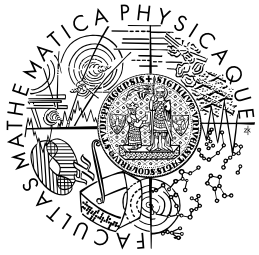

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Vojtěch Šípek

Comparison of Approaches for Querying of Chemical Compounds

Department of Software Engineering

Supervisor of the master thesis: Doc. RNDr. Irena Holubová, Ph.D.

Study programme: Computer Science

Study branch: Software Systems

Prague 2019

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date 5.5.2019 signature of the author

Title: Comparison of Approaches for Querying of Chemical Compounds

Author: Bc. Vojtěch Šípek

Department: Department of Software Engineering

Supervisor: Doc. RNDr. Irena Holubová, Ph.D., Department of Software Engineering

Abstract: The purpose of this thesis is to create an overview of research about querying chemical databases and to validate or invalidate its results. There is no found research which would compare the performance and memory usage of the best performing approaches on the same data set. In this thesis, we address this lack of information and we create an un-biased benchmark of several index building methods for subgraph querying of chemical databases. Also, we compare the results of such benchmark with the performance results of SQL and graph database.

Keywords: Chemical database, Subgraph querying, Graph database, Subgraph isomorphism

Dedication. **TODO**

Contents

Introduction

Querying is the essential utility of each database and the same applies to chemical databases. Nowadays, the largest publicly accessible databases contain around 100 million compounds. The chemical compounds can be naturally represented as graphs where atoms are represented as vertices and bonds are represented as edges. The typical chemical compound is a connected sparse graph with labeled edges and vertices where the size of the labeling alphabet for edges is less than 10 and size of the labeling alphabet for vertices is in order of low hundreds.

The size of chemical compounds is variable. The vertex count varies typically from very small compounds with less then 10 vertices to huge compounds with hundreds of vertices. These sizes multiplied by the size of the database implies that querying over such databases might be a challenging task.

The most common queries over chemical databases are exact match query, shortest path search, similarity search and substructure search which are usually used in graph databases. The latter will be the main point of interest in this thesis.

The goal of subgraph querying is to obtain a list of graphs from the database which contain the queried graph as its subgraph. The result of this process has a wide range of utilization e.g. in chemoinformatics and bioinformatics and therefore in pharmaceutic industry. Several indexing techniques have been proposed to minimize the number of subgraph isomorphism tests since it is known as NP-complete problem.

There are several benchmarks of the mentioned indexing techniques already. The problem is that all found benchmarks has been created by the authors of some of the indexing technique and therefore the intention of the benchmark is to show that the particular index is more powerful than others. There is a lack of independent benchmarks which would compare the best performing indices on the same data and on the same hardware.

In this thesis we compare the best performing indexing techniques using the same environment. We also compare these techniques with the classical SQL database performance as well as with the performance of the modern graph databases.

Structure of the Thesis

This thesis is divided into four main parts. In the first part called *Base Terms and Definitions* we define the main problem of this thesis, subgraph isomorphism, and we define the main terms related to the graph theory which are used later in the thesis.

In the second part called *Analysis of Related Work* we analyze the found literature about the algorithms for resolving subgraph isomorphism problem and

most importantly we analyze and briefly describe the indexing techniques proposed in related work.

In the third part, *Experimental Work*, several hypotheses are formulated. For their verification the author's experimental work is used. These experiments are described in detail and the issues found out during the implementation are explored.

The last part of the thesis called *Experimental Results* covers the results of experimental work and the comparison with results of related researches. We will comment on the findings and propose some directions in possible following research.

1. Base Terms and Definitions

1.1 Definitions

Definition: *Graph* $G = (V, E)$ is a ordered pair where V is a set of vertices and $E \subseteq V \times V$ is a set of edges .

Definition: *Labeled Graph* $G = (V, E, L_V, L_E, f_V, f_E)$ is an ordered 6-tuple of set of vertices V, set of edges $E \subseteq V \times V$, set of vertex labels L_V, set of edge labels L_E, function assigning the vertex labels to vertices $f_V : V \longrightarrow L_V$ and function assigning the edge labels to edges $f_E : E \longrightarrow L_E$.

Definition: Graph $G = (V, E)$ is a *Subgraph* of graph $G' = (V', E')$ if and only if $V \subseteq V'$, $E \subseteq E'$ and $((v1, v2) \in E \Longrightarrow v1, v2 \in V)$. We denote it as $G \subseteq G'$.

Definition: Graph $G = (V, E)$ is an *Induced Subgraph* of graph $G' = (V', E')$ if $G \subseteq G'$ and for all edges $e = (u, v) \in E'$, $(u \in V)\&(v \in V) \Longrightarrow e \in E$.

Definition: Graphs $G = (V, E)$ and $G' = (V', E')$ are *Isomorphic* to each other if there exists a bijection $I : V \longrightarrow V'$ so that $(v1, v2) \in E \Leftrightarrow (I(v1), I(v2)) \in E'$.

Definition: Graph G is *Subgraph Isomorphic* to graph H if there exists a subgraph $H' \subseteq H$ which is isomorphic to G.

The last four definitions can be extended for the labeled graphs intuitively.

1.2 Subgraph Querying

Due to the NP-complete nature of the subgraph isomorphism problem (is one graph subgraph isomorphic to other?), we cannot expect good results using a naive approach where we test iteratively all database records to find out whether they match the query graph or not. Usually, we need to cut down the number of these tests to the minimum.

Most of the techniques, described later in chapter 2, are working using the following pattern:

1. Based on the database statistics and approach specific heuristics, construct a database index
2. Utilizing the index structure, build a **candidate set** of graphs for particular query
3. Use a sub-graph isomorphism algorithm to filter out false positives from the candidate set to obtain **answer set**

As we cannot expect significant improvement in the verification step since it is a known NP-complete problem, most of our focus in the rest of this thesis will

be targeted on the first two steps, i.e. index construction and its utilization for the candidate set creation.

2. Analysis of Related Work

In this chapter we summarize the work done by other authors which is related to the topic of this thesis. At first, we summarize the algorithms which have been developed for subgraph isomorphism matching and their comparison. Next we describe indices which might be used for obtaining the candidate set and algorithms which are used for their construction. The next part of this chapter focuses on approaches which utilize query mechanisms of particular relational and graph databases. In the last part we provide a summary of commercially used solutions.

2.1 Subgraph Isomorphism Algorithms

This section does not provide in-depth comparison of available algorithms since it is not a main topic of this thesis.

Almost all papers related to subgraph query methods refer two algorithms - Ullmann [1] and VF2 [2]. Those two algorithms are deeply compared in the [3] benchmark where VF2 outperforms Ullmann.

In paper [4] there is a comparison of four algorithms derived from Ullmann's algorithm. These are VF2, QuickSI [5], GraphQL [6], GADDI [7] and SPath [8]. They were compared using three real-world data sets. Although all three comparisons have a different winner, it seems that the most efficient algorithm is QuickSI in an average use-case.

2.2 Index Building Methods

In the first part of this section we briefly describe algorithms for building indices on top of chemical compound databases. These are *GraphGrep* [9][10], *GIndex* [11], *GString* [12], *GraphGrepSX* [13], *GIRAS* [14], *C-tree* [15] and *GDIndex* [16]

They form just a selection from a much bigger set of applicable methods and they were picked for different reasons:

- The method is mentioned in a majority of relevant articles
- The method uses an original algorithm or data structure
- The method has excellent results in benchmarks

Some of them can be used in generic graph databases, some of them are very specific to the field of chemical compounds but with some effort they might be used also for other graph databases with a specific point of interest.

In the following sections we will briefly introduce the basic ideas behind all the previously mentioned methods.

2.2.1 GraphGrep

Very simple and intuitive indexing technique which can be used in any graph database with labeled graphs is called *GraphGrep*. The presumption is that every vertex has a defined unique ID.

For each graph in the database there is a constructed index represented as a hash table where the key is a hashed value of a *label-path* (a concatenation of the vertex/edge labels on the path) and the value is a number of unique *id-paths* (a concatenation of the vertex IDs on the path) which represent a particular *label-path* in the graph. In the hash table there are all *label-paths* which are present in the graph up to length l, where l is a parameter. This hash table is called a *graph fingerprint*.

For example the graph in Figure 2.1 would be represented in the index with $l = 3$ as depicted in Table 2.1. The numbers in the picture represent the vertex ID, characters next to each vertex represent its label.

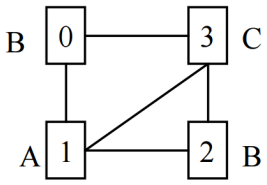

Figure 2.1: GraphGrep example graph

The same process is used for the query. The query itself is also a graph and therefore the hash table can be created too. Then, in the candidate set creation part, each graph's fingerprint is compared to the query fingerprint.

If any value in the query fingerprint is higher than value in the graph fingerprint for the same key or when some key from the query fingerprint is missing in the graph's fingerprint, it means that this graph can be filtered out from the candidate set because we know that the query cannot be its subgraph.

2.2.2 GraphGrepSX

GraphGrepSX is an improved version of GraphGrep. It uses the very same approach for obtaining all the indexed features - it takes all the paths up to the length l from all the graphs in the database. The core of the improvement is in the data structure where the index is stored.

Storing all the paths for each graph in a hash table is quite ineffective. Most of the paths appear in more than one graph and we do not need to store these duplicate keys more than once.

This method stores the paths in the suffix tree instead. Each node in the suffix tree represents a path (which is an extension of its parent) and contains a set of pairs $(graph, count)$ where $graph$ is an ID of the database record and $count$

Key	Value	Key	Value
h(A)	1	h(ABC)	2
h(B)	2	h(ACB)	2
h(C)	1	h(BAC)	2
h(AB)	2	h(BCA)	2
h(AC)	1	h(BAB)	2
h(BA)	2	h(BCB)	2
h(BC)	2	h(CBA)	2
h(CA)	1	h(CAB)	2
h(CB)	2		

Table 2.1: GraphGrep example graph fingerprint

is the number of occurrences of the represented path in the *graph*.

The way how the query is processed is very similar to the GraphGrep. It mines all the paths up to length l from the query graph and finds the matching nodes in the index tree. For each matched node we need to check whether the number of occurrences for each graph is equal or higher than the number of occurrences in the query graph. If so, we can add this database record into the candidate set. If some path from the query graph is not in the index, we can return an empty candidate set.

2.2.3 GIndex

This method utilizes the concepts of *frequent subgraphs* and *discriminative fragments*. It also comes with an innovative data structure for storing the index.

Since the number of all subgraphs grows exponentially with the size of the graph and therefore it would be impossible to index all of them, we need to prune the number of index records to be as compact and still as efficient as possible.

Because of the mentioned reasons the *frequent subgraphs* and *discriminative*

fragments concepts have a significant role.

Frequent subgraphs are all subgraphs which are contained in at least *minSup* (minimum support) graphs in the database. The survey of frequent subgraph mining can be found in [17]. Suppose we have an index from all frequent subgraphs and for each record in the index we have a set of IDs of graphs in the database in which it occurs. If the query graph q is frequent, we have the candidate set immediately. If not, we can get the candidate set as an intersection of matched graphs sets of all frequent subgraphs of q.

Utilization of pure set of frequent subgraphs with static *minSup* attribute has a couple of issues. With *minSup* set too low, we get an enormous set of frequent subgraphs. If the *minSup* is too high, the candidate set can be too large (at least *minSup*) with larger probability of false positives.

That is why the described method comes with size-increasing support function. It is a non-decreasing function which takes the graph size as an argument (defined as the number of edges) and returns the *minSup* for given size. This results in smaller *minSup* for small graphs (because of the efficiency) and bigger *minSup* for large graphs (because of the compactness). To prevent too big subgraphs in the index, it is necessary to specify a threshold starting from which the function returns infinite.

An additional pruning of the index can be done. There is a very high chance that frequent subgraph g will not be enough discriminative. It means that the candidate set of g is not significantly smaller than the intersection of candidate sets of its subgraphs.

Discriminative fragments concept brings a new metric. It measures how much discriminative the frequent subgraph is in comparison to the set of its subgraphs in the index. The discriminative ratio is defined as

$$\gamma = \frac{|\bigcap_i D_{f_{\varphi i}}|}{|D_x|}$$

where D_x is the set of graphs containing x and $D_{f_{\varphi i}}$ is the set of graphs which contain subgraphs of x which are in the index. If the discriminative ratio is close to 1, we know that the discriminative power is low.

gIndex is a prefix tree data structure. Its nodes are of 2 types - *discriminative* and *redundant*. Each node's key is a text string which represents the subgraph. It is serialized and canonized based on special application of DFS algorithm. This technique is called *DFS Coding* and is described in [18].

Discriminative nodes are both frequent (based on given *size-increasing support function*) and discriminative (based on specified γ) and they contain a list of IDs of all graphs in the database which contain the particular subgraph. Redundant nodes are present just to satisfy the structure of the *gIndex* tree.

The root of the tree is an empty graph, whose candidate set is the whole database. Level 1 of the tree is the set of vertices (graphs of size 0). Each node in the tree (from level 2) has 1 more edge than its parent (because of the canonization it has its parent's key as its prefix).

It would be very inefficient to check all subgraphs of a query graph. But, we know that if subgraph g is not present in graph G, then no superstructure of g is present in G. Also, we know that if g and h are subgraphs of G and $g \subset h$, then the candidate set generated by h is a subset of candidate set generated by g and therefore it has a bigger pruning power and usage of g is redundant.

From the two previously mentioned statements it is apparent what is the search algorithm. We need to enumerate all fragments of query graph q starting from 1-node fragments and iteratively enlarge the fragments by adding 1 edge each time. We stop this process at the point where the fragment is not in the index anymore.

Each of the fragments which were created in the last iteration can be found in the index. We only need to check whether the matched node in the index is discriminative or redundant. If it is redundant, we find the closest discriminative node on the path to root. Having the set of matched discriminative nodes in the tree, we compute an intersection of their sets of matched graphs in the database to get the desired candidate set.

2.2.4 GIRAS

As *gIndex* comes with an idea of indexing frequent and discriminative fragments, GIRAS indexes rare and discriminative fragments. The idea is to get higher pruning power and put the indexing focus on the graph features which are specific for a particular record in the database. Ultimately, to have a unique index for each graph in the database. This leads to much smaller index size.

For getting the rare fragments it utilizes the modified version of *gSpan* algorithm [18]. Although, the original *gSpan* is designed to get all subgraphs whose support in the database is *n or higher*, the modified version finds all the subgraphs whose support is equal to n.

The modified *gSpan* utilizes minimal DFS codes which were already described in *gIndex* section. It starts with an empty DFS code and in each call it finds all the possible right-most extensions from the whole database. For all of them it finds out whether they are minimal DFS codes and, if so, it checks what the support of this subgraph is. If it is equal to the specified support f, the subgraph is added into the result set. If the support is higher, we continue recursively.

Note that it returns only the minimal rare substructures with a given frequency. This is important since the extensions of these minimal rare substructures with the same frequency would not give us any more pruning power but it would increase the index size significantly.

The GIRAS itself then calls the modified gSpan. It starts for $f = 1$. After each call of modified gSpan it checks which database records are represented by the result set of gSpan. If there are database records which are not indexed yet, the modified gSpan is called iteratively with $f + 1$. Once there are all database records indexed, we are finished. The last f is called f_{min} and it is the threshold defining the meaning of rare substructure.

Although it is not discussed in the paper [14] what data structure it uses for the index representation, we found out from the source code obtained from Dr. Azaouzi, the author of the described research, that it uses very similar data structure which was described in *gIndex* section, as well as the same technique for the querying process.

2.2.5 GString

All other methods can be used in any graph database. On the other hand, GString method is very specific for the organic chemical databases (but can be internally modified to support different graph databases with specific content).

The main ideas come from the knowledge of common structures of the graphs in the database. The chemical compounds consist of 3 types of semantic structures - paths, cycles and stars (a central node with a fan-out). Each chemical compound can be converted into a graph whose nodes are not atoms but one of the mentioned structures. This converted graph is significantly smaller than the original one.

The other observation is that we can omit the hydrogens since their number can be easily computed and we can omit the labels of carbon atoms and single (saturated) bonds.

Based on previous preliminaries, each graph in the database can be shrinked to the graph of common structures. Each node contains 3 types of information:

- **Type** - path, cycle or star
- **Size** - For path and cycle it is the number of nodes, for star it is the fan-out
- **Triple** $< n_n, n_b, n_e >$ where:
  - n_n is the number of non-carbon atoms
  - n_b is the number of branches (connected paths of the length 1)
  - n_e is the number of double or triple bonds

For each such graph we can get a set of all paths up to length l. The index structure of GString method is a suffix tree of these paths, where each node is identified by tuple $< Type, Size >$ and contains a set of pointers to the *detail table* where quadruples $< n_n, n_b, n_e, id >$ are stored for matched nodes from a particular graph. The suffix tree is built from all paths up to length l from all

graphs in the database.

The candidate set is obtained as follows. The query graph itself is translated to the common structure graph by the same process which was utilized for index building. Then we just identify suffix tree nodes which were visited and use the pointers to the *detail table* in such nodes. The graph is added into the candidate set if it is represented in each visited suffix tree node and if the triple $< n_n, n_b, n_e >$ satisfies the query.

It means that for cycles, the n_n and n_e has to be equivalent in both query and database record, n_b has to be equal or lower in the query comparing to the database record. For the paths and stars all three attributes has to be same or lower in the query.

Note that the answer set of this method can be different from previous methods. Let us take a path of four carbons $c - c - c - c$ as an example of a query and assume that the benzene (cycle of six carbons) is a part of the database. The previous methods marks the benzene as a *match*. On the other hand the GString will filter it out from the candidate set because it finds out that its *common structure* graph is completely different.

However, this is a correct behavior for the chemical compound database since we can expect that if somebody asks for a path of four carbons, he or she does not expect a benzene as a result since cycles and paths have different semantics.

2.2.6 C-Tree

Contrary to the previous methods, this one does not utilize the fragments of the graph to find the candidate set. It builds the state-of-the-art tree structure where the nodes are *closures* of their children so they contain the same substructures as their whole subtrees. Also it comes with the term of *pseudo sub-isomorphism* which is similar (and weaker) to subgraph isomorphism but it can be verified in polynomial time.

The core of the C-tree method are the graph closures. Let G, G' be graphs and m be the mapping between them (graphs can contain dummy nodes for enabling mapping between graphs of different size). Let v, v' be nodes from G or G', respectively and let $m(v) = v'$. Vertex closure which corresponds to v and v' then contains a union of labels of v and v'. The very same approach is used for edges. Let e, e' be edges from G or G', respectively and let $m(e) = e'$. Edge closure which corresponds to e and e' contains a union of labels of e and e'. **Graph closure** of graphs G and G' is a tuple (VC, EC) where VC is a set of vertex closures and EC is a set of edge closures. Note that G and G' can be both graphs and graph closures.

Several approaches how to get the mapping m are described in [15] and we will not describe these in the this section to not dive too deep into the technical details.

The **C-tree** data structure is a tree where leaf nodes are graphs from the database and every internal node is a graph closure of its children. Each node has at least m children unless it is root, $m \geq 2$, and each node has at most M children, $\frac{M+1}{2} \geq m$. All operations with the tree are done in polynomial time and their implementation is analogous to those on R-trees [19]

The idea of the method is to approximate the subgraph isomorphism by a weaker statement, **pseudo subgraph isomorphism**, which can be tested in polynomial time. An important note is that pseudo subgraph isomorphism can be tested on both graphs and graph closures.

Full description of the theory behind the pseudo subgraph isomorphism would be too exhaustive for the purposes of this thesis. Very briefly, the idea is to construct a bipartite graph G between vertices of graph $G_1 = (V_1, E_1)$ and vertices of $G_2 = (V_2, E_2)$. There is an edge between $v \in V1$ and $u \in V2$ if *breadth-first search tree* around v with the paths up to the specified length n is isomorphic to the one around u. If G has a semi-perfect matching, G_1 is *level-n pseudo subgraph isomorphic* to G_2

The authors of C-tree are also proposing a recursive algorithm which can effectively obtain the information whether two nodes should be connected by an edge in the previously mentioned bipartite graph for the level n based on the bipartite graph for the level $n - 1$.

The candidate set creation process utilizes the C-tree. It goes from the root to leafs and every time it finds out that a query is not pseudo subgraph isomorphic to some node, this node and its subtrees are pruned out. Leafs which are pseudo subgraph isomorphic to the query are added to the candidate set.

The main advantage of this method is that contrary to the previous methods, this one does not loose information during the index creation time. It does not count with paths or any other fragments, the closure tree does contain all the information about all the graphs in the database. This helps to increase the level of the pruning during candidate set creation.

2.2.7 GDIndex

This method's approach is quite different to the previous ones. It tries to completely omit the verification step and therefore computationally hard usage of any subgraph isomorphism detection algorithm. It is achieved by all the subgraphs of all database records.

It uses two structures in the index:

1. Directed acyclic graph (DAG) of all subgraphs. Each node in the DAG represents a specific connected subgraph. Each such node contains also the information whether it refers an actual record in the database. There is a directed edge from node N to node M if N is a subgraph of M, N contains exactly 1 vertex less than M and N is an induced subgraph of M.

2. Lookup hash table of subgraphs. There is a record in the hash table for each node in the DAG. For hashing, the canonical form of the graph is defined. This canonical form is derived from the adjacency matrix.

Both index building and querying is straightforward. To build the index we just take each graph, add it to the DAG and by gradual removing of its vertices we repeat the same procedure for all its subgraphs. In each step we just need to check whether such node already exists in the DAG which we can easily achieve using the lookup table.

To reduce the number of subgraphs, the canonization technique is introduced and from all isomorphic subgraphs only one is used in the index. This canonization technique is very similar to the DFS codes described in *gIndex*, however, instead of minimal DFS code it is using maximal adjacency matrix serialization (but both approaches are equally strong and have the same computational difficulty).

Querying is even simpler. All we need to do is to create a canonical representation of the query graph and use the lookup table. If the particular record is not present in the index, we know that the candidate set is empty. If there is such node, we recursively iterate through all its descendants in the DAG and find all pointers to the database graphs. Since we are using hash table, we can get false positives. Therefore, for each record in the matched row of a hash table we need to compare the exact canonical code and we will use only the the record which is exact match.

The big advantage of this method is that we do not have to do the NP-complete subgraph isomorphism test since we store the subgraphs in the index and we have the canonical representation.

What we have found as a missing piece (and there is no information about this case in the paper) is that the query does not have to be an induced subgraph of any node in the database. It can be more sparse. In this case we cannot expect the exact match of the canonical code and therefore we cannot expect any results.

The possible solution to fix this problem would be to index all the subgraphs instead of just induced ones. On the other hand that would have serious impact on the index size.

2.2.8 Benchmark Results

GraphGrep, *GIndex*, *GString* and *C-Tree* have been compared in [12]. As the testing data set the AIDS Antiviral Screen Dataset [20] was used. It contains 43 000 molecules with an average number of 25 vertices.

All measured metrics except for the speed of index creation had the same winner. The *GString* algorithm outperforms the others in the size of index, accuracy of the candidate data set and the search time.

On the other hand, in [13] we can find the benchmark of the *GraphGrepSX* method which looks like a more generic version of *GString*. While in [12] *GString* outperforms *CTree* just by few percents, in [13] *GraphGrepSX* outperforms the *CTree* by the two levels of magnitude despite larger candidate sets.

In [16] there is a comparison of *GDIndex* and *C-tree* where *GDIndex* significantly outperforms *C-tree* in all measured metrics - the size of index and its construction time and the search time.

What we may question is that how *GDIndex* would perform over a database with larger graphs such as the AIDS dataset which was used in experimental parts of all other methods.

In [14] we can find a benchmark of the *GIRAS*, *C-tree*, *gIndex* and couple of other approaches. On the AIDS dataset *GIRAS* outperforms *gIndex* and *C-tree* in all query sizes. In the dataset with bigger graphs, *GIRAS* outperforms the other two methods only in larger query sizes (12 vertices and more).

What is not measured in [14] is the size of index and time needed for index construction.

2.3 Database Management Systems Utilization for Subgraph Querying

Surprisingly we have not found many articles about substructure querying in DBMS using just their native way how to structure data and their specific query language.

The first approach [21] we found is about the utilization of relational database management system and SQL queries. The second one [22] is referring about utilizing a graph DBMS, Neo4j [23], and its query language Cypher.

2.3.1 SQL Substructure Search

Contrary to typical subgraph matching algorithms which use variations of the depth-first-search algorithm, the authors of [21] come with an SQL based solution which utilizes the principles of the BFS.

In the database the molecules are described as follows. The database contains 3 tables - molecules, atoms and bonds. The bonds have an extended type column which is a string identifier that identifies bond type and types of both end atoms type (e.g. there is a unique identificator of two carbons connected by double bond).

The bond table has three indices built on top of it. The first one is built for bond type which helps us to do efficient filtering, the second one is built for

atom1_id column (a reference to the atoms table) which helps us to get all neighbours for each atom. The last index is built based on unique identifier of records in bond table by atom pairs.

When the substructure query is obtained, the minimal spanning tree is constructed. The value of each edge depends on the statistics of the database. We can say that the most rare atom-bond-atom edge has the lowest value. Also in this tree we find a root node which has the least valuable edges on it. This spanning tree will help us to construct an efficient SQL query, because thanks to the spanning tree minimality and the root selection the constraints (edges) with the highest probability of failure will be checked first.

The query itself uses only the edge table. It starts from the root of the spanning tree. For each edge there is a specification of an extended bond type and specification of a join to other instance of edge table. At the end there are edges which are not a part of a spanning tree.

As an example we can use a subgraph query where we want to find all structures which contain $O = C - N$. The bond $C - N$ is more rare in the sample database and therefore this bond is described as the first one in the query. The query itself would look as follows:

```
SELECT b1.compound_id, b1.atom1_id, b1.atom2_id, b2.atom2_id
FROM bonds b1, bonds b2
WHERE b1.bond_type = "C-N" and
      b2.atom1_id = b1.atom1_id and
      b2.bond_type = "O=C"
```

where $C - N$ means carbon and nitrogen connected by a single bond and $O = C$ means oxygen and carbon connected by a double bond.

This example is quite simple. On the other, hand we need to build an SQL query which describes the whole *Constrain Satisfaction Problem*. It means that for each pair of bonds, we have to define whether their atoms do or do not have the same IDs.

Where it is possible, we can force usage of built indices. For the first edge we should use the index built for the bond type column. For other spanning tree edges we should use the index for *atom*1_*id* column which literally does the BFS. For edges outside the spanning tree we should use the index built for *atom*1_*id*, *atom*2_*id* pair since we already know the IDs of both atoms of the edge we need to check.

2.3.2 Neo4j Substructure Search

Hoksza et al. in [22] describe their case-study of mining the protein graphs. They use the Neo4j graph DBMS to store the protein database and query it by the Cypher language.

They found out that the query time is factorial with respect to the number of edges in the query. Beginning from size 15, the queries were impossible to execute in a reasonable time and therefore they recommend the usage of Neo4j only for small subgraph queries.

They have also tried to compare their results with results for an SQL database. However, the SQL results significantly outperform Neo4j. But the comparison is not fair enough since the SQL approach used pre-computed neighborhood relations and therefore had a significant advantage in comparison with Neo4j.

However, based on this paper we can be pessimistic in case of Noe4j utilization, we should keep in mind that the database had a different structure comparing to our molecule databases which are the target of this thesis. Graphs used in the experiment have an average size of more than 500 edges. On the other hand, typical molecule databases contain significantly smaller graphs and therefore we cannot be sure that the numbers from the mentioned paper can be applied also for such databases.

2.4 Commercially Used Solutions

In this section we introduce three real-world solutions. The first one is the AMBIT project [24] which offers chemoinformatics functionality via REST web services. One of the functionality is, of course, the substructure search. This project represents a standalone solution - the querying is not dependent on any particular database management system.

The second solution, JChem Cartridge [25], is an example of an Oracle cartridge [26]. The reason why we picked this cartridge from the set of existing ones is that it has the best results in the benchmark presentation at [27].

The third solution, ABCD Cartridge [28], is a pure commercial one developed by the Johnson & Johnson company [29]. We picked this one because its architecture is well described in [28] despite the software is not publicly available .

2.4.1 AMBIT-SMARTS

AMBIT-SMARTS is a Java based software built on top of the Chemistry Development Kit (CDK [30]). It implements the whole SMARTS querying language specification [31] for querying chemical databases. It uses two indices. Both are in the form of a bitstring which is stored for each record in the database.

Each bit in the first bitstring represents whether some structure is a part of the particular record. The structures are of two kinds.

The first set of structures is selected automatically based on the database content. It considers each atom's topological layers. The first topological layer is the atom and all its neighbours. n-th topological layer is the whole (n-1)-th layer

and some or all of its neighbours. All such structures up to a selected layer level are recorded. Structures which are a part of at least 50% of database records are considered as those which will be represented in the bitstring.

The second set of the structures represented in the first index is selected by the database administrator who should be aware of what types of queries are most likely to be used in such database.

The second bitstring represents all paths up to length 7. Because the number of these paths is enormous, they are not represented directly in the bitstring, but they are at first hashed and this hashed value is added (by logical OR) to the bitstring. This concept is called *fingerprints* and it is described in [32].

2.4.2 JChem Cartridge

The JChem Cartridge is a part of the JChem package from ChemAxon [33]. It allows users to build their chemical database in the Oracle database easily. A part of the cartridge contains tools for chemical formats conversion, similarity search and sub-structure search. It also implements functions for SMARTS queries.

With regards to the substructure search it filters the database based on the fingerprints which are present for every molecule. It uses the hashed fingerprints similarly to the AMBIT-SMARTS. The keys for hashing are:

- All paths in the molecule up to a specified length
- The branching points (atoms with degree higher than two)
- All cycles

The fingerprint itself is generated based on 3 user-defined parameters:

- The length of the fingerprint
- The maximum path length (how long paths are used for generating the hash keys)
- How many bits are set to 1 for each hash key

In the documentation there is stated that for the substructure search the optimal values in most cases should be 512 bits long fingerprints, the maximum path length set to 5 or 6 and the number of bits per hash key set to 2.

The cartridge also has a tool for analyzing the efficiency of the fingerprints. As a good metric the idea of *darkness* is used. Darkness is defined as a ratio between numbers 0 and 1 in the fingerprint. The analysis tool provides the user with information about the lowest, average and highest darkness in the database and also provides a distribution. The darkness should be as low as possible, highest values should not exceed 80%, but best performance is expected under 66%.

2.4.3 ABCD Cartridge

ABCD is an integrated drug discovery informatics platform developed by the Johnson & Johnson Pharmaceutical Research & Development, L.L.C. It consists of a set of algorithms for subgraph isomorphism checking and index building and an interoperability layer, cartridge, for the Oracle database which enables the RDMS to use the algorithms and indices during the SQL query evaluation.

For the filtering it uses a set of hashed fingerprints. There are 5 types of fingerprints which are used for each molecule - atom, edge, ring, path and cluster fingerprint. For each type there is a different algorithm which generates the hash keys. Also for each hash key, the number of occurrences of a particular feature is stored.

Contrary to AMBIT it does not store the fingerprints for each record in the database. It utilizes the concept of inverted bitstrings.

The algorithm proceeds as follows. Every molecule in the database is analyzed and the set of hash keys along with the number of occurrences in that molecule are computed. The information for each key is stored as a triplet h, c, m, where h is the hash code, c is the number of occurrences, and m is the ID of the molecule in the database. The list is then traversed and for each unique hash code, h, a series of binary masks, $M(h, cmin)$, are defined, where $M(h, cmin)$ contains the IDs of the molecules for which the hash code h occurs at least $cmin$ times.

For more compact representation of the inverted bitstring there are three types of their representation where N is the size of database and K is the number of database records in the matching set:

- If $K < \frac{N}{32}$ then the representation is an array of IDs of database records which belong to the set.
- If $(N - K) < \frac{N}{32}$ then the representation is an array of IDs of database records which do not belong to the set.
- Otherwise it is stored as a classing bitstring where n-th bit represents whether n-th record belongs to the set.

3. Experimental Work

3.1 Introduction

During the research of the related work, many questions have arisen. The papers are usually very brief and they miss a lot of implementation details. Sadly, even if we tried to contact the authors, we did not get the original source code for the described methods, nor for the described benchmarks. The only exception is the *GIRAS* method where we were successful in contacting its author and we do have the complete implementation.

All the benchmarks we mentioned in the previous chapter were a part of the papers which describe each particular method. Knowing that we cannot be much surprised that each presented method outperformed the others. The question is whether we do get the same results on different data sets.

The other interesting question is how the winners of the various benchmarks would perform on the same data set. For example, when *GString* outperforms *C-tree* just by few percents in [12] and *GraphGrepSX* outperforms *C-tree* by two levels of magnitude, we cannot implicitly say that *GraphGrepSX* would outperform *GString*. There might be three reasons why this presumption might be wrong:

- The lack of knowledge of the tested data set. In most of the papers there is an information which dataset has been used. On the other hand, there is usually no information about which part of the dataset has been used since the dataset is usually cut down to only a small part of the original size. Moreover, not all the benchmarks are using the same datasets at all.
- The lack of knowledge about the implementation of the verification step. In none of the mentioned papers is an information about which algorithm has been used for the final subgraph isomorphism testing. This can cause quite a significant difference in the final query measurements (although it cannot influence in the candidate set time computing).
- We do not even know how much time the authors spent on the optimization of the code itself. Whether they cared more about the code readability and maintainability of the code or whether they did try to optimize the code as much as possible. Moreover, we do not know anything about which languages and compilers have been used.

What we did not find at all is some comparison of the performance of the described indexing techniques and utilization of SQL or graph databases. It might be interesting to see how significant difference in performance we get when we use very graph specific technique comparing to the very generic ones which the databases offers.

In the following sections we will describe what hypotheses do we found interesting to prove or disprove and we describe the process and the implementation

of those proofs.

What is probably fair to mention is that due to the brevity of the related work we cannot be sure whether we did not omit some important part of the algorithms. There have been a lot a situations where we had to improvise since we found out that some very important implementation detail has been omitted in the method descriptions. These cases will be described in following sections as well. Although, we did implement all the methods with opened mind without any endeavor to make some method better or worse, we cannot guarantee that we did not do any mistake or bad implementation decision which can influence the final benchmark results.

3.2 Hypotheses to be verified by the experimental work

In this section we will list several hypotheses which came to our mind during the related work research.

3.2.1 Hypothesis 1: GString vs GraphGrepSX

GString and *GraphGrepSX* use very similar data structures for indexing the database. The main difference is that *GraphGrepSX* uses all graph paths, whereas *GString* uses all paths in the condensed graph. Also *GString* uses heuristics which are very specific for our field of research, i.e. the organic chemical databases.

- **Hypothesis H1.1**: The index size of *GString* will be significantly smaller compared to *GraphGrepSX* due to the condensed graph usage.
- **Hypothesis H1.2**: Due to the specificity of *GString*, it will outperform *GraphGrepSX* which can be used for any graph dataset.

3.2.2 Hypothesis 2: GIRAS performance for large queries

As described in paper [14], for small queries (of size 4 and 8) the performance of *GIRAS* is about the same as *C-tree*. On the other hand, for larger queries, the performance is ten times better comparing to *C-tree* and even better results there are for the candidate set sizes. What we may question is how it will perform comparing to *GString* and *GraphGrepSX*.

- **Hypothesis H2.1**: Based on the benchmark results we expect *GraphGrepSX* will outperform *GIRAS* despite the smaller candidate sets.
- **Hypothesis H2.2**: Time to build *GIRAS* index will be significantly larger compared to other methods since the algorithm seems to be computationally complicated

3.2.3 Hypothesis 3: How the SQL and graph oriented databases perform in comparison with the domain specific solutions

We may question what performance we may get when we use an SQL or a graph database. In this case we do not need to implement any special algorithms for index building, we just use the possibilities of the databases, i.e. create a query which describes the subgraph and in case of SQL databases to build the indices to help the query process.

- **Hypothesis H3.1**: Domain specific indexes will perform much better. I.e. methods where we are building the index will perform better than SQL and graph database.
- **Hypothesis H3.2**: Graph database will perform better than SQL database because it runs completely in memory and is optimized for querying graph data.

3.3 Description of the Experimental Work

In this section we will describe the implementation details of the experimental work. Based on the uttered hypotheses we have implemented:

- *GraphGrepSX* and *GString* algorithms
- Adapter for the *GIRAS* implementation obtained from Dr. Azaouzi to be working on the same dataset
- Tools for inserting and querying an SQL and a graph database

The whole implementation has been written in Java language [34]. Most of the work uses Java version 10, a graph database adapter uses Java version 8 due to the technology dependencies.

For the chemical database parsing we use the Chemistry Development Kit [30] version 2.1.1, a Java library for working with chemical formats and data structures.

In case of verification step for the *GraphGrepSX* and *GString* algorithms, we are using the *SMARTSQueryTool* from the Chemistry Development Kit. It uses *Ullmann* [1] algorithm inside.

3.3.1 GraphGrepSX

Since the *GraphGrepSX* algorithm is very simple, the implementation was quite straight-forward.

We had to do only one change in the algorithm to make it applicable to our use-case. The original description of the algorithm expects that the suffix tree

represents the vertex label paths. Since we need to represent even the edge labels we have changed the original suffix tree presumption so that the odd levels of the suffix tree represent the vertices and the even levels of the suffix tree represent the edges.

The previous statement does not affect the maximum path length parameter l of *GraphGrepSX* algorithm. It is still valid that this parameter sets the maximum length of the index path with regards to the number of vertices, therefore the index tree will have depth up to $2l - 1$.

For our experiments, we have set the parameter l to the value of 6.

3.3.2 GString

In contrary to the previous *GraphGrepSX* description, the *GString* algorithm description offers a wide range of pieces which were not described at all. Most of the unknown parts are related to the original graph reduction process where the graph representing the atoms and bonds is transformed into a graph consisting only of nodes representing cycles, stars and paths and edges representing the connection between these structures.

The first issue which we faced was the process of extracting the cycles from the original graph. In the algorithm description there are no references on how to extract the cycles, nor which method should be used. The obvious issue is that the cycles are not necessarily independent. They can share both vertices and edges and in some cases the vertices and edges can be shared even by several cycles.

After some research we have found out that the Chemistry Development Kit has an utility for retrieving *MCB - Minimum Cycle Basis* (also known as *SSSR - Smallest Set of Smallest Rings*) described in [35]. *Cycle Basis* is defined as a set of cycles by which one can express any other cycle present in a particular graph as the result of a symmetric difference operation on the cycle basis.

The *MCB* is defined as a cycle basis which consists of the shortest possible cycles. A good example might be naphthalene which we can see in Figure 3.1. It contains three cycles, two of size 6 and one of size 10, and any pair of these can serve as a cycle basis. On the other hand, there is only one *MCB* which consist of two cycles of size 6.

In this picture it is also clearly visible why we cannot use all the cycles. If all three cycles were represented in *GString* graph, it would be very unclear what is the the relationship between these cycles and how they should be connected in *GString* graph. Also, it may lead to false positives from chemistry point of view because naphthalene consist of two aromatic cycles and it does not make sense to include an information about the cycle of the size 10.

The *MCB* finder utility requires specification of the maximum cycle size pa-

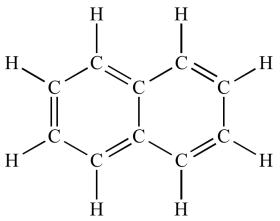

Figure 3.1: Molecule of naphthalene

rameter. This parameter defines a threshold above which the cycles are not considered as cycles. When we tried to set this threshold high enough to not omit any cycle in the testing database, we had big issues with performance and in some cases the process died on the lack of memory. Since the target of this thesis is to measure the performance in usual use-cases, we have decided to set the threshold to 10 which should cover the vast majority of real cases.

Bigger cycles are described as paths of the length equal to the cycle size. These paths begin at each point in which the cycle interfere with another *GString* structure. For each such interference there are two paths, one in each direction

Another question which arose is how to set the threshold which defines the minimum degree of an atom to be considered as a star. The original thought was to set the threshold to 3. The reason was that if we set this threshold to a higher number we get another problem to solve - how to handle path joints. We can demonstrate this problem on methyl propionate in Figure 3.2

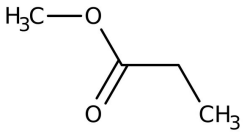

Figure 3.2: Molecule of methyl propionate

We can see that there are six possible paths, two are between the carbons (one from each side), and four between each carbon and oxygen (again, one from each side). If we define the threshold of stars to three, we do not have to handle such situations because in our algorithm, we extract the stars first and then we are finding paths connected to already found stars and cycles. In this case we would have 1 star (the atom in the middle) and two paths connected to it (the oxygen connected by double bond would be considered as a branch which is by definition a path of length 1). This would simplify the algorithm quite a lot because we would not need to handle paths which are connected to another paths.

During the testing we found out that it is possible to use this threshold but in practice, we would loose the majority of results. The reason is the same as we

described in the first chapter. If we try to query a path which may be described as $C - O - C - C - C$ there is obviously such path in methyl propionate but our algorithm would filter this candidate out, because it does not contain a path of length 5 but a star and two paths of length 2. Since this is a very common case (it is very rare that chemical compounds do contain long paths without any branching), we had to use a higher threshold and develop some logic for handling the connected paths.

What we did was to implement DFS which finds all the paths and all of these paths are included into the *GString* graph. In case of methyl propionate there would be 2 independent paths, since we do not have a connection to any other structure, we start DFS in random atom with degree 1. This is quite a special case because the molecule consist only of paths. If the methyl propionate would be connected on one end to a star or a cycle, there would be 2 nodes in the *GString* graph - one cycle/star and two paths connected to this structure.

The rest of the algorithm mimics the *GraphGrepSX* implementation including the notes described in section 3.3.1. The only difference is that due to the fact that we expect significantly smaller graphs due to the condensation process, we have set the parameter l to value of 5.

3.3.3 SQL Database

We have based our implementation on the proposal in [21]. We have chosen the Oracle Database 12c. For the Java API we have used Oracle Database JDBC driver 12.2.0.1.

Based on the mentioned paper we have designed our table with 5 columns - **ATOM1_ID**, **ATOM2_ID**, **BOND_ID**, **BOND_TYPE** and **COMPOUND_ID**.

The implementation itself is quite straightforward and it consist of two parts. The first part is a routine for the database creation. In this routine we just iterate through the whole database and for each molecule, at first, we iterate through all its atoms and assign an unique ID to each of them. Later, we iterate through all the bonds and for each we create one **INSERT** statement.

We already have the IDs of atoms and the compound ID (this comes from the original chemical DB on the input), we generate an unique ID for the bond itself. The type of bond consists of the type of each atom at the bond's end and the type of the bond itself. Each bond, if it is not symmetrical, is represented by two rows in the database, because we need to make the graph representation undirected. During the process of inserting the bonds we are updating the in-memory statistics - we are maintaining the count of rows for each bond type.

The inserts are happening in batches. We did test the performance and found out that batch size of 50 rows for one **INSERT** statement is quite optimal.

The second part of the implementation is the query building. As proposed in [21], at first we build the minimal spanning tree of the query graph. The edge value is based on the database statistics which we gather during the insert phase. For spanning tree construction we have implemented Kruskal algorithm [36].

Then, in the spanning tree we find the edge with the lowest value and from this edge we start a BFS algorithm and for each edge we add the rule into the **SELECT** statement. We also need to mark all the neighbours by stating that atom ID of one edge is equal to the atom ID of the neighbour edge. The same we have to do for non-neighbours. For each such pair we have to explicitly state that their atom IDs are not equal. The same we have to do for the bonds, we need all the bonds unique so we have to state for each pair of bonds that their IDs are not equal.

Since we are interested only in the information whether the subgraph is present in particular compound, we start the **SELECT** statement with **SELECT DISTINCT b0.COMPOUND_ID FROM ...** which returns the set of compounds matching the subgraph query which is exactly the result we need.

As an example of SQL query building process we may use the path of 4 carbons connected by single bonds. The **SELECT** statement looks like following:

SELECT DISTINCT b0.compound_id

As we use only bonds table, we need to specify that we want to join three instances of bonds table, one per each bond in the query graph:

FROM BONDS b0, BONDS b1, BONDS b2

As a next step we need to specify that all bonds are distinct:

WHERE b0.BOND_ID != b1.BOND_ID AND
b0.BOND_ID != b2.BOND_ID AND
b1.BOND_ID != b2.BOND_ID AND

As a last step, we need to specify all the constrains for each bond. At first we specify the specification which atoms are shared with previous bonds to mimic BFS. Next, we describe the type of the bond which should help a lot with pruning. In last step we have to specify that all the atoms which were not marked in the first part are distinct, i.e. every pair of atom IDs has to be distinct:

/*first bond*/
b0.BONDTYPE='C-C' AND
/*second bond*/
b1.ATOM1_ID = b0.ATOM2_ID AND b1.BONDTYPE='C-C' AND
b1.ATOM2_ID != b0.ATOM1_ID AND
b1.ATOM2_ID != b0.ATOM2_ID AND
/*third bond*/
b2.ATOM1_ID = b1.ATOM2_ID AND b2.BONDTYPE='C-C' AND
b2.ATOM2_ID != b0.ATOM1_ID AND
b2.ATOM2_ID != b0.ATOM2_ID AND
b2.ATOM2_ID != b1.ATOM2_ID

3.3.4 Graph Database

As observations of paper [22] state that the Neo4j is not performing well in subgraph querying, we have tried to look for alternative graph databases which can perform better. We found the graph analytic tool *PGX* [37].

It is not a graph database in its original meaning. It is a toolkit for graph analysis with an ability to load and store graphs from / to various data formats. It does support an SQL-like query language called *PGQL* [38] and it advertises a scalable solution with a focus on high performance.

Since it is an analytic tool and not a real database, it does not support the ACID transaction model as other databases do, but for the purposes of this thesis it does not take serious role.

Oracle did a benchmark which compares the performance of subgraph matching in PGX and Neo4j. The results are available at presentation [39] on slide 31. Note that the benchmark compares the results on so-called *hot data*. In other words, it makes sure that the graphs which are being queried are already loaded in memory to prevent result inconsistencies due to data fetching from the disk.

The results of this benchmark are quite convincing. Although there are huge differences of result times according to the query size, PGX outperforms Neo4J in all categories.

The first issue we had to solve was that although Oracle offers windows batch files for starting PGX, we were not successful to run the database. To make the results of the performance measuring as precise as possible, we did not want to use other hardware or different operating system for executing the performance testing of PGX than the hardware and system used for other performance testing of other methods.

As a viable compromise we have decided to use the Windows 10 Subsystem for Linux utility [40] and we have downloaded the Ubuntu system to be used in this way. On Ubuntu there were no issues with the PGX database usage.

The client side was implemented in Windows environment using the PGX Java client library. The implementation is quite straight-forward. Instead of creating the graph in PGX for every single molecule in the database, we create one huge graph for each 10 000 compounds where each graph component represents one molecule. This helps the performance since we do not have to load and store the a huge number of small files, we are executing query just on several big graphs instead.

For each vertex we generate a unique ID, use a label to mark the representing atom symbol and we store a molecule ID as a vertex's property. For each edge we use a label to mark the type of represented bond.

For the querying we use the PGQL language which is supported by PGX. For

each atom in the query graph we generate a unique ID and then for each bond in the query graph we insert a rule into the query where we define that the two atoms with particular IDs and of particular type are connected by a bond of a particular type. As the last thing in the query we need to state for each pair of atom IDs that they represent different vertex.

A demonstration of simplicity of PGQL usage in our case might be a query representing the path of three carbons. We start the query with the following statement which tells us that we want to select all molecule IDs (which are being stored for each vertex) except the duplicates. Note that $a1$ is an ID of the first atom in the query graph.

SELECT DISTINCT a1.moleculeId

The following part of PGQL query describes the bonds in the query graph. The colon sign means that we describe the label of the vertex or edge. We can see that in this case we are looking for two pairs of carbons connected by a single bond (represented by character **S**), where atom **a2** is shared in these two bonds.

MATCH (a1:C)-[:S]-(a2:C), (a2:C)-[:S]-(a3:C)

As the last part we need to state that all atoms **a1**, **a2** and **a3** are different. Otherwise the query would match every pair of connected carbons since **a1** and **a3** could represent the same atom

WHERE a1 <> a2 AND a1 <> a3 AND a2 <> a3

We must admit that the work with PGQL is quite intuitive and compared to building the SQL query it is way more user-friendly. On the other hand, this is quite an expected result since PGQL is designed to query graphs and SQL is designed to query generic data.

3.3.5 GIRAS

As we were successful with the request of the original implementation of the *GIRAS* algorithm, there was not much implementation needed on the side of this thesis. For the measurement we are using the original solution. We had to implement only an adapter which translates the chemical database which we are using for other methods to the format - vertex and edge lists - accepted by the *GIRAS* code.

However, during the testing we have found out that the results do not match the results of other methods. After some investigation, we have realized that the problem is not in the implementation but in the algorithm itself. The core of the problem is in the way how the structures which are being indexed are chosen.

As we mentioned in the analysis chapter, the *GIRAS* method is trying to find the rare substructures with the condition that every graph in the database has to be represented by at least one rare subgraph. We have found out that when

we create a query which should have only several results, everything works fine. On the other hand when we build a query which should match nearly the whole database, we do not get any results at all.

We did an explicit test which proves that the algorithm cannot work properly in all cases. We have created a database with 4 molecules where each of them contains a path of four aromatic carbons but in each of them there is a unique substructure which does not contain this particular path.

When we have executed a query of the mentioned path we did not get any results. When we have added a new molecule into the database which represents the query itself, i.e. a path of four aromatic carbons, and we ran the query again, the result was that the query matches all the graphs in the database.

This observation invalidates the statement in [14] that the indexing is complete. We may then question how the results of the performance measurements are valid. If we know that the indexing is incomplete, it should be also faster since the index is smaller and therefore it should take less time to use it. So even for the queries whose results are valid, it is questionable how seriously we can take the performance numbers.

4. Experimental Results

In this chapter, we present the results of the experimental work. We have measured the following statistics:

- The time needed for building the index (or database in case of SQL and graph database)
- Sum of the sizes of the index and data representation
- Time needed for obtaining the candidate set
- Time needed for verification of the candidate set
- Hit ratio of the candidate set
- Total time for query execution

Candidate set related numbers are not applicable for SQL and graph database since we are not creating candidate set during the query execution.

We have measured the results on laptop Dell Inspiron 15 7000 with Intel(R) Core(TM) i7-6700HQ CPU processor with frequency of 2.60GHz and 16GB of RAM with installed Windows 10 operating system.

As a data set we have used the first 100 000 compounds of *ChEMBL* database [41] release 24. These are the statistics of the used data set:

- **Vertex count of smallest compound:** 1
- **Vertex count of largest compound:** 548
- **Average vertex count:** 28
- **Average edge count:** 30
- **Number of vertex labels:** 18
- **Number of edge labels:** 4

For query testing, we have created four sets of queries with sizes of 4, 8, 16 and 24 vertices respectively. Each set contains 10 different queries defined in SMILES language [42]. Queries are visible in attachment Set of tested queries. All measured numbers are attached in attachment Results of measurements.

At the end of this chapter we summarize the results and use them to prove or disprove the hypotheses formulated in section 3.2.

4.1 Index building time

We define index building time as a time difference between the moment when the chemical database is loaded into the memory and the time the index/database is ready to execute queries.

In case of *GraphGrepSX*, *GString* and *GIRAS* it means the index building itself, in case of SQL and graph database it means the set of API calls to put the data into the database. The complete results are visible in graph at figure 4.1. For better graph scale we also present results with excluded SQL number in graph at figure 4.2

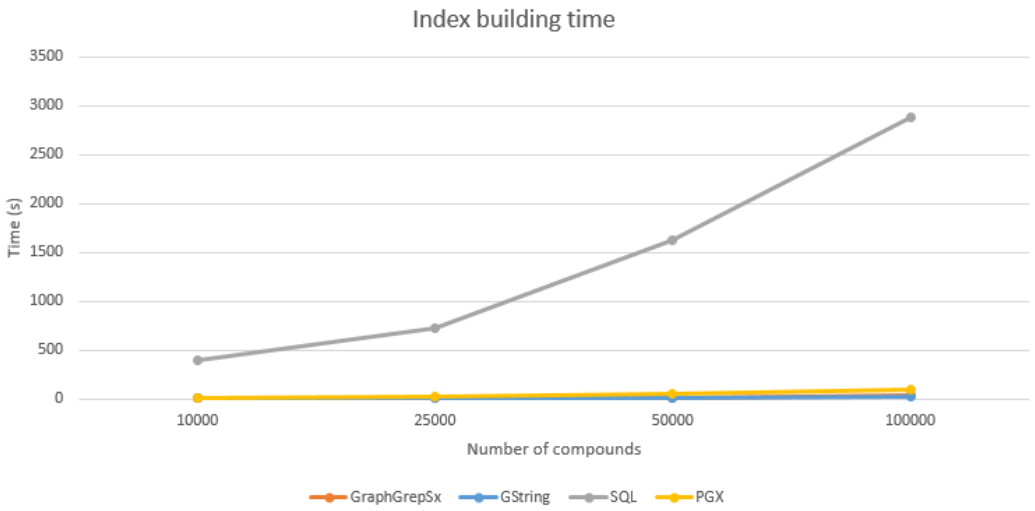

Figure 4.1: Index building time

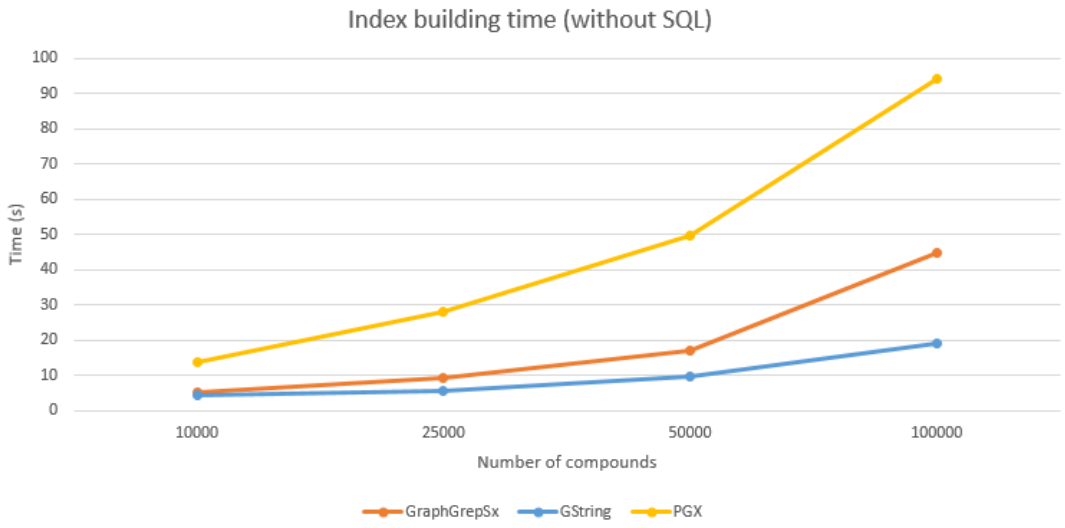

Figure 4.2: Index building time without SQL

We may see that it is significantly slower to create the database from scratch

than create just an index as it is in case of *GraphGrepSX* and *GString*. Results of database methods are quite convincing - SQL database creation time is 50 times slower compared to PGX. This is quite an expected result since PGX does work only in memory in contrary to SQL which writes all the data onto the disk.

In other two methods the difference is not so big. *GraphGrepSX* is two times slower than *GString*. There might be two reasons for this observation. The first one is that for *GString* we have used smaller parameter l compared to *GraphGrepSX*. The other explanation might be that it is worth to spend some time in condensation process because it significantly reduces the number of distinct paths in the graph and therefore it makes the index building process faster.

We are not presenting the results for *GIRAS* method. The reason is that we were not able to get the results in reasonable time. Even for the 10 000 compounds we did not get the built index even after two days of computation. The reason is that the our data set contains even small structures which are substructures of many others and therefore there are not present any rare subgraphs.

After two days of computation on 10 000 compounds we stopped at the moment where we were missing indexing of 39 compounds and the currently searched support level was 600, in other words, these 39 compounds do not contain any subgraph (of maximal size of 8 vertices) which is rare enough to be part of less than or equal to 600 compounds in the data set.

Just for verification, we had tested *GIRAS* on small datasets (hundreds of compounds) and the computation has finished in reasonable time and with expected results.

Because we were not able to build the *GIRAS* index for our data set, we do not present even the results of other metrics for this method since we had no way to measure them.

4.2 Index and data size

Since it is tricky to measure just the index size (and in graph database this term does not even makes sense) we have decided to measure the whole amount of memory needed for particular method.

In case of *GraphGrepSX* and *GString* it is the memory used by the running process after the index is built and after triggered garbage collection.

In case of SQL database we are querying the size of the index structure and **BONDS** table itself. The query looks as following:

```
SELECT sum(bytes)/1024/1024 as "SIZE in MB"
    FROM dba_segments
    WHERE segment_name='BONDS/INDEX_NAME'
```

In case of graph database we use the *getMemoryMb* method which is offered by the Java API of PGX graph representation.

The results are visible on graph 4.3. What we found as an interesting observation is the size of the *GString*'s index. After we saw these numbers we started to investigate what is the reason. It turned out that the premise that using the condensed graph to reduce the number od different paths is not valid. We have found out that the built index on the tested database contains more then one a half million nodes. The root node itself has almost 150 children, i.e. there are almost 150 different node types. This is a huge number compared to *GraphGrepSX* which contains only 21 vertex node types.

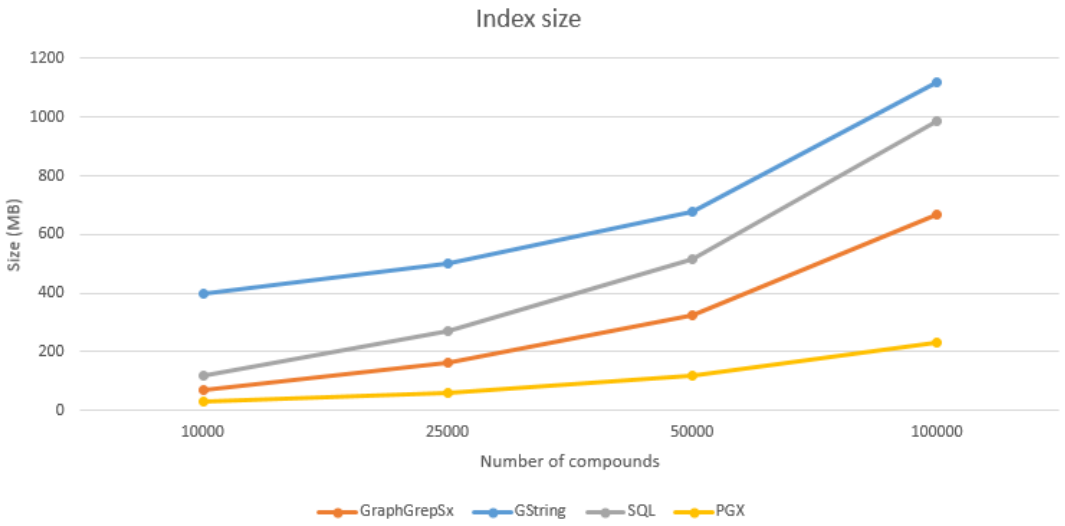

Figure 4.3: Index size

Other results are quite expected. The reason why PGX data representation is significantly smaller compared to SQL is that PGX does not build any indices. The amount of memory consumed by SQL table representation (without the indices) is about the same as for PGX.

4.3 Candidate set creation time

This metric is meaningful only for *GraphGrepSX* and *GString*. As we described in chapter 3, the candidate set concept is not applicable to SQL and graph database.

By candidate set creation we mean the time difference between the moment when a query is executed and the point in time when we finish the index utilization for particular query.

The results are visible on graph 4.4. We can see that *GString* is significantly slower compared to *GraphGrepSX*. This is most probably because of the significantly bigger index size.

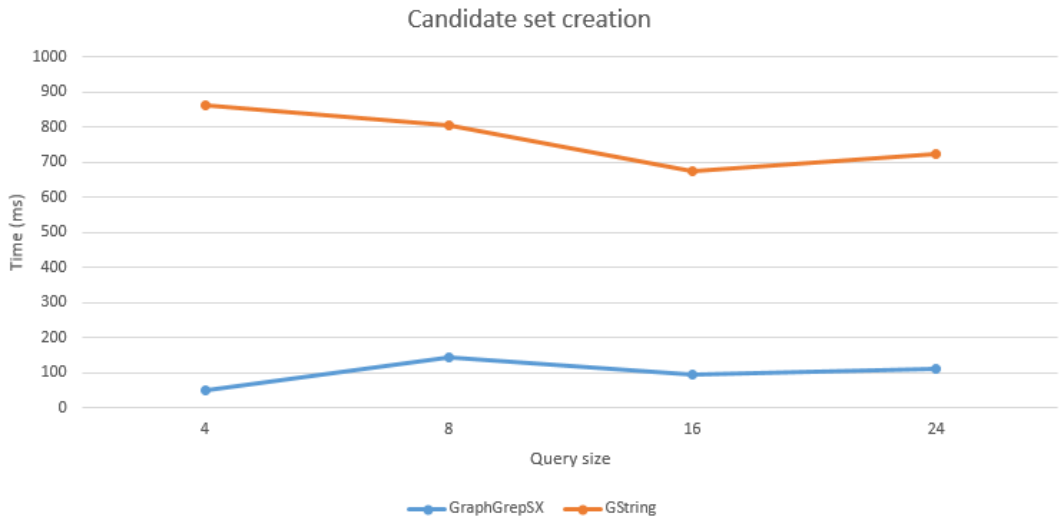

Figure 4.4: Candidate set creation time

The other interesting fact for *GString* is that due to the "stars versus paths" issue described in section 2.2.5 it is almost impossible to get meaningful results for large queries and therefore the candidate sets are cut down to almost empty sets.

4.4 Verification time

Verification time does have two different meanings in this context. For methods where we create the candidate set, we understand verification time as time needed to verification of the candidate set.

In case of SQL and graph database where we do not work with the candidate set concept we understand verification time as a time needed for executing the query since we need to verify every single record in the database.

The results are visible on graph 4.5 in which we can observe several interesting outcomes.

At first, we can be surprised by very low numbers for verification time in case of *GString*. This is caused by significantly smaller candidate set compared to *GraphGrepSX*. On the other hand, the candidate set is not smaller because of better pruning ability of *GString* index but because of the fact that *GString* invalidates even the results which are valid for other methods. This was described in detail in section 2.2.5.

The other interesting observation are the numbers for PGX. Although, the numbers for queries of sizes 4 and 8 are very good (even better than *GraphGrepSX* which is indexed) we have found out that for queries with the size bigger than 14 it is barely usable.

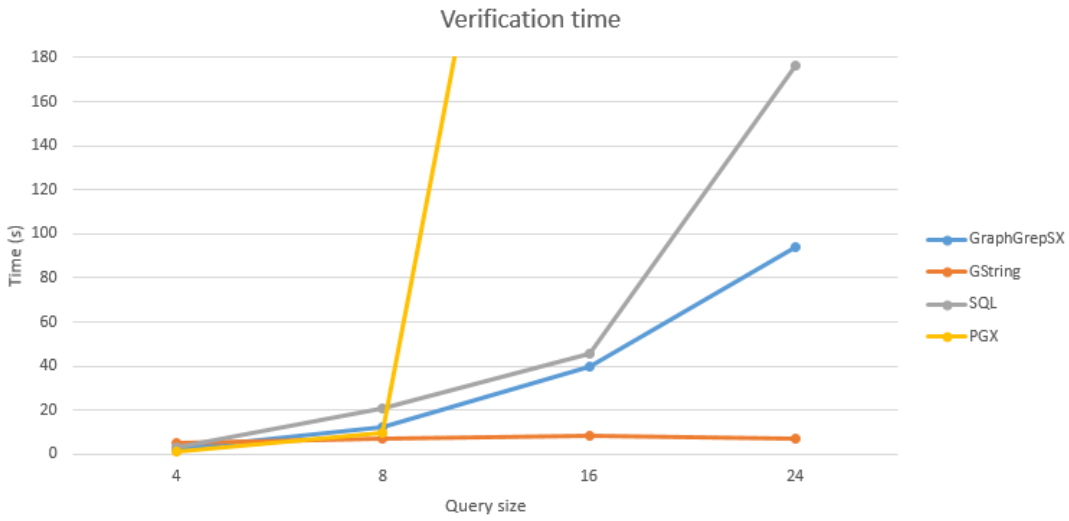

Figure 4.5: Verification time

Also, we have tried to test it even on database with 1 graph with 2 vertices and 1 edge between them. We would expect that any query will end pretty quickly because there is not much to compute. However, we have found that even on this small graph, big queries are very slow and the complexity grows exponentially, while query with 12 vertices took 46 seconds and query with 14 vertices took 50 minutes. Even after 3 hours of computation we were not able to get results for query with 16 vertices.

It seems that PGX spent a lot of time on PGQL query parsing and on creation of execution plan. We were unsure whether we did not anything wrong. Luckily, author of this thesis knows a person who works in Oracle in PGX team. This person confirmed that the query structure is correct and that it takes enormous time even on Oracle internal infrastructure. We may then doubt how valid results are described in presentation [39] where very promising numbers even for large queries are presented.

4.5 Hit ratio of candidate set

This section is only applicable for methods which create the candidate set. The metric is defined as a ratio between the candidate set size and the answer set size. It measures the quality of the index, the higher the ratio is, the better results are obtain from the index.

The results are visible on graph 4.6. We can see that for *GraphGrepSX* the efficiency of its index decreases with the query size. This is natural since the *GraphGrepSX*'s index describes only paths of length up to 6. Therefore, it is expectable that with growing size of query, the accuracy will decrease, since the indexed paths cover smaller and smaller portion of the query.

On the other hand, even queries of size 24 are not big enough to overflow the

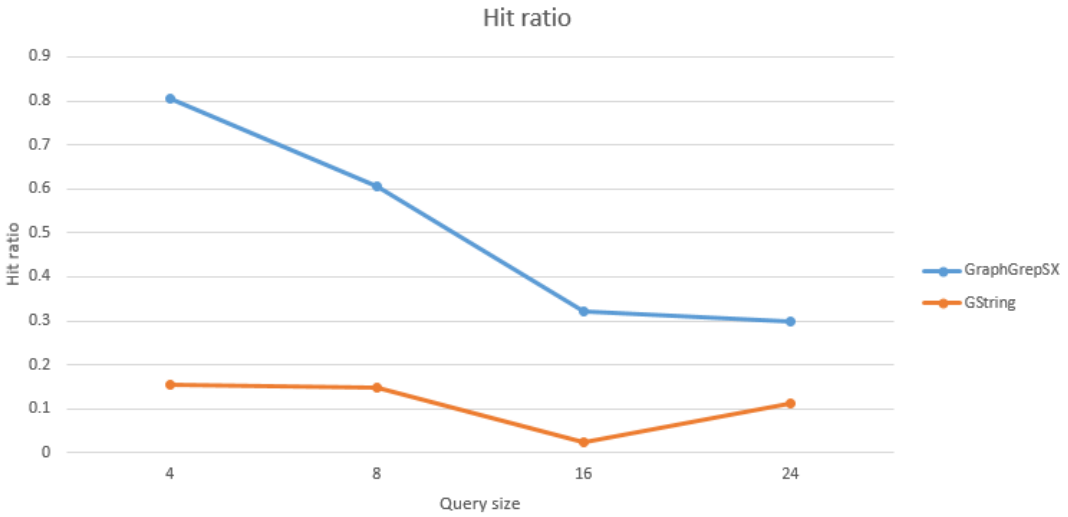

Figure 4.6: Candidate set hit ratio

capacity of *GString*. The condensed graph does not contain paths longer than 5 in these cases and therefore we would expect more or less constant hit ratio for all query sizes which matches the actual results. However, we can see that the hit ratio is significantly smaller compared to *GraphGrepSX*.

4.6 Query execution time

In this section we describe the time results for the whole query process. It can be defined as a sum of the candidate set creation time and verification time. Note that in case SQL and graph database this is equal to the verification time. For the end user, this is probably the most crucial metric.

The results are visible on graph 4.7. The first thing we can observe is that this graph is not much different to graph 4.5. In other words, the time for obtaining candidate set plays only very minor role in total query time.

The very good results of *GString* are spoiled with the fact that the results are very much different to other methods. On the other hand if the user knows what are the *GString* restrictions, it may be very efficient way of querying.

In case of small queries, the best choice seems to be PGX. The implementation is very straight-forward and most of the work is very intuitive. Also, the implementation of PGX handler is very easily improvable to work with even much bigger data sets which cannot fit into the memory.

In case of SQL, we are quite surprised that it is a viable solution. The difference in performance times to other methods is not that significant as we would expect. Also, SQL solution is the only one which does not have to fit into memory as it is.

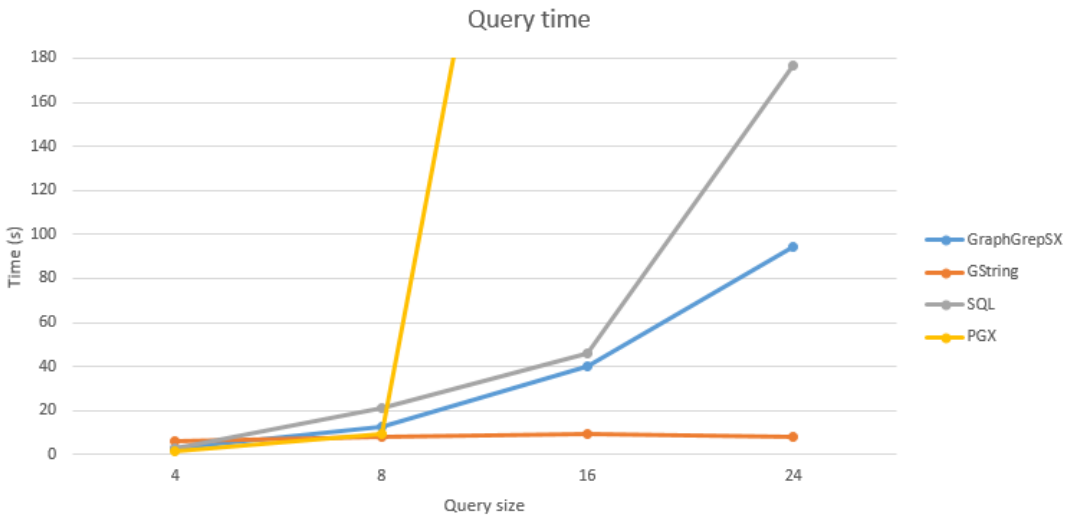

Figure 4.7: Query time

Although all other methods has their own benefits, the *GraphGrepSX* seems to be an overall winner. It is quite simple to implement, it has the best overall performance and reasonable index size and its build time.

4.7 Hypotheses results

In this section we summarize the results of the hypotheses formulated in section 3.2 in table 4.1.

Hypothesis	Result	Comments
H1.1	False	Index of *GString* is significantly larger. The number of distinct nodes in case of *GString* is much bigger compared to *GraphGrapSX* on real data set.
H1.2	Uncertain	The performance of *GString* is indeed better compared to *GraphGrepSX*. On the other hand the main reason is smaller answer set because the rules for candidate set creation are too restrictive in some cases.
H2.1	Cannot be verified	We were not able to build the *GIRAS* index in reasonable time
H2.2	True	We were not able to build the *GIRAS* index in reasonable time even for the one tenth of the tested data set size
H3.1	True	In general, both *GraphGrepSX* and *GString* perform better than SQL and PGX approaches. On the other hand for small queries, the PGX is slightly faster. Moreover, in case of SQL we did expect much worse results.
H3.2	Uncertain	The hypothesis is definitely valid for small queries, in which case the performance difference is enormous. On the other hand, for larger queries PGX starts to be barely usable due to the issues with PGQL query parsing.

Table 4.1: Hypotheses results

Conclusion

Possible continuation o research:

- What about combining the indices with graph databases?
- Do some performance comparison of commercially used solutions with the other solutions.

Bibliography

[1] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[2] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, Oct 2004.

[3] Hans-Christian Ehrlich and Matthias Rarey. Systematic benchmark of substructure search in molecular graphs - from ullmann to vf2. *Journal of Cheminformatics*, 4(1):13, 2012.

[4] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proc. VLDB Endow.*, 6(2):133–144, December 2012.

[5] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, August 2008.

[6] Huahai He and Ambuj K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 405–418, New York, NY, USA, 2008. ACM.

[7] Shijie Zhang, Shirong Li, and Jiong Yang. Gaddi: Distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 192–203, New York, NY, USA, 2009. ACM.

[8] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *Proc. VLDB Endow.*, 3(1-2):340–351, September 2010.

[9] E. S. S. Dongoran, W. K. Rahmat Saleh, and A. A. Gozali. Analysis and implementation of graph indexing for graph database using graphgrep algorithm. In *2015 3rd International Conference on Information and Communication Technology (ICoICT)*, pages 59–64, May 2015.

[10] Dennis Shasha, Jason T. L. Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 39–52, New York, NY, USA, 2002. ACM.

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

[12] H. Jiang, H. Wang, P. S. Yu, and S. Zhou. Gstring: A novel approach for efficient search in graph databases. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 566–575, April 2007.

[13] Vincenzo Bonnici, Alfredo Ferro, Rosalba Giugno, Alfredo Pulvirenti, and Dennis Shasha. *Enhancing Graph Database Indexing by Suffix Tree Structure*, pages 195–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[14] Mehdi Azaouzi and Lotfi Romdhane. A minimal rare substructures-based model for graph database indexing. volume 557, pages 250–259, 02 2017.

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

[16] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 976–985, April 2007.

[17] Chuntao Jiang, Frans Coenen, and Michele Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 28(1):75–105, 2013.

[18] Xifeng Yan and Jiawei Han. gspan: graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pages 721–724, Dec 2002.

[19] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[20] Daniel Zaharevitz (NIH/NCI). Aids antiviral screen data. `https://wiki.nci.nih.gov/display/NCIDTPdata/AIDS+Antiviral+Screen+Data`, 2015. [Online; accessed 19-May-2017].

[21] Adel Golovin and Kim Henrick. Chemical substructure search in sql. *Journal of Chemical Information and Modeling*, 49(1):22–27, 2009. PMID: 19072559.

[22] D. Hoksza and J. Jelínek. Using neo4j for mining protein graphs: A case study. In *2015 26th International Workshop on Database and Expert Systems Applications (DEXA)*, pages 230–234, Sept 2015.

[23] Neo4j database. `https://neo4j.com/`. [Online; accessed 19-May-2017].

[24] Ambit. `http://ambit.sourceforge.net/`. [Online; accessed 19-May-2017].

[25] Krisztina Vajda (ChemAxon). Jchem cartridge for oracle. `https://docs.chemaxon.com/display/docs/JChem+Cartridge+for+Oracle`, 2015. [Online; accessed 19-May-2017].

[26] Oracle. Oracle9i data cartridge developer's guide. `https://docs.oracle.com/cd/B10501_01/appdev.920/a96595/dci01wht.htm`. [Online; accessed 19-May-2017].

[27] John May (NextMove Software). Substructure search face-off: Are the slowest queries the same between tools? https://nextmovesoftware.com/blog/2015/06/01/substructure-search-face-off-are-the-slowest-queries-the-same-between-tools/, 2015. [Online; accessed 19-May-2017].

[28] Dimitris K. Agrafiotis, Victor S. Lobanov, Maxim Shemanarev, Dmitrii N. Rassokhin, Sergei Izrailev, Edward P. Jaeger, Simson Alex, and Michael Farnum. Efficient substructure searching of large chemical libraries: The abcd chemical cartridge. *Journal of Chemical Information and Modeling*, 51(12):3113–3130, 2011. PMID: 22035187.

[29] Johnson & johnson. https://www.jnj.com/. [Online; accessed 19-May-2017].

[30] The chemistry development kit. https://github.com/cdk/. [Online; accessed 19-May-2017].

[31] Daylight Chemical Information Systems Inc. Smarts - a language for describing molecular patterns. http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html. [Online; accessed 19-May-2017].

[32] Daylight Chemical Information Systems Inc. Fingerprints - screening and similarity. http://www.daylight.com/dayhtml/doc/theory/theory.finger.html. [Online; accessed 19-May-2017].

[33] Chemaxon. https://www.chemaxon.com/, 2017. [Online; accessed 19-May-2017].

[34] Oracle. Java. https://www.java.com/en/. [Online; accessed 18-April-2019].

[35] Ulrich Bauer. Minimum cycle basis algorithms for the chemistry development toolkit.

[36] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[37] Oracle. Pgx. https://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytix/overview/index.html. [Online; accessed 25-April-2019].

[38] Oracle. Pgql. http://pgql-lang.org/. [Online; accessed 25-April-2019].

[39] Oracle. An introduction to graph: Database, analytics, and cloud services. https://www.slideshare.net/JeanIhm/an-introduction-to-graph-database-analytics-and-cloud-services?fbclid=IwAR2liLiNYui4EbEJWPEmrPxji_u7jzRproo1u86G10qbabekgvLOruPPskw. [Online; accessed 25-April-2019].

[40] Microsoft. Windows subsystem for linux documentation. https://docs.microsoft.com/en-us/windows/wsl/about. [Online; accessed 25-April-2019].

[41] Chembl. https://www.ebi.ac.uk/chembl/. [Online; accessed 2-May-2019].

[42] Smiles. http://www.daylight.com/dayhtml/doc/theory/theory.smiles.html. [Online; accessed 2-May-2019].

List of Figures

List of Tables

List of Abbreviations

- DFS - Depth-First search
- DAG - Directed Acyclic Graph
- DBMS - DataBase Management System
- SQL - Structured Query Language
- BFS - Breadth-First Search
- REST - REpresentational State Transfer
- CDK - Chemistry Development Kit
- SSSR - Smallest Set of Smallest Rings
- MCB - Minimum Cycle Basis
- API - Application Programming Interface
- JDBC - Java DataBase Connectivity
- PGX - Parallel Graph analytiX
- PGQL - Property Graph Query Language
- ACID - Atomicity, Consistency, Isolation, Durability
- CPU - Central Processing Unit
- RAM - Random Access Memory

Attachments

Set of tested queries

Queries of size 4

1. c:c:c:c
2. C=NCC
3. S(=O)(=O)C
4. C=CC=C
5. N=CCC
6. CN=CO
7. SCCC
8. c:n:c:c
9. CSCC
10. CSC=C

Queries of size 8

1. COc1ccccc1
2. CCCCCCCC
3. c:c:c:c:c:c:c:c
4. S(CCC)(CCC)(C)
5. C(=O)NCCCCC
6. c1cc(C(=O))ccc1
7. CCC(=O)C(=O)NC
8. c2c(Cl)cccc2Cl
9. C(C)(C)C(C)(C)C(=O)
10. S(CNCC)(C=O)(C)

Queries of size 16

1. CNC(=O)c1cc(C(=O)CCCC)ccc1
2. CCCCCCCCCCCCCCCC
3. c:c:c:c:c:c:c:c:c:c:c:c:c:c:c:c
4. c1cccc2c1ccc3c2cccc3CC
5. S(CC)(=O)(=O)Nc1cc(CCCC)ccc1
6. C(C)(C)C(C)(C)C(C)(C)C(C)(CCCCC)
7. C(O)CCCC(=C)CCCc1ccccc1
8. CCCCCNCCCCCCCCNCC
9. c1ccccc1CCc2cc(CC)ccc2
10. Oc1cccc2Cc3ccccc3C(=O)c12

Queries of size 16

1. C(C)NC(=O)C(NC(=O)OC)CCCCNC(=O)c1ccccc1
2. CCCCCCCCCCCCCCCCCCCCCCCCCC
3. c:c:c:c:c:c:c:c:c:c:c:c:c:c:c:c:c:c:c:c:c:c:c:c:c:c
4. c1cccc2c1ccc3c2ccc4c3ccc5c4cccc5CC
5. S(C)(=O)(=O)Nc1cc(C(c2c(=O)oc(CC)cc2)CCC)ccc1
6. C(C)(C)C(C)(C)CCCC(C(NCCCCC)=O)(CCCCCC)
7. C(O)C(O)C(O)C(O)CCCC(=C)C(O)C(C)Cc1ccccc1
8. CCCCCNCCCCCCCCNCCCCCCCCCC
9. c1ccccc1CCc2cc(CCCNc3ccccc3)ccc2
10. Oc1cccc2Cc3ccc(Cc4ccccc4)c(O)c3C(=O)c12

Results of measurements

Index building

	Molecule Count	Index Size(MB)	Index Build Time(ms)
GraphGrepSX	10000	70	5056
	25000	165	9271
	50000	326	17049
	100000	669	44860
Gstring	10000	399	4225
	25000	502	5792
	50000	680	9562
	100000	1118	19038
SQL	10000	118	396306
	25000	272	722317
	50000	517	1628827
	100000	989	2891050
PGX	10000	29	13746
	25000	62	28097
	50000	117	49840
	100000	231	94142

Queries of size 4

	Query ID	Candidate Set Creation(ms)	Candidate Set Size	Verification Time (ms)	Answer Set Size
GraphGrepSX	1	92	88034	17861	88033
	2	41	4754	667	4186
	3	43	9932	532	3798
	4	35	1428	142	1424
	5	36	3048	167	2699
	6	34	245	11	192
	7	35	6037	796	5471
	8	109	42899	4506	42899
	9	38	7074	526	4113
	10	31	858	43	543
Gstring	1	851	52581	12352	45074
	2	789	39201	4275	2276
	3	982	18713	2100	890
	4	714	21199	3407	890
	5	883	39201	3872	1349
	6	1196	37302	4207	53
	7	648	49082	4957	3951
	8	878	49082	5424	17902
	9	764	49082	6004	2750
	10	945	39201	4809	192
SQL	1			22871	88033
	2			713	4186
	3			213	3798
	4			699	1424
	5			860	2699
	6			692	192
	7			867	5471
	8			1551	42899
	9			332	4113
	10			109	543
PGX	1			8559	88033
	2			971	4186
	3			700	3798
	4			815	1424
	5			383	2699
	6			454	192
	7			336	5417
	8			818	42899
	9			401	4133
	10			434	543

Queries of size 8

	Query ID	Candidate Set Creation(ms)	Candidate Set Size	Verification Time (ms)	Answer Set Size
GraphGrepSX	1	146	27694	4756	26065
	2	60	19030	31359	11142
	3	89	83704	79181	18506
	4	82	305	29	3
	5	263	15555	4230	9722
	6	110	18474	3375	17623
	7	448	429	78	275
	8	132	1966	399	1963
	9	70	10113	2448	659
	10	44	0	0	0
Gstring	1	1334	72013	10751	24888
	2	1184	17257	21549	5646
	3	763	17257	17150	2627
	4	919	18774	1975	1
	5	778	13921	4353	4135
	6	1240	53792	8483	14365
	7	835	13485	3136	185
	8	38	23489	3281	1963
	9	101	283	235	2
	10	857	15049	1675	0
SQL	1			8982	26065
	2			43451	11142
	3			124065	18506
	4			233	3
	5			3576	9722
	6			8613	17623
	7			2302	275
	8			3963	1963
	9			12970	659
	10			202	0
PGX	1			20217	26065
	2			8171	11142
	3			7348	18506
	4			4826	3
	5			5010	9722
	6			14570	17623
	7			4466	275
	8			18753	1963
	9			4703	659
	10			6239	0

Queries of size 16

	Query ID	Candidate Set Creation(ms)	Candidate Set Size	Verification Time (ms)	Answer Set Size
GraphGrepSX	1	165	11	56	3
	2	69	7727	148934	2070
	3	77	51697	216697	278
	4	143	74	564	16
	5	67	14	3	12
	6	91	6174	10725	215
	7	128	594	261	70
	8	89	6862	19024	92
	9	50	170	266	73
	10	82	342	220	340
Gstring	1	1515	12387	2661	2
	2	767	4216	34278	671
	3	631	4216	16958	4
	4	293	7893	4147	5
	5	113	49	11	0
	6	116	76	163	0
	7	1101	7850	2098	1
	8	585	3838	18784	4
	9	1567	21486	5153	17
	10	44	4913	768	295
SQL	1			6157	3
	2			132027	2070
	3			239151	278
	4			13236	16
	5			4053	12
	6			17249	215
	7			1507	70
	8			31585	92
	9			7125	73
	10			5005	340
PGX	1			∞	N/A
	2			∞	N/A
	3			∞	N/A
	4			∞	N/A
	5			∞	N/A
	6			∞	N/A
	7			∞	N/A
	8			∞	N/A
	9			∞	N/A
	10			∞	N/A

Queries of size 24

	Query ID	Candidate Set Creation(ms)	Candidate Set Size	Verification Time (ms)	Answer Set Size
GraphGrepSX	1	148	6	67	1
	2	70	4053	394131	158
	3	97	10040	516538	7
	4	207	9	2604	5
	5	95	37	125	6
	6	119	900	1100	28
	7	101	65	169	53
	8	101	3688	26886	1
	9	90	5	77	2
	10	96	23	130	19
Gstring	1	1496	2672	2089	1
	2	585	1128	27793	6
	3	719	1128	14371	0
	4	249	741	1623	0
	5	199	0	32	0
	6	759	176	500	0
	7	706	1188	814	0
	8	961	1110	23108	0
	9	1326	4147	1388	2
	10	242	147	212	18
SQL	1			6664	1
	2			211365	158
	3			1347799	7
	4			104108	5
	5			2422	6
	6			9323	28
	7			6395	53
	8			60604	1
	9			8005	2
	10			8275	19
PGX	1			∞	N/A
	2			∞	N/A
	3			∞	N/A
	4			∞	N/A
	5			∞	N/A
	6			∞	N/A
	7			∞	N/A
	8			∞	N/A
	9			∞	N/A
	10			∞	N/A