) $\in$ P and LDS(l_T) for the LDS(l) $\in$ T, where l is a label both in T and in P. For every such l, every element on position i in LDS(l_P) is $\leq$ to the element in position i on LDS(l_T) for 1$\geq$i$\leq$ |LDS(l_P)|.*

Example 2 Let us consider the subgraph isomorphism problem with T the graph on figure 3.2 and P the graph on figure 3.1. The set of labels in P, denoted as l_P, is composed of B and Y; and the set of labels in T, denoted as l_T, consists of B, Y and Gr. Clearly, $l_P \subseteq l_T$. Therefore, the first compatibility condition for this example is true.
Looking at the LDS of each l $\in$ l_P, it satisfies definition 16. For example, for l=B, LDS(l_P) has only 1 element, which is 4 at position 1, and looking at LDS(l_T), the element at position 1 is also 4.

The defined compatibility conditions are the two new tests added to SIP1. In the next subsection, we explain the implementation details of NDS.

NDS and LDS Implementation

To accommodate the notion of NDS in the SIP algorithm, we added a class called Label to represent every label in the graph and its LDS. We made slight changes to the class representing the graphs, called Graph, and finally we created a class called SIP1 which implements the same subgraph isomorphism algorithm like the easiest SIP algorithm used in [7]. It has additional trivial failures added on the top of the search to check the NDS compatibility between the target and the pattern. Below, we explain the changes for each of the three stated classes.

- Class Label
  Every label object has a name of type String and a DS which is an array of integers, sorted in non-increasing order. The degree sequence array is built by inserting the degree of each node with the corresponding label, i.e. building the sorted list incrementally.
- Class Graph
  A new field of type array of Labels is introduced to keep a record of all unique labels in the graph. While reading in the graph, we build the array of Labels, creating a new object for each unique label and building its degree sequence as explained above.
- Class SIP1
  This is the class that is called for every (p,t) pair of the dataset to check whether t contains p, i.e. this is the part of the code that solves the subgraph isomorphism problem for every such (p,t) pair.

  We have added 5 trivial failures on top of the search, which can be seen in table 4.1. They are added into the code as if statements in the same hierarchical order as shown in the table. If any of these failures is true, the algorithm terminates returning false (i.e. the corresponding (p,t) instance is not satisfiable, because it is impossible for t to contain p). For example, if the order of t is smaller than the order of the p (trivial fail 1 4.1), SIP1 terminates without going into the search and without taking any other trivial failure tests into an account.

Trivial failures hierarchy

In this subsection, we discuss on the choice of the trivial failures hierarchical structure.

- Filter 2 and 3 filter differently. Ex: p = {A,B,B,C} and t={A,B,C}. Test 2 passes, but test 3 fails
- There is no need to check whether the length of all labels in T is bigger of equal to the length of all labels in t, because test 2 and 3 would fail if this was not the case.
- test 4 contains tests 2 and 3.$Explainwhy$ Furthermore, the notion of NDS introduced those 3 tests. We decided to have them instead of 1 big test to check the effectiveness of each part of the NDS filtering.

Trivial Fail	Meaning
1	T.order $\geq$ P.order
2	T unique labels $\geq$ P unique labels
3	P labels $\subseteq$ T labels
4	NDS(P) $\subseteq$ NDS(T)
5	initial domain cardinality > 0

Table 4.1: Specification of the measured failure types

4.0.9 SIP Algorithms evaluation

This section reports on the observed performance of SIP0 and SIP1 when used with the four Big Data datasets.

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, also used by [5]. The datasets as the same as described in section 2.2. All of them are used for evaluation on indexing performance by [5, 2, 3] and only the dataset called “aids” is used by [2, 4, 6, 8].

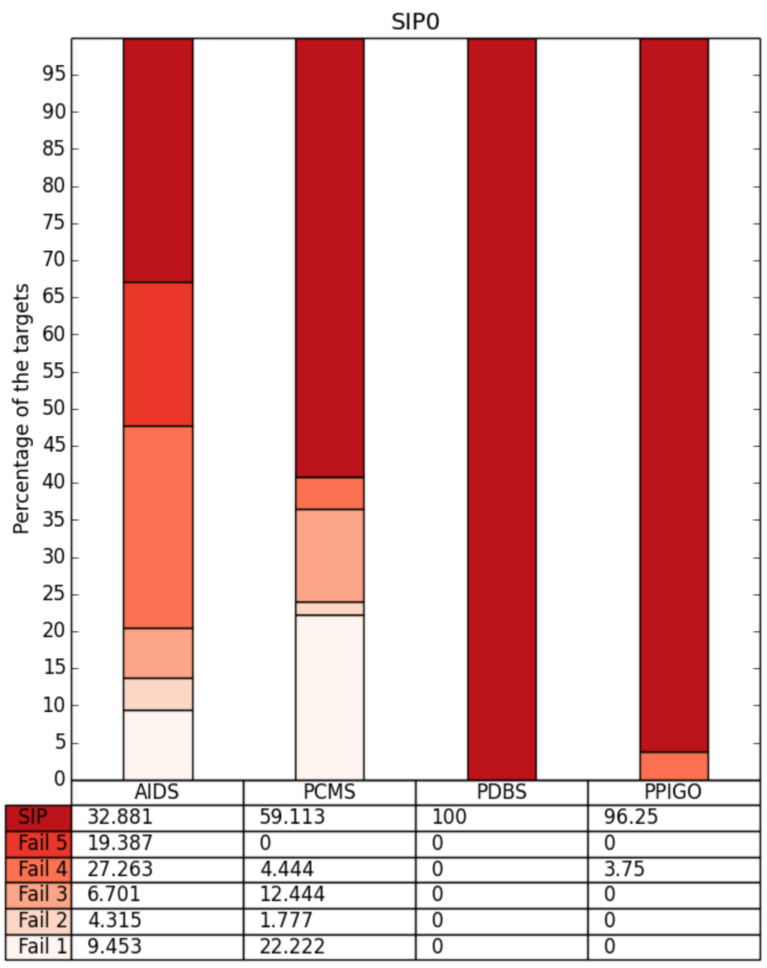
When given a set of targets T and a set of pattern P, the program reads in every t $\in$ T and every p $\in$ P and performs a SIP algorithm for each (pattern,target) pair. Run time is measured in miliseconds from when the process starts until it completes, including the time to read in all the graphs, performing all SIP tests and write out all results to a file. As stated previously, both SIP0 and SIP1 implementations are written in Java.

Trivial Failures

This subsection reports on the total filtering strength of the measured trivial failures with SIP0 and SIP1 algorithms. Table 4.1 shows in which order they are executed (from 1 to 5) for each SIP algorithm.

While running the SIP for each dataset, every time a trivial failure was observed, we recorded its number using the scale on table 4.1. Then, we computed the number of occurrences of each failure for each dataset. These numbers were then used to derive the percentage of the targets that each trivial failure eliminated when executed in the stated hierarchical order. Figure 4.1 shows the results with SIP0 (on the left) and SIP1 (on the right). Here, average percent of filtering for each failure is plotted. Additional plots that present the median, maximum and minimum percent of trivial failure filtering can be viewed from the appendix section.

It is important to note that as the failures are executed in hierarchical order, the number of graphs filtered by trivial fail x can depend on the order of execution of x with regards to the other trivial failures. With the results presented, we do not aim to make conclusions on individual performance of trivial failures. In order to study this, alternative set of experiments needs to be conducted.



	AIDS	PCMS	PDBS	PPIGO
SIP	32.881	59.113	100	96.25
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 4.1: Average filtering percentage of each method in table 4.1 for each of the datasets

While running the experiments, the following observations were made:

- The trivial failures studied perform best on the aids dataset.
  In particular, almost 70% of the targets are removed without making a call to SIP. Only on checking the order of the graphs (trivial fail #1) and the number of different labels (trivial fail #2), one is able to prove almost 14% of all targets as not satisfiable.
- All trivial failures measured do not work well on all graphs 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. There are two possible reasons for this. First, most of the instances of pdbs and ppigo might be satisfiable. If this is the case, there is nothing that can be filtered out. A second reason for these results could be that the trivial fails are not strong enough to eliminate targets from these datasets. Further investigations of the results need to be conducted to find out the reason for the obtained results and identify possible strategies to make SIP algorithms perform better.
- For some datasets (aids), NDS compatibility tests manage to filter some targets that are otherwise not filtered by the tests 1,2 and 5. For other datasets (pcms), NDS filter out part of the targets that are otherwise removed by a next trivial failure.
- NDS compatibility does not help eliminating targets before the search for pdbs and ppigo datasets.

- The dataset that shows best filtering improvement after adding failures 3 and 4 is aids.
- There are duplicate target graphs in the pcms and pdbs datasets.
  The pcms database is supposed to contain 200 targets [1]. In practice, there are only 50 unique graphs and each of them is added 4 times. The pdbs dataset is composed of 600 targets [1], but out of them only 30 are unique, each of them duplicated 20 times.

Hardness of SIP in terms of search nodes

We report on the cost of call to SIP in terms of the number of nodes (a measure of search effort) taken to solve an SIP instance. For every dataset T, for every (P,T) pair, we find every graph t $\in$ T that was not filtered by any of the trivial failures (i.e. all target graphs for which a call to SIP had to be made to check whether the pattern is contained in the target). We compute the number of graphs $|T_i|$ that are solved in given number of nodes i for $0 < i < maxN$, where $maxN$ is the maximum number of nodes taken to solve an instance for a given dataset. Figures 4.2, 4.3, 4.4 and 4.5 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 4.2, 24% of the targets are solved by using at most 2 nodes, 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 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 4.2, 4.3, 4.4 and 4.5, which present the hardest instance for each percentile in the dataset.

We derived the number of members of 1 percent of all target graphs in T by dividing —T— by 100. In cases with aids and pcms, the number of targets in the dataset was not divisible by 100. This is why for these datasets, the last percentile is slightly bigger with respect to the number of targets that form each of the rest 99% of the graphs.

The plots below help us to make the following observations:

- The easiest dataset is ppigo. Looking at figure 4.5, 88% of all targets are solved by using at most 4 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 unsatisfiable.
- The dataset called pdbs is the harder compared to ppigo and and aids and the most versatile one in terms of search effort. It is on average harder than pcms, however, the hardest instance in pcms takes more search effort than the hardest instance in pdbs. Figure 4.4 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 7152 nodes for 95 milliseconds. This instance is not satisfiable.
- The dataset with the hardest instance is pcms. The hardest SIP takes 10470 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. Namely, it is the SIP call between pattern #1 and target #629591 and it took 0 milliseconds of time. SIP(#1,#629591) is not satisfiable.
- 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 not satisfiable.
- Even the hardest instance of each dataset is solved very quickly.

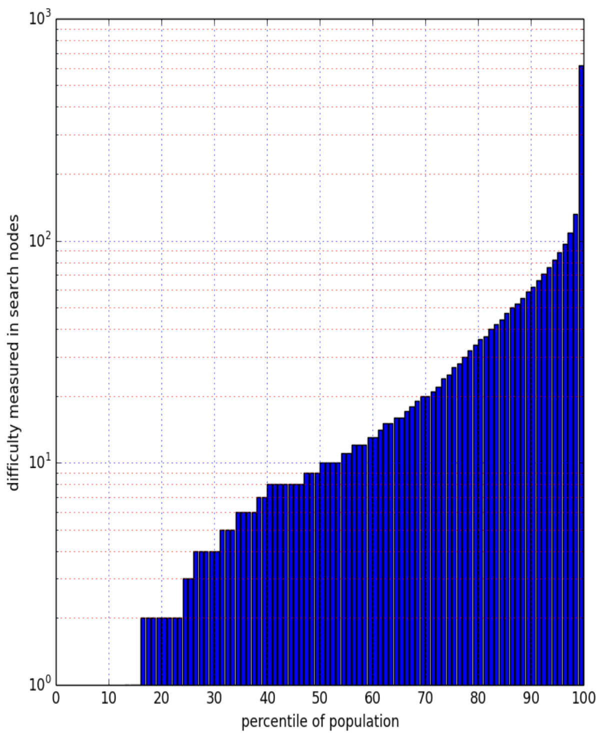


Figure 4.2: SIP on aids dataset

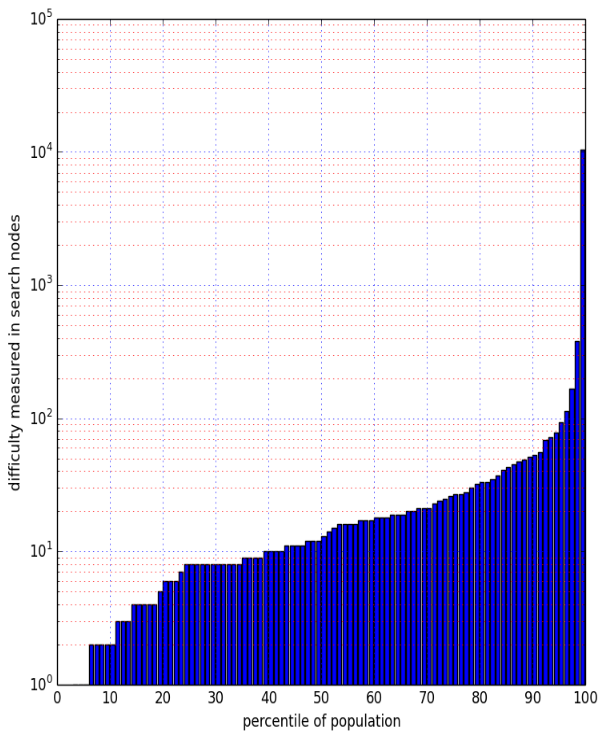


Figure 4.3: SIP on pcms dataset

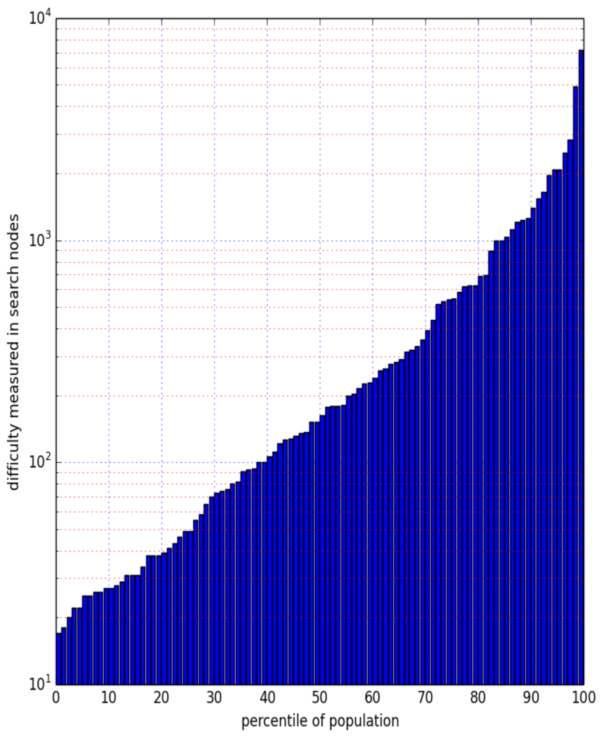


Figure 4.4: SIP on pdbs dataset

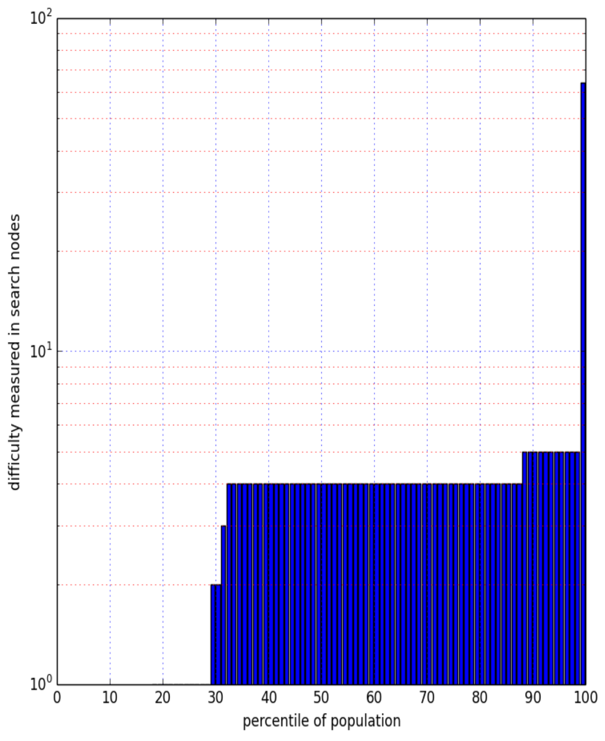


Figure 4.5: SIP on ppigo dataset

Hardness of SAT vs UNSAT SIP instances

The observation that the hardest instance of each dataset is not satisfiable arises the following question: is UNSAT SIP generally harder than SAT SIP or are our observations due to pure chance? 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 graphs below break each of the graphs discussed in the previous section (namely 4.2, 4.3, 4.4 and 4.5) further down in terms of whether the SIP instances are satisfiable or not. The blue plots represent all satisfiable SIP pairs (p,t) for every t ∈ T and every p ∈ P, where T and P are the sets of targets and patterns for a given dataset D. Similarly, the red plots represent all unsatisfiable SIP (p,t) instances of a dataset D. For each D (namely, for aids, pcms, pdbs and ppigo), the union of the blue plot (satisfiable instances, left-hand side below) and the red plot (unsatisfiable instances, right-hand side below) gives the plot for the corresponding dataset discussed in the previous section. For example, figure 4.6 ∪ figure 4.7 = figure 4.2.

Note that figures 4.12 and 4.13 contain only 4 bars each of them, i.e. the data is divided into quartiles instead of percentile. Here, each bar represents 25% of all instances for the corresponding category (SAT/UNSAT). For example, figure 4.12 that represents all satisfiable SIP instances 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 instances is too small to be scaled to percentiles.

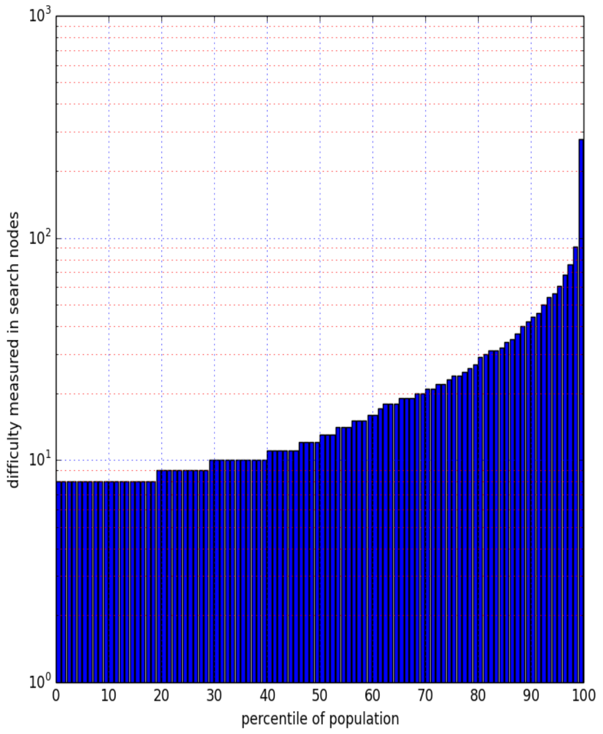


Figure 4.6: Search effort for SAT aids targets

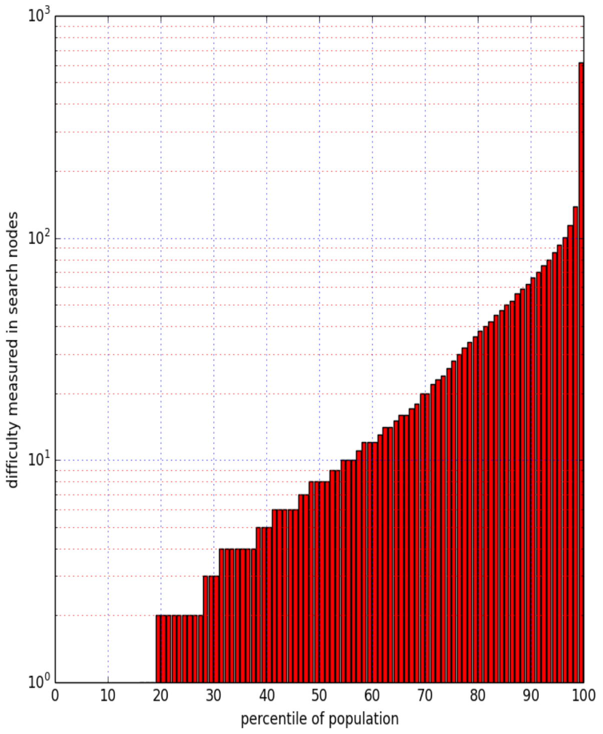


Figure 4.7: Search effort for UNSAT aids targets

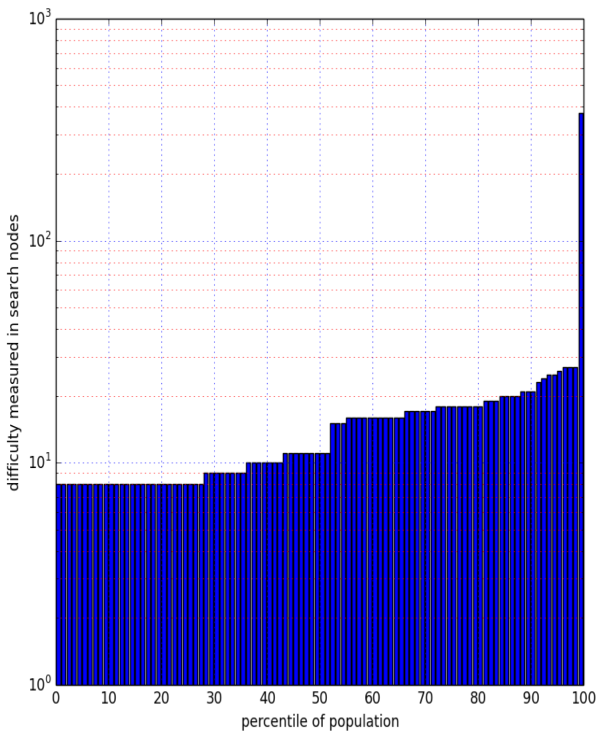


Figure 4.8: Search effort for SAT pcms targets

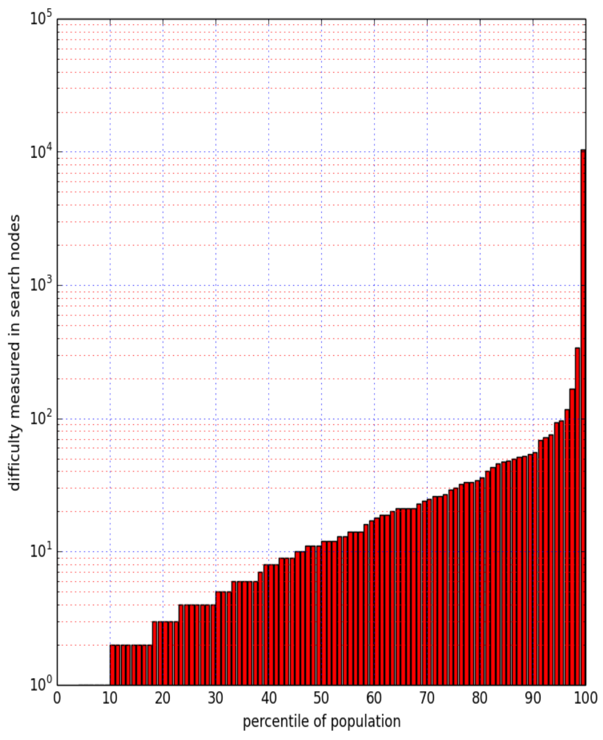


Figure 4.9: Search effort for UNSAT pcms targets

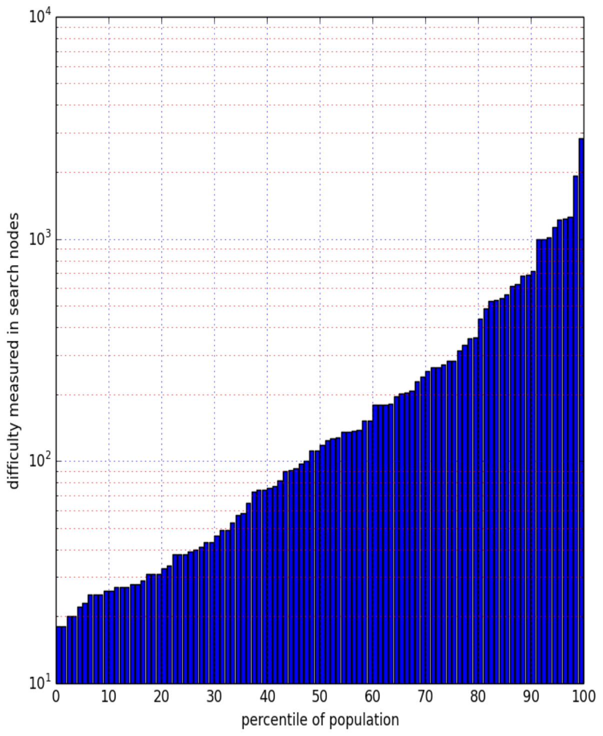


Figure 4.10: Search effort for SAT pdbs targets

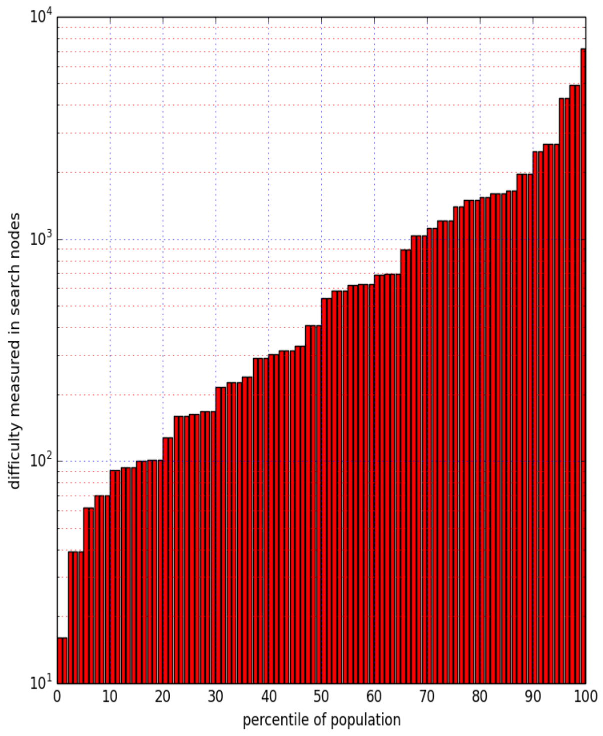


Figure 4.11: Search effort for UNSAT pdbs targets

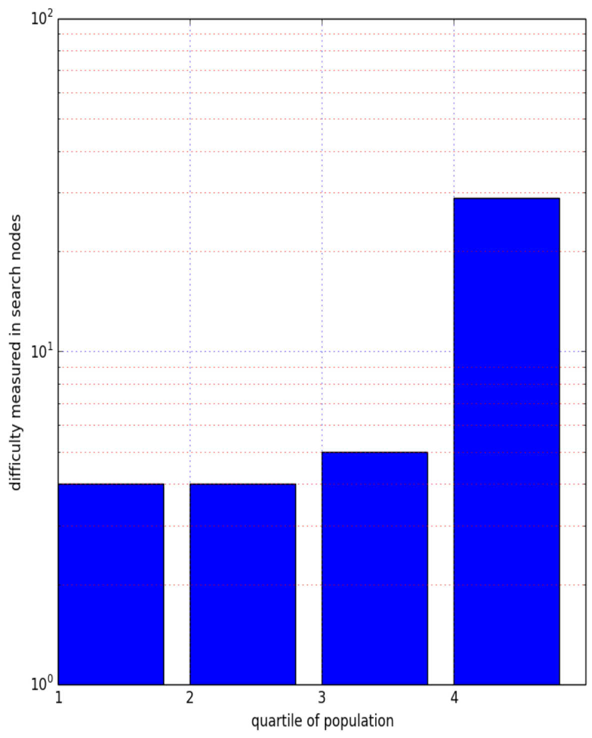


Figure 4.12: Search effort for SAT ppigo targets

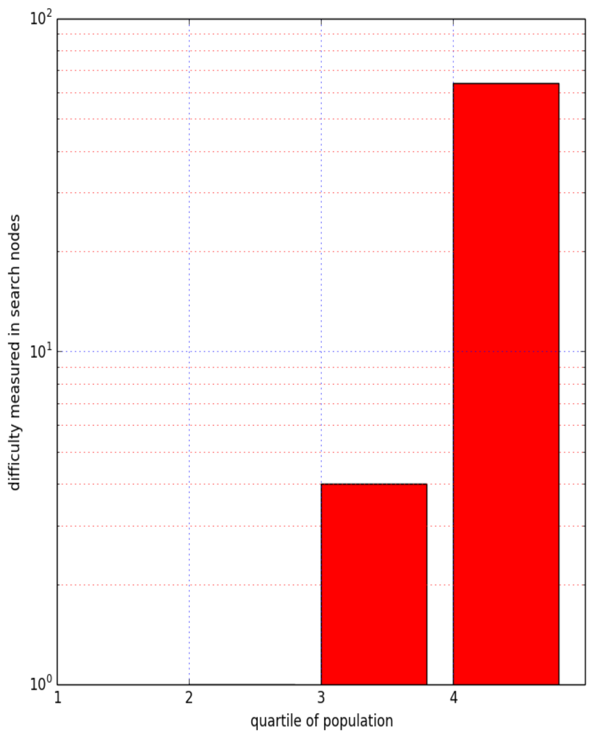


Figure 4.13: Search effort for UNSAT ppigo targets

Table 4.2 presents statistics in terms of the search effort for satisfiable (blue columns) and unsatisfiable (red columns) instances. We calculated the total number of nodes taken to solve all SIP instances for every dataset, the average, median, maximum and minimum number of nodes taken per instance. For instance, the table shows that the total number of nodes taken to solve all satisfiable SIP instances for the aids dataset is 437 108, whereas the total number of nodes taken to solve all unsatisfiable SIP instances is 2 295 724 nodes, which is almost 5 times more. Using these figures, we derive that the total number of 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 nodes.

Looking at the SAT and UNSAT figures and the table, we notice that:

- For three of the datasets, namely aids, pcms and ppigo, the easiest and hardest instances tend to be UNSAT.
- For pdbs, there is a big difference in terms of search effort between SAT and UNSAT problems. For example, the satisfiable instances are easier for every percentile of the targets than the unsatisfiable instances (4.10, 4.11). The tabulated results on figure 4.2 show that on average, SAT instances are 3 times easier than UNSAT, the SAT median is more than 4 times smaller than the UNSAT instances median and the number of nodes taken to solve the hardest SAT instance is 2 845 which is 4307 nodes less than UNSAT.
- Table 4.2 shows that 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. The reason for these results is that pdbs contains large number of solvable instances. In particular, the figures on table 2.3 show us that the average percent of satisfiable (p,T) SIP instances, where p $\in$ P, is 69.42 (i.e, for each set of (p,T) SIP problems, for 69.42% of all (p,t) pairs, where t $\in$ T, t contains p as subgraph). Similarly, the median, minimum and maximum number of SAT instances for a set of (p,T) SIP problems are respectively 83.3%, 22.67% and 96.7%. Therefore, the large search effort of SAT SIP problems for pdbs is due to their substantially larger number compared to UNSAT problems.
- There is a substantial difference in terms of search effort between the hardest SAT and the hardest UNSAT instances of the pcms dataset. In particular, this difference is 10 092 nodes, where the hardest SAT problem takes 378 nodes (4.2) and it is between pattern #32_1CY1.cm.A.out and target #1CY0.cm.A.cmap, solved for 4 milliseconds.

- For aids and pcms datasets, all unsatisfiable instances in total are about 5 times harder than the total number of all satisfiable instances (4.2).
- For datasets pdbs and ppigo, the total search effort for unsatisfiable instances is smaller than the search effort for satisfiable SIPs. In particular, for ppigo, it is twice smaller.
- As noted before, the maximum search effort is always bigger for unsatisfiable instances. For some datasets, the difference is substantial (pcms). For all four datasets, the search effort for the hardest SAT instance is at least twice easier than the search effort taken for the hardest UNSAT instance.

	Total		**Average**		**Median**		**Minimum**		**Maximum**	
aids	437 108	2 295 724	20.998	10.473	13	0	9	0	279	619
pcms	13 644	133 276	23.047	110.327	17	9	9	0	378	10 470
pdbs	894 260	854 720	321.67	1042.341	123	544	18	17	2 845	7 152
ppigo	714	312	6.932	5.473	6	2	5	0	30	65

Table 4.2: Number of nodes of search effort for each dataset. Blue for solvable and red for unsolvable SIP instances

Hardness of SIP in terms of running time

Chapter 5

Empirical Study

5.1 Filter-Verification methods

5.1.1 CT-Index

- my evaluation

5.1.2 Path-based indexing implementations

5.2 Light Filters

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] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

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

[15] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta ¿= graph. *In Proc. VLDB*, pages 938 – 949, 2007.