Faster Subgraph Isomorphism Detection by Well-Founded Total Order Indexing

Markus Weber[a], Marcus Liwicki[a], Andreas Dengel[a,b]

[a]German Research Center for Artificial Intelligence (DFKI) GmbH,
Trippstadter Straße 122, 67663 Kaiserslautern, Germany
[b]Knowledge-Based Systems Group, Department of Computer Science,
University of Kaiserslautern, P.O. Box 3049, 67653 Kaiserslautern

Abstract

In this paper an extension to index-based subgraph matching is proposed. This extension significantly speeds up the indexing time for graphs where the nodes are labeled with a rather small amount of different classes. Furthermore, the needed storage amount is significantly reduced. In order to reduce the complexity, we introduce a weight function for labeled graphs. Using this weight function, a well-founded total order is defined on the weights of the labels. Inversions which violate the order are not allowed. A computational complexity analysis of the new preprocessing is given and its completeness is proven. Furthermore, in a number of practical experiments with randomly generated graphs the improvement of the new approach is shown. In experiments performed on random sample graphs, and on real-world datasets. The number of permutations for the real-world datasets have been decreased to a fraction of 10^{-5} and 10^{-8} in average compared to the original approach by Messmer. The novel indexing strategy makes indexing of larger graphs feasible, allowing for fast detection of subgraphs.

Keywords: Graph isomorphism; Subgraph isomorphism; Tree search; Decision tree; Indexing

Graphs play a major role in structural pattern recognition. An important task in this field is to find similar structures (error-tolerant graph matching) or the same structure (exact graph matching). The focus of this paper is on the latter task, which is important if exactly the same sub-structure needs to be retrieved.

Exact graph matching is needed when the user searches for specific constellations in molecules [1], in computer vision for the recognition of 3-D objects [2, 3], shape matching in image analysis [4, 5], or room-constellations in floor plans [6]. In most applications, the retrieval result should be available in real-time and the database of reference structures does not change too often. For those situations it is advisable to build an index of the reference structures in advance.

Such a method has been proposed by Messmer et al. [7]. It builds an index using the permutated adjacency matrix of the graph. The real-time search is then based on a tree based search. While the method has shown to be efficient for reference set with small graphs, it is infeasible for graphs with more than 19 vertices.

This paper proposes a method to overcome this problem. Assuming that the number of labels for the nodes is relatively small, we introduce a well-founded total order and apply this during index building. This optimization decreases the amount of possible permutations dramatically and allows building indexes of graphs with even more than 30 vertices.

Note that a preliminary version of this paper has been published in [8]. However, the focus of [8] was on a short description of the approach and first experiments on random graphs. This paper elaborates more on the algorithm, its validity and its complexity. Furthermore, additional experiments have been performed new experiments on random graphs with 100-150 vertices. Moreover experiments on two real-world databases, i.e., the AIDS Antiviral Screen Database of Active Compounds [9] and the Mutagency database [10] have been accomplished .

The rest of this paper is organized as follows. First, Section 1 gives an overview over related work. Subsequently, Section 2 introduces definitions and notation which are used and Section 3 describes the new preprocessing step. Next, Section 4 shows that the number of computational steps is significantly decreased on random graphs as well as on realistic databases. Finally, Section 5 concludes the work.

1. Related Work

In [11, 12], a survey of the work done in the area of graph matching is given. Conte et al. [11] defines two taxonomies, one which almost contains all the graph matching algorithms proposed from the late seventies, and describes the different classes of algorithms. The second considers the types of common applications of graph-based techniques in the Pattern Recognition and Machine Vision field. Using this taxonomy, our approach can be assigns to exact matching, as it is a modified version of Messmer's method [7] which is assign to this category.

The focus of Goa et al. [12] is the calculation of error-tolerant graph-matching; where calculating a graph edit distance (GED) is an important way. Mainly the GED algorithms described are categories into algorithms working on attributed or non-attributed graphs. Ullman's method [13] for subgraph matching is known as one of the fastest methods. The algorithm attains efficiency by inferentially eliminating successor nodes in the tree search. To filter unmatched graphs, enumerated paths are used as index features in GraphGrep [14]. While TreePi [15] and FG-Index [16] use frequent subtrees/subgraphs as index features, GIndex [17] uses discriminative frequent fragments to improve filtering and reduce index size. GString [18] reduces the problem of graph querying to subsequence matching. A graph decomposition based approach is taken in Williams et al. [19] to hash canonical subgraphs for fast accessing. A similar approach is taken in SAGA [20] in which answers are generated by assembling hits of enumerated fragments. In [21], a new data model for the storage and management of graph objects has been proposed. It relies on the idea of structural unification, a novel graph representation based on minimum structures, and an indexing mechanism for storing minimum graph structures.

Bunke [22, 23] discussed several approaches in graph-matching. One way to cope with error-tolerant subgraph matching is using the maximum common subgraph as a similarity measure. Furthermore the application of graph edit costs which is an extension of the well-known string edit distances. A further group of suboptimal methods are approximative methods, they are based on neural networks [24], such as the Hopfield network, Kohonen map [25] or Potts MFT neural net. Moreover methods as genetic algorithms [26, 27], Eigenvalues [28], and linear programming [29] are used.

Recently, He and Singh proposed GraphQL as a query language for graphs [30]. GraphQL assumes an underlying optimization based on prudent access structures and cost model. Graph matching is challenging in presence of large databases [6, 23, 31, 32]. Consequently, methods for preprocessing or indexing are essential. Preprocessing can be performed by graph filtering or concept clustering. The main idea of the graph filtering is to use simple features to reduce the number of feasible candidates. Another concept, clustering, is used for grouping similar graphs.

In principle, given a similarity (or dissimilarity) measure, such as GED [33], any clustering algorithm can be applied. Graph indexing can be performed by the use of decision trees.

Messmer and Bunke [7] proposed a decision tree approach for indexing the graphs. They are using the permutated adjacency matrix of a graph to build a decision tree. This technique is quite efficient during run time, as a decision tree is generated beforehand which contains all model graphs. However, the method has to determine all permutations of the adjacency matrices of the search graphs. Thus, as discussed in their experiments, the method is practically limited to graphs with a maximum of 19 vertices. The main contribution of this paper is to improve the method of Messmer and Bunke for special graphs by modifying the index building process.

2. Definitions and notations

In this section some basic definitions are given which will be used throughout the paper.

Definition 1. *A labeled graph is a 6-tuple, $G = (V, E, L_v, L_e, \mu, \upsilon)$, where*

- *V is a set of vertices,*
- *$E \subseteq V \times V$ is a set of edges,*
- *L_v is a set of labels for the vertices,*
- *L_e is a set of labels for the edges.*
- *$\mu : V \rightarrow L_v$ is a function which assigns a label to the vertices,*
- *$\upsilon : E \rightarrow L_e$ is a function which assigns a label to the edges.*

The labels L_v set is a finite set and the labeling function μ assigns the type of an entity to a concrete vertex.

A common representation for a labeled graph is an adjacency matrix.

Definition 2. *An adjacency matrix is $n \times n$ matrix M.*

$$M = (m_{ij}), i, j = 1, ..., n, \text{ where}$$
$$m_{ii} = \mu(v_i)$$

and

$$m_{ij} = \upsilon((v_i, v_j)) \text{ for } i \neq j.$$

Figure 1 shows an example illustration of a graph and a possible corresponding adjacency matrix. Furthermore, the so called row-column representation is given. In a row-column representation the matrix is represented by its row-column elements a_i, where a_i is a vector of the form

$$a_i = (m_{1i}, m_{2i}, ..., m_{ii}, m_{i(i-1)}, ..., m_{i1}).$$

The following definition for the subgraph is given by:

Definition 3. *Given a graph $G = (V, E, L_v, L_e, \mu, \upsilon)$, a subgraph of G is a graph $G' = (V', E', \mu', \upsilon', L_v', L_e')$ such that*

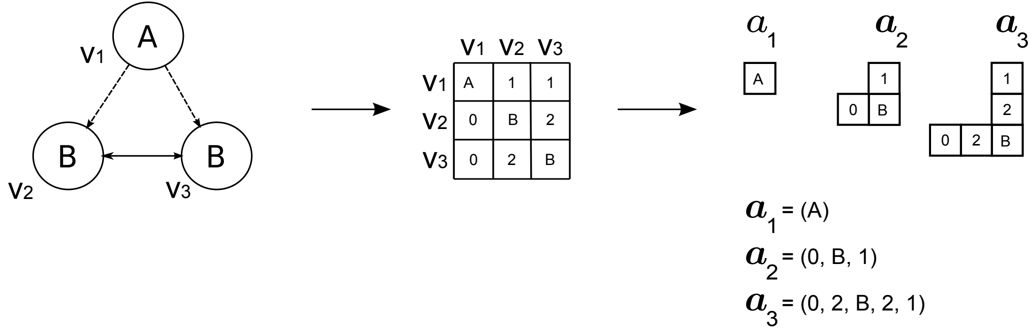


Figure 1: Row-column representation of an adjacency matrix

1. $V' \subseteq V$
2. $E' = E \cap (V' \times V')$
3. $\mu'(v) = \begin{cases} \mu(v) & \text{if } v \in V' \\ \text{undefined} & \text{otherwise} \end{cases}$
4. $\upsilon'(e) = \begin{cases} \upsilon(e) & \text{if } e \in E' \\ \text{undefined} & \text{otherwise} \end{cases}$

Let $G = (V, E, L_v, L_e, \mu, \upsilon)$ be a graph with $V = \{v_1, v_2, ..., v_n\}$. As stated above, G can also be represented by an adjacency matrix M. Note that the matrix M is not unique for a graph G. If M represents G, then any permutation of M is also a valid representation of G.

Definition 4. *A $n \times n$ matrix $P = (p_{ij})$ is a permutation matrix if*

1. $p_{ij} \in \{0, 1\}$ *for i,j = 1, ..., n*
2. $\sum_{i=1}^{n} p_{ij} = 1$ *for j = 1, ..., n*
3. $\sum_{j=1}^{n} p_{ij} = 1$ *for i = 1, ..., n*

Let G be a graph represented by an $n \times n$ adjacency matrix M and P be an $n \times n$ permutation matrix P with P^T as the transposed matrix, then the $n \times n$ matrix

$$M' = PMP^T$$

is also a valid representation of G.

Definition 5. *Let $G = (V, E, \mu, \upsilon, L_v, L_e)$ be a graph, then $A(G)$ is the set of all permuted adjacency matrices of G,*

$$A(G) = \{M_P | M_P = PMP^T \text{ where } P \text{ is a } n \times n \text{ permutation matrix}\}.$$

In order to formalize the subgraph isomorphism, we can first define the isomorphism as follows:

Definition 6. *Let G_1 and G_2 be two graphs and M_1 and M_2 their corresponding adjacency matrices. G_1 and G_2 are isomorphic if there exists a permutation matrix P, such that*

$$M_2 = PM_1P^T$$

So, the permutation matrix P can be considered as a bijective function f that maps vertices of G_1 to G_2, and vice versa. Thus finding a *graph isomorphism* between G_1 and G_2 is equivalent to finding a permutation matrix P for which Definition 6 holds true.

Definition 7. *Given two graphs G_1 and G_2, there is subgraph isomorphism from G_1 to G_2 if there exists a subgraph $S \subset G_2$ such that G_1 and S are isomorphic.*

In order to compare two adjacency matrices with different dimensions, a notation $S_{k,m}(M)$ is required which reduces the dimension of the matrix.

Definition 8. *Let $M = (m_{ij})$ be a $n \times n$ matrix. Then $S_{k,m}$ denotes the $k \times m$ matrix that is obtained from M by deleting rows $k + 1,..., n$ and columns $m + 1,...,n$ where $k, m \leq n$. That is, $S_{k,m}(M) = (m_{ij})$; $i = 1,...,k$ and $j = 1,...,m$.*

Let G_1 and G_2 be graphs with their adjacency matrices M_1 and M_2 of dimension $m \times m$ and $n \times n$ and $m \leq n$. The problem of finding a subgraph isomorphism from G_1 to G_2 is equivalent to finding a $n \times n$ permutation matrix P such that

$$M_1 = S_{m,m}(PM_2P).$$

Besides, we need a definition for orders on sets.

Definition 9. *A partial order is a binary relation $\leq$ over a set P which is reflexive, anti-symmetric and transitive, thus for all a, b and c in P, it holds that:*

- *$a \leq a$ (reflexivity);*
- *if $a \leq b$ and $b \leq a$ then $a = b$ (anti-symmetry);*
- *if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity).*

Definition 10. *A total order is a binary relation $\leq$ over a set P which is transitive, anti-symmetric, and total, thus for all a, b and c in P, it holds that:*

- *if $a \leq b$ and $b \leq a$ then $a = b$ (anti-symmetry);*
- *if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity);*
- *$a \leq b$ or $b \leq a$ (totality).*

Definition 11. *A partial or total order $\leq$ is well-founded,*

iff $(\forall\, Y \subseteq X \;:\; Y \neq \emptyset \rightarrow (\exists y \in Y \;:\; y \text{ minimal in } Y \text{ in respect to } \leq))$.

Additionally to the previous definitions, a weight function is defined which assigns weight to a label.

Definition 12. *The weight function σ is defined as: $\sigma : L_v \rightarrow \mathbb{N}$.*

Using the weight function, a well-founded total order can be defined on the labels of the vertices, for instance $\sigma(L_1) < \sigma(L_2) < \sigma(L_3) < \sigma(L_4)$. Thus the labeled graph can be extended in its definition.

Definition 13. *An ordered labeled graph consists of a 7-tuple, $G = (V, E, L_v, L_e, \mu, \upsilon, \sigma)$, where*

- *V is a set of vertices,*
- *$E \subseteq V \times V$ is a set of edges,*
- *L_v is a set of labels for the vertices,*
- *L_e is a set of labels for the edges,*
- *$\mu : V \rightarrow L_v$ is a function which assigns a label to the vertices,*
- *$\upsilon : E \rightarrow L_e$ is a function which assigns a label to the edges,*
- *$\sigma : L_v \rightarrow \mathbb{N}$ is a function which assigns a weight to the label of the vertices,*

and a binary relation $\leq$ exists which defines a well-founded total order on the weights of the labels:

$$\forall x, y \in L_v \ : \ \sigma(x) \leq \sigma(y) \ \vee \ \sigma(y) \leq \sigma(x)$$

After having introduced the basic definitions, we can use them for the further description and analysis of the algorithm.

3. Algorithm

3.1. Messmer algorithm

In [7], Messmer and Bunke proposed an index-based algorithm. Their method computes all permutations of the adjacency matrices and transforms them into a decision tree. During run-time, the adjacency matrix of the query graph is split into its row-column vectors. These vectors are now used to traverse the decision tree and find the adjecency matrix which contains this sub-structure.

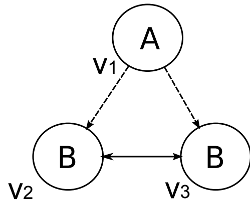
Let $G = (V, E, L_v, L_e, \mu, \upsilon)$ be a graph from the graph database and M the corresponding $n \times n$ adjacency matrix and $A(G)$ the set of permuted matrices. The total number of permutations is $|A(G)| = n!$, where n is the dimension of the permutation matrix, respectively the number of vertices.

Now, let $Q = (V, E, L_v, L_e, \mu, \upsilon)$ be a query graph and M' the corresponding $m \times m$ adjacency matrix, with $m \leq n$. If a matrix $M_P \in A(G)$ exists, such that $M' = S_{m,m}(M_P)$, the permutation matrix P which corresponds to M_P represents a subgraph isomorphism from Q to G, i.e

$$M' = S_{m,m}(M_P) = S_{m,m}(PMP^T).$$

Messmer proposed to arrange the set $A(G)$ in a decision tree, such that each matrix in $A(G)$ is classified by the decision tree. Figure 2 shows an example of the decision tree built from the permuted adjacency matrices for a graph.

Unfortunately, this approach has one major drawback. For building the decision tree, all permutations of the adjacency matrix have to be considered, thus the complexity for compiling an index for a graph G For a graph with more than 19 vertices the computational effort becomes intractable.



	v_1	v_2	v_3
v_1	A	1	1
v_2	0	B	2
v_3	0	2	B

M_1

	v_1	v_3	v_2
v_1	A	1	1
v_3	0	B	2
v_2	0	2	B

M_2

	v_2	v_1	v_3
v_2	B	0	2
v_1	1	A	1
v_3	2	0	B

M_3

	v_2	v_3	v_1
v_2	B	2	0
v_3	2	B	0
v_1	1	1	A

M_4

	v_3	v_1	v_2
v_3	B	0	2
v_1	1	A	1
v_2	2	0	B

M_5

	v_3	v_2	v_1
v_3	B	2	0
v_2	2	B	0
v_1	1	1	A

M_6

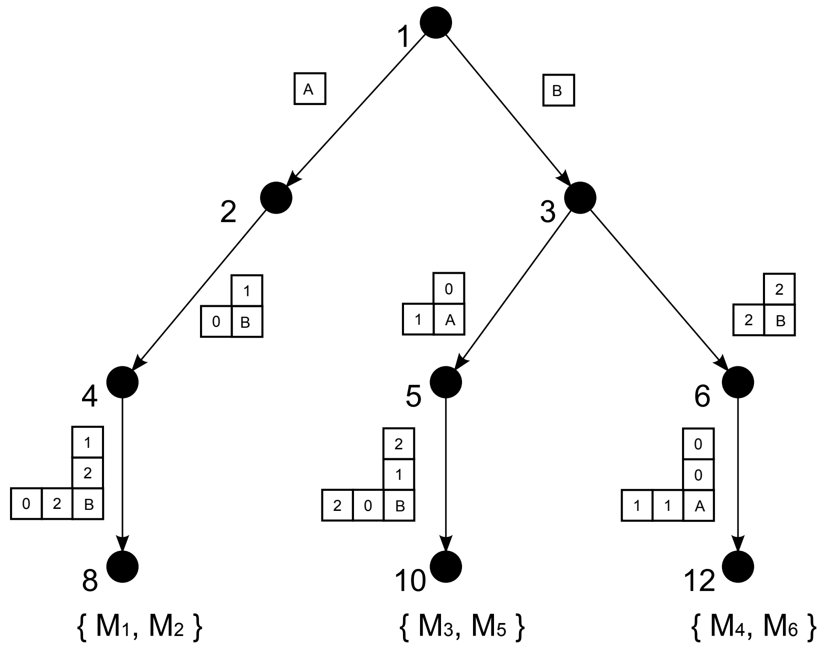


Figure 2: Decision tree for adjacency matrices

3.2. Novel algorithm

In this section, we propose a novel approach to constraint the permutation, in order to overcome the previous mentioned issue. By limiting the number of permutations larger graphs can be considered for the indexing and also the decision tree becomes more sparse.

First, a weight function σ (see Definition 12) is introduced which assigns a weight to each vertex according to its label. So each label has a unique weight and a well-founded total order (see Definition 10 and Definition 11) on the set of labels which reduces the number of allowed inversion for the adjacency matrix. Figure 3 illustrates the modified matrices and the corresponding decision tree for the example in 2. There the highlighted numbers (in red) indicate violations of the ordering (see below) if the following weights for the nodes are considered:

$$\begin{aligned} L_v =& \{A, B, C, D\} \\ \sigma(A) =& 1, \\ \sigma(B) =& 2, \\ \sigma(C) =& 3, \\ \sigma(D) =& 4. \end{aligned}$$

No inversion that violates the ordering is allowed. Hence, just the vertices which have the same label, respectively the same weights, have to be permuted and if the labels have a different weight, just the variations are required. Given the graph illustrated in Fig. 2, the following labels are assigned to the vertices,

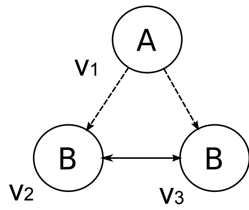
$$\begin{aligned} V =& \{V_1, V_2, V_3\} \\ \mu(V_1) =& A, \\ \mu(V_2) =& B, \\ \mu(V_3) =& B. \end{aligned}$$

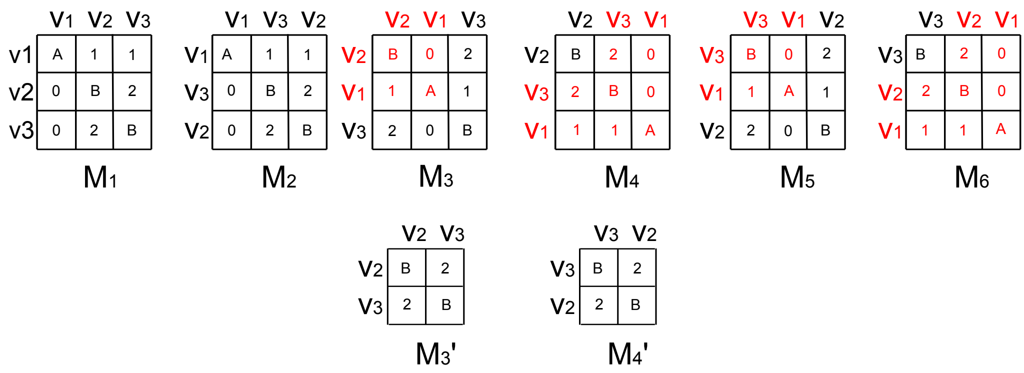
Hence, the only valid permutations are:

1. $\sigma(\mu(V_1)) \leq \sigma(\mu(V_2)) \leq \sigma(\mu(V_3))$
2. $\sigma(\mu(V_1)) \leq \sigma(\mu(V_3)) \leq \sigma(\mu(V_2))$
3. $\sigma(\mu(V_2)) \leq \sigma(\mu(V_3))$
4. $\sigma(\mu(V_3)) \leq \sigma(\mu(V_2))$

Let $VA(G)$ be the set of all valid permutations. The decision tree is built according to the row-column elements of the adjacency matrices $M_P \in VA(G)$.

The decision tree should cover all graphs from the repository. So, let R be the set of graphs $R = \{G_1, G_2, ..., G_n\}$, where n is the total number of graphs in the repository, with their sets of corresponding adjacency matrices $VA(G_1)$, $VA(G_2)$, ..., $VA(G_n)$. Now, each set of adjacency matrices has to be added to the decision tree. As a result for this very small graph, instead of 6 permutations only 4 are need and for the resulting decision tree the number of nodes are reduced from 12 to 6. The following Section 3.3 will provide a short discussion on the retrieval algorithm and Section 3.4 will prove, that even with this reduction the method is still complete and finds all solutions.





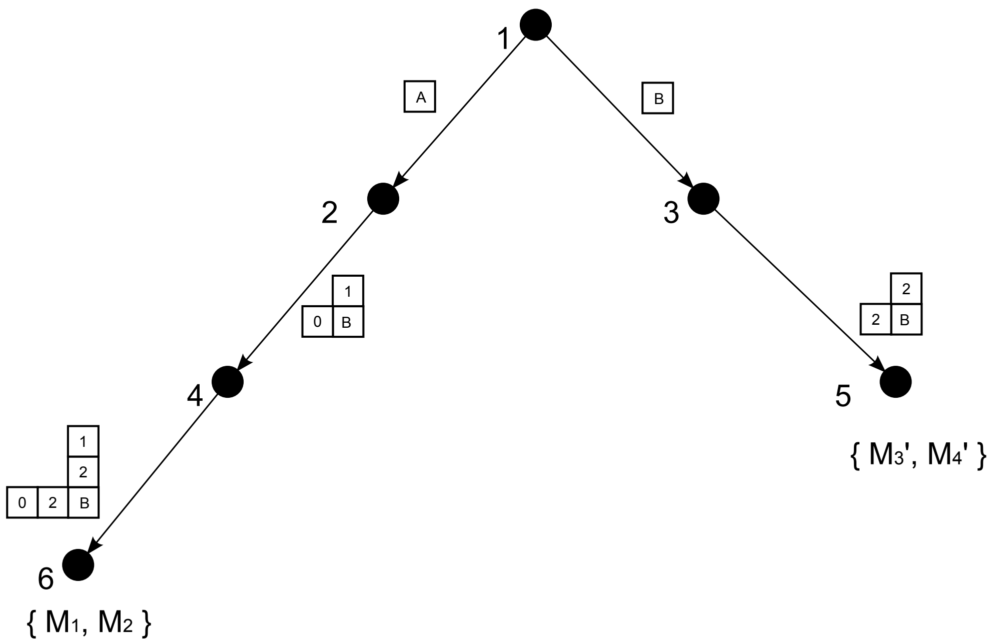


Figure 3: Modified decision tree for adjacency matrices

3.3. Retrieval Algorithm

The a priori computed decision tree acts as an index for subgraphs. So, during run-time the decision tree is loaded into memory and by traversing the decision tree, the corresponding subgraph matrices are classified. For the query graph Q the adjacency matrix M is determined following the constraints defined by ordering. Afterwards the adjacency matrix is split up into row-column vectors a_i. For each level i of the decision tree the corresponding row-column vector a_i is used to find the next node in the decision tree using an index structure. The pseudo code of Algorithm 1 displays how the results are retrieved by traversing the tree and Figure 4 provides an illustration of a sample query.

```
Algorithm 1 RETRIEVAL(Q = (V, E, μ, υ, σ, L_v, L_e), Tree)
Require: Unsorted set V of vertices, μ labeling function, σ weight function
 1: sort(Q, L_v, μ, σ)
Ensure: Vertices V are sorted according to the defined order.
 2: Let R be an empty sorted set which will contain all results.
 3: Determine adjacency matrix M from graph Q.
 4: Determine row-column list RCL from M.
 5: for i ← 1 to |RCL| do
 6:    Let m_i be the match of row-column vector a_i ∈ RCL in tree at level_i.
 7:    if m_i is empty. then
 8:       return ∅
 9:    end if
10: end for
11: Add graphs assigned to leafs of search path to R.
12: return R.
```

The retrieval algorithm solves the exact subgraph matching problem in $O(|V_q|)$, where V_q are the vertices of the query graph Q. Furthermore, with modifications using backtracking techniques it is possible to find a maximum common subgraph (MCS) [31] to realize some kind of inexact graph matching. However, this is not the focus of this paper and might be investigated in future work.

3.4. Proof of Completeness

For the proposed modified algorithm it has to be proven that the algorithm finds all solutions. The algorithm elaborated in the previous section reduces the number of valid permutations. So, it has to be shown that by leaving out permutations, no valid solution is lost.

Let $G = (V, E, L_v, L_e, \mu, \upsilon, \sigma)$ be a well-founded total ordered graph and let $A(G)$ be the set which contains all valid permutations of the graph's adjacency matrices. To be complete, the algorithm must find a solution if one exists; otherwise, it correctly reports that no solution is possible. Thus if every possible valid subgraph $S \subseteq G$, where the vertices of S fulfill the order, every corresponding adjacency matrix M has to be an element of the set $A(G)$, $M \in A(G)$.

For this reason to proof that the algorithm is complete it has to be shown that the algorithm generates all valid subgraphs $S \subseteq G$. Therefore the pseudo code of Algorithm 2 shows how the index is build. Algorithms 4 and 5 are helper functions for calculating all variations of the set of vertices in an interval.

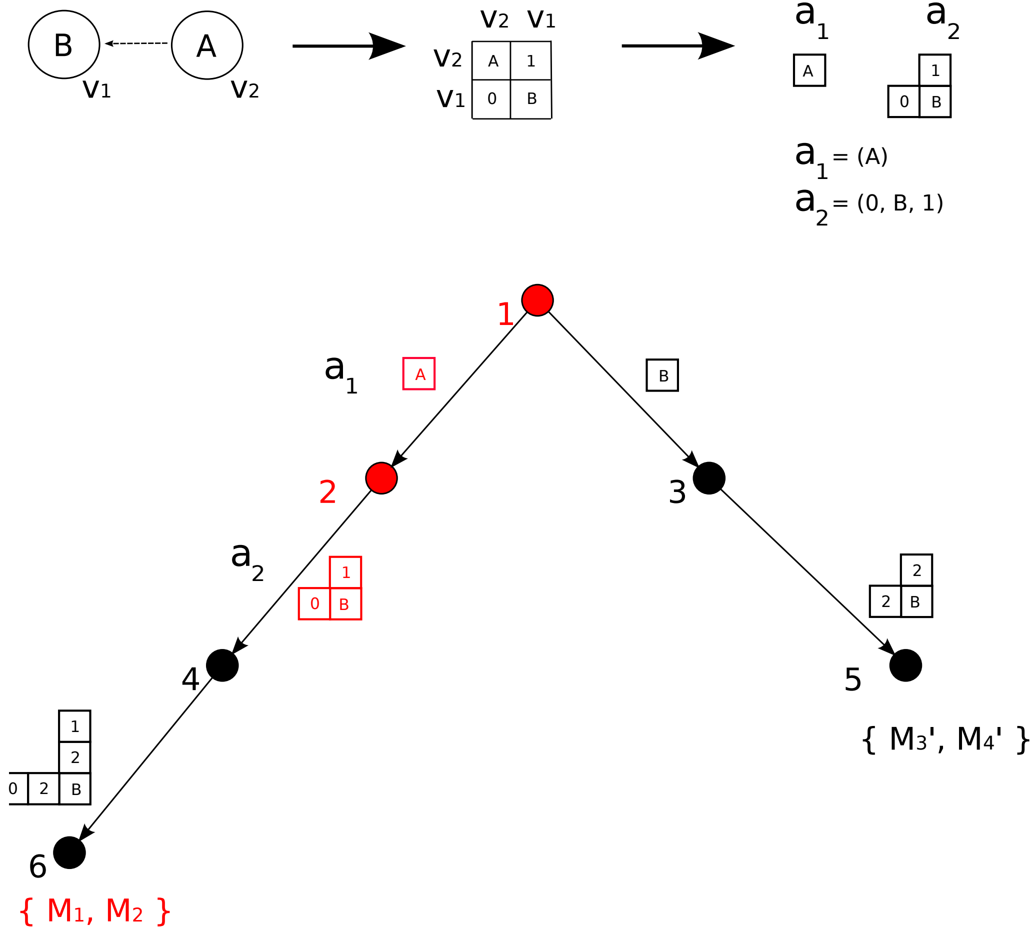


Figure 4: Retrieval using the new decision tree.

Algorithm 2 BUILD_INDEX($G = (V, E, L_v, L_e, \mu, \upsilon, \sigma)$, Tree)

Require: Unsorted set V of vertices, μ labeling function, σ weight function.

1: sort(V, L_v, μ, σ)

Ensure: Vertices V are sorted according the defined order.

2: Let O be an empty list.

3: **for all** $l_i \in L_V$ **do**

4:     Let interval $\{v_a, \ldots, v_b\}$ contain all v with $\mu(v) = l_i$

5:     $O_i \leftarrow VARIATIONS(\{v_a, \ldots, v_b\})$

6: **end for**

7: Let $AG \leftarrow O_1 \times \ldots \times O_{|L_V|}$.

8: **for all** m_i in AG **do**

9:     Add row column vector for sequence of m_i to Tree.

10: **end for**

The generation of the index starts with an unsorted set of vertices. By sorting the vertices with their associated labels using the well-founded total order, the set is ordered according the weights of the labels.

Now, the algorithm iterates over all intervals of vertices $\{v_a, ..., v_b\}$ where the labels have the same weights, $\sigma(\mu(v_a)) = \sigma(\mu(v_b))$. For each interval $\{v_a, \ldots, v_b\}_i$ all variations with respect to the order have to be determined. These variations are computed in Algorithm 3, by determining all combination of the interval $\{v_a, \ldots, v_b\}_i$ including the empty set and calculating all permutations for these combinations. Algorithm 4 and Algorithm 5 realize the algorithm proposed by Rosen [34] which computes all permutations for a defined interval. It has been proven that Rosen's algorithm computes all permutations.

Algorithm 3 VARIATIONS($\{v_a, \ldots, v_b\}$)

Require: Sorted set $V = \{v_a, \ldots, v_b\}$ of vertices, $a \leq b$.
1: Let O be an empty list.
2: Determine all combinations C for $\{v_a, \ldots, v_b\}$ including the empty set.
3: **for all** c in C **do**
4: Call $PERMUTE(c, 0, |c|, O)$.
5: **end for**
6: Return O.

Algorithm 4 $PERMUTE(V, begin, end, R)$

Require: Sorted set V of vertices and $begin < end$, with V_{end-1} being last the element.
1: Adding sequence of vertices V to R.
2: **for** $i \leftarrow end - 2$ to $begin$ **do**
3: **for** $j \leftarrow i + 1$ to $end - 1$ **do**
4: Swapping position i and j in V.
5: Call $PERMUTE(V, i + 1, end, R)$.
6: **end for**
7: Call $ROTATE(V, i + 1, end, R)$.
8: **end for**

Algorithm 5 $ROTATE(V, begin, end, R)$

1: Let $temp \leftarrow V_{end-1}$.
2: Shift elements in V in from position $begin$ to $end - 1$ one position right
3: Set $V_{begin} \leftarrow temp$.
4: Add sequence of vertices V to R.

In combinatorial mathematics, a k-variation of a finite set S is a subset of k distinct elements of S. For each chosen variation of k elements, where k is $L_{interval}$ = length of interval; $k = 1 \ldots L_{interval}$, again all permutations have to be considered.

Now, assuming there would be a valid subgraph $Q = (V', E', L'_v, L'_e, \mu, \upsilon, \sigma)$, respectively the according adjacency matrix A which depends on the alignment of the vertices. To be a valid subgraph according to Definition 3, V' has to be a subset of V, $V' \subseteq V$. Furthermore the

alignment of the vertices V' according to their labels has to fulfill the defined order, $\sigma(\mu(v_i)) \leq \sigma(\mu(v_{i+1}))$. For the alignment the intervals $\{v'_a, \ldots, v'_b\} \in V'$ where the weights of the labels have the same value $\sigma(\mu(v'_a)) = \sigma(\mu(v'_b))$ are important as they can vary. The Algorithm 3 determines all variations for intervals with the same weights for labels, thus the alignment $\{v'_a, \ldots, v'_b\}$ is considered.

This holds for each interval, thus algorithm produces all valid permutations according the well-founded total order. As the query graph Q also has to fulfill the order, its adjacency matrix A will be an element of $A(G)$, if Q is a valid subgraph of G. Thus, the solution will be found in the decision tree.

3.5. Complexity Analysis

The computational complexity analysis discussed in this section will be based on the following quantities:

$$\begin{aligned} N &= \text{the number of graphs in the graph database,} \\ M &= \text{the maximum number of vertices of a graph in the graph database,} \\ M_v &= \text{the number of vertex labels for a graph in the graph database,} \\ I &= \text{the number of vertices in the query graph,} \\ l_v &= \text{the number of vertex labels.} \end{aligned}$$

The original algorithm by Messmer [7] as well as the proposed algorithm need an intensive preprocessing, the compilation of the decision tree. Messmer's method has to compute all permutations of the adjacency matrix of the graph, thus the compilation of the decision tree for a graph $G = (V, E, L_v, L_e, \mu, \upsilon, \sigma)$ has a run time complexity of

$$O(|V|!).$$

For the size of the decision tree Messmer determined the following bounds. The sum of nodes over all the levels (without the root node) is limited to

$$O(l_v \sum_{k=0}^{M-1} \binom{M}{k} (l_2^e)^k) = O(l_v(1 + l_e^2)^M),$$

and as the decision tree becomes linearly dependent on the size of the database N, the space complexity of the decision tree is

$$O(Nl_v(1 + l_e^2)^M).$$

The processing time for the new decision tree compilation algorithm, as Algorithm 2 describes the new algorithm which compiles the decision tree. The basic idea of the algorithm is to take all labels L_v with the same weight which occur in the graph and omit their instances. So, considering the mathematical idea of the algorithm an approximation of the run time complexity would be:

$$\prod_{i=1}^{|L_v|} \left(\underbrace{|\{\forall v \in V | \mu(v) = l_i\}|!}_{n_i} + \sum_{j=1}^{n_i - 1} \binom{n_i}{j} \cdot j! \right).$$

The first term considers the permutations for all labels with the same weight, denoted by n_i. The second term describes the k-variations. As we have to consider the variations from $n_i - 1$ to 1, the sum is sufficient.

In order to simplify the equation, we can combine the two terms, as the $n_i!$ is equivalent to $\binom{n_i}{n_i} \cdot n_i! = 1 \cdot n_i! = n!$, thus we have:

$$\prod_{i=1}^{|L_v|} \left(\sum_{j=1}^{n_i} \binom{n_i}{j} \cdot j! \right).$$

Now, let n_{max} be the maximum number of vertices with the same weight. Then we have the upper bound of

$$\left(\sum_{j=1}^{n_{max}} \binom{n_{max}}{j} \cdot j! \right)^{|L_v|}.$$

A rather imprecise approximation of the sum $\sum_{j=1}^{n_{max}} \binom{n_{max}}{j} \cdot j!$ would be $(n_{max} + 1)!$, so the resulting complexity of the algorithm is

$$O(((n_{max} + 1)!)^{|L_v|})$$

Thus for the worst case - where all vertices have the same label - $n_{max} = |V|$,

$$O((|V| + 1)!)$$

which would be worse than the method proposed by Messmer and the best case - where all vertices have different labels - $n_{max} = 1$

$$O(2^{|V|})$$

To find the average case of the algorithm the distribution of the labels in the graph has to be considered. Thus the average is dependent on the distribution of the data.

4. Experiments

In order to examine the modfied algorithm in practice, we performed experiments on randomly generated graphs and two real world graph datasets taken from [32]. For the first experiment a set of random graphs with average sizes of 50, 75, 100, 125, and 150 have been analysed with each 1000 graphs.

For the second experiment, the AIDS data set was used. It consists of graphs representing molecular compounds and they are constructed graphs from the AIDS Antiviral Screen Database of Active Compounds [9]. The molecules are converted into graphs in a straightforward manner by representing atoms as vertices and the covalent bonds as edges. Nodes are labeled with the number of the corresponding chemical symbol and edges by the valence of the linkage. For the third experiment, the Mutagenicity database was used. It is one of the numerous adverse properties of a compound that hampers its potential to become a marketable drug [10]. In order to convert molecular compounds of the Mutagenicity data set into attributed graphs the same procedure as for the AIDS data set is applied. For both databases we choose the chemical symbol as the label for the vertex and the valence as the label for the edge. For the experiments we have just taken a subset of the database where the number of vertices are less than 30 in a graph. So, from the AIDS database we used 1781 graphs and from the Mutagenicity database 2681 graphs.

Table 1: Run time for compiling the decision tree for each graph

	Vertices #	**Same labels (max.)**	**Run time (minutes)**	**Per. (modfied)**	**Per. (original)**
1	10	4	1.32×10^{-3}	1.04×10^{4}	3.63×10^{6}
2	11	6	2.62×10^{-2}	1.25×10^{5}	3.99×10^{7}
3	12	8	4.75	3.51×10^{6}	4.79×10^{8}
4	11	9	3.26×10^{2}	3.95×10^{6}	3.99×10^{7}
⋮	⋮	⋮	⋮	⋮	
135	12	8	1.86×10^{1}	7.12×10^{6}	4.79×10^{8}
⌀	9.44	5.01	4.67	3.54×10^{5}	9.3×10^{7}

During these experiments we only compute the permutations needed for the original algorithm and for the proposed modified version, since this is the main difference of the two algorithms. The permutations for the modified algorithm have been determined according to the algorithm discussed in Section 3 and the formula in Section 3.5:

$$\prod_{i=1}^{|L_v|} \left(\sum_{j=1}^{n_i} \binom{n_i}{j} \cdot j! \right).$$

The original algorithm has to be calculate the permutations for all vertices ($|V|!$ permutations).

In the fourth experiment the time has been measured to add a graph to the decision tree. As this experiment is quite time-consuming on a desktop machine (Intel® Core$^{\text{TM}}$ 2 Quad CPU Q9550 @ 2.83 GHz), only the performance for 135 graphs with less than 12 vertices has been measured. The graphs have been taken from the AIDS database and the algorithm has been implemented single threaded in unoptimized Java code [1]. The results of the experiment are listed in Tab. 1.

Further experiments show that the algorithm also significantly reduces the number of permutations for random graphs (see Tab. 2). For average size of graphs we have tested the algorithm on three sets: The first set of graphs only has different labels; the second set has a limited number of vertices having the same label; and the labels of the last set were more randomly distributed. While the number of permutations is significantly reduced, with the current state of the algorithm still to many permutations are needed for larger graphs. A possible solution is to find a way to split larger graph into smaller subgraphs and then using the subgraphs for indexing without losing the completeness. Noteworthy, these experiments have just been performed to measure the hypothetical performance of the algorithm. In the real world data set used below, which fulfill conditions like having many different label categories, the proposed algorithm can be used for the exact subgraph isomorphism search for graphs with more than 19 vertices.

Also for the real world data sets we have observed an improvement (see Tab. 3 and Tab. 4). As the vertices of the molecules in the AIDS database mainly had the same labels - in average 7 out of 11 vertices - there was less improvement than on the Mutagenicity database, where only 10 out of 20 vertices had the same label in average. The average number of permutations needed

[1]The Java implementation and the IAM Graph Database: AIDS and Mutagenicity are available at: `http://www.dfki.uni-kl.de/~m_weber/subgraph-matching`

has been reduced from 6.1×10^{29} to 2.3×10^{24} on the AIDS database and from 1.7×10^{31} to 1.5×10^{23} on the Mutagenicity database, respectively.

The performance experiment has shown that even on a desktop machine with unoptimized, single-threaded Java code graphs with up to 12 vertices can be handled without any problem. Note that the time needed to compile the decision tree is still quite long (even for small problem instances) as shown in Tab. 1. However, as the method is designed for an off-line preprocessing, i.e., it would only need to be applied once and then several search operations can be applied very fast. Furthermore, the compilation could run on a server machine. Running this system on a multi-core server with optimized parallel code would significantly reduce the compilation time.

5. Conclusion

In this paper an extension for the method of Messmer's subgraph matching has been proposed. The original method is very efficient to perform exact subgraph matching on a large database. However, it has a limitation for the maximum number of vertices. The modification discussed in this paper enables to increase this limit depending on how the vertices are labeled. As the number of permutations in the preprocessing step depends on the vertices with the same labels, an analysis of the data that will be represented in graph is necessary. If there are just a few vertices with the same label, e.g. less than five, even graphs with 30 vertices can be handled. It has been proven that the modification of the method does not affect its completeness.

Noteworthy, the proposed method can be applied in several areas, such as object recognition, matching of 2D or 3D chemical structures, and architectural floor plan retrieval (room connectivity graphs). Future work will be to research strategies for choosing appropriate weight functions. Furthermore, we plan to extend this method to provide a fast method for error-tolerant graph matching and investigate strategies to split larger graph into smaller subgraphs without losing the completeness of the method. These smaller subgraphs will then be used for indexing.

References

[1] I. Schomburg, A. Chang, C. Ebeling, M. Gremse, C. Heldt, G. Huhn, D. Schomburg, BRENDA, the enzyme database: updates and major new developments, NUCLEIC ACIDS RESEARCH 32 (Sp. Iss. SI).

[2] W. Kim, A. Kak, 3-d object recognition using bipartite matching embedded in discrete relaxation, IEEE Transactions on Pattern Analysis and Machine Intelligence 13 (1991) 224–251.

[3] E. K. Wong, Model matching in robot vision by subgraph isomorphism, Pattern Recogn. 25 (1992) 287–303.

[4] J. Cheng, T. Huang, Image registration by matching relational structures, Pattern Recognition 17 (1) (1984) 149 – 159.

[5] H. Bunke, B. Messmer, Efficient attributed graph matching and its application to image analysis, 1995, pp. 44–55.

[6] M. Weber, C. Langenhan, T. Roth-Berghofer, M. Liwicki, A. Dengel, F. Petzold, a.SCatch: Semantic Structure for Architectural Floor Plan Retrieval, in: Advances in Case-Based Reasoning, Proc. of ICCBR 2010, 2010.

[7] B. Messmer, H. Bunke, A decision tree approach to graph and subgraph isomorphism detection, Pattern Recognition 32 (1999) 1979–1998.

[8] M. Weber, M. Liwicki, A. Dengel, Indexing with Well-Founded Total Order for Faster Subgraph Isomorphism Detection, in: X. Jiang, M. Ferrer, A. Torsello (Eds.), Graph-Based Representations in Pattern Recognition, Springer, 2011, pp. 185–194.

[9] Development Therapeutics Program (DTP), AIDS antiviral screen, `http://dtp.nci.nih.gov/docs/aids/aids_data.html` (2004).

[10] J. Kazius, R. McGuire, R. Bursi, Derivation and validation of toxicophores for mutagenicity prediction, Journal of Medicinal Chemistry 48 (1) (2005) 312–320.

[11] D. Conte, P. Foggia, C. Sansone, M. Vento, Thirty years of graph matching in pattern recognition, IJPRAI 18 (3) (2004) 265–298.

[12] X. Gao, B. Xiao, D. Tao, X. Li, A survey of graph edit distance, Pattern Analysis and Applications 13 (1) (2009) 113–129. doi:10.1007/s10044-008-0141-y.

[13] J. Ullmann, An algorithm for subgraph isomorphism, Journal of the ACM (JACM) 23 (I) (1976) 31–42.

[14] R. Giugno, 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. doi:10.1109/ICPR.2002.1048250.

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

[16] J. Cheng, Y. Ke, W. Ng, A. Lu, Fg-index: towards verification-free query processing on graph databases, in: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, SIGMOD '07, ACM, New York, NY, USA, 2007, pp. 857–872.

[17] X. Yan, P. S. Yu, J. Han, Graph indexing: a frequent structure-based approach, in: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, SIGMOD '04, ACM, New York, NY, USA, 2004, pp. 335–346.

[18] H. Jiang, H. Wang, P. S. Yu, S. Zhou, Gstring: A novel approach for efficient search in graph databases, in: In ICDE, 2007, pp. 566–575.

[19] D. W. Williams, J. Huan, W. Wang, Graph database indexing using structured graph decomposition, in: In ICDE, 2007.

[20] Y. Tian, R. C. Mceachin, C. Santos, D. J. States, J. M. Patel, Saga: a subgraph matching tool for biological graphs, Bioinformatics 23 (2007) 232–239.

[21] H. Jamil, Computing subgraph isomorphic queries using structural unification and minimum graph structures, in: Proceedings of the 2011 ACM Symposium on Applied Computing, SAC '11, ACM, New York, NY, USA, 2011, pp. 1053–1058.

[22] H. Bunke, Graph matching: Theoretical foundations, algorithms, and applications, Vol. 23, 2000, pp. 82–88.

[23] H. Bunke, Recent developments in graph matching, Pattern Recognition, International Conference on 2 (2000) 2117.

[24] B. J. Jain, F. Wysotzki, Solving inexact graph isomorphism problems using neural networks, Neurocomput. 63 (2005) 45–67. doi:http://dx.doi.org/10.1016/j.neucom.2004.01.189.
URL `http://dx.doi.org/10.1016/j.neucom.2004.01.189`

[25] L. Xu, E. Oja, Improved simulated annealing, boltzmann machine, and attributed graph matching, in: Proceedings of the EURASIP Workshop 1990 on Neural Networks, Springer-Verlag, London, UK, 1990, pp. 151–160.
URL `http://dl.acm.org/citation.cfm?id=646655.700099`

[26] A. D. J. Cross, R. C. Wilson, E. R. Hancock, Genetic search for structural matching, in: Proceedings of the 4th European Conference on Computer Vision-Volume I - Volume I, ECCV '96, Springer-Verlag, London, UK, 1996, pp. 514–525.
URL `http://dl.acm.org/citation.cfm?id=645309.648894`

[27] Y.-K. Wang, K. chin Fan, J. tzong Horng, Genetic-based search for error-correcting graph isomorphism, IEEE Transactions on Systems, Man, and Cybernetics: Part B - Cybernetics 27 (1997) 588–597.

[28] S. Umeyama, An eigendecomposition approach to weighted graph matching problems, IEEE Trans. Pattern Anal. Mach. Intell. 10 (1988) 695–703. doi:http://dx.doi.org/10.1109/34.6778.
URL `http://dx.doi.org/10.1109/34.6778`

[29] H. A. Almohamad, S. O. Duffuaa, A linear programming approach for the weighted graph matching problem, IEEE Trans. Pattern Anal. Mach. Intell. 15 (1993) 522–525. doi:10.1109/34.211474.
URL `http://dl.acm.org/citation.cfm?id=628301.628477`

[30] H. He, A. 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, ACM, New York, NY, USA, 2008, pp. 405–418.

[31] H. Bunke, On a relation between graph edit distance and maximum common subgraph, Pattern Recognition Letters 18 (8) (1997) 689 – 694.

[32] K. Riesen, H. Bunke, Iam graph database repository for graph based pattern recognition and machine learning, in: Proceedings of the 2008 Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition, SSPR & SPR '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 287–297.

[33] H. Bunke, K. Shearer, A graph distance metric based on the maximal common subgraph, Pattern recognition letters (1998) 255–259.

[34] K. H. Rosen, Discrete mathematics and its applications (2nd ed.), McGraw-Hill, Inc., New York, NY, USA, 1991.

Table 2: Results of random graph experiments.

Exp.	Same labels (max.)		Vertices #	Perm. (modified)	Perm. (original)	Diff. labels #
1	1	∅	50.08	5.7×10^{16}	7.2×10^{78}	50.08
2	5	∅	49.99	2.0×10^{28}	8.2×10^{78}	10.60
3	10	∅	50.04	4.7×10^{39}	6.4×10^{78}	5.56
4	1	∅	75.15	2.0×10^{24}	1.6×10^{125}	75.15
5	5	∅	75.20	7.5×10^{40}	1.9×10^{125}	15.64
6	10	∅	75.10	1.1×10^{56}	2.2×10^{125}	8.05
7	1	∅	100.10	6.6×10^{31}	7.4×10^{174}	100.11
8	5	∅	99.97	2.8×10^{53}	8.4×10^{174}	20.59
9	10	∅	100.05	4.2×10^{74}	6.4×10^{174}	10.57
10	1	∅	125.35	2.4×10^{39}	1.1×10^{227}	125.35
11	5	∅	124.55	7.0×10^{65}	7.4×10^{226}	25.52
12	10	∅	124.84	7.1×10^{90}	8.8×10^{226}	13.03
13	1	∅	150.15	8.1×10^{46}	1.7×10^{281}	150.15
14	5	∅	149.71	3.2×10^{78}	1.4×10^{281}	30.54
15	10	∅	150.00	4.5×10^{109}	1.5×10^{281}	15.55

Table 3: Results of AIDS graph database experiments.

Graph	Vertices #	Permutations (modified)	Permutations (original)	Same labels (max.)
1	12	3.51×10^{6}	4.79×10^{8}	8
2	30	1.52×10^{20}	2.65×10^{32}	14
3	9	8.45×10^{3}	3.63×10^{5}	4
4	22	2.31×10^{13}	1.12×10^{21}	11
⋮	⋮	⋮	⋮	⋮
1781	11	3.43×10^{5}	3.99×10^{7}	7
∅	11.39	2.3×10^{24}	6.1×10^{29}	7.14

Table 4: Results of Mutagenicity graph database experiments.

Graph	Vertices #	Permutations (modified)	Permutations (original)	Same labels (max.)
1	28	1.54×10^{21}	3.05×10^{29}	15
2	19	1.72×10^{10}	1.22×10^{17}	8
3	23	2.78×10^{16}	2.59×10^{22}	15
4	14	6.38×10^{6}	8.72×10^{10}	15
⋮	⋮	⋮	⋮	⋮
2682	26	1.39×10^{17}	4.03×10^{26}	11
∅	20.7	1.5×10^{23}	1.7×10^{31}	10.06