Subgraph Isomorphism in Polynomial Time

B.T. Messmer and H. Bunke
Institut für Informatik und angewandte Mathematik,
University of Bern, Neubrückstr. 10, Bern, Switzerland

Abstract

In this paper, we propose a new approach to the problem of subgraph isomorphism detection. The new method is designed for systems which differentiate between graphs that are a priori known, so-called *model graphs*, and unknown graphs, so-called *input graphs*. The problem to be solved is to find a subgraph isomorphism from an input graph, which is given on-line, to any of the model graphs. The new method is based on an intensive preprocessing step in which the model graphs are used to create a *decision tree*. At run time, the input graph is then classified by the decision tree and all model graphs for which there exists a subgraph isomorphism from the input graph are detected. If we neglect the time needed for preprocessing, the computational complexity of the new subgraph isomorphism algorithm is only quadratic in the number of input graph vertices. Furthermore, it is independent of the number of model graphs and the number of edges in any of the graphs. However, the decision tree that is constructed in the preprocessing step may grow exponentially with the number of vertices of the model graphs. Therefore, we present several pruning techniques which aim at reducing the size of the decision tree. A computational complexity analysis of the new method is given. Also, the advantages and disadvantages of the new algorithm are studied in a number of practical experiments with randomly generated graphs. Finally, the application of the algorithm in a graphic symbol recognition system is briefly discussed.

1 Introduction

Graph and subgraph isomorphism are concepts that have been intensively used in various applications. The representational power of graphs and the need to compare different graphs with each other have led numerous researchers in the past twenty years to study the problem of efficiently computing graph and subgraph isomorphisms. As of today, it is still an open question whether the general graph isomorphism problem can be solved by an algorithm that is only polynomial in time and in space [GJ79]. All algorithms that have been proposed in the literature so far have an exponential time complexity in the worst case. Also, the problem of subgraph isomorphism detection is known to be NP-complete [GJ79]. In the following, we will give a brief overview of graph and subgraph isomorphism algorithms.

There are two basic approaches that past research has taken towards the problem of graph isomorphism. The first approach is based on group-theoretic concepts and the study of permutation groups. In [Bab81], it was shown that there exists a moderately exponential bound for the general graph isomorphism problem. Furthermore, by imposing certain restrictions on the properties of the graphs, it was possible to derive algorithms that have a polynomially bounded complexity. For example, Luks and Hoffman describe a polynomially bounded method for the isomorphism detection of graphs with bounded valence [Hof82]. For the special case of trivalent graph isomorphism, it was show in [Luk82] that algorithms with a computational complexity of $O(n^6)$ exist. In [HW74] a method for the computation of the isomorphism of planar graphs is proposed that has only a linear time complexity. However, the major drawback of algorithms based on group-theoretic concepts is the fact that there is usually a large overhead and consequently a large constant factor associated with the theoretical complexity. The second approach to graph and subgraph isomorphism is more practically oriented and aims directly at developing an algorithmic procedure for isomorphism detection. Most of these algorithms are based on a state-space search with backtracking. One of the first publications in this field is the one by Corneil and Gotlieb [CG70]. A major improvement of the backtracking method was then presented by Ullman, who introduced a refinement method which reduces the search space of the backtracking procedure remarkably [Ull76]. For an overview of the publications on graph isomorphism see [Gat79, RC77]. More recent work is described in [MLL92, FFG90] where the graph isomorphism problem was reduced to the problem of clique detection by constructing an association graph for all the possible vertex mappings. And most recently, a network based approach to graph matching has been proposed by the authors [MB95b].

So far, we have only considered the problem of finding a graph or subgraph isomorphism between two graphs at a time. However, in practical applications there is often a database of graphs, so-called model graphs, and a single unknown input graph that must be tested. If the number of graphs in the database is large then the sequential testing of each model graph becomes computationally very costly. As a consequence, several systems have been proposed in the past which combine graph or subgraph isomorphism algorithms with indexing methods. The basic idea of indexing is to use specific and easily computable features of an input graph in order to select a small set of model graphs out of a large database. In [HS88], Horaud proposes to use the second immanantal polynomial of the Laplacian matrix of a graph as an index into the database of graphs. However, this index is only unique for graphs with less than 12 vertices and its computation requires $O(n^4)$ steps, where n indicates the number of vertices in the graph. Another approach is taken by Paris in [Par93] where it is proposed to calculate a structural index that is organized in a hierarchical classification network similar to the network-based graph matching described in [MB95b]. Of high interest to the present paper are the indexing approaches that are proposed in [Ike87, GB89, Spi93]. Instead of using an indexing mechanism as a preprocessor to some conventional subgraph isomorphism algorithm, the database of graphs is transformed into a decision tree. The decision tree is then used to directly and simultaneously index and match the model graphs with the input graph. However, all of the decision tree approaches that have been presented so far are strongly connected to 3D-object recognition and offer no solution to the general graph isomorphism problem.

In this paper, we propose a new method for graph and subgraph isomorphism detection. It has two important features. First, its run time is only quadratic in the number of vertices in the input graph if we neglect the time needed for preprocessing. Secondly, the time complexity is independent of the number of graphs in the database. The new method is based on the following idea. We generate the set of all permutations of the adjacency matrix of a model graph and organize this set in a decision tree. Different model graphs can be combined into the same decision tree. The decision tree is built from the model graphs in an off-line preprocessing step. At run time, it is used to efficiently determine if there is a subgraph isomorphism from an unknown input graph to one of the model graphs. The main advantage of the new method is that it is guarantueed to terminate in quadratic time. However, the trade-off for the efficient run time is the size of the decision tree. It contains, in the worst case, an exponential number of nodes. Nonetheless, we believe that the proposed method is a new contribution to the field of subgraph isomorphism detection. It is of particular interest in applications where the underlying graphs are rather small, but where almost real time behavior is required.

The rest of this paper is organized as follows. In Section 2, the basic definitions and notations are given. In Section 3, one of the standard algorithms for subgraph isomorphism is briefly described for comparison reasons. The basic idea and an overview of the new algorithm is then presented in Section 4. A more detailed description of the procedures and data structures of the new algorithm is given in Sections 5 and 6. The results of a computational complexity analysis are described in Section 7. Because of the exponential size, it is necessary to consider techniques for the reduction of the number of nodes in the decision tree. These pruning techniques are presented in Section 8. The practical performance of the new algorithm is studied in a number of experiments that are documented in Section 9. Section 10 is dedicated to the description of a prototype application in document image analysis that incorporates the new algorithm. Finally, in Section 11, a summary of the results and a discussion of future research directions conclude the paper.

2 Definitions and Notations

In this section we give the basic definitions and notations that will be used throughout the paper.

Definition 2.1: A *labeled graph* G is a 6-tuple, $G = (V, E, \mu, \nu, L_v, L_e)$, where

- V is the set of vertices,
- $E \subseteq V \times V$ is the set of edges,
- $\mu : V \rightarrow L_v$ is a function assigning labels to the vertices,
- $\nu : E \rightarrow L_e$ is a function assigning labels to the edges.

$\square$

As usual, we assume that L_v and L_e are finite sets of symbolic labels. Note that the above definition corresponds to the case of directed graphs. Undirected graphs are obtained if we require for each edge (v_1, v_2) an edge (v_2, v_1) in the opposite direction with the same label.

Definition 2.2: Given a graph $G = (V, E, \mu, \nu, L_v, L_e)$, a *subgraph* of G is a graph $S = (V_s, E_s, \mu_s, \nu_s, L_v, L_e)$ such that

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

□

Let $G = (V, E, \mu, \nu, L_v, L_e)$ be a graph with $V = \{v_1, v_2, \dots, v_n\}$. Then G can also be represented by its adjacency matrix $M = (m_{ij}), i, j = 1, \dots, n$, where $m_{ii} = \mu(v_i)$ and $m_{ij} = \nu((v_i, v_j))$ for $i \neq j$. Apparently, the adjacency matrix representation of a graph doesn't take into account loops at a vertex. However, this isn't a real restriction as loops can be represented by means of an extended set of vertex labels.

Clearly, 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 2.3: A $n \times n$-matrix $P = (p_{ij})$ is called a *permutation matrix* if

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

□

If a graph G is represented by an $n \times n$ adjacency matrix M and P is an $n \times n$ permutation matrix, then the $n \times n$ matrix

$$M' = PMP^T \tag{1}$$

where P^T denotes the transpose of P, is also an adjacency matrix of G. If $p_{ij} = 1$ then the j-th vertex in M becomes the i-th vertex in M'.

Definition 2.4: Let G_1 and G_2 be two graphs and M_1 and M_2 their corresponding adjaceny matrices. G_1 and G_2 are *isomorphic* if there exists a permutation matrix P such that

$$M_2 = PM_1P^T \tag{2}$$

□

Notice that the matrix P can be understood as a bijective function f that maps the vertices of G_1 to G_2, and vice versa. That is, $f(v_j) = v_i$ iff $m_{ij} = 1$. We will call both P and f a *graph isomorphism* between G_1 and G_2. Thus, the problem of finding graph isomorphisms between two graphs G_1 and G_2 is equivalent to finding a permutation matrix P for which (2) holds true.

Definition 2.5: Given two graphs G_1 and G_2, there is a *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. □

Definition 2.6: Let $M = (m_{ij})$ be a $n \times n$ matrix. Then $S_{k,m}(M)$ denotes the $k \times m$ matrix that is obtained from M by deleting rows $k+1, \ldots, n$ and columns $m+1, \ldots, n$, where $k, m \leq n$. That is, $S_{k,m}(M) = (m_{ij})$; $i = 1, \ldots, k$ and $j = 1, \ldots, m$. □

Using the notation introduced in the last definition, not only the concept of graph isomorphism but also subgraph isomorphism can be described in terms of adjacency matrices. Let G_1 and G_2 be graphs with adjacency matrices M_1 and M_2 of dimension $m \times m$ and $n \times n$ respectively, where $m \leq n$. There is a subgraph isomorphism from G_1 to G_2 iff there is a $n \times n$ permutation matrix P such that

$$M_1 = S_{m,m}(PM_2P^T). \tag{3}$$

Thus, the problem of finding a subgraph isomorphism from G_1 to G_2 is equivalent to finding a permutation matrix P for which (3) holds. Notice that $S_{m,m}(PM_2P^T) = S_{m,n}(P)M_2(S_{m,n}(P))^T$.

3 A Brief Review of Ullman's Algorithm

For comparison reasons, we briefly review Ullman's method [Ull76]. It can be applied to both graph and subgraph isomorphism detection. The method is based on backtracking and a refinement procedure. Out of all methods that have been mentioned in the introduction, Ullman's method is considered one of the fastest algorithm for the subgraph isomorphism problem.

The input to the algorithm consists of a model graph $G = (V, E, \mu, \nu, L_v, L_e)$ and an input graph $G_I = (V_I, E_I, \mu_I, \nu_I, L_v, L_e)$. Let M denote the $n \times n$-adjacency matrix of G and M_I the $m \times m$-adjacency matrix of G_I. We intend to find all permutation matrices P such that $M_I = S_{m,m}(PM_IP^T)$. Note that $S_{i,n}(P)$ is an $i \times n$ permutation matrix that represents a *partial matching* from the first i vertices of G onto some vertices of G_I. If $S_{i,i}(M_I) = S_{i,n}(P)M(S_{i,n}(P))^T$ then clearly $S_{i,n}(P)$ represents a graph isomorphism from the subgraph of G_I that consists of the vertices 1 to i to some subgraph of G.

Ullman's algorithm is based on the idea of finding all subgraph isomorphisms by gradually setting the permutation matrix P row by row (see Fig. 1). From Def. 2.3 in the previous section, we know that each row k in P contains exactly one non-zero entry $p_{ki} = 1$, while all other elements p_{kj} of the row k with $j \neq i$ are set to 0. The recursive procedure *Backtrack* begins by setting the first element p_{11} of the top row of P to 1 and all other elements in the top row of P to 0. If $S_{1,n}(P)$ is a partial matching that represent a

ULLMAN$(G = (V, E, \mu, \nu, L_v, L_e),\ G_I = (V_I, E_I, \mu_I, \nu_I, L_v, L_e))$

1. Let $P = (p_{ij})$ be a $n \times n$ permutation matrix, $n = |V|$, $m = |V_I|$ and M and M_I denote the adjacency matrices of G and G_I, respectively.
2. call Backtrack$(M, M_I, P, 1)$
3. procedure Backtrack(adjacency matrix M, adjacency matrix M_I, permutation matrix P, counter k)
   - (a) if $k > m$ then P represents a subgraph isomorphism from G_I to G. Output P and return.
   - (b) for all $i = 1$ to n
     - i. set $p_{ki} = 1$ and for all $j \neq i$ set $p_{kj} = 0$
     - ii. if $S_{k,k}(M_I) = S_{k,n}(P)M(S_{k,n}(P))^T$ then call Backtrack$(M, M_I, P, k+1)$

Figure 1: Algorithm *Ullman*.

subgraph isomorphism, then the procedure *Backtrack* is recursively called again and the second row of P is tentatively set. This process is continued until either m rows of P have been succesfully set and a subgraph isomorphism is found or the condition in step (3.b.ii) is not satisfied. In both cases, the procedure backtracks to the previous level and tries another setting of p_{ki}.

It is easy to see that this algorithm finds all subgraph isomorphisms from G to G_I and outputs all permutation matrices P which satisfy $M_I = S_{m,n}(P)M(S_{m,n}(P))^T$ (or $M_I = S_{m,m}(PMP^T)$). Furthermore, if G_I and G are of equal size, i.e. $m = n$, the algorithm finds all graph isomorphisms between G_I and G and outputs all permutation matrices P which satisfy $M_I = PMP^T$. For the practical experiments that are documented in Section 8, we implemented the backtracking procedure along with the refinement steps described by Ullman [Ull76]. The refinement procedure is based on the idea of forward checking whether the assignment $p_{ki} = 1$ in step (3.b.i) is locally consistent with at least one entry $p_{lj} = 1$ for each row $m \geq l > k$ in the future search process. If the refinement procedure reveals that setting $p_{ki} = 1$, i.e. mapping vertex i of G_I onto vertex k of G, will not lead to a graph isomorphism, the search may directly continue with the next column $i + 1$ and the setting $p_{k(i+1)} = 1$. Thus, it is possible to avoid unnecessary recursive calls.

It is important to note that the backtracking procedure can only be applied to two graphs at a time. If more than one model graphs are involved, it has to be called for each of them individually. Therefore, the complexity of the subgraph isomorphism detection algorithm based on backtracking is linearly dependent on the number of model graphs.

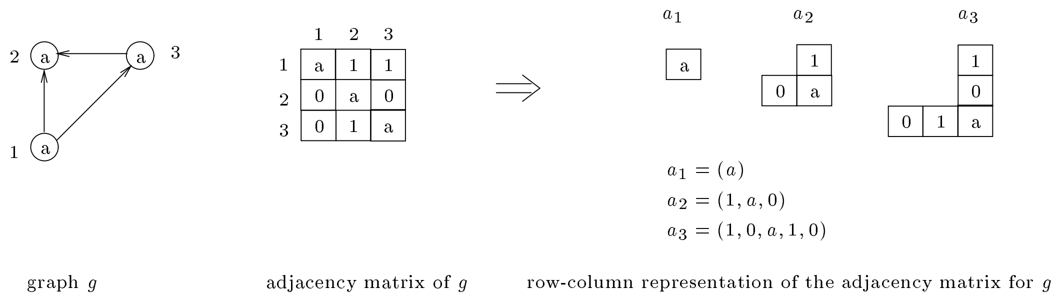


Figure 2: The row-column representation of an adjacency matrix.

4 Subgraph Isomorphism by Means of Decision Trees

The main problem of the algorithm described in the previous section lies in the fact that all permutation matrices which represent a subgraph isomorphism are calculated and generated at run time. Furthermore, the algorithm may run into dead ends and backtracking becomes necessary. In order to overcome these problems and to avoid backtracking at run time, we now propose a decision tree based approach. We assume that there is a set of model graphs that are known a priori, while the input graph becomes accessible at run time only. For each model graph we compute all possible permutations of its adjacency matrix and transform these adjacency matrices into a decision tree. At run time, the matrix of the input graph is then used to find those adjacency matrices in the decision tree, that are identical to it. The permutation matrices that correspond to these adjacency matrices represent the graph or subgraph isomorphisms that we are looking for.

Let $G = (V, E, \mu, \nu, L_v, L_e)$ be a model graph and M its corresponding $n \times n$-adjacency matrix. Furthermore, let $A(G)$ denote the set of all permuted adjacency matrices of G,

$$A(G) = \{M_P | M_P = PMP^T \text{ where } P \text{ is a } n \times n \text{ permutation matrix}\} \tag{4}$$

The total number of permuted adjacency matrices is $|A(G)| = n!$ as there are $n!$ different permutation matrices of dimension n. We are now ready to restate the subgraph isomorphism problem in terms of the set introduced above. For a model graph G with corresponding $n \times n$-adjacency matrix M and an input graph G_I with an $m \times m$-adjacency matrix M_I and $m \leq n$, determine whether there exists a matrix $M_P \in A(G)$ such that $M_I = S_{m,m}(M_P)$. If such a matrix M_P exists, the permutation matrix P corresponding to M_P describes a subgraph isomorphism from G_I to G, i.e. $M_I = S_{m,m}(M_P) = S_{m,m}(PMP^T)$. If G and G_I are of equal size, the permutation matrix P represents a graph isomorphism between G_I and G, i.e. $M_I = PMP^T$.

We propose to organize the set $A(G)$ in a decision tree such that each matrix in $A(G)$ is classified by the tree. The features that will be used for the classification process are the individual elements in the adjacency matrices. However, it is important to note that the purpose of the decision tree will be to classify adjacency matrices of input graphs. In case of subgraph isomorphism detection, these matrices will be smaller than the matrices in $A(G)$. Therefore, it is necessary to group the individual elements of the adjacency matrices in $A(G)$ into features such that the classification on each level of the decision tree is independent of the size of the adjacency matrix that is to be classified. For this

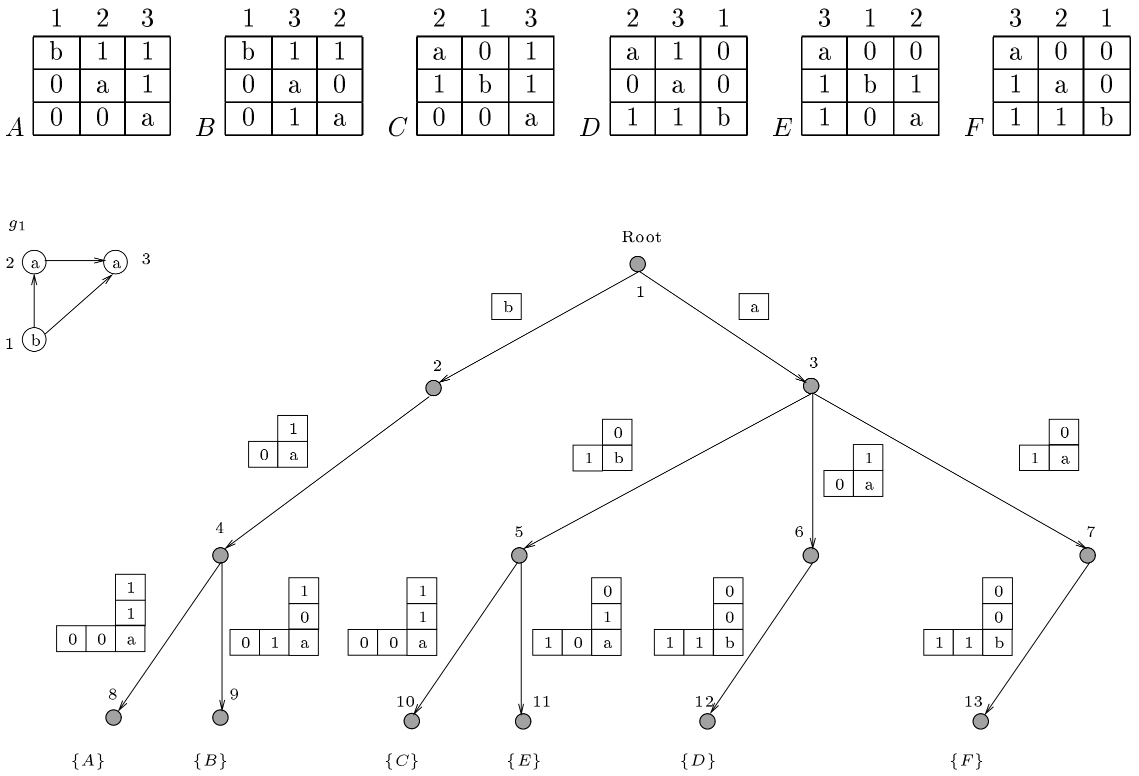


Figure 3: Decision tree for the classification of the adjacency matrices $A \dots F$ of the graph g_1.

purpose, we introduce a new notation for an $n \times n$ adjacency matrix $M = (m_{ij})$. We say that the matrix consists of an array of so-called *row-column* elements a_i, where each a_i is a vector of the form

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

The matrix can then be written as

$$M = (a_1, a_2, \dots, a_n); \quad i = 1, \dots, n$$

Fig. 2 illustrates the structure of an adjacency matrix M with regard to its row-column elements.

The decision tree is now built according to the row-column elements of each adjacency matrix $M_P \in A(G)$. At the top of the decision tree there is a single root node. The direct successor nodes of the root node constitute the first level of the decision tree. On the first level, the classification of the matrices in $A(G)$ is done according to the first row-column element a_1 of each matrix $M_P \in A(G)$. The element $a_1 = (m_{11})$ represents the label of the first vertex in each matrix in $A(G)$, with $m_{11} \in L_v$ (see Section 2). Consequently, each matrix in $A(G)$ is classified according to its first vertex label. Each branch that leads to a direct successor node of the root node is associated with a specific value for the row-column element a_1. Next, on the second level of the decision tree, the second row-column element a_2 of each matrix is used for the classification, and so on. In general, the matrices that are represented by some node on the level k are divided into classes according to the

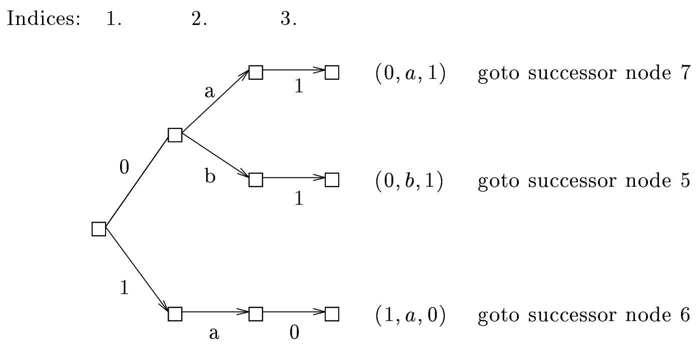


Figure 4: Dictionary and index for the decision tree node 3 in Fig. 3.

element a_k. With each matrix M_P that is represented by some node N on the level k, the corresponding permutation matrix P is also given. As M_P has been classified up to the k-th vertex, P describes a subgraph isomorphism for the subgraph with adjacency matrix $S_{k,k}(M_P)$ to G. At run time, P will describe a subgraph isomorphism for any input graph that has been classified into the node N. Finally, at the bottom of the decision tree, there are the leaf nodes. Each leaf node represents a class of identical matrices $M_P \in A(G)$. For each of these matrices, the corresponding permutation matrix is stored in the leaf node. The number of these permutation matrices in each leaf node is equal to the number of automorphisms of G. (An automorphism is an isomorphism of a graph to itself.)

In Fig. 3 a graph, g_1, and its corresponding decision tree is shown. The nodes of the decision tree are represented by shaded circles. Each directed branch from one node to another has associated with it a row-column element. At the top of Fig. 3 the set $A(g_1)$ of permuted adjacency matrices of g_1 is listed. It is easy to see that the number of automorphisms of g_1 is one. Therefore, each leaf node in the decision tree represents exactly one adjacency matrix.

An important requirement for a decision tree is that the classification on each level must be easily computable. Therefore, if a matrix M_P is to be classified according to the k-th row-column element a_k, the successor which is reached via an element a_{k_i} with $a_{k_i} = a_k$ must be easily computable. For this purpose, all row-column elements that are associated to the branches pointing from a node on level k to a node on level $k+1$ are collected in a dictionary of strings. This dictionary is organized as an index structure with $2k-1$ indices. There are exactly $2k-1$ elements m_{ij} in a row-column element a_k and each of these elements is used as an index. Looking up an element in this dictionary can be done in $2k-1$ steps. Thus finding the successor node in the decision tree at level k can be done in $O(2k-1) = O(k)$ steps. For an example, consider the node 3 in the decision tree in Fig. 3. There are three branches leading from node 3 to nodes 5, 6, and 7, respectively. The row-column elements associated with these branches are organized in a dictionary that is given in Fig. 4. In this example, a three-level index structure is required for the organization of the dictionary entries.

So far we have only discussed the structure of the decision tree with regard to a single model graph. If there are several model graphs in a database then the most trivial solution would be to build a decision tree individually for each model graph. However, it is possible to represent several model graphs by the same decision tree. On each level, the

$$A' \begin{array}{ccc} 1 & 2 & 3 \\ \hline a & 1 & 0 \\ 0 & a & 1 \\ 1 & 0 & a \end{array} \quad B' \begin{array}{ccc} 1 & 3 & 2 \\ \hline a & 0 & 1 \\ 1 & a & 0 \\ 0 & 1 & a \end{array} \quad C' \begin{array}{ccc} 2 & 1 & 3 \\ \hline a & 0 & 1 \\ 1 & a & 0 \\ 0 & 1 & a \end{array} \quad D' \begin{array}{ccc} 2 & 3 & 1 \\ \hline a & 1 & 0 \\ 0 & a & 1 \\ 1 & 0 & a \end{array} \quad E' \begin{array}{ccc} 3 & 1 & 2 \\ \hline a & 1 & 0 \\ 0 & a & 1 \\ 1 & 0 & a \end{array} \quad F' \begin{array}{ccc} 3 & 2 & 1 \\ \hline a & 0 & 1 \\ 1 & a & 0 \\ 0 & 1 & a \end{array}$$

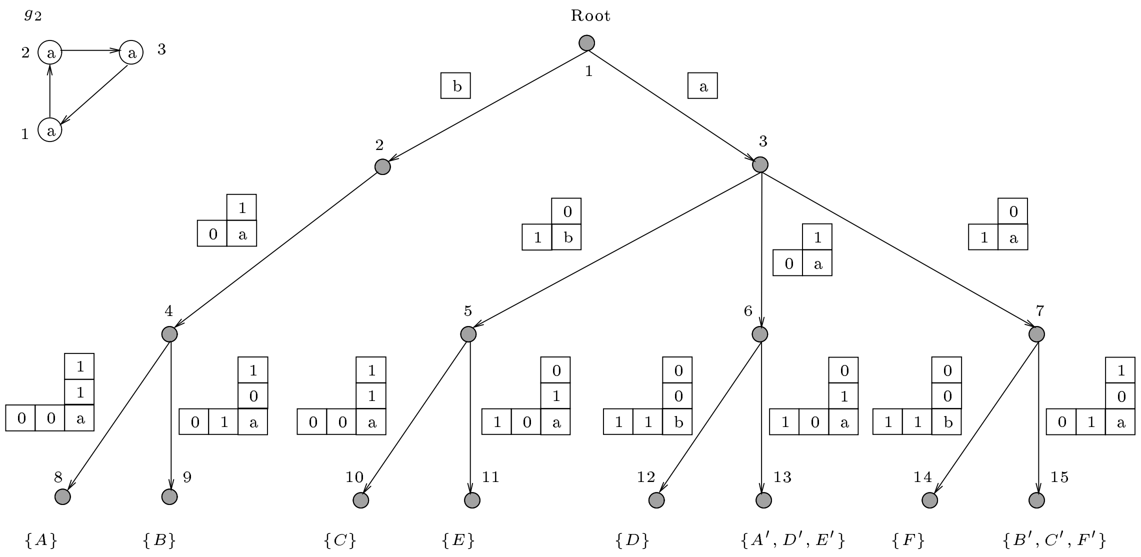


Figure 5: Decision tree for the graph g_1 in Fig 3 and the graph g_2.

classification of the adjacency matrices for a model graph is done solely on the basis of the current row-column element. No test on any level of the decision tree makes explicit use of the model graph itself. Therefore, given a set of model graphs $G_1, G_2, \ldots G_L$, the corresponding adjacency matrix sets $A(G_1), A(G_2), \ldots, A(G_L)$ can be classified and represented by the same decision tree. In Fig. 5 the decision tree for the graph g_1 of Fig. 3 and another graph, g_2, is displayed. In order to classify each of the adjacency matrices in $A(g_2)$ (given at the top of Fig. 5) only two nodes have to be added to the decision tree that corresponds to the graph g_1. As there are three automorphisms of g_2, each of the nodes 13 and 15 in Fig. 5 represents exactly three adjacency matrices.

At run time, the decision tree is directly used in order to classify the $m \times m$ adjacency matrix M_I of an unknown input graph G_I. The matrix M_I is classified on the first level according to its row-column element a_{1_I}. If there is some branch i from the root node to a successor node whose associated element a_{1_i} matches a_{1_I}, the algorithm continues with the successor node on the second level and so on. If at some point no classification is possible, then the input graph G_I is not isomorphic to any subgraph of the model graphs or any of the model graphs in the database. If each row-column element of G_I has been used in the classification process and some node N in the decision tree has been reached, then each permutation matrix that is associated with N represents a subgraph isomorphism from the input graph to one of the model graphs. If node N is a leaf node and the input graph and the model graph are of equal size then each permutation matrix associated with N

COMPILE_TREE(GRAPH $G = (V, E, \mu, \nu, L_v, L_e)$)

1. Create the root node *Root* of the decision tree if it does not yet exists.
2. For k=1 to n , where $n = |V|$
   (a) Generate all subgraphs $S_k \subset G$ with k vertices.
   (b) Call merge_tree($Root, S_k$).

Figure 6: Algorithm *compile_tree.*

describes a graph isomorphism between the input graph and one of the model graphs.

5 A More Efficient Representation of Decision Trees

The decision trees described in the last section are unnecessarily large. In this section we introduce a more compact representation. It it is based on the observation that at level k in the decision tree all subgraphs consisting of k vertices are represented, $k = 1, \ldots, n$. Notice that each of these subgraphs S is represented $k!/\alpha$ times, where α denotes the number of different automorphisms of S. Clearly, all these representations are equivalent to each other, and the information they contain is largely redundant. We now show how this kind of redundancy can be avoided. As a result, a more compact representation of decision trees is obtained.

Let N_1 and N_2 be nodes of the decision tree that both represent the same subgraph S of a model graph G and let S be given by its adjacency matrix M_S. Furthermore, let M_1 and M_2 be the adjacency matrices represented by N_1 and N_2, and P_1 and P_2 the corresponding permutation matrices such that $P_1 M_S P_1^T = M_1$ and $P_2 M_S P_2^T = M_2$. Then, there exists a permutation matrix R such that

$$M_1 = R M_2 R^T \tag{5}$$

R can be simply obtained by

$$R = P_1 P_2^T \tag{6}$$

because substituting R in Equation (5) by $P_1 P_2^T$ and also substituting M_2 by $P_2 M_S P_2^T$ yields $P_1 P_2^T (P_2 M_S P_2^T)(P_1 P_2^T)^T = P_1 M_S P_1^T = M_1$. Therefore, any adjacency matrix represented in the node N_2 can be transformed, by means of the matrix R, into a matrix that is represented in N_1. The most important conclusion from this observation is that for the decision tree node N_2 it is not necessary to classify the represented matrices further and create successor nodes of N_2. Instead, it is sufficient to classify the matrices represented in N_1 and simply refer or redirect the node N_2 to N_1. For this purpose, we introduce a *redirecting branch* in the decision tree that originates at N_2 and ends in N_1. Associated with the redirecting branch is the permutation matrix R. In general, for a subgraph $S \subset G$, we choose from the set of decision tree nodes that represent S an arbitrary node T and insert redirecting branches from all the other nodes to T. Consequently, only one of the

MERGE_TREE(NODE $Root$, GRAPH S)

1. Let S be given by its $k \times k$ adjacency matrix M.
   Generate all permutation matrices P of dimension k and the corresponding set $A(S)$ of permuted adjacency matrices of S.
2. For each $M_P = (a_1, \ldots, a_k) \in A(S)$ do
   (a) $N = Root$.
   (b) For i=1 to k do
      i. If there is a successor node N_s of N for which $a_k = a_{N_s}$
         then let $N = N_s$ else
         generate a new node N_s and make N_s a direct successor of N. Insert a_k into the dictionary that is attached to N. Let N_s represent the row-column element a_k and set $N = N_s$.
3. Choose the node N on the level $k+1$ which was created or found for the first matrix M_1 in $A(S)$ and set $T = N$. Let $M_1 = P_1 M P_1^T$.
4. For each node $N \neq T$ on the level $k+1$ that represents S, insert a redirecting branch to the node T. Let $M' \in A(S)$ be an adjacency matrix classified in N and P' the corresponding permutation matrix such that $M' = P' M P'^T$. Then, associate the permutation matrix $P_1 P'^T$ to the redirecting branch from N to T (see Equation 6).

Figure 7: Algorithm *merge_tree*.

decision tree nodes representing S will be used for the further classification of matrices of the graph G.

We now describe a procedure for compiling a decision tree from a given model graph. The basic idea of the compilation scheme is to create a decision tree for all subgraphs of the model graph. In the procedure we gradually increase the size of the subgraphs that are incorporated in the decision tree. As it was discussed above it is sufficient that for each subgraph there is only one node in the decision tree that is used for further classification. In the compilation procedure *compile_tree* given in Fig. 6, we start by creating a single root node for the decision tree, provided that the decision tree is yet empty[1]. Next, we successively generate the subgraphs of G starting with subgraphs of size one and ending with the graph G itself. For each of these subgraphs S_k where k denotes the number of vertices, a procedure *merge_tree* is called.

In the first step of the procedure *merge_tree*, given in Fig. 7, the set $A(S)$ of all permuted adjacency matrices is generated. These matrices are then classified by the already existing decision tree as far as possible. Note that the decision tree may at this point already incorporate different model graphs or subgraphs of model graphs. The classification of a

[1]A non-empty decision tree will be encountered when we compile a decision tree for a set of model graphs.

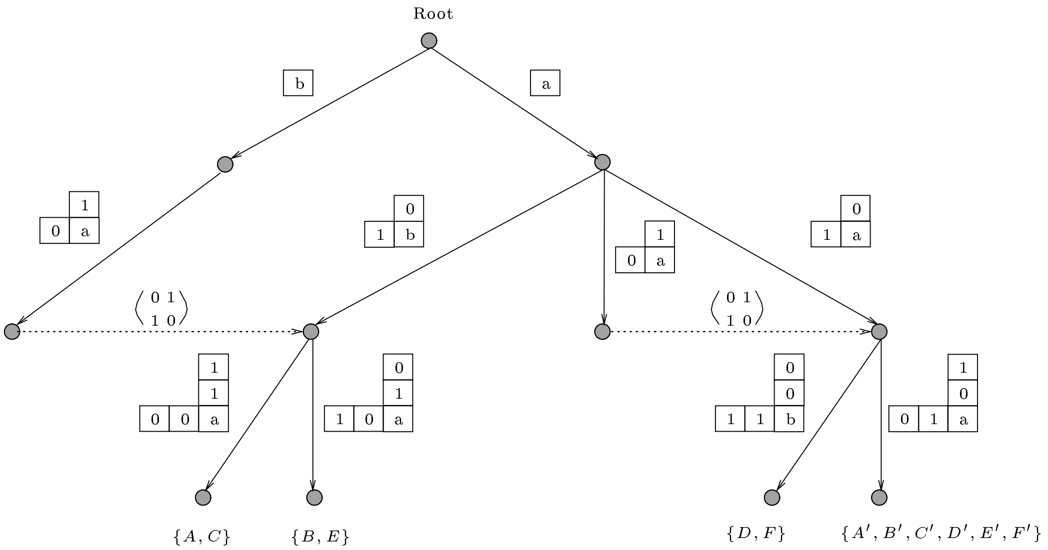


Figure 8: Compact decision tree for the classification of the adjacency matrices $\{A, \ldots, F\}$ and $\{A', \ldots, F'\}$ of the graphs g_1 and g_2, respectively (see also Figs. 3 and 5).

matrix is performed in the steps (2.a) and (2.b) by consulting successively each of its row-column elements. If for some node N on level k, there is no classification of the current matrix, i.e. there is no successor node with a row-column element that corresponds to the row-column element a_k of the current matrix, then a new node N_s must be inserted into the decision tree. The new node N_s must be made a direct successor node of N. Furthermore, the element a_k must be inserted into the dictionary that is attached to N. When all matrices in $A(S)$ have been succesfully classified, the redirecting branches are inserted. By default, the node that is generated for the first matrix $M_1 \in A(S)$ is taken as the representative node T. In step (4), all other nodes that have been created for matrices in $A(S)$ are then redirected to T.

In Fig. 8, the decision tree for the graphs g_1 and g_2 resulting from the algorithm *compile_tree* is displayed. There are two redirecting branches in this decision tree (denoted by dotted lines). Associated with both branches is the permutation matrix $\binom{0\ 1}{1\ 0}$. When applied to the 2×2 submatrices $S_{2,2}(A)$, $S_{2,2}(B)$ and $S_{2,2}(D)$ of g_1, the matrices $S_{2,2}(C)$, $S_{2,2}(E)$ and $S_{2,2}(F)$ result. For the graph g_2, the redirecting branch transforms the matrices $S_{2,2}(A')$, $S_{2,2}(D')$ and $S_{2,2}(E')$ into the matrices $S_{2,2}(C')$, $S_{2,2}(F')$ and $S_{2,2}(B')$. Notice that by introducing redirecting branches into the decision tree it is possible to save three decision tree nodes.

So far, we have only discussed the compilation of decision trees for single model graphs. However, the general structure of a decision tree is independent of the number of model graphs that are classified by the tree. Therefore, it is not necessary to build a new decision tree for every new model graph. Instead the same decision tree can be extended and used for any number of model graphs in the database. The only extension that is needed is to call the algorithm *compile_tree* for each model graph.

6 Decision Tree Traversal

The decision tree structure that was previously described can now be used to get a very efficient graph and subgraph isomorphism algorithm. Let $G_1, \ldots, G_L$ be a set of model graphs represented by a decision tree and G_I an unknown input graph. We assume that the input graph is represented by its adjacency matrix $M_I = (a_1, a_2, \ldots, a_m)$ given in row-column format. The algorithm *decision-tree* (see Fig. 9) now tries to find out whether there exists a matrix $M \in A(G_i)$ such that $M_I = S_{m,m}(M)$ by classifying M_I according to its row-column elements. The algorithm starts at the root node of the decision tree and first classifies M_I according to its first element a_1. If this step is succesful, the classification is continued on the next level. In general, if the process is on level k and N is the current node of the decision tree, then the successor node of N which represents a_k must be found. This is done by looking up the element a_k in the dictionary of row-column elements that is attached to N. If there is an element a_{k_N} in the dictionary that matches a_k perfectly, the process follows the branch from N to the successor node N_s that represents the row-column element a_{k_N}. If no such element can be found in the dictionary then M_I cannot be classified by the decision tree and it follows that G_I is not isomorphic to any subgraph of the model graphs G_i.

In step (2.d) of the algorithm it is checked whether the current node N_s has an outgoing redirecting branch. If this is the case then we follow this redirecting branch. Accordingly, the matrix M_I of the input graph must be permuted by applying the permutation matrix R that is attached to the redirecting branch. Note, however, that if N is on level k then R is a $k \times k$ permutation matrix because it was created at compilation time for a subgraph of size k (see previous section). On the other hand, M_I is an $m \times m$-adjacency matrix of the input graph G_I with $m \geq k$. In order to apply R to M_I, it is necessary to extend R to an $m \times m$ matrix by adding $m - k$ rows and $m - k$ columns. The rows and columns are to be copied from an $m \times m$ identity matrix. The extended matrix R is denoted by R'. Using R' it is possible to adjust M_I according to the redirecting branch by means of the operation $M_I = R' M_I R'^T$. The classification process can then be continued. The algorithm terminates either in step (2.b) when it is detected for the first time that there is no subgraph isomorphism from the input graph to any of the model graphs, or in step (3) when the last row-column element a_m of M_I has been processed and some node N has been reached. In the latter case, the matrix M_I is identical to all matrices M_i of the model G_i that are represented in N. If N is not a leaf node then the set of permutation matrices that are stored in N represents all subgraph isomorphisms from the graph G_I to G_i. If, on the other hand, N is a leaf node and G_I and G_i are of equal size then the set of permutation matrices in N represents all graph isomorphisms between G_I and G_i.

It is easy to see that the new algorithm for graph isomorphism traverses the decision tree without the need for backtracking and therefore has a time complexity which is only polynomial in the number of vertices of the input graph. Furthermore, the algorithm is clearly independent of the number of model graphs that are represented in the decision tree. In order to avoid any misunderstanding, it is important to note that the new algorithm cannot be used for the detection of subgraph isomorphisms from the model graphs to the input graph. This can be concluded from the fact that the algorithm uses the adjacency matrix of the input graph in its original form and tries to classify each row-column element sequentially. If the input graph contains an extraneous vertex, then the classification of

DECISION_TREE(NODE *Root*, GRAPH G_I)

1. Let G_I be given by its adjacency matrix $M_I = (a_1, \ldots, a_m)$ and let $m = |V|$ and $N = Root$.
2. For k=1 to m do
   (a) Look up the dictionary of row-column elements that is attached to node N and find an entry a_{k_N} such that $a_{k_N} = a_k$.
   (b) If no such element in the dictionary is found, the graph G is not isomorphic to any subgraph of the model graphs or any of the model graphs themselves represented by the decision tree. Exit with failure.
   (c) If an element $a_{k_N} = a_k$ is found in the dictionary, then follow the branch marked by a_{k_N} to the node N_s.
   (d) If there is a redirecting branch from N_s to some node N' then set $M_I = R'M_IR'^T$, where R' is the extended $m \times m$ permutation matrix associated to the redirecting branch (see text). Set $N = N'$.
   Else, $N = N_s$.
3. For each matrix M_P of a model G_i (with corresponding $n \times n$ adjacency matrix M_i) that is represented by the node N
   (a) If $m < n$ then the associated permutation matrix P describes a subgraph isomorphism from G_I to G_i, i.e. $M_I = S_{m,m}(M_P) = S_{m,m}(PM_iP^T)$.
   (b) If $m = n$ then the associated permutation matrix P describes a graph isomorphism from the input G_I to the graph G_i, i.e. $M_I = PM_iP^T$.

Figure 9: Algorithm *decision_tree*.

the adjacency matrix may fail at any stage of the decision tree.

7 Complexity Analysis

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

N =the number of model graphs in the database,
M=the maximum number of vertices in a model graph,
I =the number of vertices in the input graph,
l_v =the number of vertex labels,
l_e =the number of edge labels.

7.1 Computational Complexity of the Conventional Algorithm

For comparison reasons, we first analyze the space and time complexity of the conventional, backtracking based subgraph isomorphism method. The best case for this algorithm is given when the vertices of the model graph are uniquely labeled such that on each level of the recursion there is only one possible assignment of the current input graph vertex to a model graph vertex. Thus, the time complexity is in the best case bounded by

$$\begin{array}{c}\text{(best case time complexity)}\\ O(NIM)\end{array} \tag{7}$$

The space complexity of the conventional algorithm is in the best case bounded by

$$\begin{array}{c}\text{(best case space complexity)}\\ O(M^2 I)\end{array} \tag{8}$$

The worst case for the conventional algorithm arises when the model graphs and the input graph are unlabeled, undirected and highly connected. In this case, there are on each level I possible assignments for the model graph vertices and each of these assignments can be expanded. Therefore, the worst case time complexity of the conventional algorithm is bounded by

$$\begin{array}{c}\text{(worst case time complexity)}\\ O(I^M M^2 N)\end{array} \tag{9}$$

On the other hand, the space requirements in the worst case do not exceed the requirements in the best case, because each model graph is treated individually and the space allocated for each level in the search tree can be reused when backtracking occurs.

$$\begin{array}{c}\text{(worst case space complexity)}\\ O(M^2 I)\end{array} \tag{10}$$

7.2 Computational Complexity of the New Algorithm

Unlike the conventional algorithm for which no preprocessing steps are necessary, the new algorithm involves intensive preprocessing, namely the compilation of the decision tree for the model graphs. In the following, we first consider the computational complexity of the preprocessing step and then we analyze the run time complexity of the new algorithm.

With regards to practical applications it is important to note that the decision tree which is built in the preprocessing step must be present in the main memory at run time. Therefore, the size of the decision tree is of great importance to any application incorporating the new algorithm. A useful measure for the size of the decision tree is the number of nodes. According to the compilation algorithm described in Section 5 the decision tree for a model graph is built level by level. The nodes on the level $k+1$ are created by isolating all subgraphs of size k of the model graph and classifying each of these subgraphs in the decision tree[2]. However, regardless of the number of permutations, there is only one node for each subgraph of size k that is finally used as an ancestor node to other

[2]The root node is located on level 1. Subgraphs of size 1 are represented on then level 2 and so on.

nodes in the decision tree. Therefore, the number of nodes on each level with successor nodes is limited by $O(\binom{M}{k})$. Each of these nodes may be the parent node of numerous successor nodes that represent different row-column elements. Any row-column element a_{k+1} that is associated to a branch that originates in a node on level $k+1$ consists of $2k$ edge label entries and one vertex label entry. Consequently, there are at most $O(l_e^{2k} l_v)$ different row-column elements, each being represented by a distinct node in the decision tree. The total number of nodes on the level $k+2$ following the level $k+1$ is therefore bounded by $O(\binom{M}{k} l_e^{2k} l_v)$. The sum of the nodes over all the levels (without the root node) is then bounded by

$$O(l_v \sum_{k=0}^{M-1} \binom{M}{k} (l_e^2)^k) = O(l_v(1+l_e^2)^M) \tag{11}$$

If there are several model graphs in the database, the decision tree becomes linearly dependent on N, the size of the database:

$$\begin{gathered}\text{(space complexity of decision tree)}\\ O(N l_v (1+l_e^2)^M)\end{gathered} \tag{12}$$

This is the worst case upper bound for directed, labeled graphs. A special case, however, are unlabeled, undirected graphs. For these graphs, which are notably the worst possible case for the conventional algorithm, the size of the decision tree is bounded by the following expression:[3]

$$O(N3^M) \tag{13}$$

The theoretical run time performance of the new algorithm can be estimated by analyzing the algorithm *decision-tree* in Section 6. The adjacency matrix of the input graph is classified step by step by the decision tree. For a graph with M vertices, the adjacency matrix contains M row-column elements $a_1, a_2, \ldots a_M$. The classification of a matrix on level $k+1$ according to the row-column element a_{k+1} requires that the row-column element a_N in any of the successor nodes of the current node, which matches a_{k+1} perfectly, is found. The element a_{k+1} is an array consisting of $2k$ edge labels and one vertex label. At construction time, the row-column elements represented in the successor nodes of the current node have been organized in a dictionary. The dictionary is indexed such that a_{k+1} can be classified by performing exactly $2kl_e + l_v$ comparisons. This computational effort must be done on each level of the decision tree. Additionally, it may occur that on each level of the decision tree a redirecting branch is encountered. Consequently, on each level the vertices of the input graph must be reordered according to the permutation matrix that is associated with the redirecting branch. Reordering the position of M vertices on M levels requires M^2 computational steps. Therefore, the theoretical complexity of the new subgraph isomorphism algorithm is bounded by

$$\begin{gathered}\text{(best and worst case time complexity of decision tree traversal)}\\ O(M(2Ml_e + l_v) + M^2) = O(M^2 l_e + M l_v)\end{gathered} \tag{14}$$

[3] For unlabeled graphs the set of edge labels l_e consists only of the null label and a uniform label, which indicates the presence of an edge. Consequently, $|l_e| = 2$ and $|l_v| = 1$. Furthermore, the adjacency matrix of an undirected graph is symmetric and there are at most $O(l_e^k l_v)$ different row-column elements on level k.

Notice that the above complexity analysis of the new algorithm with respect to subgraph isomorphism detection also applys to graph isomorphism detection.

Again, a special case can be observed for unlabeled, undirected graphs, where the run time performance is bounded by

$$O(M^2) \tag{15}$$

It is important to note that in both the general and the special case of unlabeled graphs the computational complexity of the new algorithm is polynomially bounded. Furthermore, the theoretical performance is completely independent of the connectivity of the graphs and the number of model graphs that are represented by the decision tree. Unlike the conventional, backtracking-based algorithm (and any other subgraph isomorphism algorithm that has been proposed in the literature up to this moment) the new algorithm is guaranteed to find all graph and subgraph isomorphisms in quadratic time. Naturally, for regular complete graphs, the *listing* of all subgraph isomorphism may take exponential time, as their number is exponential. The new algorithm delivers in quadratic time the number of subgraph isomorphisms and a pointer to the set of all permutation matrices that represent subgraph isomorphisms. The main problem of the new approach, however, is that the size of the decision tree grows exponentially with the size of the model graphs. Therefore, in order to render the new algorithm applicable for practial tasks, it is necessary to find ways of making the decision tree more compact.

8 Pruning the Decision Tree for Practical Applications

In Section 5, the compilation of a complete and functional decision tree for the classification of adjacency matrices was described. In the following, we will refer to decision trees that are built according to this compilation as *complete* decision trees. The main advantage of complete decision trees is that graph isomorphisms can be detected in quadratic time and that subgraph isomorphisms from input graphs that are subgraphs of model graphs can also be found in quadratic time. However, the disadvantage of a complete decision tree is its exponential size. We will now propose two different techniques in order to reduce the size of the decision tree. The first method is based on the idea of pruning the breadth of the complete decision tree. The properties of the new algorithm based on a breadth-pruned decision tree are similar to those of the algorithm based on a complete decision tree. However, the time complexity for graph isomorphism detection based on a breadth-pruned decision tree is increased from $O(M^2)$ to $O(M^3)$. Also, the breadth-pruned decision tree does no longer support subgraph isomorphism detection. The second method aims at pruning the depth of the decision trees. Depth-pruned decision trees will no longer guarantee a graph isomorphism detection in polynomial time. However, they are very suitable for applications where the number of model graphs is large.

8.1 Breadth-Pruning the Decision Tree

The compilation of a decision tree for a model graph G consists in generating all permutations of the adjacency matrix of G and incorporating a classification path in the decision tree for each of these matrices. Consequently, at run time the adjacency matrix of the

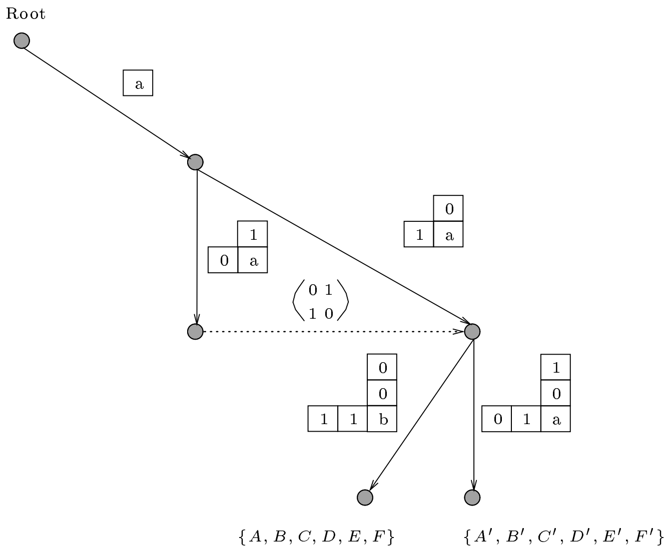


Figure 10: The decision tree of Fig. 8 (for the graph g_1 and g_2) after breadth-pruning was performed.

input matrix is taken in its original form and classified directly by the decision tree. However, by allowing additional operations on the input matrix at run time, the number of nodes in the decision tree can be reduced remarkably. In the following, we briefly outline two sets of transformations of this kind.

The first set of transformations simply requires that the vertices of the input graph are ordered such that each vertex is connected to at least one other vertex that appears earlier in the ordering. For the adjacency matrix of the input graph given as a vector of row-column elements $(a_1, \ldots, a_m)$, this means that each row-column element a_i contains at least two non-zero entries. It is straightforward to show that the adjacency matrix of any connected graph can be transformed such that the above condition holds. This problem is equivalent to finding a spanning tree of a graph and thus it can be solved in quadratic time[Eve79]. We can then reduce the number of permuted adjacency matrices $A(G)$ that must be classified by the decision tree. Namely, all permutations of the model matrix for which the above condition does not hold, may be discarded at compilation time. For graphs with a low connectivity, the number of decision tree nodes that are pruned by this technique can be very high. As the transformation of the input graph has only quadratic time complexity, the graph isomorphism algorithm based on the pruned decision tree has the same computational complexity as before. Also, both graph and subgraph isomorphism detection are still supported by a decision tree that was subject to this type of breadth-pruning technique.

The second technique for pruning the breadth of a decision tree is based on the following observation. Let $M_I = (a_1, \ldots, a_m)$ be the adjacency matrix of an input graph G_I. Assume that M_I is to be classified by a decision tree that was built for a model graph G. Furthermore, assume that M_I has been classified up to the level $k + 1$ and the node N is the current node in decision tree. In the next step, the algorithm will try to find a successor node of N according to the $k + 1$-th element of M_I, a_{k+1}. However, at this point, the decision tree traversal algorithm may be rewritten in the following manner. Note that there are exactly $(n-k)!$ different permutation matrices of G_I for which the first k row-column elements $(a_1, \ldots, a_k)$ are identical. Consequently, when M_I has been classified up to the element a_k, there are in fact $n-k$ different successor nodes, each representing a permutation of M_I for which the element a_{k+1} is different but the elements $a_1, \ldots, a_k$ are identical. Therefore, a modified version of the graph isomorphism algorithm may rotate the $n-k$ last columns and rows of the input matrix M_I and try each of the $n-k$ row-column elements a_{k+1} in order to classify the matrix. Naturally, in a complete decision tree, each of the row-column elements will allow a succesful classification. However, with the modified version of the graph isomorphism algorithm it is no longer necessary to incorporate all $n-k$ successor, but it is sufficient to build a single successor node. The modified algorithm will still be able to classify the input matrix by trying all $n-k$ possible rotations of the last rows and columns of the input matrix. If this pruning scheme is applied on each level of the decision tree, then the number of nodes in the tree will be reduced by a factor of n, where n indicates the total number of vertices in the represented model graph. Hence, in case of an unlabeled, undirected graph, the size of the decision tree will be bounded by $O(3^n/n)$ (see Equation 13). The computational complexity of the graph isomorphism algorithm on the other hand is increased by a factor of n due to the permutations of the input matrix which must be performed on each level of the decision tree. Thus, the run time complexity of the new algorithm for undirected, unlabeled graphs on the basis of a pruned decision tree is bounded by $O(n^3)$ (see Equation 15). The construction of a decision tree which is pruned according to this technique is based on a two step procedure. First, the complete decision tree is constructed, and afterwards the maximal number of nodes is pruned according to the technique described above. (Consequently, the applicability of breadth-pruned decision trees is limited by the exponential size of the corresponding complete decision tree.) The main drawback of this type of breadth-pruning technique, however, is that the resulting decision tree can only be used for graph isomorphism and no longer for subgraph isomorphism detection. By allowing only one possible successor for each matrix represented by a decision tree node, some subgraphs of the represented model graph will no longer be present in the decision tree. Consequently, at run time, they will not be detected by the modified algorithm.

In Fig. 10, the decision tree for the graphs g_1 and g_2 of Fig. 8 is displayed after both techniques for breadth-pruning have been applied. The first technique has no effect because the graphs g_1 and g_2 are completely connected. The second technique, however, reduces the size of the decision tree remarkably. Note that the left branch departing from the root node is no longer necessary since all adjacency matrices can be classified by the right branch at the expense of reordering the vertices. Consequently, the decision tree for the graphs g_1 and g_2 of Fig. 5 now contains only 6 nodes instead of 11.

For the remainder of this paper, we will generally assume that a breadth-pruned decision tree was subject to both the first and the second pruning technique that were presented in this section.

8.2 Depth-Pruning the Decision Tree

In the previous sections we have shown that the new algorithm based on the decision tree approach has very interesting properties. However, its main disadvantage is the exponential growth of memory and, consequently, the limitation to graphs of moderate size. In this section, we briefly introduce a depth-pruning technique, which makes the new algorithm also suitable for applications that deal with larger graphs.

A common observation in practical graph applications is that the branching factor of a graph isomorphism search tree, such as it is used in Ullman's algorithm, is large at the top of the tree and becomes gradually smaller for deeper levels of the search tree. This is especially true for labeled graphs where only at the beginning of the search tree there are several possible matchings, while at the lower part of the search tree no backtracking is necessary anymore. In model based 3D-object recognition, for example, where graphs represent 3D objects and graph isomorphism is used to recognize objects and establish their position, the search for a graph isomorphism is often terminated after three or four vertices of an object have been matched, because these vertices are sufficient to uniquely determine the location of the object. Based on this observation, we propose to prune the depth of a decision tree in the following manner.

Instead of compiling a complete decision tree for a model graph G of size n, we only build a decision tree for all subgraphs S_k of G of size k, where $k < n$. The depth of the decision tree is therefore limited by k. At run time, the unknown input graph G_I is classified by the new algorithm described in Section 6. When a leaf node on the level k of the decision tree is reached, a subgraph of size k of the input graph has been succesfully matched onto one or several subgraphs of the model graph. To get a complete subgraph isomorphism, the search process must then continue with a conventional algorithm. However, the search space of the conventional algorithm has been greatly reduced as the the partial matching found by the decision tree can be used as an initialization of the conventional search algorithm.

The most important aspect of the depth-pruned decision tree approach, however, is its capability of indexing a model graph out of a set of models. Clearly, a pruned decision tree may incorporate different model graphs. At run time, an input graph that is classified with the algorithm *decision_tree* on the basis of the depth-pruned decision tree will invoke a single leaf node of the decision tree. Only model graphs that are represented by this leaf node will be subject to being further tested, for example, by a conventional algorithm. The depth-pruned decision tree is thus a very efficient indexing method for large databases of graphs.

9 Practical Experiments

In order to examine the efficiency of the new algorithm in practice, we have performed a number of experiments with randomly generated graphs. In Section 7 it was shown that the complexity of the new algorithm is quadratic for both graph and subgraph isomorphism detection if the underlying decision tree is complete. For practical applications, however, it will be necessary to prune the decision tree according to one or several pruning techniques described in Section 8. Thus, all the experiments documented in this section

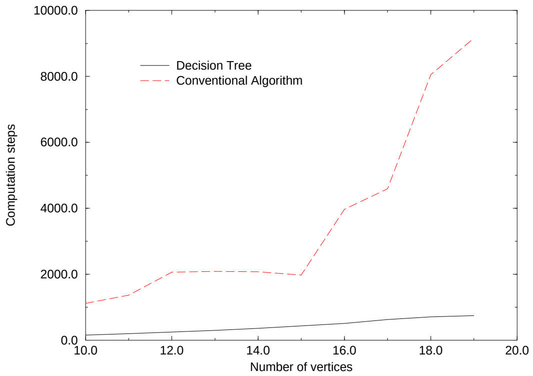


Figure 11: Computation steps for a growing number of vertices (first experiment).

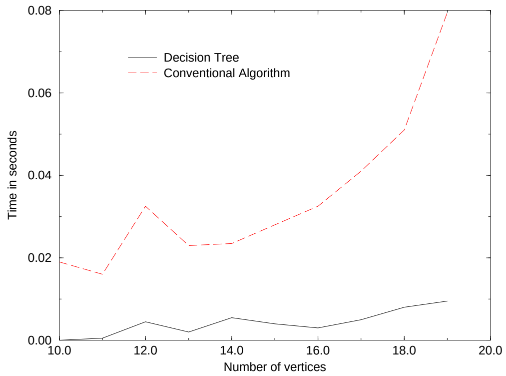


Figure 12: Computation time in seconds for a growing number of vertices (first experiment).

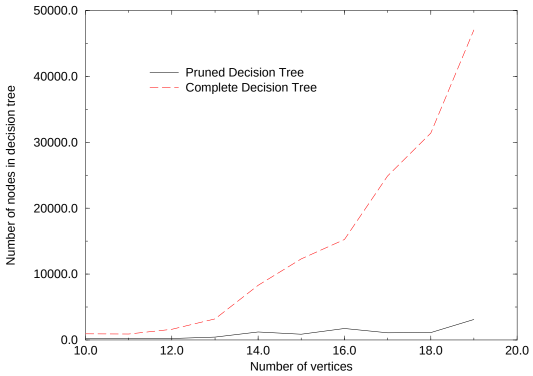


Figure 13: Number of nodes in the decision tree for a growing number of vertices (first experiment).

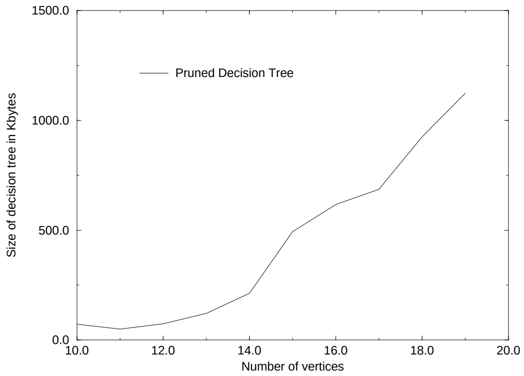


Figure 14: Disk space occupied by the decision tree for a growing number of vertices (first experiment).

were performed with respect to graph isomorphism detection and the underlying decision trees were subject to either breadth-pruning or both breadth- and depth-pruning. For each experiment, we generated one or more model graphs and used these model graphs to create isomorphic input graphs. All of the graphs generated for the experiments in this section were undirected and unlabeled. The new decision tree algorithm and the conventional algorithm based on Ullman's refinement procedure were both implemented in C++ and run on a SUN Sparc10 Workstation. In order to compare the performance of the algorithms a measure for the computational effort was needed. We defined that a basic computation step is the comparison of one model graph vertex and its incident edges to one input graph vertex and its incident edges. The performance was then measured by counting the number of basic computation steps that were performed while searching for all graph isomorphisms. Additionally, for each experiment, we measured the absolute computation times of both algorithms. The sizes of the decision trees in terms of the number of nodes and the required disk space are also given for each experiment.

In the first experiment, we wanted to demonstrate the efficiency but also the limits of the new algorithm with regard to the size of the graphs that can be handled. For this purpose, we randomly generated a sequence of model graphs with a growing number of vertices. A copy of each model graph with randomly permuted vertices was used as input graph. We started with a graph consisting of 10 vertices and 15 edges and ended up with a graph consisting of 19 vertices and 28 edges. The number of computational steps that were performed by the new algorithm and the conventional algorithm in order to find all graph isomorphisms between model and input graph are plotted in Fig. 11. The corresponding computation times are displayed in Fig. 12. Clearly, the new algorithm is much more efficient than the backtracking algorithm as predicted. Notice that the computation time for a graph with 19 vertices and 28 edges was ten times longer with the conventional algorithm than with the new algorithm. The cost of the run time efficiency of the new algorithm is, however, the size of the decision tree. In Fig. 13 the number of nodes in the breadth-pruned and the complete decision tree for a growing number of vertices in the model graphs is displayed. The growth of the complete decision tree is clearly exponential. While there were only 945 nodes in a complete decision tree for a graph with 10 vertices, there were already 47'050 nodes in a tree for a graph with 19 vertices. The breadth-pruned decision tree, on the other hand, is much smaller. In Fig. 14 the disk space that is occupied by the breadth-pruned decision tree is displayed. Notice that for a graph with 19 vertices, the breadth-pruned decision tree required 1.2 Mbyte of disk space. Consequently, the complete decision tree, which is 16 times larger in terms of nodes, would require approximately 20 Mbyte of disk space. The compilation time for a decision tree for a model graph with 10 vertices took only two second, while the compilation for a model graph with 19 vertices took roughly 30 minutes on a Sparc10 Workstation. (The compilation of a model graph consisting of 20 vertices did not finish within 60 minutes and was aborted after more than 200 Mbytes of main memory were taken up by the program.) Although the results for the breadth-pruned decision tree indicate that graphs with 19 vertices and 28 edges can be represented by decision trees of reasonable size, it is important to remember that for the construction of the breadth-pruned decision tree, the complete decision tree must be created first (see Section 8). Consequently, although the breadth-pruned decision tree is reasonably small, the necessary previous construction of the complete decision tree prevents the new method from being applied to graphs containing more than 19 vertices. In Section 8, we showed that the computational complexity of the new algorithm based on a breadth-pruned decision tree is cubic in the number of vertices of the input graph, while for a complete decision tree the complexity is only quadratic. However, in practice, the run time performances of the new algorithm based either on breadth-pruned or on complete decision trees are almost identical. (This observation is not illustrated here.)

In the second experiment, we were interested in the influence of the database size on the performance of the new algorithm. We first generated a single model graph consisting of 11 vertices and 33 edges and then gradually added new model graphs of the same size until there were 30 graphs in the database. With every new model that was added to the exisiting database, a corresponding input graph was generated and the computational effort for finding all graph isomorphisms from the input graph to the model graphs in the database was measured for both algorithms. The results of the second experiment are document in Figs. 15 and 16. As was to be expected from the computational complexity

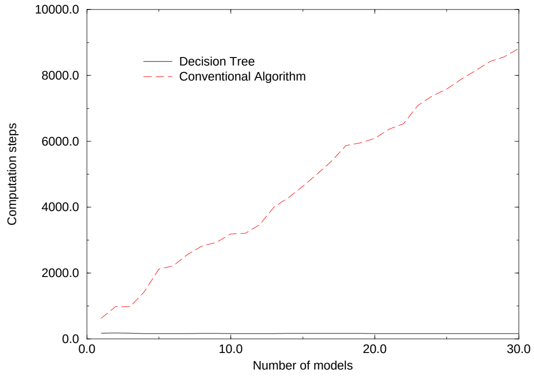


Figure 15: Computation steps for a growing number of model graphs (second experiment).

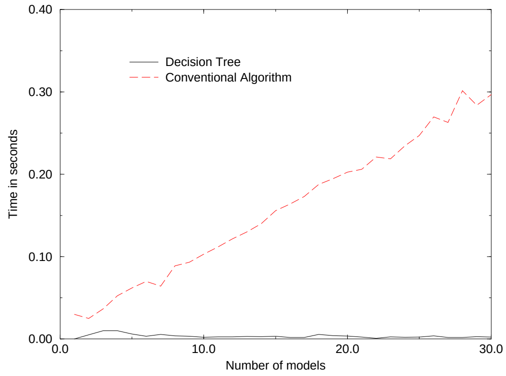


Figure 16: Time in seconds for a growing number of model graphs (second experiment).

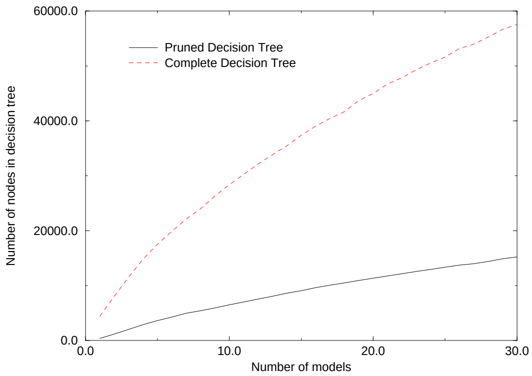


Figure 17: Number of nodes in the decision tree for a growing number of model graphs (second experiment).

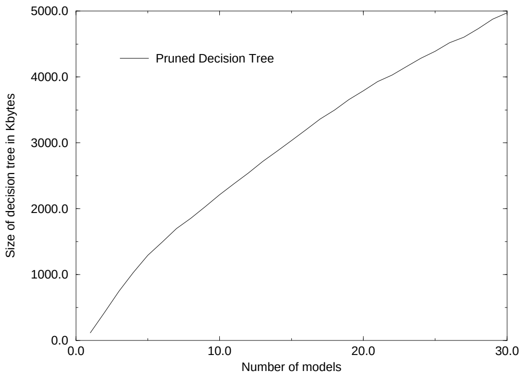


Figure 18: Disk space occupied by the decision for a growing number of model graphs (second experiment).

analysis, the decision tree algorithm's performance was completely independent of the number of models in the database. The conventional algorithm, on the other hand, had to perform a graph isomorphism search for each model in the database individually and was therefore linearly dependent on the size of the database. While the computation time of the decision tree algorithm varied between 0.005 and 0.01 seconds, the conventional algorithm required increasingly more time for a growing database. In the end, for 30 model graphs, the conventional algorithm's computation time was approximately 0.3 seconds. For practical applications, such as 3D-object recognition, where graphs are restricted in size but the number of object models is large, this result is of great importance. From the computational complexity analysis, we expect that the size of the decision tree is linearly dependent on the number of model graphs. In practice, due to the fact that common subgraphs of different model graphs can be represented by the same decision tree structures, the size of the decision tree is often only sublinearly dependent on the number of model graphs. In Fig. 17 the size of the decision tree for a growing number of model

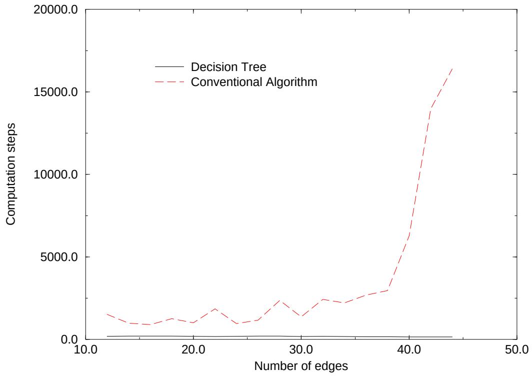


Figure 19: Computation steps for a growing number of edges (third experiment).

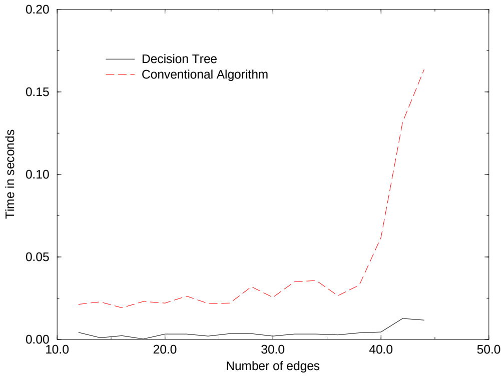


Figure 20: Computation time in seconds for a growing number of edges (third experiment).

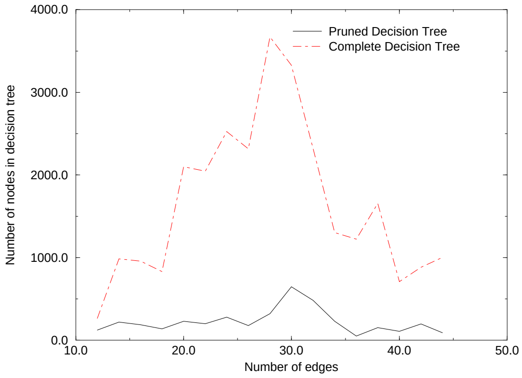


Figure 21: Number of nodes in the decision tree for a growing number of edges (third experiment).

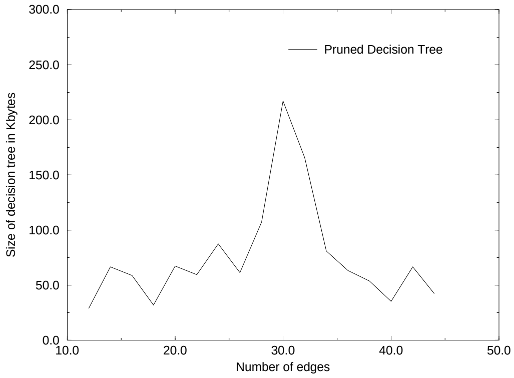


Figure 22: Disk space occupied by the decision tree for a growing number of edges (third experiment).

graphs is displayed. It can be observed that the growth rate of the decision tree becomes indeed smaller with every new model graph that is added to the database. Although both the pruned and the complete decision tree grow with the number of model graphs, the complete decision tree is on the average four times as large as the pruned decision tree. In Fig. 18 the disk space occupied by a breadth-pruned decision tree for a growing number of model graphs is displayed. Note that a database consisting of 30 model graphs – each of which contains 11 vertices and 33 edges – requires 5 Mbytes of disk space. Thus, we conclude that a database containing 100 model graphs will occupy probably less than 15 Mbytes. Consequently, with the run time performance being independent of the number of model graphs, the new algorithm is an interesting option for many practical applications.

Apart from the influence of the size of the graphs and the size of the database, the complexity of the graph isomorphism problem is also strongly dependent on the number

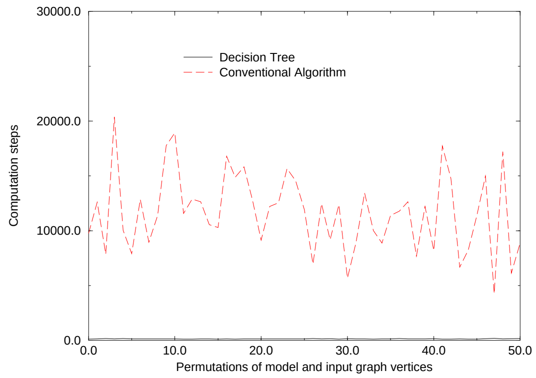


Figure 23: Computation steps in dependence of the vertex order (fourth experiment).

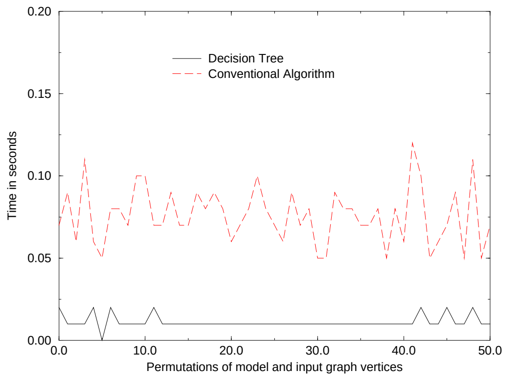


Figure 24: Time in seconds in dependence of the vertex order (fourth experiment).

of edges in a graph. In a complete graph, for example, where each vertex is connected to all other vertices, the number of automorphisms is exponential in the number of vertices. Thus, any algorithm attempting to list all graph isomorphisms from one complete graph to another isomorphic complete graph will take exponential time. In order to compare the performance of the new and the conventional algorithm for varying degrees of connectivity of the model graph, we performed a third experiment. In this third experiment, the number of vertices in the model graph was constantly kept at 11 while the number of edges was varied between 11 and 44. Notice that an undirected graph with 11 vertices can contain 55 edges at most. However, for such complete or nearly complete graphs, the enumeration of all graph isomorphisms is an exponential problem and of little interest in practice. Therefore, the number of edges for this experiment was restricted to 44. There was exactly one model graph in the database. The performance of both algorithms with respect to the number of computational steps and the computation time are displayed in Figs. 19 and 20, respectively. Again, the experimental results confirm the theoretical complexity analysis, i.e., the performance of the decision tree method is completely independent of the number of edges in the model graph, while the conventional approach requires more computation steps and time when the number of edges grows[4]. On the other hand, the size of the decision tree is strongly dependent on the the number of edges in a graph. The size of the tree in terms of nodes is displayed in Fig. 21 while the required disk space of the pruned decision tree is given in Fig. 22. In Fig. 21 we observe that the size of the complete (and also the pruned) decision tree reaches its maximum when there are approximately 30 edges present in the graph (consisting of 11 vertices). For more than 30 edges in the graph, the size of the tree decreases again. This behavior has not been explained in the computational complexity analysis. The reason for the growth in the beginning and the subsequent decrease of the decision tree's size lies in the influence of the number of automorphisms. Note that there is a correlation between the number of automorphism of a graph and the average number of successor nodes of a decision tree node. Consequently, if the number of automorphisms in a graph is large, the number of

[4]Notice, however, that the computation time of the decision tree varied slightly due to the fact that at the end of each run, the number of detected graph isomorphisms was individually counted by the new algorithm.

successor nodes of the decision tree nodes is large, too. On the other hand, the number of leaf nodes for a graph G is limited by $|A(G)|/\alpha$, where α is the number of automorphisms of G. While $|A(G)|$ is constant for all graphs of a given size, α is dependent on the edges and labels in the graph. Therefore, if the number of edges and consequently α is increased, the total number of leaf nodes in the decision tree decreases. The combination of these two effects results in the curve shown in Fig. 21.

In the fourth experiment documented in Fig. 23 and 24 we investigated the robustness of the new algorithm towards different orderings of the vertices of the model and the input graphs. It is known that the performance of conventional graph isomorphism methods based on backtracking is strongly dependent on the ordering of the model graph vertices. If, for example, the vertices are ordered such that backtracking occurs at an early stage in the search process, the computational performance will be better than on an ordering which imposes late backtracking. For the fourth experiment, we therefore generated randomly a model graph consisting of 11 vertices and 33 edges. A copy of the model graph was used as input graph. Next, both model and input graph vertices were randomly permuted and after each permutation the conventional and the new algorithm were used to calculated all graph isomorphisms between the model and the input graph. There are $11! = 39'916'000$ possible permutations of 11 vertices. In Fig. 23 and 24 only the results of 50 permutations are plotted. But it can be clearly seen that the performance of the new algorithm is fairly independent of the ordering of the vertices, while the conventional algorithm's performance varies remarkably from one permutation to the next.

So far, we have studied the behavior of the new algorithm on the basis of breadth-pruned decision trees. Though these decision trees guarantuee graph isomorphism in cubic time, they are only applicable for graphs with maximally 19 vertices. Consequently, for larger graphs, it will be necessary to prune the depth of the decision trees as described in Section 8.2. In the fifth experiment, we first examined how depth pruning can be used to reduce the size of a decision tree. For a model and an input graph consisting of 19 unlabeled vertices and 28 edges, we created decision trees with varying depth, starting with depth 7 and ending with the complete decision tree of depth 19. For each of these decision trees, an identical copy of the model graph was used as input graph and its adjacency matrix was classified according to the algorithm *decision_tree* described in Section 6 and the modification for depth-pruned decision trees described in Section 8.2. In Figs. 25 and 26 the computation steps and the computation time are displayed for the fifth experiment. Naturally, the performance of the conventional algorithm was constant in this experiment. The performance of the new algorithm, however, was strongly dependent on the depth of the underlying decision tree. For example, for a decision tree with depth six, the new algorithm required 0.6 seconds and over 45'000 computation steps compared to 0.005 seconds and 780 computation steps when a decision tree of depth 19 was used. Due to the large number of isomorphic subgraphs of size 6 in a graph with 19 vertices, the decision tree that was pruned at depth 6 could not limit the search space. However, the efficiency of the new algorithm increased rapidly with the growing depth of the underlying decision tree. Particularly, for a decision tree of depth 11, the new algorithm was already as efficient as for a tree of depth 19. Therefore, we conclude that for graphs with 19 vertices, it is sufficient to construct decision trees with depth 11. In Fig. 27, the size of the decision tree in terms of nodes is displayed. For comparison reasons, the size of the complete decision tree is also displayed. Note that by pruning the decision trees at depth 11 it is possible to

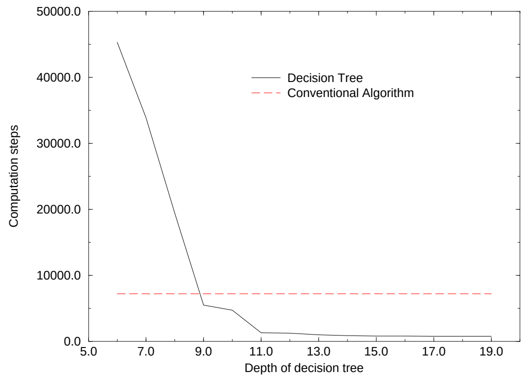


Figure 25: Computation steps in seconds in dependence of the depth of the decision tree (fifth experiment).

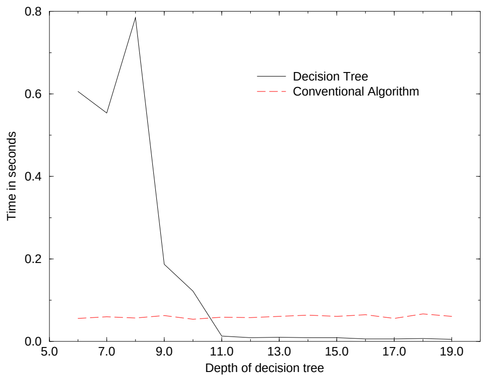


Figure 26: Detection time in seconds in dependence of the depth of the decision tree (fifth experiment).

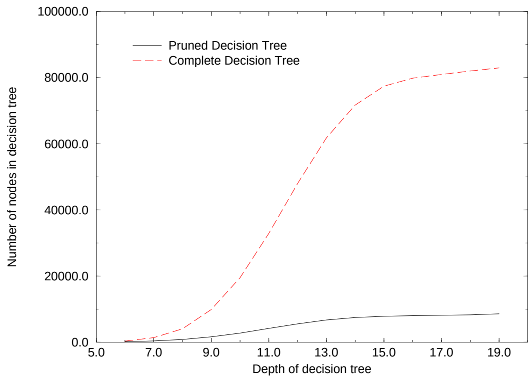


Figure 27: Size of the decision tree for a growing compilation depth (fifth experiment).

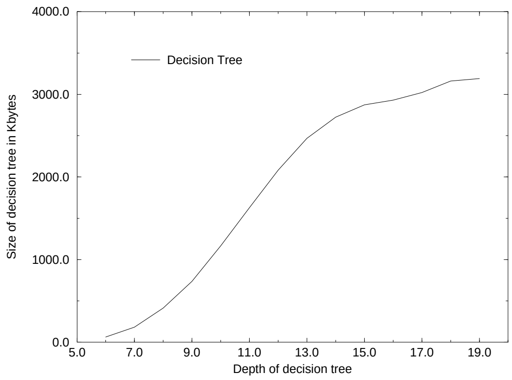
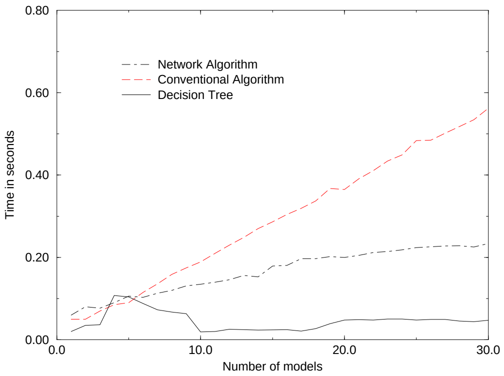


Figure 28: Disk space occupied by the decision tree for a growing compilation depth (fifth experiment).

save more than half of the total number of nodes. Analogously, a decision tree of depth 11 occupies only 1.5 Mbyte of disk space compared to 3 Mbyte for a decision tree of full size (see Fig. 28). Consequently, the depth-pruning technique renders an efficient algorithm applicable for graphs with more than 19 vertices.

In the theoretical complexity analysis and in practice, it was demonstrated that the new algorithm's run time performance is independent of the number of model graphs for both complete and breadth-pruned decision trees (see second experiment). In the case of depth-pruned decision trees, however, the new algorithm is no longer independent of the database size. However, depth-pruned decision trees represent a powerful means for indexing a database of model graphs. Particularly, the model graphs may consist of more than 19 vertices due to the fact that depth-pruning is applied. Furthermore, if the vertices are labeled, even decision trees with small depth become very effcient in indexing a database of models. In order to demonstrate this effect, we performed a sixth experiment.



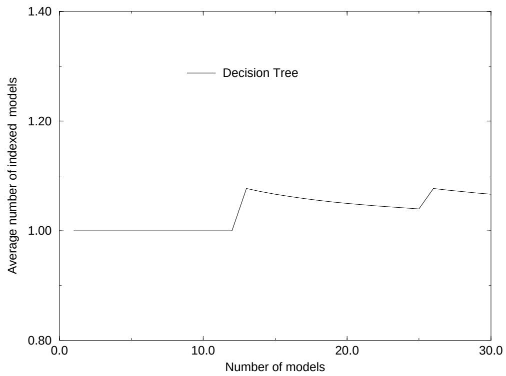
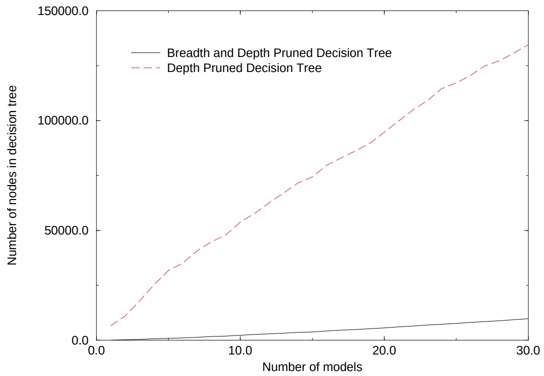


Figure 29: Detection time in seconds for a growing number of models (sixth experiment).

Figure 30: Average number of models that are indexed by the decision tree for a growing number of models (sixth experiment).



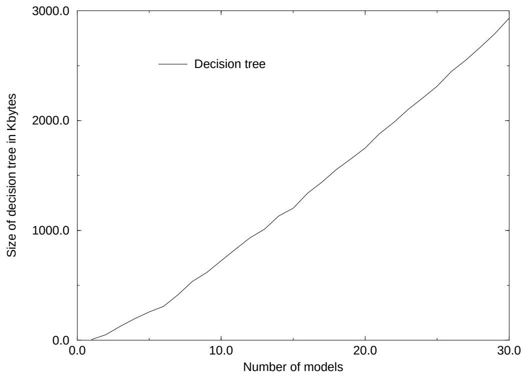


Figure 31: Size of the decision tree when only depth-pruning and when both depth- and breadth-pruning are applied (sixth experiment).

Figure 32: Disk space occupied by the decision tree when both depth and breadth-pruning are applied (sixth experiment).

We randomly generated model graphs consisting of 20 vertices with 5 different labels and 30 edges. Initially, the database of model graphs was set to contain a single model. We then gradually increased the number of model graphs until there were 30 model graphs in the database. The database of graphs was represented by a decision tree that was built to the 6th level, i.e. all subgraphs of the models with six vertices were compiled into the decision tree. Each model graph was then used as input graph and its adjacency matrix was classified by the depth-pruned decision tree. When a leaf node was reached in the tree traversal algorithm, all the model graphs represented by this leaf node were then further tested by the conventional algorithm. However, the conventional algorithm did not perform a complete search from scratch, but was initialized with the subgraph isomorphisms that were represented in the encountered leaf node. In Fig. 29 the time in seconds for the detection of all graph isomorphisms between the input graphs and the model graphs is plotted. For comparison reasons, we also applied the conventional algorithm on its own and the network based algorithm that was developed by the authors and has proven to be efficient for labeled graph matching [MB95b]. Note that both the conventional algorithm and the network based algorithm required more time than the new algorithm based on the depth-pruned decision tree. The reason for the behavior of the new algorithm is illustrated in Fig. 30 where the number of indexed model graphs for each size of the database is given. Note that even for 30 models in the database, the average number of models that are indexed was never larger than 2. In the first experiment it was demonstrated that the new algorithm can currently handle graphs with up to 19 vertices when breath pruning is applied. Based on the size of a decision tree for a single graph of this size, we can expect that a complete decision tree for a graph with 20 vertices would contain approximately 150'000 nodes. Thus, it would not be possible to compile the model graphs in the sixth experiment. By depth-pruning the decision tree, however, even databases of graphs with 20 vertices become managable for the new algorithm. In Fig. 31 the number of nodes in the depth-pruned decision tree for the model graphs of the fifth experiment is shown. Notice that the size of the decision tree was even further reduced by applying not only depth pruning but also breadth pruning. The disk space in kilobytes that was occupied by the depth- and breadth-pruned decision tree is given in Fig. 32.

We conclude that both breadth- and depth-pruning are valuable techniques, which render the new algorithm applicable for practical applications. The breadth-pruning technique reduces the number of nodes on each level of the decision tree but preserves the basic quality of the decision tree for polynomial graph isomorphism detection (not, however, for subgraph isomorphism detection). On the other hand, the depth-pruning technique only creates decision trees up to a certain depth. A depth-pruned decision tree no longer guarantuees polynomial graph isomorphism. However, its main advantage lies in the efficient indexing of the database of model graphs.

10 A Practical Application Based on the New Algorithm

In order to demonstrate the applicability of the decision tree based algorithm, we integrated the new algorithm into a system for the interpretation of line drawings. The system has been previously described by the authors in [MB95a]. The main purpose of the system is to interpret technical line drawings by locating predefined symbols. The drawings and symbols are required to consist of straight line segments. Internally, both symbols and drawings are represented by labeled graphs such that each line segment corresponds to a vertex in the graph and each intersection of line segments corresponds to an edge of the graph. The edges are labeled with the angle between two line segments. In Fig. 33 the symbols that are known to the system are displayed. Note that the largest symbol is symbol 3 which contains 24 different line segments.

The symbols in Fig. 33 were all compiled into a decision tree. The decision tree was breadth-pruned and also depth-pruned with depth six, i.e., only subgraphs of size six were compiled into the decision tree. The decision tree for the whole database of symbols occupied disk space of 1.242 megabyte (8.4 megabyte if no breadth-pruning was applied). Note that in the experiments with randomly generated graphs (Figs. 29 to 32) the depth-pruned decision tree representing 24 model graphs each consisting of 20 vertices and 30

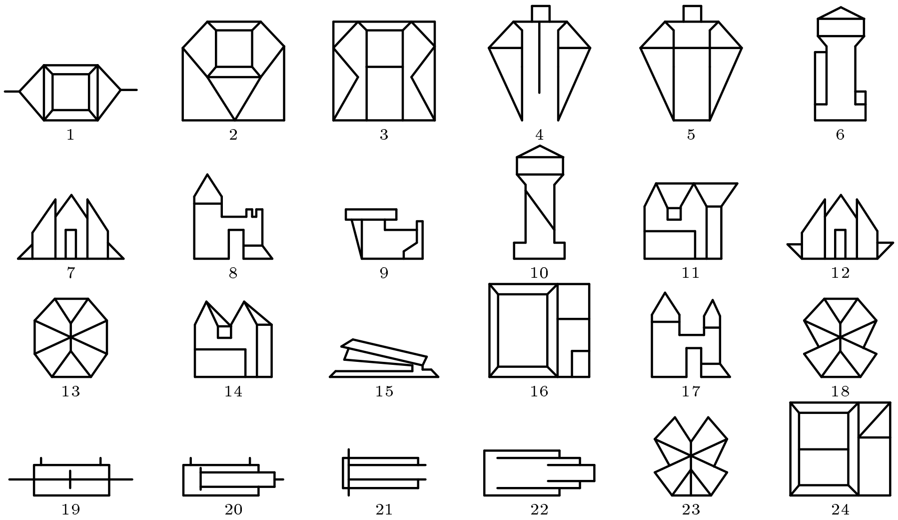


Figure 33: The database of symbols.

Symbols		1	2	3	4	5	ave.	speedup
Decision tree	seconds :	0.09	0.11	0.1	0.14	0.59	0.206	16
	steps :	1901	2362	3148	2972	4752	3027	-
Conv. Algorithm	seconds :	2.74	2.89	4.46	3.15	3.66	3.38	1
	steps :	22659	26420	117749	29558	49744	49226	-

Figure 34: Detection time and computation steps of the new and the traditional algorithm for five symbols.

edges, occupied approximately 2.2 megabyte of disk space.

We performed a number of experiments by using the symbols as line drawings and measuring the time necessary in order to detect all graph isomorphisms from the line drawing to the symbols in the database. For comparison reasons, we ran the same experiments also with the conventional algorithm. In Fig. 34 the time in seconds and the number of computational steps for both the new algorithm and the conventional algorithm are given. The last column of the table in Fig. 34 indicates that the application of the new algorithm resulted in an average speedup factor of 16 when compared to the conventional algorithm.

11 Summary and Conclusions

We have presented a new algorithm for the problem of graph and subgraph isomorphism detection that is based on the decision tree paradigm. In the computational complexity analysis, it was shown that the new algorithm has a worst case run time complexity that is only quadratic in the size of the graphs that are to be compared. Furthermore, the algorithm is completely independent of the number of model graphs that are represented by the decision tree. On the other hand, the complexity analysis also revealed that the size of the decision tree grows exponentially with the size of the represented graphs.

In order to make the new method also applicable for larger graphs, we proposed two different pruning techniques. The combination of both the breadth- and the depth-pruning technique results in the creation of very compact decision trees. Although the depth-pruned decision trees do no longer guarantuee graph isomorphism in polynomial time, they are very suitable for applications where large databases of graphs are involved. Particularly, the depth-pruned decision trees can be used to efficiently index models in the database.

The results of the theoretical analysis and the influence of the pruning techniques have been studied in a number of practical experiments with randomly generated graphs. The advantage of the new algorithm in terms of computational performance was demonstrated in these experiments for graphs with up to 19 vertices and databases containing up to 30 model graphs.

Despite the exponential complexity, we believe that there are potential applications of the new graph matching algorithm. It is particularly of interest if the underlying graphs are rather small but computation time is critical. We also believe that the general approach of using decision trees for graph matching is worth further research. Topics of future research may include the adaption of the method to the general subgraph isomorphism and to the error-tolerant subgraph isomorphism problem. Another interesting aspect will be a possible mapping of decision tree structures to neural networks as indicated in [Set90]. By such a mapping it would be possible to parallelize the decision tree and thus arrive at linear time complexity for the subgraph isomorphism detection based on the decision tree approach.

Acknowledgement

This work is part of a project of the Priority Program SPP IF, No: 5003-34285, funded by the Swiss National Science Foundation.

References

[Bab81] L. Babai. Moderately exponential bound for graph isomorphism. In F. Gecseg, editor, *Lecture Notes in Computer Science: Fundamentals of Computation Theory*, pages 34–50. Springer Verlag, 1981.

[CG70] D.G. Corneil and C.C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM*, 17:51–64, 1970.

[Eve79] S. Even. *Graph Algorithms*. Pitman Publ., 1979.

[FFG90] B. Falkenhainer, K.D. Forbus, and D. Gentner. The structure-mapping engine: Algorithms and examples. *Artificial Intelligence*, 41:1–63, 1989/90.

[Gat79] G. Gati. Further annotated bibliography on the isomorphism disease. *Journal of Graph Theory*, pages 96–109, 1979.

[GB89] T. Glauser and H. Bunke. Generation of decision trees from CAD-models for recognition tasks. In H. Burkhardt, K.H. Höhne, and B. Neumann, editors, *Mustererkennung 1989*, pages 334–340. Hamburg, 1989. (in German).

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman and Company, 1979.

[Hof82] C.M. Hoffman. *Group-theoretic Algorithms and Graph Isomorphism.* Springer Verlag, 1982.

[HS88] R. Horaud and T. Skordas. Structural matching for stereo vision. In *Proc. 9.th ICPR*, pages 439–445, 1988.

[HW74] J.E. Hopcroft and J.K. Wong. Linear time algorithm for isomorphism of planar graphs. In *Annual ACM Symposium on Theory of Computing*, pages 172–184, 1974.

[Ike87] K. Ikeuchi. Generating an interpretation tree from a CAD model for 3D-object recognition in bin-picking tasks. *International Journal of Computer Vision*, 1(2):145–165, 1987.

[Luk82] E.M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, pages 42–65, 1982.

[MB95a] B.T. Messmer and H. Bunke. Automatic learning and recognition of graphical symbols in engineering drawings. In *Proceedings Int. WorkShop on Graphics Recognition, University Park, Pennsylvania*, 1995.

[MB95b] B.T. Messmer and H. Bunke. A new method for efficient error-correcting subgraph isomorphism. In D. Dori and A. Bruckstein, editors, *Syntactic and Structural Pattern Recognition.* World Scientific Publ. Company, Singapore, to appear in 1995.

[MLL92] S.H. Myaeng and A. Lopez-Lopez. Conceptual graph matching: a flexible algorithm and experiments. *Journal of Experimental and Theoretical Artificial Intelligence*, 4:107–126, April 1992.

[Par93] S. Paris. Structural recognition using an index. In S. Impedovo, editor, *Proceedings of the 7'th Int. Conf. on Image Analysis and Processing: Progress in Image Analysis and Processing III*, pages 258–265. World Scientific, 1993.

[RC77] R.C.Read and D.G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1:339–363, 1977.

[Set90] I. K. Sethi. Entropy nets: From decision trees to neural networks. In *Neural Networks: Theoretical Foundations and Analysis*, pages 275–283. IEEE Press, 1990.

[Spi93] L. Spirkovska. Three-dimensional object recognition using similar triangles and decision trees. *Pattern Recognition*, 26(5):727–732, 1993.

[Ull76] J.R. Ullman. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23(1):31–42, 1976.