CT-Index: Fingerprint-based Graph Indexing Combining Cycles and Trees

Karsten Klein [1], Nils Kriege [2], Petra Mutzel [3]

Department of Computer Science, Technische Universität Dortmund, Germany

[1] karsten.klein@cs.tu-dortmund.de

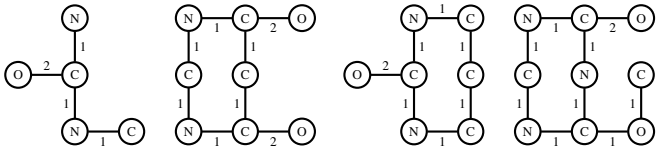
[2] nils.kriege@cs.tu-dortmund.de

[3] petra.mutzel@cs.tu-dortmund.de

***Abstract*— Efficient subgraph queries in large databases are a time-critical task in many application areas as e.g. biology or chemistry, where biological networks or chemical compounds are modeled as graphs. The NP-completeness of the underlying subgraph isomorphism problem renders an exact subgraph test for each database graph infeasible. Therefore efficient methods have to be found that avoid most of these tests but still allow to identify all graphs containing the query pattern. We propose a new approach based on the filter-verification paradigm, using a new hash-key fingerprint technique with a combination of tree and cycle features for filtering and a new subgraph isomorphism test for verification. Our approach is able to cope with edge and vertex labels and also allows to use wild card patterns for the search. We present an experimental comparison of our approach with state-of-the-art methods using a benchmark set of both real world and generated graph instances that shows its practicability. Our approach is implemented as part of the Scaffold Hunter software, a tool for the visual analysis of chemical compound databases.**

I. Introduction

Graphs are a versatile data structure that is used to represent data in many application areas such as biology, chemistry, software development or business process modeling. Biological or social interaction networks for example allow a natural interpretation as graphs and the structure of chemical compounds can be modeled as graphs by associating a vertex with each atom and an edge with each chemical bond. A frequent task then is to query a database containing such graphs for the existence of a given subgraph pattern, either exact or approximative, e.g. to search for chemical compounds containing a specific substructure associated with biological activity [1]. Fig. 1 shows a small example of a graph database and a query graph contained in two of the three database graphs. Real world databases may contain hundreds of thousands or even millions of graphs, e.g. ChEBI [2], a database containing chemical entities of biological interest, contains nearly $600,000$ entries, and the PubChem Compound database [3] contains about 27 million entries. Both databases are accessible via a substructure search interface. As the subgraph isomorphism problem, which asks if a given graph G_1 is contained as a subgraph in another graph G_2, is NP-complete [4], it is infeasible to conduct this test for each graph in the database. In order to efficiently query the database, approaches that allow to omit most of the isomorphism tests are therefore needed. A typical pattern to achieve this is the so-called *Filter-and-Verification* paradigm, which proceeds in two steps. First, a fast *filtering step* is performed, that significantly reduces the size of the potential solution set by eliminating candidates with a heuristic, then a *verification step* tries to verify the subgraph relationship for the remaining candidates by a subgraph isomorphism test.



(a) Query Q (b) G_1 (c) G_2 (d) G_3

Fig. 1. An example database $\mathcal{D} = \{G_1, G_2, G_3\}$ and a query graph Q with the answer set $\mathcal{D}_Q = \{G_2, G_3\}$. The graph G_1 contains all paths contained in the pattern graph Q, but obviously Q is not a subgraph of G_1.

Approaches that use the Filter-and-Verification paradigm differ in the way the two steps are implemented, and can be evaluated and compared by the quality of their filter step and the running time of both steps. The quality of the filtering depends on the size of the remaining candidate set compared to the final solution — the closer the candidate set is to the final set of graphs containing the search pattern, the less work has to be done for verification. Comparing vertex properties like degree or attached labels of the query graph with those from the database graphs may already allow to filter out candidates early from the query process, but such simple criteria typically are not sufficient to remove a significant number of graphs from the candidate set. Clearly, no candidate graph containing the query pattern should be eliminated during filtering, such that the resulting candidate set comprises all graphs that contain the query graph. In addition to the quality and running time of the filtering itself, the space and runtime requirements of a potentially required preprocessing computation need to be considered. Such a preprocessing can be used to perform part of the work needed for online query processing in a preceding offline phase in order to further speed up the query.

A popular way to perform the filtering is to build up an index structure in a preprocessing step that classifies the database graphs according to specific subgraphs, also called *features*. A graph that contains the query pattern also has to contain all features that the pattern contains, otherwise the graph does not

need to be considered further. The features have to be selected such that the filtering is sufficiently discriminative and at the same time the feature set is as small as possible to minimize build-up, storage and testing requirements. Feature selection, extraction and comparison need to be done efficiently in order to speed up the subgraph search significantly. The overall quality of the index method therefore is determined by the index construction and the query processing time, the size of the constructed index structure and the resulting filtering strength.

There are already a number of methods published that can be classified according to how the features are obtained. Two main approaches can be distinguished:

Data-mining based approaches first analyze the database and extract features that occur frequently in the database [5]. A drawback of the data-mining based approaches is that they are computationally demanding, and that updates to the database also often require an update of the selected features, as otherwise the effectivity may degrade. One way to decrease the computational work is to restrict the mining to a subset of the database graphs [6], with the hope that a deliberate selection will not compromise the quality of the index too much. However, in many cases the dataset will consist of subsets with substantially different graphs and frequent subgraphs, like e.g. molecular graphs from different test series or libraries that are dynamically added to the database, and it will therefore be difficult to maintain a certain quality of the feature index without regular recomputation from scratch.

Non-data-mining approaches use exhaustive feature enumeration and are typically restricted to specific classes of graphs considered for feature extraction. In order to keep the computational requirements of the feature handling low, often only simple graph classes are used to select the features, e.g. by restricting the feature set to paths (e.g. GraphGrep [7]). For path-based indexes, all paths up to a certain length contained in the database are used to create the index. At query time, paths are extracted from the query pattern and the index is used to identify all database graphs containing those paths. Restriction to simple feature structure of limited size may simplify handling of the features for filtering, as corresponding problems like canonicalization are much easier to handle. It also helps to keep the feature set size reasonable, but as the features are not selected according to their selectivity with regard to the database graphs, there is a tradeoff between the size of the feature set and its filtering strength.

While data-mining based approaches pay for the good filtering strength with large computational demands, index structures derived without data-mining, and based solely on simple features like paths, tend to be large in order to comprise a sufficiently discriminative feature set.

A common way to use the feature set for filtering is to characterize graphs as a bit array of fixed size, or *fingerprint*, where each entry indicates if the graph contains a specific feature, and to simply compare the pattern's representation with the representation of each database graph, removing all graphs that do not support each of the features from the query pattern.

TABLE I
PATH-BASED 16-BIT HASH-KEY FINGERPRINT OF Q.

Path string s	$h(s)$	**Path string** s	$h(s)$
C	6	N	5
C1N	3	N1C1N	14
C2O	10	N1C2O	3
C1N1C	1	O	7
Fingerprint:	0101011100100010		

Instead of reserving one bit for each feature in the feature set, *hash-key fingerprints* are constructed by using a hashing scheme to map extracted features to bit positions of a constant size bit array. This may reduce the storage requirements and allows to generate the features directly from the graphs instead of defining them in advance, as a hash-key fingerprint may represent sets of arbitrary size and with arbitrary elements. Table I shows an example of a hash-key fingerprint representing all paths with up to 3 vertices contained in the graph depicted in Fig. 1(a). Note that each path is associated with two possibly different string encodings and the lexicographically smaller one of these is used to uniquely represent the path. Using hash-key fingerprints, different paths may be mapped to the same position and thus some information on contained features is lost. Optimization techniques for fingerprints are well known: Daylight [8] proposes a technique called *folding* which starts with a large fingerprint and reduces the size by dividing the fingerprint in two halves which are combined by logical OR. This step is repeated until a minimum utilization is reached, i.e. at least a certain percentage of the bits in the fingerprint is set to 1. The resulting *variable-sized fingerprints* can still be used for filtering by folding the query fingerprint until it reaches the same size. This procedure allows to minimize the influence of collisions while keeping the index size small at the same time.

An alternative index concept that is used in many approaches is to build up an inverse index structure that stores for each feature the database graphs in which it is contained. The intersection of the graph sets corresponding to the pattern's features then determines the candidate set.

In addition to subgraph detection based solely on the graph structure, real world graphs are often annotated by additional labels, e.g. to model atom types in chemical compounds, and this information also needs to be considered for the query. Due to the specific properties of graphs from different application areas, the subgraph search may be less demanding than in the general case. Chemical compound graph structures for example typically possess low vertex degrees, and their vertex and edge labels may allow to prune the search space for a subgraph isomorphism efficiently. It is often necessary to perform the query using wildcards, that allow arbitrary annotations at specified locations of the query graph, e.g. when chemical compounds are required that contain certain functional substructures but there is no further constraint on the full compound structure.

Our main contribution is an algorithm for subgraph queries

in graph databases, which uses a new hash-key fingerprint technique based on both tree and cycle features for filtering and an adapted subgraph isomorphism test for verification. The rest of the paper is organized as follows. After a short description of the necessary terminology in Sect. I-A and an overview on related work in I-B, our new approach CT-Index is presented in Sect. II, and the results and discussion of our experimental comparison can be found in Sect. III. We conclude the paper with a short summary on the results and remarks on open problems.

A. Preliminaries

In this section the terminology used throughout this article is introduced. We refer to simple undirected graphs with vertex and edge labels.

Definition 1 (Labeled Graph) *A* labeled graph $G = (V, E, l)$ *consists of a set of* vertices $V =: V(G)$ *and* edges $E \subseteq \{X \subseteq V || X| = 2\} =: E(G)$ *together with a function* $l : V \cup E \to \Sigma$ *defining the* labels *of vertices and edges.*

Please note that in the following an edge $\{u, v\}$ is denoted by (u, v) or (v, u), both refer to the same edge. The set of vertices *adjacent* to a vertex v is denoted by $adj(v) = \{u \in V|(u, v) \in E\}$ and the *degree* of v is defined as $deg(v) = |adj(v)|$. A *path* of length n is a sequence of vertices $(v_0, \dots v_n)$ such that $(v_i, v_{i+1}) \in E$ for $0 \leq i < n$. A path with no repeated vertices is called *simple*. A *cycle* is a path of length $n > 1$ where $v_0 = v_n$. A cycle with no repeated vertices except v_0 and v_n is called *simple*. A graph is *connected* if at least one path between any pair of vertices exists. A *(free) tree* is a connected graph containing no cycles. A graph $G' = (V', E', l')$ is said to be a *subgraph* of G, denoted by $G' \subseteq G$, if $V' \subseteq V$, $E' \subseteq E$ and l' is the function l restricted to the domain $V' \cup E'$. A subgraph $G' = (V', E')$ of G is said to be *induced* by V' if $E' = \{(u, v) \in E|u, v \in V'\}$. The subgraph G' of $G = (V, E)$ induced by $V' \subseteq V$ is denoted by $G[V']$.

An isomorphism of labeled graphs is a mapping that preserves vertex labels, edge labels and adjacencies.

Definition 2 (Graph Isomorphism) *Let* $G_1 = (V_1, E_1, l_1)$ *and* $G_2 = (V_2, E_2, l_2)$ *be two graphs. A* graph isomorphism *is a bijection* $\varphi : V_1 \to V_2$ *such that* $\forall u, v \in V_1 : (u, v) \in E_1 \Leftrightarrow (\varphi(u), \varphi(v)) \in E_2$ *and* $l_1((u, v)) = l_2((\varphi(u), \varphi(v))$ *and* $\forall v \in V_1 : l_1(v) = l_2(\varphi(v))$. *Two graphs* G_1, G_2 *are said to be* isomorphic*, written* $G_1 \simeq G_2$*, if a graph isomorphism between* G_1 *and* G_2 *exists.*

The subgraph isomorphism problem is to decide for two given graphs G_1 and G_2 whether G_2 contains a subgraph that is isomorphic to G_1.

Definition 3 (Subgraph Isomorphism) *Let* $G_1 = (V_1, E_1, l_1)$ *and* $G_2 = (V_2, E_2, l_2)$ *be two graphs. A* subgraph isomorphism *is an injection* $\varphi : V_1 \to V_2$ *such that* $\forall u, v \in V_1 : (u, v) \in E_1 \Rightarrow (\varphi(u), \varphi(v)) \in E_2$ *and* $l_1((u, v)) = l_2((\varphi(u), \varphi(v))$ *and* $\forall v \in V_1 : l_1(v) = l_2(\varphi(v))$. *If a subgraph isomorphism from* G_1 *to* G_2 *exists,* G_1 *is* subgraph-isomorphic *to* G_2*, written* $G_1 \lesssim G_2$ *and* $G_1 \lesssim G_2$ *otherwise.*

Informally speaking there is a subgraph isomorphism from G_1 to G_2 if G_1 is contained in G_2. Throughout this article we refer to the graph G_1 as *pattern* or *query graph* and to G_2 as *database* or *host graph*.

The *subgraph search problem* can be described as follows: Given a graph database $\mathcal{D} = \{G_1, \dots, G_n\}$ consisting of n graphs and a search pattern Q, find all the graphs in $\mathcal{D}$ that contain a subgraph isomorphic to Q. Formally one is interested in the *answer set* $\mathcal{D}_Q = \{G \in \mathcal{D}|Q \lesssim G\}$.

B. Related Work

There is already a significant number of approaches for the subgraph search problem, including both data-mining based and non-data-mining based methods for index construction, see Table II for a non-exhaustive overview.

Within most existing approaches the verification step can be accomplished by applying an arbitrary subgraph isomorphism test, although some approaches try to reuse information from the filtering step to speed up the verification step (e.g. TreePi [9], SING [10]) or even avoid the verification step if the query graph is isomorphic to an indexed feature (FG-Index [11]). Ullmann proposed in [12] an algorithm for subgraph isomorphism testing that is based on backtracking and tries to match a vertex from the pattern to a vertex of the testgraph in each step. Due to its simplicity it has been one of the preferred algorithms in software implementations for a long time, it is for example used for verification in the Closure-Tree approach [13]. Cordella et al. described in [14] the algorithm VF2, which showed good performance in experimental comparisons. It is considered the fastest subgraph isomorphism algorithm known and is for example used in the GraphGrepSX [15] and the SING approach. Battiti and Mascia proposed a new pruning technique based on the VF2 approach that aims at reducing the state space that has to be visited during the matching process [16]. As the order of the matching operations in a subgraph isomorphism test can have a critical impact on the performance, Batz et al. presented a search plan driven technique that generates an ordering of matching operations specific for a single host graph [17].

The main difference of existing filtering methods lies in the selection of features for filtering. Feature sets can be divided into paths, trees, and general graphs. These features may either be enumerated exhaustively or selected based on certain criteria, like frequency when data-mining is applied. GraphGrep [7] uses path features for indexing, which have limited pruning power, as the characteristical graph structures might not be captured, see Fig. 1. Trees have been identified as convenient features because of their good pruning power and favorable runtime of frequent tree mining compared to frequent graph mining [9], [20]. Whereas TreePi utilizes an indexing approach solely based on frequent tree features, Tree+Δ in addition tries to achieve the filtering strength of general graphs by

TABLE II
SUMMARY ON GRAPH INDEXING APPROACHES.

	Features	Feature selection
Closure-Tree [13]	–	–
Daylight FP [8]	Paths	Exhaustive enumeration
GCoding [18]	–	–
GraphGrep [7]	Paths	Exhaustive enumeration
GraphGrepSX [15]	Paths	Exhaustive enumeration
GDIndex [19]	Graphs	Exhaustive enumeration
gIndex [6]	Graphs	Frequent Graph Mining
SING [10]	Paths	Exhaustive enumeration
Tree+Δ [20]	Trees+Graphs	Frequent Tree Mining, add. Graphs "on-demand"
TreePi [9]	Trees	Frequent Tree Mining
FG-Index [11]	Graphs	Frequent Graph Mining

adding discriminative graph features on-demand, without prior extensive graph mining. TreePi also applies an additional pruning that utilizes distances between features. gIndex [6] allows general graphs up to a specified size as features, and performs an expensive data-mining preprocessing to derive a set of frequent features called *discriminative features*. GDIndex [19] uses all induced subgraphs of the database graphs for indexing, and is therefore only suitable for small graphs due to the costly index construction. Recently, a new approach especially designed to work with large graphs was presented [10]. SING exploits locality information to reduce the candidate set and also uses these information in an adapted version of VF2 for verification.

A special case is GString [21], as it uses structures with a semantical meaning in a specific application area, namely organic chemistry, for index construction. Strings that code the relevant features of a graph are constructed and the filtering step hence is transformed to a string matching problem.

There are also two approaches that are not based on features, ClosureTree and GCoding. GCoding is based on a spectral encoding method for filtering. ClosureTree proposes to employ a closure tree structure, where each internal node is a graph closure of its children, and the children of the leaves are the database graphs. Based on a traversal of the closure tree, a powerful filtering can be performed with the drawback of the computational efforts needed due to the approximate matching involved.

II. CT-INDEX

CT-Index is based on the idea to exploit more complex features than paths without adhering to frequent subgraph mining. In addition we propose a new subgraph isomorphism algorithm for verification. The system supports wildcards for vertex and edge labels. Vertices with the wildcard $\star$ may be mapped to vertices with arbitrary labels, just as edges with the wildcard $\star$ match all edges independent of their label. Furthermore we support the wildcards $[s_1, \ldots, s_n]$ and $![s_1, \ldots, s_n]$ to restrict the mapping of a vertex to vertices v with $l(v) \in \{s_1, \ldots, s_n\}$ and $l(v) \notin \{s_1, \ldots, s_n\}$, respectively. The matching condition for wildcard vertices is directly verified with our subgraph isomorphism algorithm, while all vertices and edges with wildcards are discarded and deleted in the query graph for filtering. Thus smaller, possibly diconnected fragments remain for feature extraction. A similar approach is used by GraphGrep [7] to cope with wildcards. The loss of information caused by deletion of vertices and edges makes the best possible utilization of the remaining fragments necessary.

A. Filter

We describe a new technique to efficiently and accurately filter out database graphs that do not contain the search pattern. It follows the feature-based approach and combines exhaustive feature enumeration with a compact storage technique.

Our approach uses all trees and cycles of a graph not exceeding a specified maximum size. Compared to paths, trees are able to capture additional structural information of graphs. Another distinct characteristic of graphs are cycles, which also often are substructures of particular relevance in many application areas, but are neglected when using only trees as features. Thus we also enumerate all simple cycles of a graph up to the maximum size and unambiguously store them as feature. An advantage of this combination over using arbitrary graphs is the favorable manageability of these features: Using graphs as features in an indexing system relies on a unique representation of graphs, called *canonical form*. Based on the canonical form of a graph it is possible to derive a unique string $C(G)$ without difficulty that unambiguously encodes G, such that for two graphs G and H $C(G) = C(H)$ holds if and only if $G \simeq H$. A function C with this property is commonly called a *certificate*. While for general graphs the problem of finding a certificate is far from trivial, for trees as well as for cycles the problem can be solved in linear time [22], [23]. This concept allows us to identify isomorphic feature graphs and encode them by the same string. As a result we can efficiently compute and store the set of features contained in pattern or database graphs.

We adopt the concept of hash-key fingerprints [8] to store the multitude of features. This certainly comes at the cost of accuracy: Each element is mapped to a distinct position in the bit array by a hash-function, thus different elements may be mapped to the same position. Collisions are in general unavoidable as the number of different features in a whole database of graphs typically exceed the size of the fingerprint. However, the decline in index quality may be bearable when feature set and fingerprint size are selected carefully. The data structure suits well to the Filter-and-Verification approach as false positives will in any case be identified by the verification step.

Feature-based graph indexing is essentially based on subset queries: Let Q be a pattern graph, H a database graph and $\mathcal{F}(G)$ denote the set of features contained in a graph G. If $\mathcal{F}(Q) \nsubseteq \mathcal{F}(H)$ we can conclude that H does not contain the graph Q, otherwise H may contain Q. When using fingerprints to represent sets, checking for subsets can easily be achieved by inexpensive bit operations. Let $B(\mathcal{S})$ be the fingerprint representation of the set $\mathcal{S}$ created by mapping each element

$s \in \mathcal{S}$ to a well-defined position $h(s)$ in the fingerprint via a hash-function h. Let $\wedge$ denote a bitwise AND-operation. If the condition

$$B(\mathcal{F}(Q)) \wedge B(\mathcal{F}(H)) \neq B(\mathcal{F}(Q)) \quad (1)$$

holds one may conclude $\mathcal{F}(Q) \not\subseteq \mathcal{F}(H)$ and hence $Q \lesssim H$, since Q must have at least one feature not contained in H. On the other hand, if (1) does not hold, it has to be verified whether H contains Q or not by an expensive subgraph isomorphism check. If (1) holds and $Q \lesssim H$, H is said to be a *false-positive*. Combining the feature-based approach with a fingerprint data structure there are two reasons leading to false-positives:

- The features used to build the index are not able to capture the structure of the graph and the query in an appropriate manner to distinguish the subgraphs of H from Q.
- Although $\mathcal{F}(Q)$ has a feature not contained in H, condition (1) does not hold due to collisions.

The loss of information caused by the use of hash-key fingerprints seems to be justifiable by the compact nature and convenient processing of bit arrays as long as the amount of false positives does not increase significantly due to collisions.

1) Tree Features: Using trees as features for graph indexing requires an unambiguous representation that can be mapped by a hash-function to a position in the fingerprint. Finding a canonical string encoding for trees is less computational demanding than for general graphs but certainly more sophisticated than for paths. The *graph canonization problem*, i.e. finding the canonical form of a graph, can be solved in time $O(n)$ for trees, while no polynomial time algorithm for general graphs is known. The problem of tree canonization has been extensively studied in the field of frequent tree mining, see e.g. [22] for an overview.

An unambiguous string encoding of a tree can be obtained in three steps: In the first two steps the canonical form of the tree is calculated and in the last step a string encoding of the resulting tree is derived. The canonical form of a tree can be viewed as a rooted ordered tree. The root as well as the ordering must unambiguously be determined by the structure and the labels of the original (free) tree. In a first step the root is identified as the *center* of the tree. The center can be computed by repeatedly removing all leaf vertices of the tree until a single vertex or two adjacent vertices remain. In the first case a unique center is identified that serves as the root r for the next step. Otherwise we split the tree by removing the edge connecting the two remaining vertices to obtain two trees, each of which is rooted at one of the center vertices. In this case the ordering is independently defined on both trees as well as the string encodings, which are recombined in the last step to form a single string encoding of the original tree.

For a rooted tree we use the term *parent* of v to denote the vertex $u =: p(v)$ adjacent to v on the path from v to the root r and the term *children* of v to denote the vertices having v as parent. To obtain a rooted ordered tree an unique ordering of the children of each vertex must be defined. Based on an ordering of edge and vertex labels we can define an ordering on the vertices. To decide if $u < v$ for two siblings u and v with parent p we first compare the labels of the edges (p, u) and (p, v). If they are equal we compare the vertex labels $l(u)$ and $l(v)$. If this still does not yield a result we start to compare the subtrees to solve the order. Formally the following conditions must hold.

Definition 4 *Let $u \neq r$, $v \neq r$ be two vertices of a tree, $c_1^u, c_2^u, ..., c_n^u$ the ordered children of u and $c_1^v, ..., c_m^v$ of v respectively. $u < v$ holds if and only if*

1) $l((u, p(u))) < l((v, p(v)))$ *or if equal*
2) $l(u) < l(v)$ *or if also equal*
3) $\exists k : c_k^u < c_k^v \ \wedge \ \forall i < k : c_i^u = c_i^v$ *or else*
4) $n < m$.

Otherwise $u = v$ holds if neither $u < v$ nor $v < u$.

An algorithm to apply this ordering best solves the problem bottom-up, first sorting the vertices at the lowest level and moving up stepwise to the root. This, however, does not yield a linear time algorithm. See [24] for an algorithm that computes a similar ordering for unlabeled trees, which can also be extended to cope with vertex labels (from a finite alphabet Σ) [22]. For our implementation we chose to stay with a straightforward bottom-up implementation as the trees occurring as features usually are small and vertex and edge labels are diverse and hence the order can be solved quickly.

Having computed the canonical form, the tree can easily be encoded by an unique string. Traversing the tree in depth-first order, outputting vertex and edge labels and an additional symbol $\$ \notin \Sigma$ to mark backtracking steps gives a string representing the rooted ordered tree. For example, the subtree of the query graph depicted in Fig. 1(a) formed by the central node with label `C` together with the three adjacent nodes is encoded by the string `C1N$1N$2O$$`.

In the case that the center of the original tree consists of two vertices u and v, step two yields two ordered trees T_u and T_v. A string encoding is computed separately for both. Concatenating both strings, adding the label of the removed edge in between, yields a string encoding of the original tree. By comparing u and v according to the ordering imposed by Def. 4 (ignoring the parent edge condition 1) we decide which string comes first.

To find all tree features of a graph we directly enumerate all subtrees up to a given size contained in the graph by Algorithm 1.

Theorem 1 *Algorithm 1 enumerates each subtree of G exactly once.*

Proof: We show that duplicates are avoided and that each subtree contained in G is processed at least once:

Assume the algorithm enumerates the trees T and T' with $E(T) = E(T') \subseteq E(G)$ and $|E(T)| = n$. Obviously both edge sets must have been obtained by adding the edges in two different orders. Let $(e_1, \ldots, e_k, \ldots e_n)$ be the first sequence and $(e'_1, \ldots, e'_k, \ldots e'_n)$ the second, such that k is the first

Algorithm 1: Enumeration of subtrees.

Input : Graph G, max. tree size $maxT$

```
for v ∈ V do                                  ▷ Enumerate trees of size 0
    T = ({v}, ∅)
    PROCESS(T)
B ← ∅                                         ▷ Blocked edges
T = (∅, ∅)
for e ∈ E do
    EXTENDTREE(T, e)
```

Procedure ExtendTree(T, e)

Input : Tree T

Data : Graph G, max. size $maxT$, blocked edges B

```
B ← B ∪ {e}
T ← (V(T) ∪ e, E(T) ∪ {e})                    ▷ Add edge e to T
PROCESS(T)
if |E(T)| < maxT then
    F ← {(u, v) ∈ E(G) | u ∈ V(T), v ∉ V(T)} ∩ B
    for e ∈ F do
        EXTENDTREE(T, e)
    B ← B \ F
```

position with $e'_k \neq e_k$. Let S be the tree induced in T by $\{e_1, \ldots, e_{k-1}\}$. When extending S we may assume without loss of generality that EXTENDTREE (S, e_k) is executed before EXTENDTREE (S, e'_k). When S is finally extended by e'_k the condition $e_k \in B$ holds within all the following recursive function calls starting from S. Thus $\forall i \in \{k, \ldots, n\} : e'_i \neq e_k$ and hence $e_k \notin E(T')$ contradicting the assumption.

On the other hand for each subset of $E(G)$ inducing a tree T there is at least one allowed sequence generating $E(T)$. Let $C = F \cap E(T) \neq \emptyset$ be the set of edges belonging to T selectable in a function call. Consider the sequence of extension always using the edge $e \in C$ found first: None of these edges is blocked nor violating the tree condition. ■

2) Cycle Features: Utilizing cycles as features basically follows the same concept: Each cycle must be represented by an unambiguous string. Obviously there are $2n$ possible strings derived from a cycle of n vertices: Cutting the cycle at one of the n vertices and outputting the vertex and edge labels clockwise or anti-clockwise yields $2n$ possible strings. The lexicographically smallest string is selected to represent the cycle. For example, the cycle contained in the graphs G_1 and G_2 shown in Fig. 1 is encoded by the string `C1C1C1N1C1N1`.

The problem of finding the lexicographically smallest string is closely related to the *circular string linearization* problem. A string where the first character is assumed to directly follow the last character is known as *circular string*. The circular string linearization problem is to cut a given circular string $(c_1, c_2, \ldots, c_n)$ at some character k such that $(c_k, \ldots c_n, c_1, \ldots c_{k-1})$ is the lexicographically smallest string possible. This problem is known to be solvable in time $O(n)$, e.g. by means of suffix trees [23]. However, our implementation follows the naive approach comparing $2n$ strings as the cycles used as feature are typically of small size.

The cycle features of a graph are generated by enumeration of all simple cycles up to a given size. A large number of duplicates is avoided by starting the search at a vertex s considering only vertices v with $v > s$ regarding an arbitrary ordering of the vertices, a technique which is also applied in [25].

B. Verification

The verification step checks all candidates computed in the filtering step by an exact subgraph isomorphism test. In this section we propose a new backtracking algorithm which is particularly suitable for the specifics of the verification step, i.e.

1) a single pattern graph is searched in a number of database graphs and
2) background knowledge on the database graphs is available.

Our approach is similar to VF2 [14] to some extent but neglects terminal pruning which turned out to be less restrictive for subgraph isomorphism than for induced subgraph isomorphism. However, the concept of adding adjacent vertices whenever possible was adopted and integrated in a separate preprocessing step combined with a simple yet promising heuristic for further optimization.

Before an expensive subgraph isomorphism check is invoked at all, another heuristic is used to discard some further candidates: Let $G_1 = (V_1, E_1)$ be the pattern graph and $G_2 = (V_2, E_2)$ a host graph, obviously $G_1 \lesssim G_2$ must hold if $|V_1| > |V_2|$ or $|E_1| > |E_2|$.

1) Algorithm: Backtracking algorithms for subgraph isomorphism testing typically follow a common approach: The vertices of the pattern graph are mapped one after the other to a vertex of the host graph. In each step it is checked if the adjacencies of the mapped vertices are preserved in the host graph and in addition *pruning rules* may be applied to foresee that a mapping can not be extended to a subgraph isomorphism in a subsequent step. In both cases backtracking is performed.

A *vertex sequence* $s = (v_1, \ldots, v_n)$ of the pattern graph $G_1 = (V_1, E_1, l_1)$ with $\{v_1, \ldots, v_n\} = V_1$ defines the order in which the vertices are mapped to the vertices of the host graph $G_2 = (V_2, E_2, l_2)$. The algorithm starts with an initial empty partial mapping, adding a vertex pair (v_i, w) with $v_i \in V_1$ and $w \in V_2$ in the i-th step such that $\varphi(v_i) = w$.

For each vertex v_i in the sequence s we define the *connecting edges* of v_i regarding s as the set of edges incident to some vertex preceding v in s. We store the vertex sequence together with the connecting edges, forming an *extended vertex sequence*.

Definition 5 (Extended vertex sequence) *Let $G = (V, E, l)$ be a graph and $(v_1, \ldots, v_n)$ a vertex sequence of G. The sequence $S = ((v_1, E_1), \ldots, (v_n, E_n))$ with the connecting*

edges $E_i = \{(v_i, v_j) \in E | i < j\}$ *of* v_i*, for* $1 \leq i \leq n$*, is called* extended vertex sequence *of* G.

Note that each position k of the sequence is related to the subgraph G^k of G induced by $V^k = \{v_j \in V | j \leq k\}$ and $G^k = G[V^k] = (V^k, \bigcup_{i=1}^{k} E_i)$ holds.

Algorithm 3 builds a subgraph isomorphism stepwise by extending a subgraph isomorphism from $G_1^i \subseteq G_1$ to G_2 to a subgraph isomorphism from $G_1^{i+1} \subseteq G_1$ to G_2 in the i-th step, if possible. Consequently the algorithm stops when a subgraph isomorphism of $G_1^n = G_1$ to G_2 is found (line 1).

Algorithm 2: MATCH() function

Input : Number of mapped vertices i.
Data : Graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, extended vertex sequence S of G_1
Output: `true`, if $G_1 \lesssim G_2$, `false` otherwise.

```
1 if i = n then                          ▷ Subgraph isomorphism found
2     return true
3 v ← v_{i+1}
4 for k ∈CANDIDATES(E_{i+1}) do
5     if PRESERVESSGI(v, k, E_{i+1}) then
6         φ(v) ← k                       ▷ Extend mapping
7         if MATCH(i + 1) then return true
8         φ(v) ← NIL                     ▷ Undo extension
9 return false
```

The function PRESERVESSGI(v_i, k, E_i) (line 5) assures that the extension of the mapping by the pair (v_i, k) forms a subgraph isomorphism from G^i to G_2 by checking if

- both vertices have the same label, i.e. $l_1(v_i) = l_2(k)$ and
- φ remains an injection and
- for all $e \in E_i$ a corresponding edge $e' \in E_2$ with $l_1(e) = l_2(e')$ exists.

In addition a degree pruning rule is applied by rejecting all pairs (v_i, k) with $deg(v_i) > deg(k)$, avoiding some unnecessary mappings that must fail in a subsequent step.

The function CANDIDATES(E_i) also utilizes the connecting edges to compute the set of candidates to which v may be mapped to. If $E_i \neq \emptyset$ then an arbitrary edge $(u, v) \in E_i$ is chosen and the candidates are restricted to $adj(\varphi(u))$. Otherwise all vertices of the host graphs are taken into account. Note that the precomputed connecting edges are useful for the implementation of both functions.

2) Vertex Sequence Optimization: An extended vertex sequence is computed once and then used for all subgraph isomorphism tests of the verification step. Though basically any vertex sequence is valid with our algorithm, the ordering of the vertices may have strong impact on the branching degree at each step of the recursion. We propose a heuristic to minimize the degree and hence to reduce the running time of the algorithm. As all subgraph isomorphism tests of the verification step may benefit from a convenient sequence it is promising to spend some runtime into its optimization. We define two criteria by which some sequences are preferred to others.

The first criterion assures that each vertex in the sequence is adjacent to a predecessor in the sequence whenever possible.

Definition 6 (Connected vertex sequence) *A vertex sequence* $(v_1, \ldots, v_n)$ *of a graph* $G = (V, E)$ *is* connected *if and only if for* $1 \leq j \leq n$*:*

$$\exists i < j : (v_i, v_j) \in E \quad \textit{or} \quad \forall k \geq j : \forall i < j : (v_i, v_k) \notin E.$$

An extended connected sequence can be computed with breadth-first-search in time $O(n)$.

However, the restriction to connected vertex sequences typically still leaves a large number of valid sequences. We propose a second criterion to chose from these sequences: A priority is assigned to each vertex according to its label, such that vertices with frequent labels have low priority and vertices with rare labels have high priority. Let $h : \Sigma \to \mathbb{Z}$ be a function such that $h(c)$ is the number of vertices v with $l(v) = c$ in the whole database and H be the total number of vertices. Note that h and H can be calculated in the preprocessing step without great effort and stored with the index. A priority $p(v)$ is assigned to a vertex v according to

$$p(v) = \begin{cases} H^{-1} & \text{if } l(v) = \star \\ (\sum_{l \in L} h(l))^{-1} & \text{if } l(v) = [s_1, s_2, ..., s_n] \\ (H - \sum_{l \in L} h(l))^{-1} & \text{if } l(v) = ![s_1, s_2, ..., s_n] \\ h(l(v))^{-1} & \text{otherwise,} \end{cases}$$

where $L = \{s_1, \ldots, s_n\}$ is the set of allowed/forbidden labels.

Priorities are taken into account by a slight modification of breadth-first-search: The outer loop of the BFS iterates over the vertices ordered descending by their priority and the queue is replaced by a priority queue. This assures that whenever there are multiple choices preserving the condition of Def. 6 the next vertex selected has maximum priority of all the vertices selectable. This optimization might further reduce the search space. The intuition is that vertices with rare labels are more difficult to map and thus misleading mappings are identified earlier and widely branched search trees are avoided.

For a connected pattern graph with n vertices and a host graph with m vertices and a maximum degree of d the algorithm yields a worst-case runtime of $O(md^n)$ using a connected vertex sequence. Thus the algorithm is particularly suitable for graphs of bounded degree as e.g. molecular graphs.

III. Experimental Results

In this section, we compare the new approach with state-of-the-art techniques for graph indexing, namely GCoding, gIndex, GraphGrepSX and Closure-Tree. GraphGrepSX is the latest release in a line of methods using exhaustive enumerations of paths, while gIndex is a prominent example of a data-mining-based approach. Closure-Tree and GCoding both do not use the concept of features, but have shown promising results in earlier experiments. Implementations of these indexing systems were provided by the authors for comparison purpose. We also wanted to include TreePi in our

experiments as another representative of data-mining based methods, but unfortunately were not able to obtain a working version early enough for our test runs. We implemented our indexing system using trees and cycles as described in Sect. II as well as a path-based fingerprint comparable to the Daylight method.

The experiments are conducted on two kinds of data sets: The first consists of graphs derived from chemical structural data related to an AIDS Antiviral Screen published by the National Cancer Institute [26]. The second dataset is composed of synthetic graphs which were created following the algorithm proposed in [5].

The proposed indexing techniques were implemented in Java and compiled and executed with Sun Java JDK v1.6.0, GraphGrepSX was compiled with GCC v4.3.3, the other tools where used as provided by the authors. All tests were conducted on an Intel Core i5 machine at 2.67GHz with 8GB of RAM.

A. Configuration

Both types of datasets are annotated with vertex and edge labels. The implementations of Closure-Tree and GraphGrepSX do not support edge labels which are yet required for many applications like, e.g. substructure search in chemical compound databases. To overcome this drawback we use the same approach as [18], [19]: Each edge e is split by an additional vertex v encoding the label of the edge, i.e. $l(v) = l(e)$.

Queries for both datasets are extracted from the database graphs in the same manner: First a database graph is selected according to uniform distribution, from which a subgraph is extracted that serves as query pattern. The subgraph is build by first selecting an edge of the graph according to uniform distribution and growing the subgraph edge by edge until the specified size is reached. In each step an edge incident to a vertex of the current subgraph is selected and added to the subgraph. We use queries of size 4, 8, 12, 16, 20 and 24. For each size $1,000$ query graphs are created with the procedure described above.

For gIndex we chose the same settings as used in [6]: The maximum fragment size $maxL$ is set to 10 and the minimum support for the largest fragments Θ is set to $0.1|\mathcal{D}|$. GraphGrepSX allows to configure the maximum length of path used as features. Since GraphGrepSX is used on instances with additional vertices encoding edge labels that are not supported natively we set this value to 15. This allows to find paths with up to 8 vertices and 7 edge label encoding dummy vertices lying in between. For Closure-Tree we used the default settings, i.e. the *Neighbor Biased Mapping* algorithm to create graph closures, a minimum number of fan-outs $m = 20$ and a maximum number of fan-outs $M = 2m - 1$.

The path-based fingerprint indexing system uses a fingerprint size of 4096 bits and paths with a maximum length of 8 comparable to the settings used for GraphGrepSX. This indexing system is denoted by P8E12. For the approach using trees and cycles as described in Sect. II we set the maximum tree size to 5 and 6, respectively, and the maximum cycle size to 8. The indexing system using trees of size 5 and a fingerprint size of 2048 bits is denoted by T5C8E11 and the index with a maximum tree size 6 uses a fingerprint size of 4096 bits and is denoted by T6C8E12.

B. AIDS Antiviral Screen Dataset

This section reports on experiments performed on molecular graphs derived from chemical compounds of the AIDS Antiviral Screen dataset [26]. Graphs naturally represent the structural formula of chemical compounds. The atomic symbols and type of bonds are encoded with vertex and edge labels, respectively. All compounds of the dataset come with explicit hydrogen atoms. We decided to keep hydrogen atoms and convert the structural data to molecular graphs without any modification. This results in graphs with 45.7 vertices and 47.7 edges on average and a maximum of 438 vertices and 441 edges. The majority of the vertices are labeled with H (44%), C (40.6%), O (7.7%) and N (5.3%) with a total of 63 distinct vertex labels. Less than 1% of the vertices are labeled by any of the other labels not mentioned above. The majority of the edges represent single and double bonds, only a few bonds with higher orders occur.

The database contains a total of $42,687$ structures, some of which are disconnected. Although our indexing system also works with disconnected graphs, we removed them for the experimental comparison to avoid possible difficulties with other indexing systems. We also removed 40 graphs with more than 255 edges because these graphs tend to cause problems with the implementation of gIndex. From the remaining graphs of the dataset, $10,000$ where selected randomly to form the graph database. Another dataset consisting of $40,000$ graphs was created to analyse the scalability to large databases.

1) Preprocessing: In this section we analyse the preprocessing time, the runtime required to process the graph database to create the index, as well as the size of the index data structure. The dataset was split into parts of different size to report the dependency on the database size.

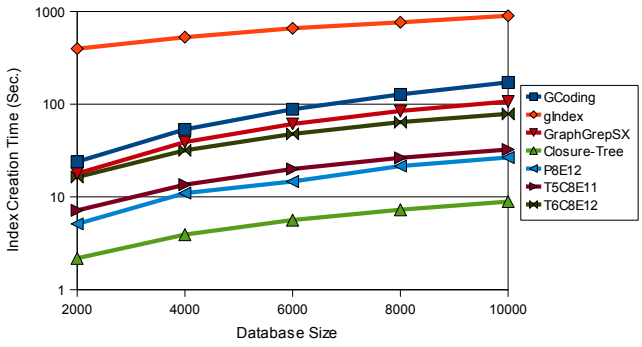


Fig. 2. Preprocessing time on the AIDS dataset.

Fig. 2 shows the index creation time for the indexing systems in comparison. While Closure-Tree performs best, the fingerprint indices can also be build rapidly. Indexing trees of size up to 5 and cycles of a maximum length of 8 takes

only slightly more time than indexing path with a maximum length of 8. Increasing the allowed size to 6 increases the runtime noticeably. However T6C8E12 remains competitive compared to the other indexing systems. gIndex, the only data-mining-based approach in comparison, needs by far the longest runtime to build the index due to the expensive feature mining step.

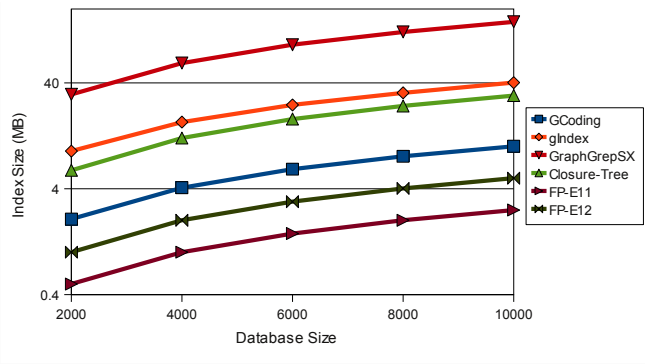


Fig. 3. Index size on the AIDS dataset.

When comparing the index size, one should take in consideration that these values are the sizes of the index files as created by the implementations provided by the authors. While the fingerprint index naturally uses a binary file format, GCoding and gIndex use plain text files. Closure-Tree and GraphGrepSX also use a binary file format. This has a considerable impact on the index size. However, Fig. 3 clearly shows that the index size of the fingerprint-based approach is convenient compared to other state-of-the-art indexing systems. The size of a fingerprint-based index only depends on the fingerprint size of a single graph and not on features used. Thus we do not distinguish between T6C8E12 and P8E12, but provide the index size for a fingerprint of 2048 bits (FP-E11) and 4096 bits (FP-E12).

Even FP-E12 shows a reasonable index size of approximately 5 MB for a database consisting of $10,000$ graphs. All indexing systems show a linear growth of the index size in relation to the database size.

2) Filtering Strength: The key property of an indexing system following the Filter-and-Verification paradigm is its filtering strength, i.e. the ability to minimize the number of false-positives included in the candidate set.

Closure-Tree is not able to answer $1,000$ queries within a time limit of 2 hours, so that it is excluded from this comparison. This might indicate that Closure-Tree does not scale well and experiences difficulties with dummy vertices encoding edge labels.

Fig. 4 depicts the average answer set sizes, denoted by OPT, as well as the average size of the candidate sets the different indexing systems yield. P8E12 of course performs worse than GraphGrepSX as both approaches index paths up to length 8, whereas GraphGrepSX does not suffer from collisions and in addition does not only store the presence of a feature but also its multiplicities. This comes at the cost of a considerable

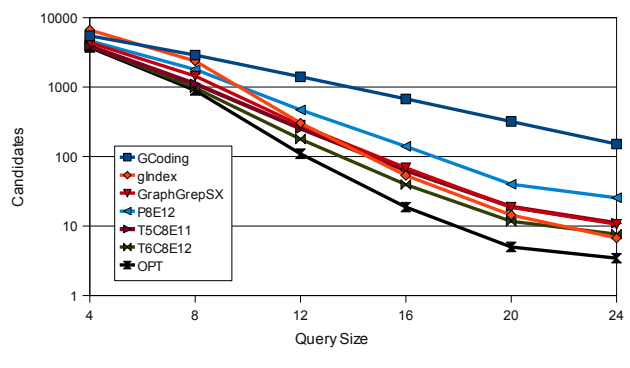


Fig. 4. Average number of candidates on the AIDS dataset.

larger index size as shown in Sect. III-B.1. Nevertheless GraphGrepSX in turn is outperformed by the fingerprint index T6C8E12, which is shown to be the best indexing system in comparison for almost all query sizes. Although it is well known and apparently clear that trees allow a better filtering strength than paths, this clear result could not be predicted beforehand: The maximum tree size used is smaller than the maximum path length in GraphGrepSX, a loss of information has to be taken by the compact fingerprint data structure and furthermore no multiplicities are stored with T6C8E12. The result is a clear evidence of the excellent filtering strength of trees on molecular graphs. This also suggests that fingerprints are an appropriate technique to cope with a multitude of features and avoid an oversized index at the same time. gIndex, on the other hand, adopts the contrary data-mining approach to reduce the amount of features. The filtering strength of gIndex increases with the size of the query graphs and reaches the same level as T6C8E12 for queries of size 24, but performs worse with smaller query graphs. This result can possibly be explained by the use of larger features that, in some cases, capture the structure of large query graphs well, while a large number of smaller trees fail. However, for the major part of the input instances T6C8E12 is shown to have a better filtering strength than gIndex. Especially small queries are not well handled by gIndex, while T6C8E12 exhibits a consistently good filtering strength.

The fingerprint index T5C8E11 naturally performs worse than T6C8E12 but still reasonable, almost at the same level as GraphGrepSX. GCoding by contrast shows a weak filtering strength for almost all query sizes.

3) Response Time: The overall response time includes the runtime of the filtering step as well as the runtime of the verification, which in turn depends on the filtering strength of the index and the used subgraph isomorphism algorithm. gIndex does not support the verification of the candidate set and is hence not included in this comparison.

Fig. 5 shows the response time for $1,000$ queries per query size. All indexing system in general response to large queries more rapidly than to small queries – this can easily be explained by the size of the answer set of these queries: Small queries lead to a large answer set and thus require an extensive

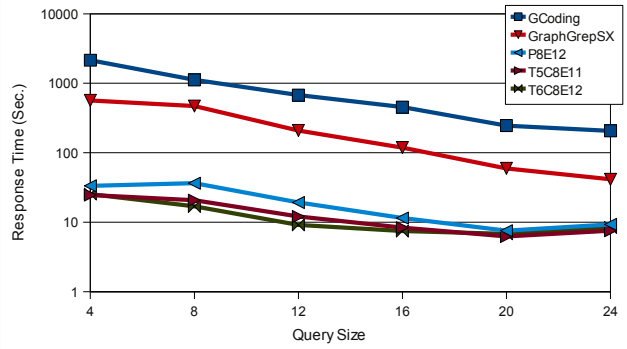


Fig. 5. Total response time for $1,000$ queries per size on the AIDS dataset.

verification step. The superior overall response time of the fingerprint indexing systems can only in parts be explained by the filtering strength (see Fig. 4) of the indices. In fact all the three indexing system using the subgraph isomorphism algorithm described in Sect. II-B perform quite well.

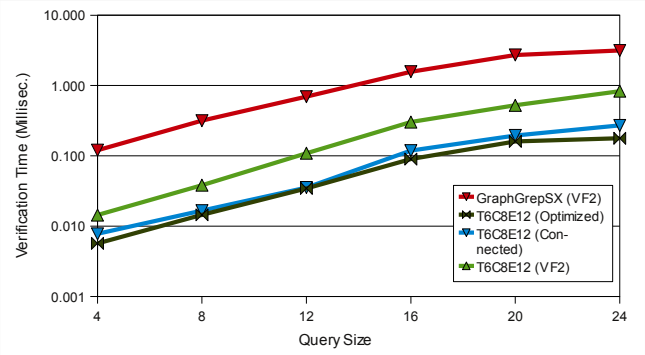


Fig. 6. Average verification time for a single subgraph isomorphism check on the AIDS dataset.

Thus the major part of the excellent response time must be due to the savings in the verification step. The runtime of the subgraph isomorphism test is analyzed in detail in Fig. 6 showing the average runtime of a single subgraph isomorphism test involved in the queries from Fig. 5. GCoding uses Ullmann's algorithm, but does not output a separate runtime of the verification step and was therefore excluded from this comparison. GraphGrepSX relies on the vflib implementation of the well known VF2 algorithm. However, it must be noted that in this test VF2 operates on larger graphs due to the dummy vertices inserted to encode edge labels. Furthermore these additional vertices may decrease the effectivity of VF2's pruning procedure based on terminal sets. We have implemented the VF2 algorithm and combined it with our indexing system for a fair comparison. Fig. 6 shows the results for CT-Index combined with VF2, our new subgraph isomorphism algorithm with a connected vertex sequence and with a vertex sequence optimized by vertex label frequencies. The results show that our approach is superior compared to VF2 in the context of a single pattern graph searched in a multitude of graphs. The optimization heuristic further reduces the runtime.

4) Large Databases: We performed further tests on a large dataset consisting of $40,000$ graphs taken from the AIDS Antiviral Screen dataset. gIndex can not be used directly on this dataset due to the expensive data-mining step. Thus we used the incremental update function to build the feature set from a subset consisting of $10,000$ graphs of the database and added the other graphs later without changing the feature set. Unfortunately gIndex failed nevertheless due to an overflow error. The other indexing systems scale well on the larger

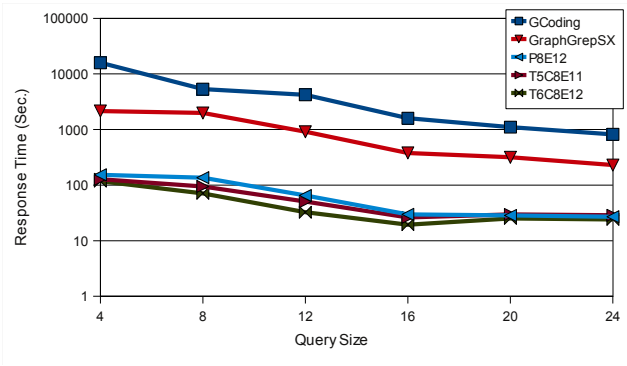


Fig. 7. Total response time for $1,000$ queries per size on the AIDS dataset ($40,000$ graphs).

dataset. Fig. 7 shows the total response times. Again CT-Index outperforms the other systems. The increase of the response time is mainly due to an increase of the runtime of the verification step. However, the filtering strength of the indexing systems remain comparable to the filtering strength observed for the database consisting of $10,000$ graphs. The growth of the answer set size results in a larger number of subgraph isomorphism test that can not be avoided and hence the total response time increases.

C. Synthetic Graph Dataset

For the synthetic graph dataset a generator was used which creates graphs by composing them in an iterative manner by insertion of smaller graphs from a pre-defined *seed pool*. In a first step the seed pool of S graphs with average size I is established by creating seed fragments randomly: For each seed fragment the number of edges is determined by Poisson distribution with mean I. Edges are added one by one, in which a new edge connects two existing vertices with probability C and connects an existing vertex with a new one otherwise. For each fragment a weight is drawn from an exponential distribution with unit mean. The sum of all weights is normalized to 1 by scaling all weights by the same factor. A graph is assembled by choosing seed fragments and adding them to the graph until a maximum size is reached. A seed is selected to be included in a graph with probability determined by its weight. The maximum size is a poisson distributed random variable with mean T. A seed is inserted into the graph by randomly adding at least one edge connecting a vertex from the graph with a vertex from the seed fragment.

The number of additional edges is a Poisson random variable with mean 1. Vertex and edge labels are chosen randomly from L_V distinct vertex labels and L_E distinct edge labels, respectively. The probabilities of the labels are determined in the same manner as the probabilities of the seed fragments. This leads to labels that occur with diverse frequencies – a characteristic observed in a broad range of real-world graphs.

The number of different labels and their distribution has wide influence on the graph indexing system: On one side the number of different subgraphs contained in database graphs considerably increases with the number of different labels. This fact has direct impact on the feature set used by indexing approaches using exhaustive enumeration like GraphGrep, GDIndex and our indexing system. Since with our approach an oversized feature set leads to a large number of collisions and thus to a loss of information, the maximum feature size and the fingerprint size have to be adjusted. With GraphGrepSX the index size increases which is controllable by the maximum path length.

In general, graphs with diverse labels can be indexed easily because each graph tends to feature certain characteristics that only a few database graphs have. Consequently Yan et al. observed in [6] that an index based on paths performs just as good as an index using more complex features applied to graphs with many labels. The results suggest that especially graphs with few labels are difficult and of interest for experimental comparison.

Therefore the following settings were used throughout the experiments: $D = 10,000$, $T = 50$, $I = 15$, $S = 100$, $L_V = 3$, $L_E = 2$, $C = 0.1$.

1) Results: We have omitted an in-detail comparison of the preprocessing step on synthetic graphs but present the analysis of the filtering strength in this section. However, it is worth mentioning that all indexing system required significant more time on the synthetic dataset than on the molecular graph dataset. To build the index for the full database consisting of $10,000$ graphs gIndex required approximately 44 minutes, while the other indexing system needed less than 5 minutes. The fastest indexing system T5C8E11 and P8E12 required less than a minute. With GraphGrepSX as well as T6C8E12 building the index took circa 3 minutes, GCoding required almost 5 minutes. This result indicates that the graph mining step performed by gIndex is quite costly on the synthetic dataset with few labels.

The size of the candidate sets created by the different indexing systems is depicted in Fig. 8. The performance of the indices is overall similar to the performance observed on the real-world dataset. Again a wide discrepancy between OPT and the number of candidates created by GCoding can be observed. The path-based indexing methods GraphGrepSX and P8E12 perform similar with GraphGrepSX having an advantage over the fingerprint-based index. The tree-based indices perform better than the path-based indices for a broad range of query sizes. In this comparison gIndex performs extremely well for large query sizes, nearly reaching the optimal candidate size. This might be explained by the inherent

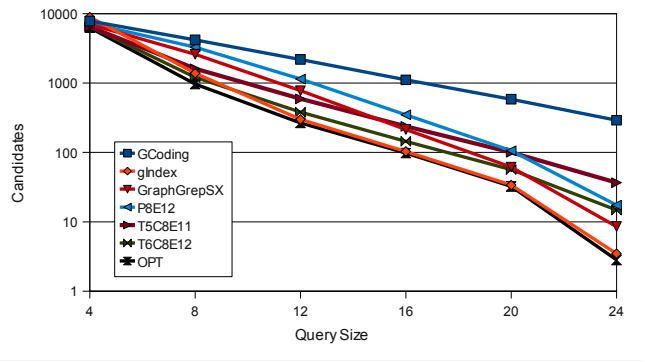


Fig. 8. Average number of candidates on the synthetic dataset.

structure of the graphs which were build from a small number of fragements.

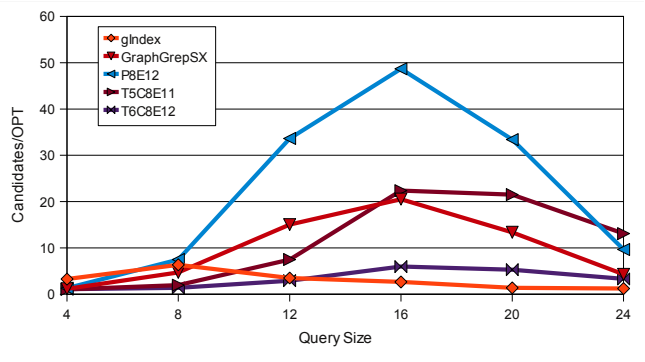


Fig. 9. Average Candidate-OPT-ratio on the synthetic dataset.

We further investigate the filtering strength by calculating the average ratio of the number of candidates and the answer sets size by the equation

$$\frac{1}{|\mathcal{Q}|} \sum_{Q \in \mathcal{Q}} \frac{|C(Q)|}{|D(Q)|}$$

for each set of queries $\mathcal{Q}$ of the same size. Fig. 9 shows the results. gIndex and T6C8E12 both show a good filtering strength despite the fact that the synthetic graphs contain few distinct labels. As already indicated by Fig. 9 gIndex has the advantage over T6C8E12 regarding large query sizes with 16 or more edges, whereas T6C8E12 shows better results using small pattern graphs consisting of 12 edges or less. Such queries are typicallly contained in a large number of graphs and thus an extensive verification step is inevitable and causes a large response time as depicted in Fig. 10. Larger pattern graphs on the other hand have a small answer set and lead to admissible response times even with indices having a weak filtering strength. This observation is a good argument to turn the attention particularly on small query graphs where our approach outperforms all other indexing techniques. Furthermore for all indexing systems in comparison allowing wildcards requires the deletion of vertices and edges of the query graph and typically leads to smaller fragments. The results indicate

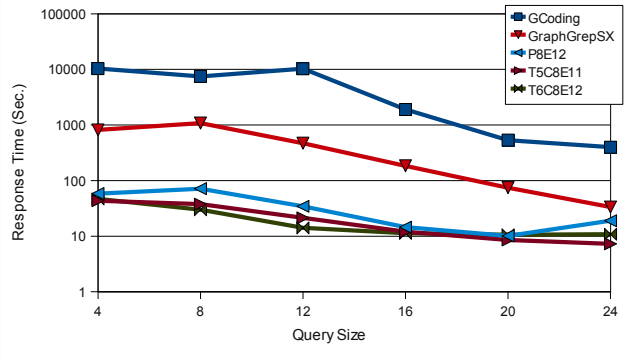


Fig. 10. Total response time for 1,000 queries per size on synthetic dataset.

that CT-Index performs superior on these degenerated query graphs.

IV. Conclusion

We present a new indexing based on a combination of tree and cycle features to speed up subgraph queries in graph databases. Our index is created by exhaustive feature enumeration up to a specified feature size, and uses a fingerprint technique to store the graph characteristics. Our experimental evaluation shows that these features allow a better filtering without an additional computational overhead.

Thus CT-Index provides a viable alternative to existing approaches and is competitive to methods based on data-mining. At the same time the convenient manageability of fingerprints and the scalability to large databases are notable characteristics of our technique. In addition further improvements for fingerprint-based indices are known which can be incorporated into our indexing systems: The technique of folding reduces the need of parameter optimization and keeps the index size low. Furthermore the filtering step can be speed up by fast index structures for bit arrays (see e.g. [27], [28]). Such techniques are likely to pay off when searching in databases with millions of graphs as planned for the future development.

The proposed indexing system was integrated into Scaffold Hunter [29], an open-source tool for the analysis and exploration of chemical databases, and proved to be practical for real-world applications.

Acknowledgment

We would like to thank the authors who provided their software for comparison purpose, namely Dennis Shasha and Rosalba Giugno for GraphGrepSX, Xifeng Yan for gIndex, Lei Zou for GCoding, Huahai He and Ambuj K. Singh for Closure Tree and Shijie Zhang for TreePi.

References

[1] J. M. Barnard, "Substructure searching methods: Old and new," *Journal of Chemical Information and Computer Sciences*, vol. 33, no. 4, pp. 532–538, 1993.

[2] K. Degtyarenko, J. Hastings, P. de Matos, and M. Ennis, "ChEBI: An open bioinformatics and cheminformatics resource." *Curr Protoc Bioinformatics*, vol. Chapter 14, no. 26, pp. 14.9.1–14.9.20, 2009.

[3] "The PubChem project," http://pubchem.ncbi.nlm.nih.gov/.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[5] M. Kuramochi and G. Karypis, "An efficient algorithm for discovering frequent subgraphs," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 9, pp. 1038–1051, 2004.

[6] X. Yan, P. S. Yu, and J. Han, "Graph indexing based on discriminative frequent structure analysis," *ACM Trans. Database Syst.*, vol. 30, no. 4, pp. 960–993, 2005.

[7] R. Giugno and D. Shasha, "GraphGrep: A fast and universal method for querying graphs," in *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, vol. 2, 2002, pp. 112–115 vol.2.

[8] "Daylight theory manual v4.9," http://www.daylight.com, Jan. 2008.

[9] S. Zhang, M. Hu, and J. Yang, "TreePi: A novel graph indexing method," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, 2007, pp. 966–975.

[10] R. D. Natale, A. Ferro, R. Giugno, M. Mongiovì, A. Pulvirenti, and D. Shasha, "SING: Subgraph search in non-homogeneous graphs," *BMC Bioinformatics*, vol. 11, p. 96, 2010.

[11] J. Cheng, Y. Ke, and W. Ng, "Efficient query processing on graph databases," *ACM Trans. Database Syst.*, vol. 34, no. 1, pp. 1–48, 2009.

[12] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[13] H. He and A. K. Singh, "Closure-Tree: An index structure for graph queries," in *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*. IEEE Computer Society, 2006.

[14] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 10, pp. 1367–1372, 2004.

[15] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha, "Enhancing graph database indexing by suffix tree structure," in *Pattern Recognition in Bioinformatics*, ser. Lecture Notes in Computer Science, T. Dijkstra, E. Tsivtsivadze, E. Marchiori, and T. Heskes, Eds. Springer Berlin / Heidelberg, 2010, vol. 6282, pp. 195–203.

[16] R. Battiti and F. Mascia, "An algorithm portfolio for the sub-graph isomorphism problem," in *SLS'07: Proceedings of the 2007 international conference on Engineering stochastic local search algorithms*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 106–120.

[17] G. V. Batz, M. Kroll, and R. Geiß, "A first experimental evaluation of search plan driven graph pattern matching," in *AGTIVE*, 2007, pp. 471–486.

[18] L. Zou, L. Chen, J. X. Yu, and Y. Lu, "A novel spectral coding in a large graph database," in *EDBT '08: Proceedings of the 11th international conference on Extending database technology*. New York, NY, USA: ACM, 2008, pp. 181–192.

[19] D. W. Williams, J. Huan, and W. Wang, "Graph database indexing using structured graph decomposition," *Data Engineering, International Conference on*, vol. 0, pp. 976–985, 2007.

[20] P. Zhao, J. X. Yu, and P. S. Yu, "Graph indexing: tree + $\Delta \geq$ graph," in *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 938–949.

[21] H. Jiang, H. Wang, P. S. Yu, and S. Zhou, "GString: A novel approach for efficient search in graph databases," *Data Engineering, International Conference on*, vol. 0, pp. 566–575, 2007.

[22] Y. Chi, R. Muntz, S. Nijssen, and J. Kok, "Frequent subtree mining – an overview," 2005.

[23] D. Gusfield, *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

[24] A. V. Aho and J. E. Hopcroft, *The Design and Analysis of Computer Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1974.

[25] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM J. Comput.*, vol. 4, no. 1, pp. 77–84, 1975.

[26] "National Cancer Institute - DTP AIDS antiviral screen dataset," http://dtp.nci.nih.gov/docs/aids/aids_data.html, 1999.

[27] J. Zobel, A. Moffat, and K. Ramamohanarao, "Inverted files versus signature files for text indexing," *ACM Trans. Database Syst.*, vol. 23, no. 4, pp. 453–490, 1998.

[28] S. Helmer and G. Moerkotte, "A performance study of four index structures for set-valued attributes of low cardinality," *The VLDB Journal*, vol. 12, no. 3, pp. 244–261, 2003.

[29] S. Wetzel, K. Klein, S. Renner, D. Rauh, T. I. Oprea, P. Mutzel, and H. Waldmann, "Interactive exploration of chemical space with Scaffold Hunter," *Nature Chemical Biology*, vol. 5, no. 8, pp. 581–583, 2009.