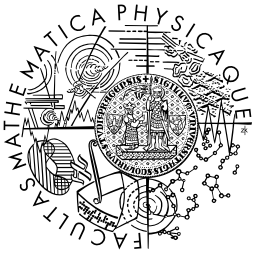

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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 2017

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 ..... date ..... 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: Abstract.

Keywords: key words

Dedication.

Contents

Introduction

Querying is the essential utility of each database. The same applies to chemical databases which growth speed is enormous and therefore there is a pressure on efficiency of the querying process. Since the chemical compounds are typically represented as graphs the most common queries on 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 this thesis is to compare the efficiency of querying methods which have been already proposed in other papers. This includes comparison of algorithms and also the utilization of native query mechanisms of both graph and relational databases.

We will focus on the general performance of those approaches as well as on the particular cases where some approach might be better than others.

Subgraph Querying

The goal of subgraph querying is to obtain a list of graphs from the database which contains 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.

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

Most of the techniques described below 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 some sub-graph isomorphism algorithm to filter out false positives from the candidate 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.

Definitions

Definition: *Graph* $G = (V, E)$ is an ordered of set of vertices V and set of edges E which are unordered pairs of elements from V.

Definition: *Labeled Graph* $G = (V, E, L_V, L_E, f_V, f_E)$ is an ordered of set of vertices V, set of edges E which are unordered pairs of elements from 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 mark 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.

Structure of the Thesis

This thesis is divided into three main parts. In the first part we will analyze the subgraph querying problem. We will define the basic terms, list the algorithms for resolving subgraph isomorphism problem and most importantly we will analyze the related work.

In the second part several hypothesis will be uttered. For their verification the author's experimental work will be used. These experiments will be described and the issues found out during the implementation will be explored.

The last part of the thesis will cover 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. 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 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.

As stated before the typical workflow in subgraph querying process has two parts:

- **Candidate set creation** - on the basis of a pre-built index the whole database is pruned to obtain as small set of graphs as possible which contains all records which satisfy the specified query.
- **Verification** - the process of isolating false positives from the candidate set. In this phase the algorithm for subgraph isomorphism recognition has to be used.

Since subgraph isomorphism problem is NP-complete, we cannot expect significant improvement in the verification phase which implies that good pruning of database is essential for effective subgraph querying.

1.1 Subgraph Isomorphism Algorithms Survey

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 the Ullmann.

In [4] there is a comparison of four algorithms derived from Ullmann algorithm. These are VF2, QuickSI [5], GraphQL [6], GADDI [7] and SPath [8]. They were compared on 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.

1.2 Comparison and Summary of 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* [14], *GraphGrepSX* [15], *GIRAS* [16], *C-tree* [12] and *GDIndex* [13]

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 specific point of interest.

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

1.2.1 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 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 1.1 would be represented in the index with $l = 3$ as depicted in Table 1.1. The numbers on the picture represent the vertex ID, characters next the each vertex represent its label.

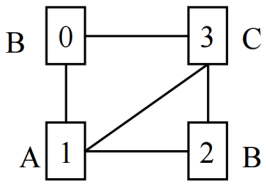

Figure 1.1: GraphGrep example graph

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.

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 1.1: GraphGrep example graph fingerprint

1.2.2 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 subgraphs 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 with smaller *minSup* for small graphs (because of the efficiency) and bigger *minSup* for large graphs (because of the compactness). It is necessary to have specified limit starting from which the function returns infinite to not have too big subgraphs in the index.

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 comparing 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 the subgraph g is not present in the 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 clear what is the search algorithm. We need to enumerate all fragments of the 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.

1.2.3 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 fanout). 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 those paths, where each node is identified by tuple $< Type, Size >$ and contains a set of pointers to the table where quadruples $< n_n, n_b, n_e, id >$ are stored for matched nodes from a particular graph. The suffix tree is filled up by 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.

1.2.4 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 is storing the paths in the suffix tree instead. Each node in the suffix tree represent 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 $count$ 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 empty candidate set.

1.2.5 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 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 which support in database is *n or higher*, the modified version finds all the subgraphs which support is equal to *n*.

This 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 input parameter f, the 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 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 the $f = 1$. 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 gSpan modified 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[16] what data structure it is using for the index representation, we found out from the source code obtained from Dr. Azaouzi, 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.

1.2.6 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 en-

abling 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'. A similar approach is applied to edges. **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.

The several approaches how to get the mapping m are described in[12] and we will not describe those 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 on 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 query is not pseudo subgraph isomorphic to some node, this node and all its subtree is 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 in contrary to 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 the candidate set creation.

1.2.7 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 has 1 less vertex than M and N has the same edges as M except those incident with the missing vertex.

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 has 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.

1.2.8 Benchmark Results

GraphGrep, *GIndex*, *GString* and *C-Tree* has been compared in [14]. 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 [15] we can find the benchmark of the *GraphGrepSX* method which looks like a more generic version of *GString*. While in [14] *GString* outperforms *CTree* just by few percents but in [15] *GraphGrepSX* outperforms the *CTree* by the two levels of magnitude despite larger candidate sets.

In [13] 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 [16] we can see the benchmark of the *GIRAS*, *C-tree*, *gIndex* and couple of other approaches. We can see that on 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 [16] is the size of index and time needed for index construction.

1.3 Summary of 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.

1.3.1 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 breadth-first-search.

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 on bond type which helps us to do efficient filtering, the second one is built on *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 on bond type column. For other spanning tree edges we should use the index on $atom1\_id$ column which literally does the BFS. For edges outside the spanning tree we should use the index built on $atom1\_id$, $atom2\_id$ pair since we already know the IDs of both atoms of the edge we need to check.

1.3.2 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 the 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 the 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 on 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.

1.4 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 the 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 the 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
.

1.4.1 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 those 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].

1.4.2 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. 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 similar 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 *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%.

1.4.3 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.

2. Description of Experimental Work

During the research of the related work, many questions arise. 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 the 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 the *C-tree* just by few percents in [14] 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 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 non 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 NOSQL databases. It might be interesting 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 proves.

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 has 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. Experimental Results

Conclusion

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] 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.

[13] 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.

[14] 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.

[15] 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.

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

[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].

List of Figures

List of Tables

List of Abbreviations

Attachments