Preface

This work is concerned with aspects and problems of comparing attributed, relational graphs with each other. Several new algorithms for the comparison of graphs by means of exact and error-correcting graph or subgraph isomorphism detection are presented. The assumption common to these algorithms is that one or several of the graphs involved in the comparison process are always known before execution time. Hence, it is possible to perform a preprocessing on these graphs. The objective of such a preprocessing is to derive a representation of the a priori known graphs which allows the construction of fast run time algorithms for graph matching. Originally, the idea of improving the efficiency of graph matching algorithms by means of preprocessing came from the domain of rule-based production systems. In these systems, a possibly large number of rules must be matched with a data set. Due to the fact that many of these rules have common structures, it was proposed to compile them off-line into a compact representation (RETE-network) and use this representation at run time for the matching process instead of the rules. Naturally, this idea can also be applied to graph-based systems. There are two types of special representations for graphs presented in this thesis. The first representation, the so-called *decomposition approach*, is conceptually related to the representations used in the production systems and has been designed to solve both the exact and the error-correcting graph matching problem. The second representation, the so-called *decision tree approach*, not only tries to represent common substructures compactly, but also incorporates properties of the run time process into the preprocessing.

Chapters 2, 3, 4 and 5 of this thesis are completely self contained and can be read independently of each other. They include all necessary definitions, the formal description of the presented algorithms, computational complexity analysis and the results of the practical experiments. In Chapter 2, the decomposition approach along with a run time algorithm for exact subgraph isomorphism detection is given. In Chapter 3, the same decomposition approach is presented along with an algorithm for error-correcting subgraph isomorphism detection. In Chapter 4, the decision tree approach to exact subgraph matching is described. Finally in Chapter 5, a complete application in the domain of graphical symbol recognition based on graph matching with the decomposition approach is described. (This material was presented in 1995 at the *Workshop on Graphics Recognition* in Pennsylvania and published in the corresponding proceedings.) Notice that in each of these chapters, the most common definitions concerning attributed graphs are repeated. The references of each chapter are collected at the end of the thesis.

From the beginning of 1993 until the end of 1995 I have been working full time on the topic of graph matching algorithms. For making this possible by giving me financial support, I would like to thank the National Science Foundation of Switzerland who sponsored this work under the project No. 5003-34285 of the priority program in computer science, SPP IF. For scientific and intellectual support, I owe many thanks to the supervisor of this thesis, Prof. Horst Bunke. Most of the novel ideas presented here and many more that could not be pursued due to lack of time, originated in talks and meetings we had. Also, I would like to thank my collegues Bernard Achermann and Thomas Bebie for helpful discussions on both professional as well as social issues.

Bern, November 24, 1995 Bruno T. Messmer

Abstract

In this thesis, the problem of efficiently comparing labeled graphs to each other is studied. It is assumed that there is a set of a priori known model graphs that must be compared with a single input graph which is given only at run time. Two new approaches to this problem are presented that are based on the idea of preprocessing the model graphs off-line and deriving a special representation which supports an efficient comparison with the input graph at run time. The first approach consists of a preprocessing method in which the model graphs are decomposed off-line into smaller subgraphs and the subgraphs are recorded in a special decomposition representation. In this decomposition subgraphs that appear multiple times within the same or in different model graphs are represented only once. Hence, at run time, these subgraphs must be compared only once to the input graph. Based on this decomposition representation two algorithms for comparing the model graphs to the input graph are proposed. The first algorithm detects all exact graph and subgraph isomorphisms from any of the models to the input graph in time that is only sublinear in the number of the model graphs. The second algorithm is designed for the detection of error-correcting graph and subgraph isomorphisms from any of the models to the input graph. In addition to being only sublinearly dependent on the number of model graphs, this algorithm can be combined with a future error estimation scheme that greatly improves the run time performance. The superiority of both algorithms over conventional methods is shown in a computational complexity analysis as well as in practical experiments. The second approach presented in this thesis is based on the idea of deriving a decision tree from the model graphs. At run time, the decision tree is used to detect exact graph and subgraph isomorphisms from the input graph to the model graphs in polynomial time. Moreover, the graph or subgraph isomorphism detection based on the decision tree is completely independent of the number of model graphs that are represented by the decision tree. However, the drawback of this approach is that the decision tree may become exponentially large. Hence, a number of pruning techniques that aim at reducing the size of the decision tree are proposed. Finally, an application of the proposed new algorithms for graph matching in the domain of graphical symbol recognition is described.

Contents

Chapter 1

Introduction

1.1 The Problem of Graph Matching

Graphs and, in particular, labeled or attributed graphs are a very powerful and universal tool that is widely used in computer applications. Depending on the requirements of an application, there are numerous methods for the examination and analysis of graphs such as finding the shortest path, detecting Hamiltonian cycles, coloring the vertices of a graph and many more [Deo74, Rea79, Eve79]. One of the most important problems when using graphs is the comparison of graphs with each other. For example, in computer vision, when graphs are used for the representation of 3-D objects, the recognition of known model objects in an unknown scene can be done by trying to find correspondences between the graphs representing the models and the graph representing the scene. Trying to find such correspondences is generally referred to as graph matching. There are three different classes of graph matching that will be considered in this report, namely graph isomorphism, subgraph isomorphism, and error-correcting subgraph isomorphism. In the following, an informal description of each of the three graph matching classes is given.

When matching two graphs G_1 and G_2 by means of *graph isomorphism* we are looking for a bijective mapping between the vertices of G_1 and G_2 such that the structure of the edges is preserved by the mapping function. When such a mapping function can be found then G_1 and G_2 are isomorphic. If one of the graphs involved in the matching process is larger than the other, i.e. G_2 contains more vertices than G_1, then we are looking for a *subgraph isomorphism* from G_1 to G_2. That is, we are interested in finding a subgraph S of G_2 such that G_1 and S are isomorphic. Finally, in many applications, the encoding of objects as attributed graphs will not be perfect due to the presence of noise and distortions. Hence, it is necessary to introduce an error model and incorporate the concept of errors into graph matching. The graphs are then compared to each other by means of *error-correcting subgraph isomorphism.* Notice that the definition of an error model is strongly application dependent.

One of the drawbacks of graph matching is its computational complexity. For the graph isomorphism problem it is up to this day an open question whether it belongs to the complexity class P or NP [GJ79, BC79]. All algorithms that have been developed so far for the general graph isomorphism problem require in the worst case exponential time. For the subgraph isomorphism problem and also the error-correcting subgraph isomorphism problem it is well known that it is NP-complete [GJ79]. Consequently, no algorithm could be constructed that guarantees to find (error-correcting) subgraph isomorphisms in polynomial time. However, research in the past twenty years has shown that there are methods for graph matching that behave reasonably well on the average in terms of performance and become computationally intractable only in a few cases. Moreover, if the constraints of graph matching are loosend, then it is possible to find solutions in polynomial time by using approximate algorithms. In the following, an overview of the methods for graph, subgraph and error-correcting subgraph isomorphism detection that have been proposed by various authors in the past is given. (Notice that some of the references will be repeated in the introductions to Chapters 2, 3 and 4.)

The graph isomorphism problem has been the focus of intensive research for over three decades [Gat79, RC77]. Due to the fact that it is not yet known whether the graph isomorphism problem is in P or in NP, there are basically two approaches that have been taken in order to find an efficient algorithm. The first approach is based on group-theoretic concepts and aims at classifying the adjacency matrices of graphs into permutation groups. With this it was possible to prove that there exists a moderately exponential bound for the general graph isomorphism problem [Bab81]. Furthermore, by imposing certain restrictions on 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 shown in [Hof82, Luk82] that algorithms with a computational complexity of $O(n^4)$ exist. In [HW74] a method for the computation of the isomorphism of planar graphs is proposed that requires time that is only linear in the size of the graphs. Although these methods are very interesting from a theoretical point of view, they are usually not applicable in practice due to a large constant overhead. For this reason, the second approach is more practically oriented. It aims directly at constructing graph isomorphisms in a procedural manner. Notice that all of the following methods apply to both graph and subgraph isomorphism detection. One of the best known methods for graph and subgraph isomorphism detection is based on depth-first backtracking search, first described in [CG70]. Informally speaking, the method works as follows. Given two graphs G_1 and G_2, the vertices of G_1 are mapped one after the other onto the vertices of G_2 and after each mapping it is checked whether the edge structure of G_1 is preserved in G_2 by the mapping. If all vertices of G_1 are successfully mapped onto vertices of G_2 and G_1 and G_2 are of equal size then a graph isomorphism is found. If G_1 is smaller than G_2 then a subgraph isomorphism from G_1 to G_2 is found. Although this method performs well for small graphs, the number of required steps explodes combinatorially when the graphs grow. Hence, Ullman proposed in [Ull76] to combine backtracking with a forward checking procedure which greatly reduces the number of backtracking steps. For a detailed description of Ullman's algorithm see Appendix A.2. A comprehensive analysis of the performance of different forward-checking and looking-ahead procedures for backtracking is given in [HE80]. Similar to forward-checking procedures, discrete relaxation methods aim at gradually reducing the number of possible mappings for each vertex by only allowing vertex to vertex mappings that are locally consistent [KR79, KK91]. The advantage of discrete relaxation methods is that they can be easily parallelized. However, because only local consistency is checked, ambiguities must be resolved in the end by applying again a backtracking procedure. Another interesting approach to graph and subgraph isomorphism detection is described in [FFG90, MLL92, HS89]. It is based on the idea of building a so-called association graph in which each consistent vertex to vertex mapping is represented by a node and each locally consistent pair of vertex to vertex mappings is represented by an edge between the corresponding nodes. Graph or subgraph isomorphisms are found by searching for maximal cliques in the association graph. In Appendix A.1 a description and a computational complexity analysis of graph matching by clique detection is given.

All of the algorithms described so far are optimal algorithms. That is, they are guaranteed to find all graph and subgraph isomorphisms from a given graph G_1 to another graph G_2. The main problem with these algorithms, however, is that for large graphs they may require exponential time. Approximate or continuous optimization algorithms, on the other hand, do not always find the optimal solution but require only polynomial time. In [HHVN90] an approximate algorithm based on relaxation and simulated annealing is presented. In [JS89, BHS89, FZ91] it is proposed to encode sequences of vertex to vertex mappings as chromosomes. A genetic algorithm is then used to optimize the pool of chromosomes such that the encoded vertex to vertex mappings represent perfect or close to perfect graph or subgraph isomorphisms. One of the advantages of approximate methods is that they can also be applied in noisy environments. For example, given a graph G_1 that represents a known model and another graph G_2 that represents a distorted image of the known model, i.e., some edges in G_1 are not present in G_2, a method based on genetic algorithms or simulated annealing will not simply fail (as it is the case with optimal methods) but generate solutions that are as close as possible to graph or subgraph isomorphism. Hence, these approximate methods are a first step towards algorithms for error-correcting subgraph isomorphism detection.

One of the requirements of error-correcting subgraph isomorphism is the definition of the errors that are to be taken into account. Probably the best known error correction model for graph matching is similar to the model used in string edit distance [WF74]. It is based on the idea of introducing graph edit operations. For each possible error type a corresponding graph edit operation is defined. In order to model the fact that certain error types are more likely than others, cost functions are assigned to the edit operations. The definition of the cost functions is strongly application dependent. The graph edit operations are then used to correct errors in the graphs. Thus, informally speaking, an error-correcting subgraph isomorphism is defined as the sequence of edit operations with minimal cost that must be applied to one of the graphs such that a subgraph isomorphism exists. As in the case of exact subgraph isomorphism detection, there are optimal and approximate algorithms for error-correcting subgraph isomorphism detection. In the problem of error-correcting subgraph isomorphism an algorithm is optimal if it is guaranteed to find the sequence of edit operations with minimal costs such that a subgraph isomorphism exists. Most of the optimal algorithms proposed so far are based on the A^* algorithm [Nil80]. Given a graph G_1 and a possibly distorted input graph G_2, a search tree is expanded such that each state in the tree corresponds to a partial mapping of vertices in G_1 onto vertices in G_2. At the top of the search tree, the first vertex of G_1 is mapped onto every vertex of G_2. Each such mapping and the corresponding cost of the edit operation is a state in search tree. The generation of successor states is then guided by the cost of the edit operations. That is, the vertex mapping with the least cost is extended by mapping a new vertex of G_1 onto every vertex of G_2 that has not yet been used. Eventually, all vertices of G_1 are mapped onto vertices of G_2 and an error-correcting subgraph isomorphism is found. A formal description of an error-correcting subgraph isomorphism algorithm based on A^* is given in Appendix A.3. The performance of such an algorithm strongly depends on the number of states that are expanded in the search tree. By introducing a heuristic future cost estimation function, the size of the search tree can be greatly reduced. In [Won90, TF79, EF84, SF83, SH81] various heuristic functions have been proposed for A^* based error-correcting subgraph isomorphism detection.

Similar to exact subgraph isomorphism detection, the problem with optimal algorithms for error-correcting subgraph isomorphism detection is that they require exponential time in the worst case. Furthermore, the search tree may also exponentially grow for large graphs. Hence, approximate methods must be applied. In [KCP92, CKP95] a method based on probabilistic relaxation is described. The basic idea is that each vertex to vertex mapping is assigned a certain probability that reflects the edit cost and the local consistency of the mapping. Similar to discrete relaxation, the probabilities of each mapping are then corrected until a maximum probability for a set of vertex to vertex mappings results. The method was tested on fairly large graphs and interesting results were obtained. However, as expected, the optimal solution was missed in some cases. Another continuous optimization approach is based on neural networks. In [FLD94, MF90] it was proposed to solve the error-correcting subgraph isomorphism problem by representing each vertex to vertex mapping by a neuron in a Hopfield network and optimize the output of the network. Another method using the Kohonen network was also presented in [XO90]. However, the problem with neural network approaches to graph matching is that the optimal solution may be missed due to the fact that the network stabilizes in a local minimum. A special case of error-correcting graph matching is the weighted graph matching problem in which two completely connected graphs of equal size with weights assigned to the edges must be matched onto each other such that the weight differences in the edges are minimized. A linear programming method providing an approximate solution to this problem is presented in [AD93]. The method was originally designed for problems in the domain of operations research. Finally, in [Ume88] an algorithm based on the eigen-decomposition of the adjacency matrices of the weighted graphs is proposed. While this method is very fast, it can only be applied to graphs with completely different eigen-values. Furthermore, only small distortions in the input graph can be handled.

There are numerous applications in which exact or error-correcting graph matching plays a relevant role. In fact, many of the algorithms described above have been developed with a particular application in mind. One of the earliest applications of exact subgraph isomorphism detection was in the field of chemical documentation and the analysis of chemical structures [RB79]. More recently, graph matching has also been proposed for the retrieval of cases in case-based reasoning [BL88, Poo93] and for the analysis of semantic networks in combination with graph grammars [Ehr92]. In machine learning graph matching is used for the learning common substructures of different concepts [BM88, Fis90, CH94]. However, most applications of graph matching have been documented in the fields of pattern recognition and computer vision. For example, in pattern recognition subgraph isomorphism detection was successfully applied to Chinese character recognition [LRS91], the interpretation of schematic diagrams [BA83, LKG90, MB96], seal verification [LK89] or it was combined with evidence based systems for shape analysis [PCB94]. In computer vision, it was mainly used for the localization and identification of 3-D objects [GB90, HS88, Won90, CH81, WLR89, CK92, Won92].

There are two major problems that practical applications using exact or error-correcting graph matching are confronted with. The first problem is the computational complexity of graph matching. As mentioned before, the time required by any of the optimal algorithms listed above may in the worst case become exponential in the size of the graphs. The approximate algorithms, on the other hand, have only polynomial time complexity, but do not guarantee to find the optimal solution. For some of the applications in the previous paragraph, this may not be acceptable. The second problem is the fact that all of the algorithms for graph matching mentioned so far can only be applied to two graphs at a time. Therefore, if there is more than one model graph that must matched with an input graph (as, for example, in many computer vision applications), then the conventional graph matching algorithms must be applied to each model-input pair sequentially. As a consequence, the performance is linearly dependent on the size of the database of model graphs. For applications dealing with large databases, this may be prohibitive.

1.2 New Approaches to Graph Matching

In this thesis, we propose two new approaches to graph matching that are particularly efficient with regard to the problems mentioned at the end of the previous section. There is a common assumption and a common idea behind both approaches. The assumption is that the graph matching problem always involves one or several graphs, so-called model graphs, that are a priori known and a so-called input graph that becomes available only at run time. That is, the objective of the graph matching process will be to find a match from each of the a priori known model graphs to the input graph. The common idea of both approaches is to compute a special representation of the model graphs in a preprocessing step. The aim of the preprocessing step is to gather information about the model graphs that can be used at run time to speed up the actual graph matching process.

The first approach was especially designed for applications dealing with large databases of model graphs. It is based on the idea of compiling the model graphs off-line into a compact representation. This representation is then used at run time in order to efficiently detect exact or error-correcting subgraph isomorphisms from the model graphs to the input graph. For the generation of the compact representation, the model graphs are decomposed into smaller subgraphs and these subgraphs are recorded in a special decomposition structure. In this decomposition structure, subgraphs that appear multiple times in the same or different model graphs are represented exactly once. Hence, at run time, these subgraphs are matched only once onto the input graph. Based on this representation of the model graphs, two new algorithms for graph matching are proposed. The first algorithm finds all exact graph or subgraph isomorphisms from the model graphs to the input graph. The second algorithm detects all error-correcting graph or subgraph isomorphisms from the model graphs to the input graph. Both algorithms work according to the same principle. Given an input graph and the compact representation of the model graphs, (error-correcting) subgraph isomorphisms from the subgraphs of the model graphs to the input graph are first searched for. Then, the (error-correcting) subgraph isomorphisms that are found for the small subgraphs are combined into (error-correcting) subgraph isomorphisms for the complete model graphs. Clearly, the advantage of using the compact representation instead of the model graphs themselves is that the computational effort of finding (error-correcting) subgraph isomorphism from the common subgraphs to the input graph must be done only once. As a consequence, the complexity of both the exact and the error-correcting algorithm is only sublinearly dependent on the number of the model graphs. In fact, if the model graphs are very similar or even identical, the computational complexity becomes independent of the number of model graphs. Furthermore, the algorithm for error-correcting subgraph isomorphism detection can be combined with a future cost estimation procedure that works on the decomposition representation of the model graphs. The application of this new estimation procedure greatly improves the performance of the error-correcting subgraph isomorphism algorithm. The computational complexity analysis of both algorithms shows that they are superior to conventional algorithms for large databases of graphs and, in the case of the error-correcting algorithm, also for single model graphs. These results are confirmed in a number of practical experiments.

The second approach was designed for applications that are time critical and require real time or near real time performance of the graph matching process. It is based on the idea of deriving a decision tree from the model graphs. In this decision tree, all permutations of the adjacency matrices of the model graphs are represented. At run time, the decision tree is used to classify the adjacency matrix of an input graph by looking up its rows and columns and traversing the decision tree accordingly. If the adjacency matrix can be classified successfully then a graph or a subgraph isomorphism from the input graph to one of the model graphs has been found. The traversal of the decision tree can be performed in a number of steps that is only quadratic in the size of the input graph. Furthermore, it is completely independent of the number of the model graphs that are represented by the decision tree. Hence, the subgraph isomorphism algorithm based on the decision tree approach has a worst case time complexity that is only quadratic in the number of vertices of the input graph and is completely independent on the size of the database. The drawback of this approach, however, is the size of the decision tree. It is exponential in the number of vertices of the model graphs. In order to reduce the size of the decision tree, various pruning techniques are proposed. The efficiency of the decision tree based algorithm and the pruning techniques is demonstrated in a number of practical experiments.

1.3 Organization

This thesis is organized in the following manner.

In **Chapter 2**, a new algorithm for the problem of exact graph and subgraph isomorphism detection from the models to the input graph is presented. After an introduction, in which a brief review of the most common traditional algorithms is given, the concepts of attributed graphs and exact graph and subgraph isomorphism are defined. Next, the off-line decomposition of the model graphs and the corresponding compact representation is presented. Based on this representation, the new algorithm NA for exact graph and subgraph isomorphism detection is given. The properties of this algorithm are studied both in a theoretical complexity analysis and in practical experiments.

In **Chapter 3**, a new algorithm for the problem of error-correcting graph and subgraph isomorphism detection is proposed. The algorithm is based on the decomposition structure that is introduced in Chapter 2. First, various conventional methods for error-correcting graph matching and practical applications of these methods are reviewed. Then, after a formal definition of error-corrections and error-correcting subgraph isomorphism, the new algorithm *NSG* is presented. In the computational complexity analysis it is shown that *NSG* outperforms the conventional A^*-based approach. This result is confirmed by practical experiments.

In **Chapter 4**, a decision tree based algorithm for the problem of exact graph and subgraph isomorphism detection from an input graph to a set of a priori known model graphs is presented. In the introductory section, the advantages and disadvantages of conventional algorithms and especially group-theoretic approaches are examined. Next, the construction of the decision tree from the model graphs is explained. Based on this decision tree, the new polynomial algorithm for graph and subgraph isomorphism detection is introduced and its properties are studied both theoretically and practically.

In **Chapter 5**, a practical application of the algorithms given in Chapters 2 and 3 is described. The purpose of the application is to recognize and automatically learn graphical symbols in engineering drawings. First, the representation of the graphical symbols and drawings as attributed graphs is given. Then, the symbol recognition process is formulated as a search for error-correcting subgraph isomorphisms from the symbol graphs to the drawing graph. For the automatic acquisition of new symbols, a learning scheme is proposed which is based on the decomposition based algorithm for error-correcting subgraph isomorphism detection. Finally, the system is tested on a number of sample scenes.

In **Chapter 6** the advantages and disadvantages of the algorithms presented in Chapters 2, 3 and 4 are summarized and conclusions are drawn. Furthermore, future research directions are discussed.

There are two appendices to this thesis. In **Appendix A** three conventional algorithms for graph matching are described along with a detailed computational complexity analysis. The first algorithm is based on the idea of maximum clique detection and can be applied for the largest common subgraph problem as well as the exact graph and subgraph isomorphism problem. The second algorithm is Ullman's algorithm for exact graph and subgraph isomorphism detection. The third algorithm is an A^*-based method for the detection of error-correcting subgraph isomorphisms. This algorithm is enhanced by a future error estimation function specifically developed for graphs.

Appendix B is the manual to the graph matching toolkit *GUB* which incorporates the new algorithms. The installation and the application of the C++ classes and methods in the toolkit are explained in detail.

Chapter 2

Efficient Subgraph Isomorphism Detection - A Decomposition Approach

2.1 Introduction

Graphs are a powerful and universal data structure useful in various subfields of science and engineering. When graphs are used for the representation of objects, the problem of comparing different objects to each other can be formulated as the search for correspondences between the attributed graphs representing the objects. Such correspondences can be established by subgraph isomorphism detection.

The concept of subgraph isomorphism detection has been applied in a variety of fields. For example, it has been used in chemical documentation systems for the comparison of molecular structures [RB79], in case-based-reasoning for the retrieval of cases from the case base [BL88, Poo93], in semantic networks in combination with graph grammars [Ehr92], or in machine learning, where it is used in order to learn common substructures of different concepts [BM88, Fis90, CH94, MB96]. In pattern recognition, subgraph isomorphism detection was applied to Chinese character recognition [LRS91], the interpretation of schematic diagrams [Bun82, LKG90] and seal verification [LK89]. It was also combined with evidence based systems for shape analysis [PCB94]. In computer vision, subgraph isomorphism detection was used for the localization of 3D objects in images [GB90, HS88, CH81, WLR89, CK92] and in robot vision [Won92]. The main problem with subgraph isomorphism detection is the fact that it is an NP-complete problem [GJ79]. In other words, the time to detect a subgraph isomorphism between two graphs is in the worst case exponential in the number of vertices of these graphs. In the following, we give a short review of methods for subgraph isomorphism detection that have been proposed in the past.

The most common technique to establish a subgraph isomorphism is based on backtracking in a search tree. In order to prevent the search tree from growing unnecessarily large, different refinement procedures such as the one by Ullman [Ull76], forward-checking and looking-ahead [HE80], or discrete relaxation [KK91] have been proposed. These techniques are fairly stable and perform well in most cases. Another approach is taken in [FFG90, MLL92], where each possible mapping of a vertex of the first graph onto a vertex of the second graph is recorded in an association graph and the detection of a possible graph match is performed by maximal clique detection. Finally, in [Bla94] it is proposed to partition the graphs according to lattice theory in order to reduce the computational complexity of the subgraph isomorphism problem. All these methods provide an optimal solution to the graph matching problem, but may in the worst case become computationally intractable.

Continuous optimizations methods on the other hand aim at providing a solution within reasonable time. However, they may not always find the optimal solution, i.e., the mapping of model graph vertices to input graph vertices found by a continuous optimization method not necessarily represents a subgraph isomorphism. In [KU88] the advantages and disadvantages of continuous optimizations methods such as neural networks compared to the optimal backtracking methods are examined. Other continuous optimization approaches include the application of simulated annealing [HHVN90], genetic algorithms [JS89, BHS89, FZ91] and probabilistic relaxation [CKP95].

The methods for subgraph isomorphism detection mentioned so far work on only two graphs at a time. However, in many applications there is more than one model graph that must be matched with the input graph. Consequently, it is necessary to apply the subgraph isomorphism algorithm to each model-input pair, resulting in a computation time that is linearly dependent on the size of the database. This dependency may become prohibitive if the number of model graphs is large. Hence, some form of organization or indexing on the database of model graphs is needed. In [SH82, SKP93, SH92, BR92] a hierarchical organization of the database is proposed which clusters the model graphs into similarity classes. The given input graph is then used to index the database of model graphs by traversing the hierarchy. However, such an indexing can only work if the input graph is isomorphic to one of the model graphs. If the input graph is larger than the models or contains more than one instance of a model graph, the indexing process will not work. Another hierarchical approach which does not have this drawback was proposed in [SB95], where at the root of the hierarchy there is a supergraph, consisting of different subgraphs of the model graphs. At run time, the input graph is matched with the supergraph and the hierarchy is traversed according to this initial match. The disadvantage of this method, however, is that the root graph may become much larger than the individual model graphs and thus the first matching process may be more time consuming than the sum of each individual subgraph isomorphism detection process between a model graph and the input. Finally, some interesting approaches have been applied in the domain of computer vision, where multi-level indexes are computed by precalculating all possible views of an object into a view-description network [Par93, Lev92]. This network can then be used to efficiently index the database of model graphs However, the scheme has not yet been generalized to arbitrary graphs.

Many of the reviewed algorithms above have interesting properties. However, no technique has been described, which solves the problem of subgraph isomorphism detection and the organization of large graph databases at the same time for general labeled graphs. In this chapter, we propose a new approach to the problem of subgraph isomorphism detection from a set of model graphs to an input graph. Our algorithm is somewhat similar to the RETE algorithm for forward-chaining rule-based systems [For82]. It is based on a compact representation of the model graphs. The representation is created off-line by decomposing the model graphs into a set of subgraphs. These subgraphs are the basic elements of the new representation. If a subgraph in the decomposition appears multiple times in the same or in different model graphs, it is represented only once. At run time, the new representation is used to efficiently detect subgraph isomorphisms from the models to the input graph in the following manner. First, subgraph isomorphisms for the subgraphs in the representation are detected. These subgraph isomorphisms are then recursively combined to form subgraph isomorphisms for the complete model graphs. Due to the fact that common subgraphs of different models are represented only once, they are matched exactly once with the input graph. Thus, the new algorithm is only sublinearly dependent on the number of the model graphs. The behavior of the new algorithm is studied both theoretically in a computational complexity analysis and practically in a number of experiments with randomly generated graphs.

This chapter is organized in the following manner. In Section 2.2 we provide basic definitions and notations. In Section 2.3, the new algorithm is described in detail. In Section 2.4, the computational complexity of the new algorithm is analyzed. The results of the theoretical complexity are confirmed in a number of practical experiments documented in Section 2.5. Finally, we conclude the chapter with a summary and some remarks on the applicability of the new algorithm in various domains. In the appendix, Ullman's algorithm which is used as a benchmark in this paper is briefly described, along with a computational complexity analysis.

2.2 Definitions and Notations

The algorithms presented in this chapter work on labeled graphs. Let L_V and L_E denote the set of vertex and edge labels, respectively.

Definition 2.1: A *graph* G is a 4-tuple $G = (V, E, \mu, \nu)$, 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.

□

In this definition, the edges are directed, i.e. there is an edge from v_1 to v_2 if $(v_1, v_2) \in E$. For graphs with undirected edges, we require $(v_2, v_1) \in E$ for any edge $(v_1, v_2) \in E$. The empty graph, i.e. the graph with an empty set of vertices, will be denoted by $\emptyset$.

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

1. $V_s \subseteq V$
2. $E_s = E \cap (V_s \times V_s)$
3. μ_s and ν_s are the restrictions of μ and ν to V_s and E_s, respectively, i.e.,

$$\mu_s(v) = \begin{cases} \mu(v) & \text{if } v \in V_s \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\nu_s(e) = \begin{cases} \nu(e) & \text{if } e \in E_s \\ \text{undefined} & \text{otherwise} \end{cases}$$

□

From this definition it is easy to see that, given a graph G, any subset of its vertices uniquely defines a subgraph of G. We use the notation $S \subseteq G$ to indicate that S is a subgraph of G.

Definition 2.3: Given a graph $G = (V, E, \mu, \nu)$ and a subgraph $S = (V_s, E_s, \mu_s, \nu_s)$ of G, the *difference* of G and S is the subgraph of G that is defined by the set of vertices $V - V_s$. We denote the difference of G and S by $G - S$. □

Definition 2.4: Given two graphs $G_1 = (V_1, E_1, \mu_1, \nu_1)$, $G_2 = (V_2, E_2, \mu_2, \nu_2)$, where $V_1 \cap V_2 = \emptyset$, and a set of edges $E' \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$ with a labeling function $\nu : E' \rightarrow L_E$, the *union* of G_1 and G_2 with respect to E' is the graph $G = (V, E, \mu, \nu)$ such that

1. $V = V_1 \cup V_2$

2. $E = E_1 \cup E_2 \cup E'$

3. $\mu(v) = \begin{cases} \mu_1(v) & \text{if } v \in V_1 \\ \mu_2(v) & \text{if } v \in V_2 \end{cases}$

4. $\nu(e) = \begin{cases} \nu_1(e) & \text{if } e \in E_1 \\ \nu_2(e) & \text{if } e \in E_2 \\ \nu(e) & \text{if } e \in E' \end{cases}$

□

The union of two graphs G_1 and G_2 with respect to a set of edges E according to Def. 2.4 will be denoted by $G_1 \cup_E G_2$.

Definition 2.5: A bijective function $f : V \to V'$ is a *graph isomorphism* from a graph $G = (V, E, \mu, \nu)$ to a graph $G' = (V', E', \mu', \nu')$ if:

1. $\mu(v) = \mu'(f(v))$ for all $v \in V$.
2. For any edge $e = (v_1, v_2) \in E$ there exists an edge $e' = (f(v_1), f(v_2)) \in E'$ such that $\nu(e) = \nu(e')$, and for any $e' = (v'_1, v'_2) \in E'$ there exists an edge $e = (f^{-1}(v'_1), f^{-1}(v'_2)) \in E$ such that $\nu(e') = \nu(e)$.

□

Definition 2.6: An injective function $f : V \to V'$ is a *subgraph isomorphism* from G to G' if there exists a subgraph $S \subseteq G'$ such that f is a graph isomorphism from G to S. □

Notice that graph isomorphism is a special case of subgraph isomorphism. For the remainder of this chapter, we will assume that there is a number of a priori known graphs, the so-called *model graphs*, and an *input graph* that is given on-line. The input and model graphs will be also referred to as input and models, for short. The problem to be solved is to find all subgraph isomorphisms from the models to the input.

2.3 Decomposition Based Subgraph Isomorphism

2.3.1 Overview of the Method

Given a set of model graphs $G_1, \ldots, G_L$ and an input graph G_I, we want to find all subgraph isomorphisms from any of the models to the input graph. Under a naive strategy, we would match the input graph sequentially to each model using, for example, Ullman's algorithm. The main disadvantage of this approach is that it is linearly dependent on the number of model graphs. Moreover, it is inefficient if different model graphs have common substructures, because these substructures will be matched with the input graph for each model repeatedly. In order to overcome this inefficiency, we propose a different approach.

Instead of matching each model graph individually onto the input graph, we recursively decompose the model graphs off-line into smaller subgraphs. At run time, these subgraphs are matched onto the input graph, and all detected subgraph isomorphisms are combined to form subgraph isomorphisms for complete model graphs. This idea is similar to the RETE matching algorithm for forward chaining production systems [For82, LS92]. The main advantage of this scheme is that subgraphs that appear multiple times in the same or in different model graphs must be matched only once onto the input. Consequently, the corresponding subgraph isomorphism detection process will be more efficient than the sequential matching of the input graph with each of the models.

The new approach consists of two parts. First, there is an off-line process in which the model graphs are recursively decomposed and the resulting subgraphs are represented by a special data structure. The second part is an on-line process, in which an input graph is matched with the model graphs, which are represented by the data structure generated in the first step. In the following, we first describe the off-line decomposition of the model graphs and the data structure for their representation. Next, the new subgraph isomorphism algorithm that is based on this representation and an example will be given.

2.3.2 Decomposing the Model Graphs

The main idea of the new approach is to recursively decompose the model graphs into smaller subgraphs in an off-line processing step. At run time, the subgraph isomorphism problem is solved in a divide-and-conquer fashion. That is, we first look for subgraph occurrences of parts of the models in the input graph. All such occurrences are then successively combined to form subgraph isomorphisms for the complete models.

Definition 2.7: Let $B = \{G_1, \dots, G_L\}$ be a set of model graph. A *decomposition* of B, $D(B)$, is a finite set of 4-tuples (G, G', G'', E), where

1. G, G' and G'' are graphs with $G' \subset G$ and $G'' \subset G$
2. E is a set of edges such that $G = G' \cup_E G''$
3. For each G_i there exists a 4-tuple $(G, G', G'', E) \in D(B)$ with $G = G_i$; $i = 1, \dots, L$.
4. For each 4-tuple $(G, G', G'', E) \in D(B)$ there exists no other 4-tuple $(G_1, G_1', G_1'', E_1) \in D(B)$ with $G = G_1$.

5. For each 4-tuple $(G, G', G'', E_1) \in D(B)$

   (a) if G' consist of more than one vertex then there exists a 4-tuple $(G_1, G'_1, G''_1, E_1) \in D(B)$ such that $G' = G_1$

   (b) if G'' consists of more than one vertex then there exists a 4-tuple $(G_2, G'_2, G''_2, E_2) \in D(B)$ such that $G'' = G_2$

   (c) if G' consists of one vertex then there exists no 4-tuple $(G_3, G'_3, G''_3, E_3) \in D(B)$ such that $G' = G_3$

   (d) if G'' consists of one vertex then there exists no 4-tuple $(G_4, G'_4, G''_4, E_4) \in D(B)$ such that $G'' = G_4$.

$\square$

Informally speaking, a decomposition is a recursive partitioning of graphs into smaller subgraphs, starting with complete models and terminating at the level of single vertices. The first component in a 4-tuple (G, G', G'', E) is the graph to be decomposed, G' and G'' are its two parts, and E are the edges in G between G' and G'' (see Conditions 1 and 2 in Def. 2.7). Condition 3 in Def. 2.7 makes sure that every model in B is decomposed, and Condition 4 implies that a decomposition is unique. By means of Condition 5 it is guaranteed that a decomposition is complete, i.e. the process of partitioning a graph into two parts is continued until individual vertices are reached. If several models $G_i, G_j, \ldots$ have a common subgraph G, or if G occurs multiple times in one model G_i, it is sufficient to represent G just by one 4-tuple (G, G', G'', E) in $D(B)$. This property not only leads to a compact representation of a set of models, B, by means of the decomposition $D(B)$, but it also is the key to an efficient matching procedure at run time.

The decomposition of a set of models will be used to guide the search for subgraph isomorphisms from the models to the input. If there is a 4-tuple (G, G', G'', E) in $D(B)$, then subgraph isomorphisms from G' and G'' to the input will be searched for first. Once such subgraph isomorphisms have been found, they will be combined, whenever possible, into subgraph isomorphisms from G to the input. This procedure is started with subgraphs G' and G'' that consist of single vertices only, and is recursively continued until the level is reached where G represents a complete model.

Apparently, there exist many different decompositions for a given set of models. This property holds even in the case where the set of models consists of only a single graph. One could now define an optimal decomposition being, for example, one that contains the minimum number of 4-tuples, or one where the largest subgraph G that occurs in all models is represented by a 4-tuple (G, G', G'', E). However, the computation of such an optimal decomposition is a highly exponential problem [CH94]. In this chapter we propose a decomposition algorithm which usually does not generate an optimal decomposition but is computationally inexpensive.

DECOMPOSITION(B)

1. let $B = \{G_1, \ldots, G_L\}$ and $D(B) = \emptyset$
2. for $i = 1$ to L
   decompose(G_i)

Figure 2.1: Algorithm *decomposition*.

In Fig. 2.1 the algorithm *decomposition* is displayed. The input to the algorithm is a set B of models that are to be decomposed and represented by the decomposition $D(B)$. In the beginning, $D(B)$ is empty. The basic idea is to sequentially consider one model after the other and to decompose each model G such that subgraphs of previously decomposed model graphs are being reused for the decomposition of G. For this purpose, the procedure *decompose* given in Fig. 2.2 is called sequentially for each model graph G. Note that the decomposition $D(B)$ – or D if B is not explicitly mentioned – is considered a global variable which retains its contents for each call to *decompose*. The task of the procedure *decompose* is to find the largest subgraph S_{max} in the model graph G that is already represented in D[1]. If S_{max} is isomorphic to G then G is already represented in D and the algorithm exits. If G consists of a single vertex only, it cannot be decomposed any further and the algorithm exits. Otherwise, G is decomposed into S_{max} and $G - S_{max}$. Clearly, S_{max} has been previously treated by the algorithm and hence only $G - S_{max}$ must be further decomposed by calling the algorithm recursively. If at some point in the recursion, no subgraph of G is already represented by $D(B)$, we randomly choose a subgraph S_{max} of G, for example one that consists of half the vertices of G, for further decomposition. Finally, the tuple $(G, S_{max}, G - S_{max}, E)$ is a added to D.

Although the algorithm *decomposition* will usually not generate an optimal decomposition, it was shown in practical experiments that this has no significant influence on the performance of the run time algorithm (see Section 2.5). Very important, however, is the fact that the algorithm *decomposition* is incremental, i.e., given a set B of model graphs that are represented by the decomposition $D(B)$, a new model graph G_{L+1} can be added to the database by simply calling *decompose(G_{L+1})*. Thus, $D(B)$ can be updated incrementally without the need for a complete recomputation of $D(B)$ from scratch. This is particularly of interest in applications where

[1]In order to find the largest subgraph in G that is already represented, it is necessary to apply a subgraph isomorphism algorithm. As the decomposition is an off-line process, some conventional algorithm such as Ullman's algorithm may be used. However, for the complexity analysis in Section 2.4, we will assume that the new algorithm *NA* described in Section 2.3.3 is applied in the decomposition process.

DECOMPOSE(G)

1. let D be a decomposition and $S_{max} = \emptyset$.
2. if G consists of only one vertex then exit.
3. for all (G_i, G'_i, G''_i, E_i)
   if G_i is a subgraph of G and S_{max} is smaller than G_i then let $S_{max} = G$.
4. if S_{max} is isomorphic to G then exit.
5. if no subgraph S_{max} was found and G consists of more than one vertex then
   (a) choose randomly a subgraph S_{max} of G.
   (b) decompose(S_{max})
6. decompose($G - S_{max}$)
7. add $(G, S_{max}, G - S_{max}, E)$ to D where E is the set of edges between S_{max} and $G - S_{max}$ in G.

Figure 2.2: Procedure *decompose*.

large databases of graphs are involved and new model graphs must be added to the database at run time.

2.3.3 Subgraph Isomorphism Based on Graph Decomposition

The decomposition of a set of model graphs presented in the previous section is the basis for an efficient algorithm that detects subgraph isomorphisms from a set of model graphs to an input. Instead of matching each model individually onto a given input, the new algorithm first finds all occurrences of the individual vertices of the model in an input graph. These occurrences are then recursively merged into larger structures until the level of complete model graphs is reached. There are two basic problems that must be solved in this scheme. First, as the smallest component of a graph is a single vertex, there must be a procedure for the detection of subgraph isomorphisms from single vertices to an input graph. Secondly, given a decomposition $D(B)$ and a 4-tuple $(G, G', G'', E) \in D(B)$, if all subgraph isomorphisms from G' and G'' to the input graph have been found, they must be combined into subgraph isomorphisms from G to the input graph. For this purpose,

VERTEX_TEST(v, l, G_I)

1. let $G_I = (V_I, E_I, \mu_I, \nu_I)$, $F = \emptyset$, and $l = \mu(v)$.
2. for all $v_I \in V_I$
   (a) if $l = \mu(v_I)$ then set $f(v) = v_I$ and $F = F \cup \{f\}$.
3. return F.

Figure 2.3: Procedure *vertex_test*.

a procedure for the combination of subgraph isomorphisms is required.

In Fig. 2.3 the procedure *vertex_test* is given. It returns all mappings of a single vertex v with label l onto an input graph G_I. The procedure simply consists of a loop over all vertices of G_I in which the label of each input graph vertex v_I is compared to the label of the model graph vertex v. If the labels are identical then a subgraph isomorphism from v to G_I has been found and can be added to the set of subgraph isomorphisms F.

In Fig. 2.4 the procedure for the combination of subgraph isomorphisms is given. The procedure takes as input two graphs S_1, S_2, an input graph G_I, a set of edges E with a corresponding edge labeling function and two sets of functions F_1, F_2 which contain all subgraph isomorphisms from S_1 and S_2 to G_I, respectively. Note that each edge $e \in E$ describes an edge between S_1 and S_2, i.e. $e = (v_1, v_2)$ and $v_1 \in V_1, v_2 \in V_2$, or $v_1 \in V_2, v_2 \in V_1$. In order to combine two functions $f_1 \in F_1$ and $f_2 \in F_2$, there are two conditions that must be satisfied. First, the images of f_1 and f_2 must be disjoint, i.e. $f_1(V_1) \cap f_2(V_2) = \emptyset$. Otherwise, the combination of f_1 and f_2 will not be an injective function. Secondly, it must be ensured that each edge that is specified in the set E is mapped correctly onto edges in G_I and vice versa. Thus, for each edge $e = (v_1, v_2) \in E$ there must be an edge $e_I = (f_1(v_1), f_2(v_2)) \in E_I$ and $\nu(e) = \nu_I(e_I)$. Also, for each edge $e_I = (v_I, v_I') \in E_I$ between $f_1(V_1)$ and $f_2(V_2)$ there must be an edge $e = (f_1^{-1}(v_I), f_2^{-1}(v_I')) \in E$ with $\nu_I(e_I) = \nu(e)$. If both conditions are satisfied, the functions f_1 and f_2 can be combined into a subgraph isomorphism from $S_1 \cup_E S_2$ to the input. When all combinations of functions in F_1 and F_2 have been tested, the procedure terminates by outputting the set F_s of subgraph isomorphisms from the union graph $S_1 \cup_E S_2$ to G_I.

Based on the decomposition of a set of model graphs and the procedures *vertex_test* and *combine*, we can formulate the new subgraph isomorphism algorithm (Fig. 2.5). The input to the algorithm consists of a decomposition $D(B)$, which represents the model graphs $B = \{G_1, \ldots, G_L\}$ and an input graph G_I. Informally

COMBINE($S_1, F_1, S_2, F_2, E, G_I$)

1. let $S_1 = (V_1, E_1, \mu_1, \nu_1)$, $S_2 = (V_2, E_2, \mu_2, \nu_2)$ and $F = \emptyset$.
2. for all pairs f_1, f_2 where $f_1 \in F_1$ and $f_2 \in F_2$
   (a) test the conditions (1) and (2):
      (1) $f_1(V_1) \cap f_2(V_2) = \emptyset$.
      (2) for each edge $e = (v_1, v_2) \in E$ there exists an edge $e_I = (f_1(v_1), f_2(v_2)) \in E_I$ with $\nu(e) = \nu_I(e_I)$ and for each edge $e_I = (f_1(v_I), f_2(v'_I)) \in E_I$ between $f_1(V_1)$ and $f_2(V_2)$ there exists an edge $e = (f_1^{-1}(v_I), f_2^{-1}(v'_I)) \in E$ with $\nu_I(e_I) = \nu(e)$.

   (b) if both (1) and (2) are true then let the subgraph isomorphism $f : V_1 \cup V_2 \rightarrow V_I$ from $S_1 \cup_E S_2$ to G_I be defined as follows:

$$f(v) = \begin{cases} f_1(v) & \text{if } v \in V_1 \\ f_2(v) & \text{if } v \in V_2 \end{cases}$$

   Add f to the set F, i.e., $F = F \cup \{f\}$.

3. output F.

Figure 2.4: Procedure *combine*.

speaking, the algorithm must first search for subgraph isomorphisms from the smallest components described in the decomposition $D(B)$ to the input graph G_I and then gradually combine them into larger subgraph isomorphisms. In order to keep track of the components that have been matched already with the input graph, each subgraph S, S' or S'' occurring in a 4-tuple (S, S', S'', E) in $D(B)$ can be marked with one of three different tags. In the beginning, all subgraphs in the decomposition are marked *unsolved*. As soon as a subgraph has been tested for subgraph isomorphisms with the input graph, it is either marked *alive* or *dead*. If the search for subgraph isomorphisms was successful, then the subgraph is marked *alive* and all detected subgraph isomorphisms are associated with it. Otherwise, the subgraph is marked *dead* and no subgraph isomorphisms are associated with it. First, in steps 3a-b, the algorithm loops over all components of the model graphs that consist of only one vertex and calls the procedure *vertex_test* for each of these components. Notice that if a vertex – or, more precisely, a particular vertex label – appears multiple times in the same or different model graphs, the decomposition represents this vertex only

NA($D(B), G_I$)

1. let $D(B) = \{(S_1, S_1', S_1'', E_1), \ldots, (S_n, S_n', S_n'', E_n)\}$, and P be the set of all graphs occurring in $D(B)$, i.e. $P = \cup_{i=1}^{n}\{S_i, S_i', S_i''\}$.
2. for all $S \in P$ mark S as *unsolved*.
3. for all $S = (V_s, E_s, \mu_s, \nu_s) \in P$ with $|V_s| = 1$
   (a) $F_s = vertex\_test(v, \mu_s(v), G_I)$ where $\{v\} = V_s$.
   (b) if $F_s = \emptyset$ mark S *dead* else mark S *alive* and associate F_s to S.
4. while there are $S \in P$ which are marked *unsolved* do
   (a) choose $(S, S_1, S_2, E) \in D(B)$ where S is marked *unsolved* and S_1, S_2 are both marked *alive*. If no such entry can be found then goto 5.
   (b) let F_1,F_2 be the set of subgraph isomorphism associated with S_1 and S_2, respectively.
   (c) $F_s = combine(S_1, F_1, S_2, F_2, E, G_I)$
   (d) if $F_s = \emptyset$ then mark S *dead* else mark S *alive* and associate F_s to S.
5. for each model graph G_i which is marked *alive*, output the subgraph isomorphisms that are associated with G_i.

Figure 2.5: Algorithm *NA*.

once and thus *vertex_test* is called exactly once for this type of vertex. Next, in steps 4a-d, each graph S is considered which is marked *unsolved* but decomposed into two subgraphs S_1, S_2 which have been previously tested and successfully matched to the input graph. The subgraph isomorphisms that are associated with S_1 and S_2 are combined into possible subgraph isomorphisms for S by calling the procedure *combine*. If the set of subgraph isomorphisms returned by *combine* is empty then S is marked *dead* and, consequently, for each model graph which contains S as a subgraph, there exists no subgraph isomorphism to the input graph. On the other hand, if the returned set F of subgraph isomorphisms is not empty then S is marked *alive*. This process continues until either all 4-tuples in $D(B)$ have been tested or no 4-tuple can be found in step 4a for which both subgraphs are marked *alive*. In this case, the search for further subgraph isomorphisms can be terminated immediately. Finally, in step 5, all subgraph isomorphism that have been found for the model graphs represented by $D(B)$ are output.

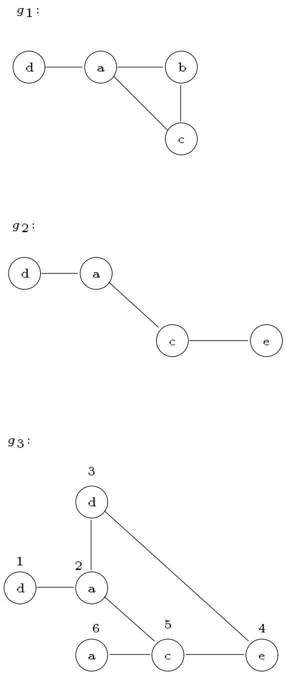

Figure 2.6: Two model graphs g_1, g_2 and an input graph g_3.

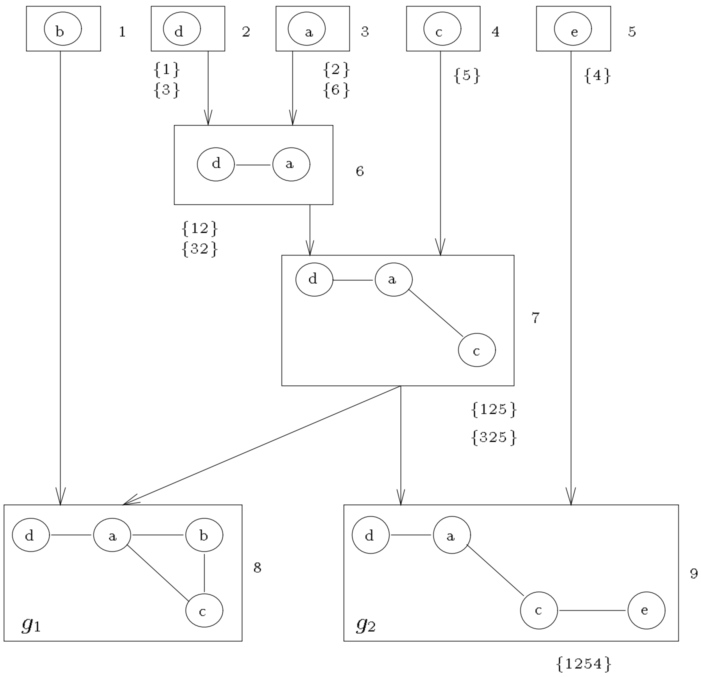

Figure 2.7: Decomposition of graphs g_1, g_2 from Fig. 2.6 displayed as a network structure.

It is easy to see that the new algorithm finds all subgraph isomorphisms from each of the model graphs $G_1, \ldots, G_L$ to the input graph G_I. Furthermore, if a subgraph S is part of several model graphs and represented by the decomposition, i.e., there is a 4-tuple (S, S_1, S_2, E), then the computation of all subgraph isomorphisms from S to the input graph G_I is done only once.

2.3.4 An Example

In order to illustrate the new algorithm, we give a detailed example. In Fig. 2.6 two model graphs, g_1 and g_2, and an input graph, g_3, are shown. In Fig. 2.7 the decomposition of g_1 and g_2 is graphically represented by a network. The network is displayed with vertices of the model graphs in the top layer. Below are subgraphs of the models that consist of more than one vertex. The complete models are represented at the bottom of the network. The numbers to the right of each network node are identifiers. At the top of the network, the five different vertex labels occurring in g_1 and g_2 are represented. Notice that, for example, the vertex with label d represented in node 2 appears in both g_1 and g_2. It appears, however, only once in the decomposition. On the second level in Fig. 2.7, a subgraph that is decomposed into two vertices with label d and a, respectively, and an edge connecting them is represented. This subgraph occurs in both g_1 and g_2. In network node 7, the largest common subgraph of g_1 and g_2 is represented. It is decomposed into the subgraph represented in node 6, a vertex represented in node 4, and an edge. Finally, network nodes 8 and 9 represent models g_1 and g_2, respectively.

At run time, given the input graph g_3, the algorithm NA described in the previous section first tries to find all subgraph isomorphisms for the vertices represented in the top nodes of the network. For example, the vertex labeled d can be matched onto the vertices 1 and 3 of g_3. Consequently, there are two subgraph isomorphisms associated with the node 2 (denoted by $\{1\}$ and $\{3\}$ below the node 2 in Fig. 2.7). Next, for each subgraph in the decomposition whose components have already been successfully tested, the algorithm tries to combine the subgraph isomorphisms of its components. In node 6, the subgraph isomorphisms $\{1\}$ and $\{3\}$ associated to node 2, and $\{2\}$ and $\{6\}$ associated to node 3 are combined. However, only the combinations $\{12\}$ and $\{32\}$ are valid subgraph isomorphisms from the graph in network node 6 to the input graph g_3. Because of the existence of these two subgraph isomorphisms, network node 6 is marked alive and, consequently, the subgraph in node 7 will be tested next. This process continues until a single subgraph isomorphism, $\{1254\}$, has been found from the graph g_2 to the input graph g_3. It can be easily verified that no other subgraph isomorphism from either g_1 or g_2 to g_3 exists.

2.4 Computational Complexity Analysis

In this section we give a detailed analysis of the run time complexity of the new algorithm for subgraph isomorphism detection and of the algorithm for the decomposition of the model graphs. The analysis will be based on the following quantities:

L = the number of model graphs in the database,
n_1 = the number of vertices of a subgraph that is common to all models,
n_2 = the number of vertices that are unique to each model,
n = the total number of vertices of a model graph, i.e. $n = n_1 + n_2$,
m = the number of vertices in the input graph,

Due to the fact that the decomposition algorithm makes extensive use of some subgraph isomorphism algorithm (step 3 in Fig. 2.2), its performance is strongly dependent on the performance of the applied algorithm. For this reason, we first study the computational complexity of the subgraph isomorphism algorithms and then examine the complexity of the proposed decomposition algorithm.

In our analysis we only consider the idealized situation where the model graphs have a single subgraph of size n_1 in common. In practice the situation will usually be more complicated due to the existence of subgraphs that are common to some, but not all models. Furthermore, for reasons of convenience, we assume that the model graphs are decomposed as follows. Given a model graph G and S, the subgraph of

G which is common to all the graphs, we assume that the decomposition contains an entry $(G, S, G-S, E)$. I.e., the graph G is decomposed into the common subgraph S and the difference graph $G-S$. The graph S is then decomposed into a subgraph S_i that consists of a single vertex and the difference graph $S - S_i$, which is again decomposed into a single vertex and the remaining difference graph. This process is continued until the difference graph itself consists of a single vertex only. The same decomposition scheme is applied to the graph $G-S$. Consequently, for a common subgraph of n_1 vertices, there are $n_1 - 1$ 4-tuples (S_1, S_1', S_1'', E) where S_1, S_1' and S_1'' are subgraphs of S. S_1 consists of k, S_1' of $k-1$ and S_1'' of one vertex, $k = 2, \ldots, n_1$. Analogously, there are $n_2 - 1$ 4-tuples in the decomposition of the difference graph $G-S$, which consists of n_2 vertices.[2] Based on this decomposition of the model graphs it is possible to analyze the computational complexity of the new algorithm *NA* given in Fig. 2.5. The computation steps that are performed can be estimated in terms of the number of calls to the procedure *vertex_test* times the complexity of *vertex_test*, and the number of calls to the procedure *combine* times the complexity of *combine* itself. In Table 2.1 an overview of the theoretical boundaries for both the new algorithm and, for comparison reasons, Ullman's algorithm [Ull76] is given. (A complexity analysis of Ullman's algorithm is given in the Appendix.)

The best case for both algorithms arises when each of the vertices of the model graphs is uniquely labeled. For a single model graph, there will be n different vertices and consequently, the procedure *vertex_test* will be called n times. Each call will test m vertices of the input graph and thus cause a total number of $O(mn)$ steps. The number of matchings found by *vertex_test* of a single model graph vertex onto the input graph will be bounded by $O(m/n)$. There will $O(n)$ calls to the procedure *combine*, i.e., for each 4-tuple (S_1, S_1', S_1'', E) there will be a call $combine(S_1', F_1', S_1'', F_1'', E, G_I)$ where S_1' consists of $k-1$ and S_1'' of one vertex. Because all vertices are uniquely labeled, F_1' contains at most one subgraph isomorphism and there will be $O(m/n)$ subgraph isomorphisms in F_1''. In the procedure *combine* each mapping in F_1' is combined with each mapping in F_1''. Comparing the images of the mappings takes $O(k)$ steps and testing each edge between S_1' and S_1'' takes $O(k)$ steps with $k \leq n$. Thus, the total computational complexity for matching a single model graph against an input graph is in the best case bounded by

$$O(mn + n^2 m/n) = O(mn) \tag{2.1}$$

When L model graphs must be matched onto the input graph, there will be $O(mn_1)$ steps performed for the common subgraph S and $O(Lmn_2)$ steps for each difference graph $G_i - S$ of the L model graphs. Additionally, for each model graph the subgraph isomorphisms found for the common subgraph and the difference graph must be combined, requiring $O(Ln_1n_2)$ steps at most. Thus, the computational complexity

[2] It must be mentioned that the algorithm *decomposition* given in Section 2.3.2 usually does not decompose the model graphs the way it is assumed here. However, the results of the complexity analysis were confirmed in a number of practical experiments documented in Section 2.5.

	New algorithm	Ullman's algorithm
All models are completely different: $n_1 = 0$, $n_2 = n$		
best case	$O(Lmn)$	$O(Lmn)$
worst case	$O(Lm^n n^2)$	$O(Lm^n n^2)$
All models are identical: $n_1 = n$, $n_2 = 0$		
best case	$O(mn)$	$O(Lmn)$
worst case	$O(m^n n^2)$	$O(Lm^n n^2)$

Table 2.1: Best and worst case complexities of the new and the traditional algorithm.

of the new algorithm for finding all subgraph isomorphisms from L model graphs to an input graph is in the best case bounded by

$$O(mn_1 + Lmn_2 + Ln_1n_2) \tag{2.2}$$

In the extreme case where there is no common subgraph among the model graphs, i.e. $n_1 = 0$ and $n_2 = n$, the complexity is bounded by

$$O(Lmn) \tag{2.3}$$

On the other hand, in the limit when all models are identical, i.e. $n_1 = n$ and $n_2 = 0$, the complexity of the new algorithm becomes independent of the number of models and is only bounded by

$$O(mn) \tag{2.4}$$

The worst case for both the new and the traditional algorithm arises when all vertices in the model graphs have the same label and each vertex is connected to each other vertex. Given a single model graph, the new algorithm makes only one call to the procedure *vertex_test* as there is only one vertex label. Each of the model vertices is then successfully matched onto $O(m)$ input graph vertices and consequently there are $O(m)$ subgraph isomorphism returned by *vertex_test*. These subgraph isomorphisms constitute the input to the first call of the procedure *combine* in which $O(m^2)$ subgraph isomorphisms are found at the expense of $O(n)$ steps. Next, according to the decomposition described at the beginning of this section, there will be n calls to *combine*. For each call, the number of subgraph isomorphisms that must be combined will be increased by a factor of m. Consequently, the total number of steps for a single model graph amounts in the worst case to

$$O(m + nm^n n) = O(m^n n^2) \tag{2.5}$$

For a database of L model graphs, the total effort is the sum of the effort for testing the common subgraph and the effort for testing the subgraphs that are unique to each model graph. The common subgraph is tested exactly once at the expense of $O(m^{n_1}n_1^2)$ while the unique subgraph must be tested for each model graph individually at the expense of $O(m^{n_2}n_2^2)$. By adding $O(Ln_1n_2m^n)$ steps necessary for the combination of the subgraph isomorphisms, we arrive at a total complexity for L model graphs of

$$O(m_1^n n_1^2 + Lm_2^n n_2^2 + Ln_1n_2m^n) \tag{2.6}$$

In the extreme case, when the database consists of completely different model graphs only, i.e. $n_1 = 0, n_2 = n$, the above formula becomes

$$O(Lm^n n^2) \tag{2.7}$$

On the other hand, if all models are identical, i.e. $n_1 = n, n_2 = 0$, the size of the database, L, disappears in Eq. 2.6 and the worst case complexity of the new algorithm becomes

$$O(m^n n^2) \tag{2.8}$$

When compared to the computational complexity of Ullman's algorithm given in the second column of Table 2.1, we note that both in the best and worst case the new algorithm is faster than the traditional algorithm by a factor of L for similar model graphs in the limit. On the other hand, given a set of completely disjoint model graphs both algorithms have the same computational complexity. In practice, we can expect the computational complexity somewhere between $O(mn)$ and $O(Lm^n n^2)$.

While the run time complexity of the new subgraph isomorphism algorithm is of general importance, the complexity of the decomposition algorithm is only relevant to applications which cannot neglect the time spent for preprocessing. For example, in applications where some models are known beforehand but others are only acquired at run time, the time spent for decomposing and adding the new models to the existing decomposition may be restricted. Thus, the performance of the decomposition algorithm will become important.

Given a set of models $B = \{G_1, \ldots, G_L\}$ and a decomposition $D(B)$ of B, a new model G is decomposed and added to $D(B)$ by calling the procedure *decompose(G)*. The basic idea of this procedure is to find a tuple (G_i, G_i', G_i'', E) in $D(B)$ such that G_i is a subgraph of G and maximal in $D(B)$. In case of success, the model G is decomposed into G_i and the difference graph $G - G_i$ which will be decomposed itself by recursively calling the procedure *decompose*. Clearly, the computational complexity of the decomposition process depends on the number of calls to *decompose* and the complexity of the search for the largest subgraph in *decompose*. It is easy to see that for a graph G with n vertices, any decomposition of G consists of at most $O(2n) = O(n)$ subgraphs. As each of these subgraphs must be decomposed itself (unless it is already represented in $D(B)$) the number of calls to *decompose* is at most $O(n)$. In each call to *decompose*, a search for the largest subgraph of G that is already represented in $D(B)$ is performed. This search process requires that for each tuple (G_i, G_i', G_i'', E) in $D(B)$ a subgraph isomorphism from G_i to G is computed. Apparently, if these subgraph isomorphisms are computed with a conventional algorithm such as Ullman's algorithm, then the complexity of the search process will be linearly dependent on the number of tuples in $D(B)$. However, due to the fact that the algorithm NA was especially designed to solve the subgraph isomorphism problem based on a decomposition of the model graphs, it can also be applied in the decomposition algorithm itself. That is, given a decomposition $D(B)$ and a model graph G, the decomposition algorithm calls the algorithm NA in order to find all subgraph isomorphisms from graphs in $D(B)$ to G. Consequently, the computational complexity of the search process in *decompose* depends on the computational complexity of the new algorithm NA (see Table 2.1). Note that the number of vertices in the input graph is denoted by m in Table 2.1. However, when NA is applied in the decomposition algorithm, the input graph is in fact a model graph and, therefore, $m = n$. This means that in the best case, when all the models are identical and their vertices are uniquely labeled, the number of steps performed in *decompose* is bounded by $O(n^2)$ (Eq. 2.4). As there are $O(n)$ recursive calls to *decompose* the total number of steps for the decomposition of a model graph with n vertices is in the best case bounded by $O(n^3)$. Notice that in order to decompose a complete set of model graphs from scratch, the procedure *decompose* is sequentially called for each of these models. Thus, in the best case, the computational complexity for decomposing a set of L models is bounded by

$$O(Ln^3) \tag{2.9}$$

From the previous complexity analysis of the algorithm NA, we know that the worst case arises when all the graphs are completely different and the vertices of each graph are identically labeled. Thus, according to the worst case complexity of algorithm NA given in Eq. 2.7, the number of steps in *decompose* is bounded by $O(Lm^n n^2) = O(Ln^{n+2})$ for $m = n$. As there are $O(n)$ subgraphs in a decomposition of a model with n vertices, the computational complexity of the decomposition algorithm is in the worst case bounded by $O(Ln^{n+3})$. Again, if there is a set of models that must be decomposed from scratch, the procedure *decompose* will be called for each of these models individually and the worst case complexity will be bounded by

$$O(L^2 n^{n+3}) \tag{2.10}$$

Notice that in spite of the exponential worst case complexity of the proposed decomposition algorithm, it is our experience that it performs reasonably well for most applications (see Chapter 5).

Experiments		Parameters for the graph generation				
Nr.	Fig.	Vertices	Edges	Labels	Common	Database size
1	2.8,2.9,2.10,2.11	10 - 50	12 - 60	10	-	1 - 20
2	2.12,2.13,2.14	50	60	10	5 - 45	1 - 20
3	2.15,2.16	50	60	4 - 40	-	1 - 20
4	2.17,2.18,2.19	50	50 - 120	10	-	1 - 10
5	2.20,2.21,2.22	50-300	60 - 360	10	-	1 - 20
6	2.23	50-500	60-600	30	-	1
7	2.24	50	60	10	-	1-100
8	2.25	50	60	10	30	20

Table 2.2: Parameters of the various experiments.

2.5 Experimental Results

In order to examine the performance of the new algorithm in practice, we have performed a number of experiments with randomly generated graphs. The new algorithm (NA) and Ullman's refinement procedure (UA) were both implemented in C++ and run on a SUN Sparc10 Workstation. The code of NA consists of roughly 7000 lines compared to 2500 lines of UA.

All model and input graphs were randomly generated. The parameters in the random graph generation process were

- the number of vertices
- the number of edges
- the number of different vertex labels (all edges were unlabeled and undirected)
- the number of graphs in the database
- the size of the common subgraph that was contained in all models.

In our experiments, we automatically generated model graphs based on these parameters. From each model graph a corresponding isomorphic input graph was derived by copying and permuting its vertices and edges. Both NA and UA were then used to detect all graph isomorphisms from the models to the inputs. In order to account for the random nature of the underlying graphs, each experimental run was repeated ten times and the average computation time was recorded as a result. By varying different parameters in the graph generation process, the behavior of the new algorithm was studied. In Table 2.2, an overview of the experiments is given, indicating the different parameter values used in each specific experiment.

In the first experiment, we were interested in measuring the influence of the size of the graphs and the database on the performance of both algorithms. We started

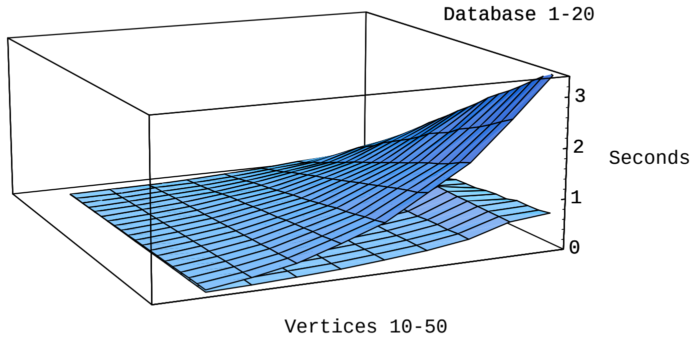

Figure 2.8: Time in seconds for increasing the number of vertices and the size of the database. (The lower plane denotes the time of NA, while the upper plane represents the time of UA.)

with a database consisting of a single graph with 10 vertices and 12 edges. The number of vertex labels was set to 10. We gradually increased the number of graphs in the database to 20 and we also increased the number of vertices from 10 to 50 together with the number of edges from 12 to 60. In Fig. 2.8 the results for NA (lower plane) and UA (upper plane) are shown. Apparently, the performance of UA became worse than that of NA with an increasing graph as well as an increasing database size. In Fig. 2.9 a cut through the 3D-plot along the axis for one model in the database is shown. We can observe that for a single model graph, UA is slightly faster than NA. However, if the database contains more than one model graph, the ability to compactly represent common substructures will make NA eventually faster than UA. This is confirmed in Fig. 2.10, where the number of graphs in the database was fixed at 20. The observation that UA's performance decreases faster than that of NA for a growing number of vertices is explained by the fact that by increasing the size of the model graphs while leaving the number of labels constant, there will be a growing number of substructures that occur multiple times within the same or different models. These substructures need to be considered only once by NA. UA, however, must search for these substructures each time they appear in a model graph.

In Fig. 2.11 a cut through Fig. 2.8 with the number of vertices fixed at 50 shows the influence of the database size more closely. As was expected from the complexity analysis, UA was linearly dependent on the size of the database, while NA's behavior was almost independent of the number of models.

In this first experiment, there were between 1 and 5 vertices with identical labels in each graph on the average. Therefore, many of the generated model graphs had some random substructures in common. In the second experiment, the effect of sharing common substructures among models was examined more closely. For this purpose, we generated model graphs consisting of 50 vertices and approximately 60 edges and increased the number of models from 1 to 20 and the size of the common

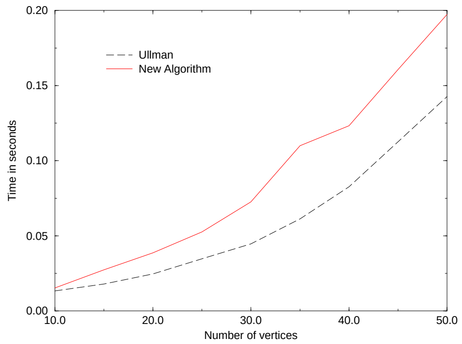

Figure 2.9: Cut through Fig. 2.8 along the vertex axis for one model graph in the database.

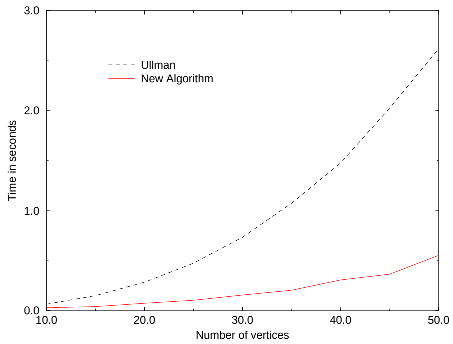

Figure 2.10: Cut through Fig. 2.8 along the vertex axis for 20 model graphs in the database.

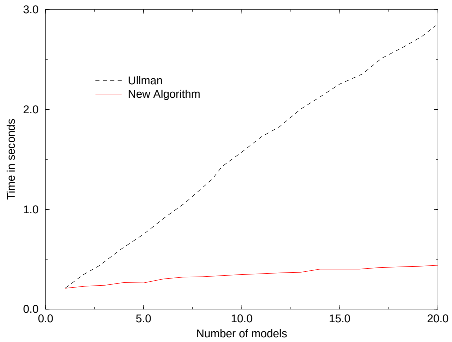

Figure 2.11: Cut through Fig. 2.8 along the database axis for 50 vertices in the model graphs.

subgraph from 5 to 45 vertices. Except for the common subgraph, all models were disjoint, i.e., except for the vertices of the common subgraph for which 15 different labels were allocated, the vertices of the model graphs were all uniquely labeled. The results of the second experiment are given in Fig 2.12 where the lower plane at the front corner denotes the times of NA while the upper plane represents UA. Clearly, the intersection of the two planes indicates that for a small or no common subgraph in the different models the performance of NA was slightly worse than that of UA. But for an increasing size of the common subgraph, the time needed by NA decreased and became much less than that required by UA.

In Fig. 2.13, a cut through Fig. 2.12 along the common subgraph axis is displayed. The size of the database was set to 20. We observe that if the common subgraph contained less than 15 vertices, NA performed worse than UA. However, for larger common subgraphs the time required by NA decreased rapidly while UA's time requirement increased. Note that in both Fig. 2.12 and 2.13 the database contained at most 20 models. It may be expected that even for small subgraphs there is a size of the database for which NA outperforms UA. Fig. 2.14 demonstrates this behavior.

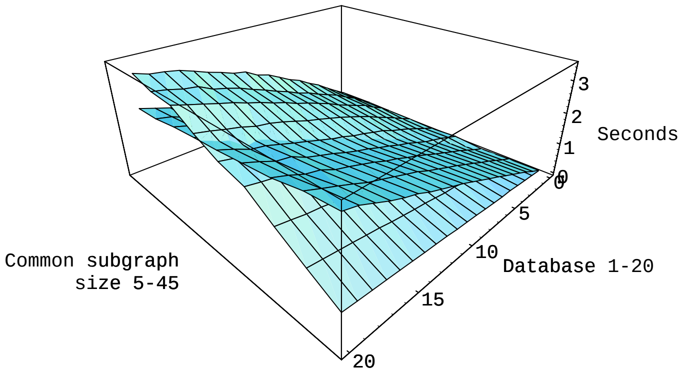

Figure 2.12: Time in seconds for an increasing number of vertices in the common subgraph and an increasing size of the database. (The lower plane in the front corner denotes the time of NA, while the upper plane represents the time of UA.)

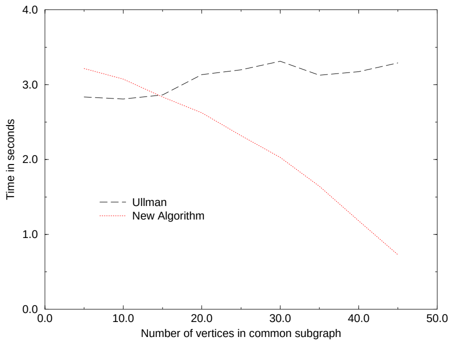

Figure 2.13: Cut through Fig. 2.12 along the common subgraph axis with the size of the database set to 20.

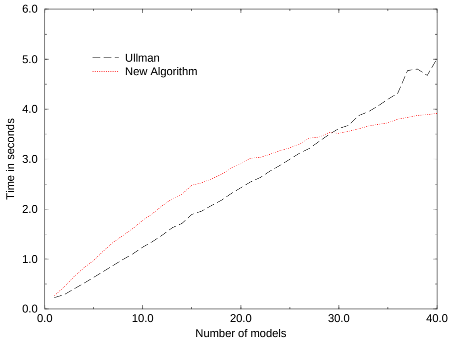

Figure 2.14: Cut through Fig. 2.12 along the database axis with the size of the common subgraph set to 5.

It shows a cut through Fig. 2.12 along the database axis with the common subgraph size set to 5 vertices. As an extension of the results shown in Fig. 2.12, the size of the database was varied between 1 and 40. For a database size of 30 models the run time of the two algorithms was identical and for larger databases NA performed better than UA. This indicates that NA is more efficient than UA even for small common subgraphs provided that the number of models in the database is sufficiently large.

In the first and second experiment, we always kept the number of vertex labels constant. In the third experiment, this number was varied. All generated graphs contained 50 vertices and 60 edges. The number of vertex labels was growing from 1 to 40 and again, we increased the size of the database from 1 to 20. The results are displayed in Fig. 2.15. The lower (upper) plane in the right corner corresponds to NA (UA). It can be observed that the two planes intersect each other. That is, the performance of NA became worse than that of UA when less than 5 labels were present. The line of intersection of the two planes grows with the number of

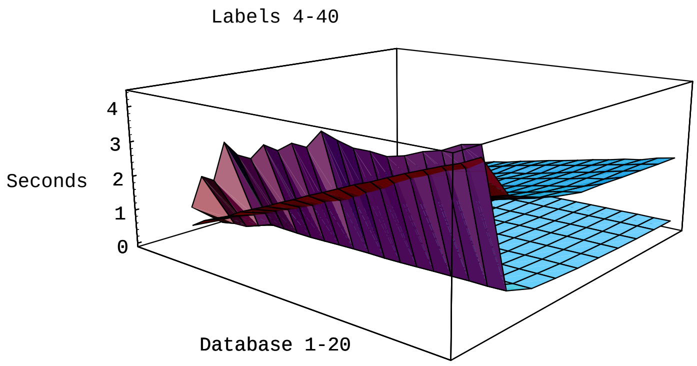

Figure 2.15: Time in seconds for increasing number of labels and size of the database. (The lower plane in the right corner denotes time of NA while the upper plane represents UA.)

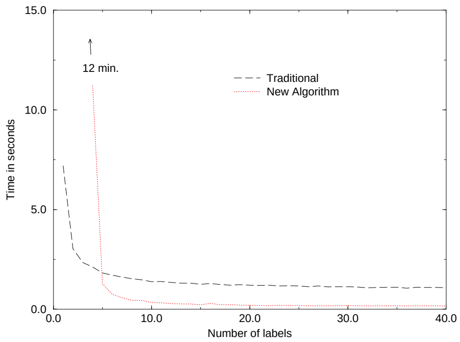

Figure 2.16: Cut through Fig. 2.15 along the labels axis with 10 model graphs in the database.

graphs in the database as expected. In Fig. 2.16 a cut through Fig. 2.15 along the axes for the labels is shown, with the database size set to 10. Between 10 and 40 labels the performance of NA and UA changed only minimally and NA was always faster. However, for less than 5 labels, NA took much more time to finish than UA. The worst case emerged for NA when there was only one vertex label, i.e., all vertices were identically labeled. In this case, NA required an average of 12 minutes in order to detect all graph matches. This decrease of the performance of NA is due to the fact that for unlabeled graphs the number of matches that are found for small subgraphs of the model graphs is usually very large. Thus, the concept of the new algorithm to first test the smaller components of the model graphs and then to combine them into subgraph isomorphisms for larger components is a major disadvantage when the underlying graphs are unlabeled. Consequently, the worst case behavior, as described in the complexity analysis section can be observed in Fig. 2.15 and 2.16. Obviously, NA tends towards the worst case behavior faster than UA for a decreasing number of labels.

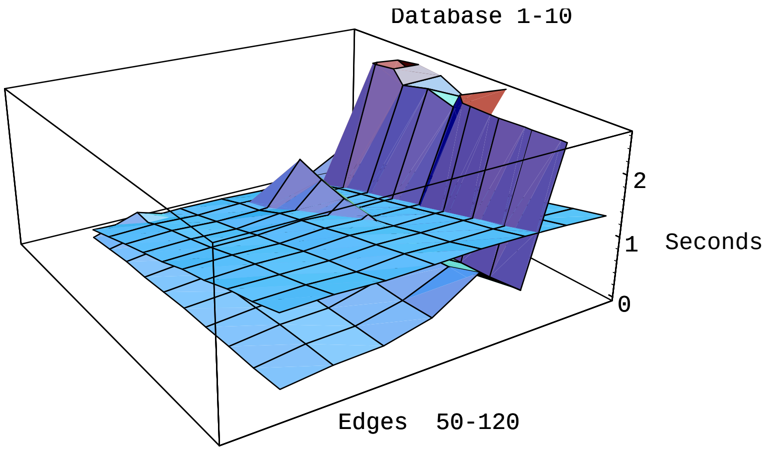

Figure 2.17: Time in seconds for increasing number of edges and size of the database. (The lower plane in the left corner denotes the time of NA while the upper plane represents UA.)

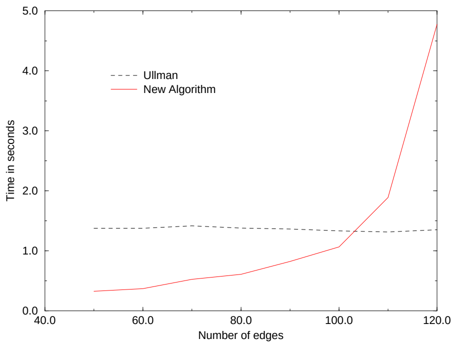

Figure 2.18: Cut through Fig. 2.17 along the edges axis with 10 model graphs in the database.

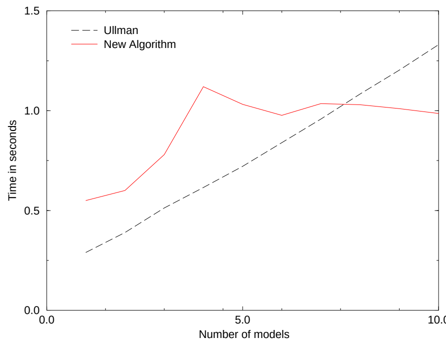

Figure 2.19: Cut through Fig. 2.17 along the database axis with 100 edges in the model graphs.

In the fourth experiment, the influence of the connectivity, i.e. the number of edges in a graph, was examined. In all of the previous experiments, the average degree of each vertex was kept at 2.5. For the experiment documented in Fig. 2.17, the graphs consisted of 50 vertices, 10 vertex labels and a gradually increasing number of edges, starting at 50 and ending at 120. Similar to the previous experiments, we also varied the number of models in the database from 1 to 10. In Fig. 2.17, the lower plane (upper plane) in the left corner denotes the times of NA (UA). Apparently, the performance of NA decreased with a growing number of edges in the graphs. In Fig. 2.18 a cut through Fig. 2.17 is taken along the axis denoting the number of edges with the number of graphs in the database fixed at 10. There was practically no influence of the number of edges on the performance of UA in the considered range. However, the computation time of NA increased fast with the number of edges. This behavior is due to the fact that with a high connectivity, the number of small subgraphs of a graph that can be matched onto themselves is large. On the other hand, the tendency of UA to spend more time than NA for larger databases was again confirmed in Fig. 2.19, where a cut of Fig. 2.17 along the axis denoting the size of the database is shown. The number of edges was constantly set to 100. We observe that for less than 7 models in the database, NA was slower than UA. For more than 7 models, the sharing of common subgraphs in NA and the linear dependency of UA on the size of the database resulted in UA taking more time than NA.

While in the first four experiments we examined the behavior of NA for the case of graph isomorphism, the fifth experiment was especially devoted to subgraph isomorphism detection. The models generated for this experiment consisted of 50 vertices, 10 different vertex labels and 60 edges. For each model graph, a corresponding input graph was created by copying the model and subsequently adding a growing number of vertices and edges. Thus, it was ensured that there existed at least one subgraph isomorphism from the model to the input. The number of additional vertices in the input graph was varied between 0 and 250, i.e., in the beginning, the input graphs consisted of 50 and in the end of 300 vertices. At the same time, the number of models in the database was also increased from 1 to 20. The results of the fifth experiment are given in Fig. 2.20. Note that the lower plane in the left corner denotes the time of NA while the upper plane represents UA. As expected, both NA and UA used more time for a growing input graph and a growing database. However, while UA performed better than NA when there were only few models in the database, the linear dependency on the database size caused UA to take more time than NA when there were more than ten models. In Fig. 2.21, a cut through Fig. 2.20 along the axis of the input graph for a database containing one model is given. It confirms that NA did not perform as well as UA for finding all subgraph isomorphisms from a model to a growing input graph. On the other hand, in Fig. 2.22, a cut through Fig. 2.20 along the database axis with the input graph constantly set to 300 vertices reveals that due to the compact representation of the models, NA used less time than UA when the database size was increased.

In all the experiments documented so far, the performance of UA and NA were studied for model graphs and databases of moderate sizes. The next two experiments were devoted to testing the behavior of both algorithms for very large graphs on the one hand and very large databases on the other hand. In Fig. 2.23 the size of the model and input graph was increased from 50 to 500 vertices along with the number of edges that was increased from 60 to 600 edges. There were 30 different vertex labels used, i.e., for a graph with 300 vertices there were on the average 10 vertices with the same label. The database consisted of a single model graph. We can observe that the performance of UA decreased much faster than NA's performance. While UA matched graphs with 250 vertices within 30 seconds, NA was capable of matching graphs twice as large in the same time. This behavior can again be explained by the fact that with an increasing number of vertices (and a

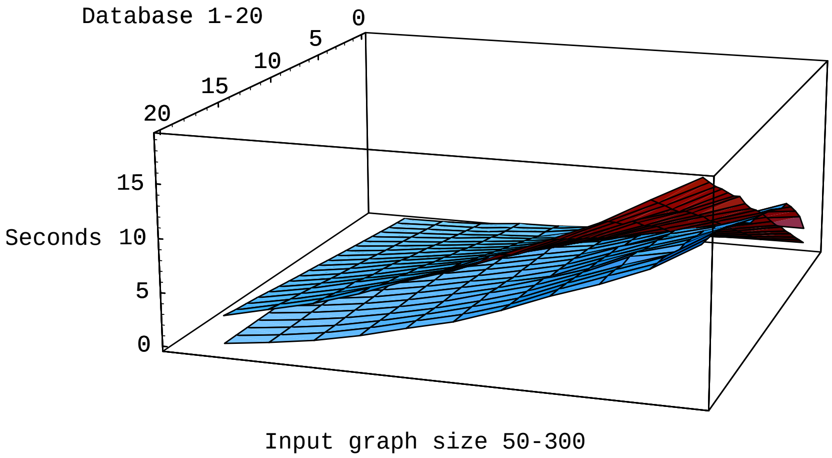

Figure 2.20: Time in seconds for increasing number of vertices in the input and size of the database. (The lower plane in the left corner denotes the time of NA while the upper plane represents UA.)

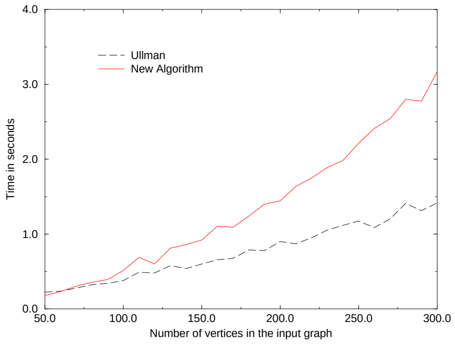

Figure 2.21: Cut through Fig. 2.20 along the input graph axis with 1 model in the database.

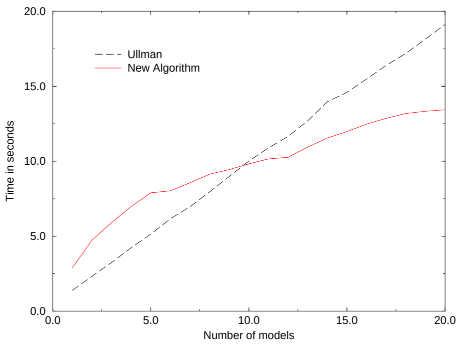

Figure 2.22: Cut through Fig. 2.20 along the database axis with 300 vertices in the input graph.

constant number of labels) the number of substructures that appear multiple times inside the same model graph grows. Although the algorithm *decomposition* that was proposed for the off-line decomposition of the models usually does not detect all of these substructures and represents them only once, the resulting decomposition is still fairly compact and allows NA to perform better than UA. Most important, however, is that *decomposition* is computationally inexpensive. For example, the decomposition of a graph with 50 vertices and 60 edges took only 0.21 seconds, and for a graph with 500 vertices and 600 edges only 70 seconds were required (not documented here).

Next, in Fig. 2.24, the number of model graphs in the database was increased from 1 to 100. As in the very first experiment (Fig. 2.8) the graphs consisted of 50 vertices and 60 edges with 10 vertex labels. Note that with NA even large databases

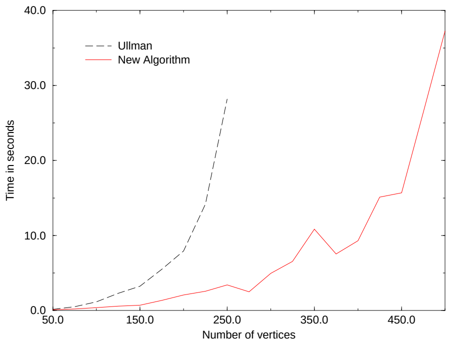

Figure 2.23: Computation time for an increasing number of vertices in the model graphs.

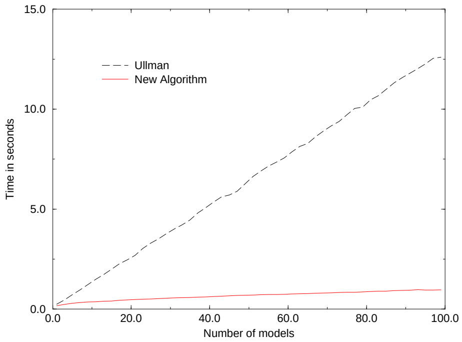

Figure 2.24: Computation time for an increasing number of models in the database.

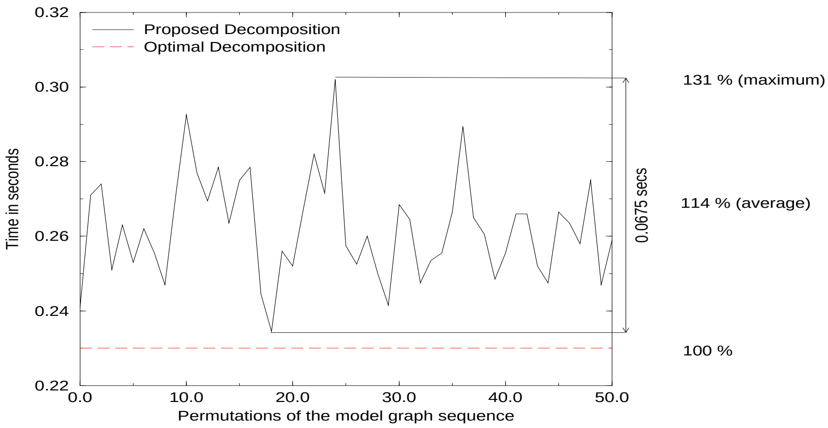

Figure 2.25: Computation time for various decompositions of the set of model graphs.

remain tractable (about 1 sec) while UA requires time that is linearly dependent on the database (15 sec for 100 models).

Finally, in the eighth experiment, we studied the influence of the decomposition algorithm on the performance of NA. In Section 2.3.2 it was observed that there exists a large number of different decompositions of a set of model graphs. These decompositions differ in terms of compactness and also in terms of the run time performance of NA. For example, it can be hypothesized that the optimal decomposition, i.e. the decomposition which contains a 4-tuple (G_c, G'_c, G''_c, E_c) for the largest subgraph G_c that appears in all the model graphs, will guarantee the optimal run time performance of NA. However, as mentioned before, the computation of the optimal decomposition is a highly exponential task. Thus, in Section 2.3.2, we proposed the algorithm *decomposition* which is computationally inexpensive. The drawback of this algorithm, however, is that it does not necessarily generate an optimal decomposition for a given set of models. Furthermore, as the models are treated sequentially, the resulting decomposition is dependent on the order of the models in the database. In order to study the difference in the performance of NA for an optimally decomposed database of models, on the one hand, and for a non-optimal decomposition of the same database on the other hand, the following experiment was performed. We generated a database containing 20 model graphs, each consisting of 50 vertices, 60 edges and 10 different vertex labels. Furthermore, a subgraph G_c with 30 vertices and 40 edges that appeared in all the model graphs was explicitly defined. Thus, it was possible to simulate the result of an optimal decomposition algorithm by taking the largest common subgraph G_c directly from the graph generation process. The input graphs were isomorphic copies of the model graphs. The performance of NA for detecting all graph isomorphisms from the models to the inputs based on this optimal decomposition was compared to its performance when the proposed, non-optimal algorithm *decomposition* was applied. In Fig. 2.25 the computation time of NA for both types of decompositions is given for various permutations of the models in the database. While the optimal decomposition was independent of the order of models, the proposed decomposition algorithm generated different decompositions for different orderings of the models and the performance of NA varied accordingly. With the optimal decomposition, NA performed faster than with the decompositions produced by algorithm *decomposition*. However, the average (maximum) difference in the performance is 14% (31%) compared to the optimal decomposition. Considering the fact that UA required more than 2 seconds in this experiment (not documented here), i.e. 870% more than NA based on the optimal decomposition, we can conclude that NA outperforms UA independent of the decomposition strategy that is used. Hence, it can be argued that the advantages of the proposed decomposition algorithm (incremental update, computationally inexpensive) outweigh the disadvantages (non-optimal decomposition).

2.6 Concluding Remarks

In this chapter a new algorithm for efficient subgraph isomorphism detection was proposed. The new algorithm is based on the idea of finding common subgraphs among different model graphs or within a single model graph. In an off-line preprocessing step these common subgraphs are recorded and at runtime they are used to efficiently detect subgraph isomorphisms from all model graphs to some input graph. In a theoretical complexity analysis it was shown that in the limit, when all model graphs are highly similar, the new method becomes independent of the size of the database. In order to verify the results of the theoretical complexity analysis, we performed a number of practical experiments with randomly generated graphs. The experiments confirmed that for model graphs with some common substructures the new algorithm is only sublinearly dependent on the size of the database. Furthermore, it was observed that for large model graphs the new algorithm is faster than traditional algorithms due to reappearing substructures that naturally evolve in large graphs. On the other hand, the experiments also documented that for unlabeled, highly connected graphs the new algorithm performs poorly.

We conclude that the new algorithm presented in this chapter is highly recommended for applications where we are faced with large databases of large labeled graphs with restricted connectivity. By contrast, the new algorithm should not be applied to problems dealing with unlabeled, highly connected graphs, to databases of completely disjoint graphs, or to problems where the input graph is considerably larger than the model graphs. We feel that the new algorithm is a substantial contribution to the problem of efficient subgraph matching. The algorithm has been presented in a general form without the explicit use of domain features or heuristics. Therefore, it can be regarded a generic tool that is applicable to various tasks. The extension of the new algorithm to error-tolerant subgraph isomorphism detection is currently under investigation. At last, it is to be mentioned that there is a potential for parallelization inherent in the new approach, which may be used for a further improvement of the performance[KS89].

Chapter 3

An New Algorithm For Error-Correcting Subgraph Isomorphism Detection

3.1 Introduction

Due to their representational power, attributed graphs are widely used in various domains of computer science. Particularly in computer vision and pattern recognition, they have been used to represent complex structures such as Chinese characters[LRS91], hand-drawn symbols[LKG90], aerial road network images [CKP95], 3D-objects[Won90, CS95], and others. In many applications, these complex structures must be classified, detected, or compared to each other by means of some matching scheme. By using attributed graphs for the representation, the matching process can be formulated as a search for graph or subgraph isomorphisms. However, real world objects are often affected by noise such that the graph representations of identical objects may not exactly match. Thus, it is necessary to integrate the concept of error correction into the matching process. Depending on the problem domain, correspondences between the models and the input graph can then be established by searching for either error-correcting graph isomorphisms between the models and the input, error-correcting subgraph isomorphisms from the models to the input, or error-correcting subgraph isomorphisms from the input to the models. Because error-correcting graph isomorphism is a special case of error-correcting subgraph isomorphism, and because input graphs are often larger than model graphs, we are particularly interested in the problem of error-correcting subgraph isomorphism detection from a set of models to an input graph.

It is well known that subgraph isomorphism detection is an NP-complete problem [GJ79]. I.e., the number of computational steps required to detect all subgraph isomorphism from one graph to another is exponential in the size of the underlying graphs. Consequently, error-correcting subgraph isomorphism detection is also in

NP and generally harder than exact subgraph isomorphism detection.

In the past, various approaches to error-correcting subgraph isomorphism detection have been proposed. The most common approach is based on tree search with the A^* algorithm [Nil80]. The search space of the A^* algorithm can be greatly reduced by applying heuristic error estimation functions. In the domain of computer vision, numerous heuristics have been proposed[Won90, TF79, EF84, SF83, SH81]. All of these methods are guaranteed to find the optimal solution but require exponential time in the worst case. Continuous optimization methods, on the other hand, are polynomially bounded in the number of computation steps but may fail to find the optimal solution. For example, in [KCP92, CKP95] a probabilistic relaxation scheme is described, which works well for large graphs, but may miss the optimal match in some cases. Other approaches are based on neural networks such as the Hopfield network[FLD94] or the Kohonen map[XO90]. However, all of these approximate methods may get tracked in local minima and miss the optimal solution. A special case of error-tolerant graph matching is the weighted graph matching problem, for which an interesting linear programming solution is presented in [AD93]. Related to this work is the approach based on eigen-decomposition described in [Ume88]. Again, both techniques have only a polynomial time complexity, but are not guaranteed to find an optimal solution.

An additional problem in many applications is that there are not only one, but several a priori known model graphs that must be matched onto a single input graph. The methods for error-tolerant graph matching mentioned so far work on only two graphs at a time. Thus, for databases which contain more than one model graph it is necessary to apply the graph matching method to each model-input pair, resulting in a linear dependency on the size of the database. This may be prohibitive for large databases. In order to overcome this problem, some authors proposed to organize the database of graphs such that the number of error-correcting subgraph isomorphism searches can be reduced. For example, in [SH82, SKP93, SH92] a hierarchical organization of the database was proposed, where the hierarchy is determined by clustering the model graphs into similarity classes. For a given input graph, the hierarchy is traversed by first matching the input graph onto the root of the hierarchy and then choosing the branch that represents the class of model graphs which are most similar to the input. The indexed class of model graphs is then again clustered into similarity classes or, if only few models are left, each of them is directly matched with the input graph. Another hierarchical organization was proposed in [SB95], where at the root of the hierarchy a supergraph, consisting of different distinct subgraphs of the model graphs is placed and matched against the input. The disadvantage of this scheme, however, is that the root graph may become much larger than the individual model graphs and thus the first matching process may be more time consuming than the sum of each individual graph match between a model and the input.

In this chapter, we present a new approach to the problem of error-correcting subgraph isomorphism detection between a database of model graphs and an unknown input graph. The new method is an extension of the algorithm for exact subgraph isomorphism detection that was presented in Chapter 2. The approach is based on a compact representation of the model graphs. This representation is derived from the model graphs in a off-line preprocessing step. In this preprocessing step, the model graphs are decomposed into smaller subgraphs and represented in terms of these subgraphs. If a subgraph appears multiple times within the same model graph or in different models, it will be represented only once. Hence, the resulting representation of the models is very compact. At run time, this compact representation is used to efficiently detect error-tolerant subgraph isomorphisms from the model graphs to the input. Common subgraphs that are part of different model graphs are matched only once with the input. Consequently, the complexity of the new algorithm is only sublinearly dependent on the size of the database. Furthermore, the algorithm can be combined with a very efficient future cost estimation technique.

The rest of this chapter is organized as follows. In Section 3.2 the main definitions and notations that will be used throughout the chapter are given. In Section 3.3 an introductory example is described demonstrating one of the traditional methods for error-correcting subgraph isomorphism detection. An overview of the new algorithm is given in Section 3.4.1. A more detailed description is provided in Section 3.4.2. The future cost estimation function that is based on the compact representation is then presented in Section 3.4.3. A practical example illustrates the new algorithm in Section 3.4.4. In Section 3.5, we analyze the computational complexity of the new algorithm in comparison with a traditional method. The results of the theoretical analysis are confirmed in a number of practical experiments in Section 3.6. In the light of the practical experiments the advantages and disadvantages of the new algorithm are briefly summarized in Section 3.7 and finally, in Section 3.8, some concluding remarks about the applicability of the new algorithm in practice and future research directions are given.

3.2 Definitions and Notations

The algorithms presented in this chapter work on labeled graphs. Let L_V and L_E denote the set of vertex and edge labels, respectively.

Definition 3.1: A *graph* G is a 4-tuple $G = (V, E, \mu, \nu)$, where

- V is the set of vertices,
- $E \subseteq V \times V$ is the set of edges,
- $\mu : V \to L_V$ is a function assigning labels to the vertices,

- $\nu : E \to L_E$ is a function assigning labels to the edges.

□

In this definition, the edges are directed, i.e. there is an edge from v_1 to v_2 if $(v_1, v_2) \in E$. For graphs with undirected edges, we require that $(v_2, v_1) \in E$ for any edge $(v_1, v_2) \in E$.

Definition 3.2: Given a graph $G = (V, E, \mu, \nu)$, a *subgraph* of G is a graph $S = (V_s, E_s, \mu_s, \nu_s)$ such that

1. $V_s \subseteq V$
2. $E_s = E \cap (V_s \times V_s)$
3. μ_s and ν_s are the restrictions of μ and ν to V_s and E_s, respectively, i.e.,

$$\mu_s(v) = \begin{cases} \mu(v) & \text{if } v \in V_s \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\nu_s(e) = \begin{cases} \nu(e) & \text{if } e \in E_s \\ \text{undefined} & \text{otherwise} \end{cases}$$

□

From this definition it is easy to see that, given a graph G, any subset of its vertices uniquely defines a subgraph of G. We use the notation $S \subseteq G$ to indicate that S is a subgraph of G.

Definition 3.3: Given a graph $G = (V, E, \mu, \nu)$ and a subgraph $S = (V_s, E_s, \mu_s, \nu_s)$ of G, the *difference* of G and S is the subgraph of G that is defined by the set of vertices $V - V_s$. □

The difference of a graph G and a subgraph S of G is denoted by $G - S$.

Definition 3.4: Given two graphs $G_1 = (V_1, E_1, \mu_1, \nu_1)$, $G_2 = (V_2, E_2, \mu_2, \nu_2)$, where $V_1 \cap V_2 = \emptyset$, and a set of edges $E' \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$ with a labeling function $\nu' : E' \to L_E$, the *union* of G_1 and G_2 with respect to E' is the graph $G = (V, E, \mu, \nu)$ such that

1. $V = V_1 \cup V_2$
2. $E = E_1 \cup E_2 \cup E'$

3. $\mu(v) = \begin{cases} \mu_1(v) & \text{if } v \in V_1 \\ \mu_2(v) & \text{if } v \in V_2 \end{cases}$

4. $\nu(e) = \begin{cases} \nu_1(e) & \text{if } e \in E_1 \\ \nu_2(e) & \text{if } e \in E_2 \\ \nu'(e) & \text{if } e \in E' \end{cases}$

□

The union of two graphs G_1 and G_2 with respect to a set of edges E' according to Def. 4 will be denoted by $G_1 \cup_{E'} G_2$.

Definition 3.5: A bijective function $f : V \rightarrow V'$ is a *graph isomorphism* from a graph $G = (V, E, \mu, \nu)$ to a graph $G' = (V', E', \mu', \nu')$ if:

1. $\mu(v) = \mu'(f(v))$ for all $v \in V$.
2. For any edge $e = (v_1, v_2) \in E$ there exists an edge $e' = (f(v_1), f(v_2)) \in E'$ such that $\nu(e) = \nu'(e')$, and for any $e' = (v_1', v_2') \in E'$ there exists an edge $e = (f^{-1}(v_1'), f^{-1}(v_2')) \in E$ such that $\nu(e) = \nu'(e')$.

□

Definition 3.6: An injective function $f : V \rightarrow V'$ is a *subgraph isomorphism* from G to G' if there exists a subgraph $S \subseteq G'$ such that f is a graph isomorphism from G to S. □

Apparently, graph isomorphism is a special case of subgraph isomorphism. For the remainder of this chapter, we will assume that there is a number of a priori known graphs, the so-called *model graphs*, and an *input graph*, which is given on-line. The input and model graphs will be also referred to as input and models, for short.

Real world objects which are represented by graphs may be affected by noise and distortions. In order to compare the undistorted model graphs to a distorted input graph and decide which of the models is most similar to the input, it is necessary to define a distance measure for graphs. Similar to the string matching problem where edit operations are used to define the string edit distance[WF74], we define a subgraph edit distance that is based on the idea of compensating the distortions in the input graph by means of edit operations that are applied to the model graphs[BA83]. I.e., the graph edit operations are used to alter the model graphs until there exist subgraph isomorphisms to the input graph. To each of the graph edit operations, a certain cost is assigned. The subgraph distance from a model to an input graph is then defined to be the minimum cost taken over all sequences of edit operations that are necessary for the compensation of the distortions in the input graph. It can be concluded that the smaller the subgraph distance between a model and an input graph is, the more similar they are. We consider the following distortions in a graph: vertex and edge label distortions, missing vertices, and missing or extraneous edges. Notice that it is not necessary to consider extraneous vertices in the input graph as we are searching for subgraph isomorphisms from the model to the input graph. Thus extraneous vertices in the input graph are automatically ignored. For each type of distortion given above, a corresponding graph edit operation is defined.

Definition 3.7: Given a graph $G = (V, E, \mu, \nu)$, a *graph edit operation* δ on G is any of the following:

- $\mu(v) \to l$, $v \in V$, $l \in L_V$: substituting the label $\mu(v)$ of vertex v by l (for the correction of vertex label distortions).
- $\nu(e) \to l'$, $e \in E$, $l' \in L_E$: substituting the label $\nu(e)$ of edge e by l' (for the correction of edge label distortions).
- $v \to \$$, $v \in V$: deleting the vertex v from G (for the correction of missing vertices)[1].
- $e \to \$$, $e \in E$: deleting the edge e from G (for the correction of missing edges).
- $\$ \to e = (v_1, v_2)$, $v_1, v_2 \in V$: inserting an edge between two existing vertices v_1,v_2 of G (for the correction of extraneous edges).

□

In order to model the fact that certain distortions are more likely than others, each graph edit operations δ is assigned a certain cost $C(\delta)$. The costs of the graph edit operations are strongly application dependent and must be defined on the basis of heuristic knowledge.

The five edit operations in Def. 3.7 are powerful enough to transform any graph G into a subgraph of any other graph G'. Therefore, it is always possible to find a sequence of edit operations that transform a model graph G into another graph G' such that a subgraph isomorphism f from G' to a distorted input graph G_I exists.

Definition 3.8: Given a graph $G = (V, E, \mu, \nu)$ and an edit operation δ, the edited graph, $\delta(G)$, is a graph $\delta(G) = (V_\delta, E_\delta, \mu_\delta, \nu_\delta)$ with

1. $V_\delta = \begin{cases} V - \{v\} & \text{if } \delta = (v \to \$) \\ V & \text{otherwise} \end{cases}$

[1]Note that all edges in G that are incident with the vertex v are deleted, too.

2. $E_\delta = \begin{cases} E \cup \{e\} & \text{if } \delta = (\$ \rightarrow e) \\ E - \{e\} & \text{if } \delta = (e \rightarrow \$) \\ E \cap (V_\delta \times V_\delta) & \text{otherwise} \end{cases}$

3. $\mu_\delta(v) = \begin{cases} l & \text{if } \delta = (\mu(v) \rightarrow l) \\ \mu(v) & \text{otherwise} \end{cases}$

4. $\nu_\delta(e) = \begin{cases} l & \text{if } \delta = (\nu(e) \rightarrow l) \\ \nu(e) & \text{otherwise} \end{cases}$

□

Definition 3.9: Given a graph $G = (V, E, \mu, \nu)$ and a sequence of edit operations $\Delta = (\delta_1, \delta_2, \ldots, \delta_k)$, the edited graph, $\Delta(G)$, is a graph $\Delta(G) = \delta_k(\ldots \delta_2(\delta_1(G))) \ldots)$. □

The total cost of the transformation of G into $\Delta(G)$ is given by $C(\Delta) = \sum_{i=1}^{k} C(\delta_i)$. We can now combine the concepts of an edited graph and a subgraph isomorphism into the concept of an error-correcting subgraph isomorphism.

Definition 3.10: Given two graphs G and G', an *error-correcting (ec) subgraph isomorphism* f from G to G' is a 2-tuple $f = (\Delta, f_\Delta)$ where

1. Δ is a sequence of edit operations such that there exists a subgraph isomorphism from $\Delta(G)$ to G',

2. f_Δ is a subgraph isomorphism from $\Delta(G)$ to G'.

□

The cost of an *ec* subgraph isomorphism $f = (\Delta, f_\Delta)$ is the cost of the edit operations Δ, i.e. $C(f) = C(\Delta)$. It is easy to see that there is usually more than one sequence of edit operations Δ such that a subgraph isomorphism from $\Delta(G)$ to G' exists and, consequently, there is usually more than one error-correcting subgraph isomorphism from G to G'. For our distance measure, we are particularly interested in the error-correcting subgraph isomorphism with minimum cost. The subgraph distance is thus defined as follows.

Definition 3.11: Let G and G' be two graphs. The *subgraph distance* from G to G', $d(G, G')$, is given by the minimum cost taken over all error-correcting subgraph isomorphisms f from G to G':

$$d(G, G') = MIN_\Delta\{C(\Delta) \mid \text{ there is an } ec \text{ subgraph isomorphism } f = (\Delta, f_\Delta) \text{ from } G \text{ to } G'\}$$

□

The *ec* subgraph isomorphism f associated with $d(G, G')$ is called the optimal error-correcting (*oec*) subgraph isomorphism from G to G'. Notice that the subgraph distance according to Def. 3.11 is in general not symmetric, i.e. $d(G, G') \neq d(G', G)$. Therefore, it is important to note that the proposed algorithm is designed to compute the graph distance from the model graph G to the input graph G_I, $d(G, G_I)$, and not vice versa.

3.3 Introductory Example

In order to compute the graph distance between two graphs G and G' according to Def. 3.11, both the minimal sequence of edit operations, Δ, and the corresponding subgraph isomorphism which matches $\Delta(G)$ onto G' must be found. It is easy to think of a brute force method in which all possible sequences Δ of edit operations are applied to the model graph G with a subsequent test whether there exists an exact subgraph isomorphism from $\Delta(G)$ to G'. Clearly, this straightforward approach is too costly to be applied in practice.

In the past, the most common approach to *oec* subgraph isomorphism detection was based on a state-space search by means of an algorithm similar to A^* [Nil80, SH81]. For comparison reasons, we briefly describe this so-called *traditional* method. For a detailed description, see Appendix A.2. The basic idea of a state-space search is to have states representing partial solutions of the given problem and to define transitions from one state to another such that the latter state represents a more complete solution than the previous state. For each state s there is an evaluation function $f(s)$ which describes the quality of the represented solution. The states are expanded according to the value of f. In the case of *ec* subgraph isomorphism detection, given a model graph $G = (V, E, \mu, \nu)$ and an input $G_I = (V_I, E_I, \mu_I, \nu_I)$, a state s in the search space represents a partial matching from G to G_I. A partial matching of k vertices of G onto k vertices of G_I is an *ec* graph isomorphism from a subgraph of G to a subgraph of G_I and is denoted by a set of vertex tuples, $\{(v_1, v_{s_1}), (v_2, v_{s_2}), \ldots, (v_k, v_{s_k})\}$ with $v_i \in V, v_{s_i} \in V_I$. The tuple (v_i, v_{s_i}) indicates that the vertex v_i of G is matched with the vertex v_{s_i} of G_I. It is important to note that a partial matching $\{(v_1, v_{s_1}), \ldots, (v_k, v_{s_k})\}$ implies a number of edit operations.

Figure 3.1: Two labeled graphs.

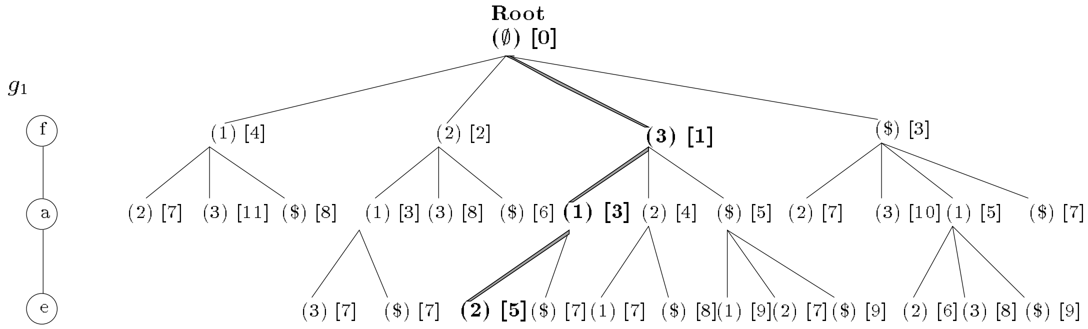

Figure 3.2: Search tree expanded by A^* when matching g_1 onto g_2 (see Fig. 3.1).

For example, if vertex v_i is matched with vertex v_{s_i} and the vertices have different labels, then the edit operation which substitutes the label $\mu(v_i)$ with the label $\mu_I(v_{s_i})$ is implied. In order to represent the deletion of a vertex, the symbol \$ is introduced, such that the tuple $(v_i, \$)$ denotes the deletion of the vertex v_i from the model graph G. Each partial matching therefore implies a set of edit operations and the cost of these edit operations can be used to define the evaluation function $f(s)$ of the state s representing the partial matching. The transition from one state s to another state s' is the extension of the partial matching in s, $\{(v_1, v_{s_1}), (v_2, v_{s_2}), \ldots, (v_k, v_{s_k})\}$, by an additional tuple, $(v_{k+1}, v_{s_{k+1}})$, matching the vertex v_{k+1} onto some vertex $v_{s_{k+1}}$. The costs of the edit operations implied by the new tuple are added to the costs that were assigned to s. In the beginning of the search process, the search space contains only the root state which represents an empty matching. All possible transitions from the root state to successor states are then generated by matching the first vertex v_1 of G onto each of the vertices v_I of G_I or by deleting v_1. From these new states, the state representing the partial matching with the lowest cost is then chosen for expansion and all corresponding successor states are generated. The process terminates when either a state representing an *oec* subgraph isomorphism from G to G_I has been reached or all states in the search space have edit costs that exceed a given acceptance threshold.

For example, consider the graphs g_1, g_2 in Fig. 3.1 and the problem of finding the *oec* subgraph isomorphism from g_1 to g_2. Notice that the vertices of g_1 and g_2 are labeled with letters of the Latin alphabet while the edges are unlabeled. Additionally, the vertices of g_2 are uniquely numbered for the purpose of identification. The costs for the edit operations are defined as follows. The substitution of a label l_1 by a label l_2 is defined to be the distance of the letters l_1 and l_2 in the order of the alphabet. For instance, if $l_1 = a$ and $l_2 = g$, the cost for substituting a by g, or g by a, is 6. The cost for deleting a vertex is constantly set to 3 while the cost for deleting and inserting an edge is set to 1. The search space that is expanded by the traditional algorithm is displayed in Fig. 3.2. On the left of the search space, the graph g_1 is redrawn with its vertices given in the order in which they are matched. Each state s is indicated by a partial matching $(v)[c]$, where (v) denotes the number of the vertex of g_2 that is matched with the corresponding vertex of g_1 and $[c]$ is the total cost that is implied by the partial matching represented in the state s and its predecessor states. The states are expanded according to the cost c. In the beginning, there is only the root state, $(\emptyset)[0]$, for which all successors states are generated. Next, the state with the lowest costs, namely the third state from the left, (3)[1], is expanded, and so on. The path to the state representing the *oec* subgraph isomorphism is depicted in bold face in Fig. 3.2. Note that the *oec* subgraph isomorphism from g_1 to g_2 matches the vertex f onto the vertex g, the vertex a onto the vertex b and the vertex e onto the vertex d. The edge between a and f in g_1 is deleted and an edge between f and e in g_1 inserted. The total costs of the edit operations amounts to 5 units.

Clearly, the number of states in the search space may grow exponentially in the worst case. Thus, various researchers have proposed to use a more sophisticated evaluation function which considers not only the cost of the represented partial matching, but also the future cost that is implied by this partial matching[SH81, TF79, Won90]. By applying a so-called *lookahead* procedure in order to estimate the future cost, the number of expanded states can be greatly reduced. In this chapter, the algorithm described above was implemented along with the heuristic lookahead procedure described in [Won90]. When applied to the example in Fig. 3.2, the influence of the lookahead procedure reduces the number of expanded states from now 29 to 10. However, the lookahead procedure is of quadratic complexity and must be executed for each state in the search space. Furthermore, the algorithm can only match two graphs at the same time. Therefore, if the database contains more than one model graph, the traditional algorithm must be applied to each model. This may become prohibitive to practical applications if the number of model graphs is large.

3.4 A New Algorithm for Error-correcting Subgraph Isomorphism Detection

3.4.1 Overview

In the previous section it was shown that, given a set of models $G_1, \ldots, G_L$ and an input graph G_I, the traditional algorithm can only match a single model graph G_i to the input graph G_I at a time. We now propose a new method in which the model graphs are not treated individually but integrated in a common data structure. This data structure is then directly used to find the *oec* subgraph isomorphisms from any of the models to the input.

The basic idea of the new method is to recursively decompose the model graphs off-line into smaller subgraphs. In particular, each model graph is decomposed into two subgraphs and each of these subgraphs is again decomposed until the remaining subgraphs consist of only one vertex. The subgraphs resulting from this decomposition process are recorded and the model graphs are represented in terms of these subgraphs. The main advantage of this scheme is that subgraphs which appear in different model graphs are represented only once and can be reused for each of the model graphs. Hence, this representation of the model graphs is fairly compact. Notice that this representational scheme is similar to the RETE network used in forward-chaining production systems [For82, LS92]. At run time, given an unknown input graph, the search for *ec* subgraph isomorphisms is first performed for the smallest subgraphs of the model graphs. Next, the *ec* subgraph isomorphisms with the least edit costs are recursively combined to form *ec* subgraph isomorphisms for the complete model graphs. Due to the fact that common subgraphs of different model graphs are represented only once, it is sufficient to match these common subgraphs only once onto the input graph. Consequently, the new method is only sublinearly dependent on the number of model graphs. Furthermore, the efficiency of the method can be greatly enhanced by the addition of a future cost estimation technique.

In the following, we first introduce the representation of the model graphs and the new *oec* subgraph isomorphism detection algorithm in detail. Next, the future cost estimation technique is described and finally, the mode of operation of both the matching algorithm and the future cost estimation are illustrated by means of an example.

3.4.2 The Matching Algorithm

The Data Structures

The new algorithm works on a set $B = \{G_1, \ldots, G_L\}$ of model graphs, that are known a priori, and an input graph that becomes available at run time only. In an off-line preprocessing step, the model graphs are compiled into a compact representation. At run time, this representation is used to efficiently detect the *oec* subgraph isomorphism f_i from a model graph G_i to the input graph such that the corresponding graph distance $d(G_i, G_I)$ is minimal over all model graphs, i.e.

$$C(f_i) = d(G_i, G_I) = MIN_j\{d(G_j, G_I)|j = 1, \ldots, N\} \tag{3.1}$$

The compilation of the model graphs in the preprocessing step is based on the idea of decomposing each model graph into two distinct subgraphs. Then, each of these subgraphs is again decomposed into smaller subgraphs until all subgraphs consist of single vertices and cannot be decomposed any further.

Definition 3.12: Let $B = \{G_1, \ldots, G_L\}$ be a set of model graphs. A *decomposition* of B, $D(B)$, is a finite set of 4-tuples (G, G', G'', E), where

1. G, G' and G'' are graphs with $G' \subset G$ and $G'' \subset G$
2. E is a set of edges such that $G = G' \cup_E G''$
3. For each G_i there exists a 4-tuple $(G, G', G'', E) \in D(B)$ with $G = G_i$; $i = 1, \ldots, L$.
4. For each 4-tuple $(G, G', G'', E) \in D(B)$ there exists no other 4-tuple $(G_1, G'_1, G''_1, E_1) \in D(B)$ with $G = G_1$.
5. For each 4-tuple $(G, G', G'', E_1) \in D(B)$
   (a) if G' consist of more than one vertex then there exists a 4-tuple $(G_1, G'_1, G''_1, E_1) \in D(B)$ such that $G' = G_1$
   (b) if G'' consists of more than one vertex then there exists a 4-tuple $(G_2, G'_2, G''_2, E_2) \in D(B)$ such that $G'' = G_2$
   (c) if G' consists of one vertex then there exists no 4-tuple $(G_3, G'_3, G''_3, E_3) \in D(B)$ such that $G' = G_3$
   (d) if G'' consists of one vertex then there exists no 4-tuple $(G_4, G'_4, G''_4, E_4) \in D(B)$ such that $G'' = G_4$.

□

Informally speaking, a decomposition is a recursive partitioning of graphs into smaller subgraphs, starting with complete models and terminating at the level of single vertices. The first component in a 4-tuple (G, G', G'', E) is the graph to be decomposed, G' and G'' are its two parts, and E are the edges in G between G' and G'' (see Conditions 1 and 2 in Def. 3.12). Condition 3 in Def. 3.12 makes sure that

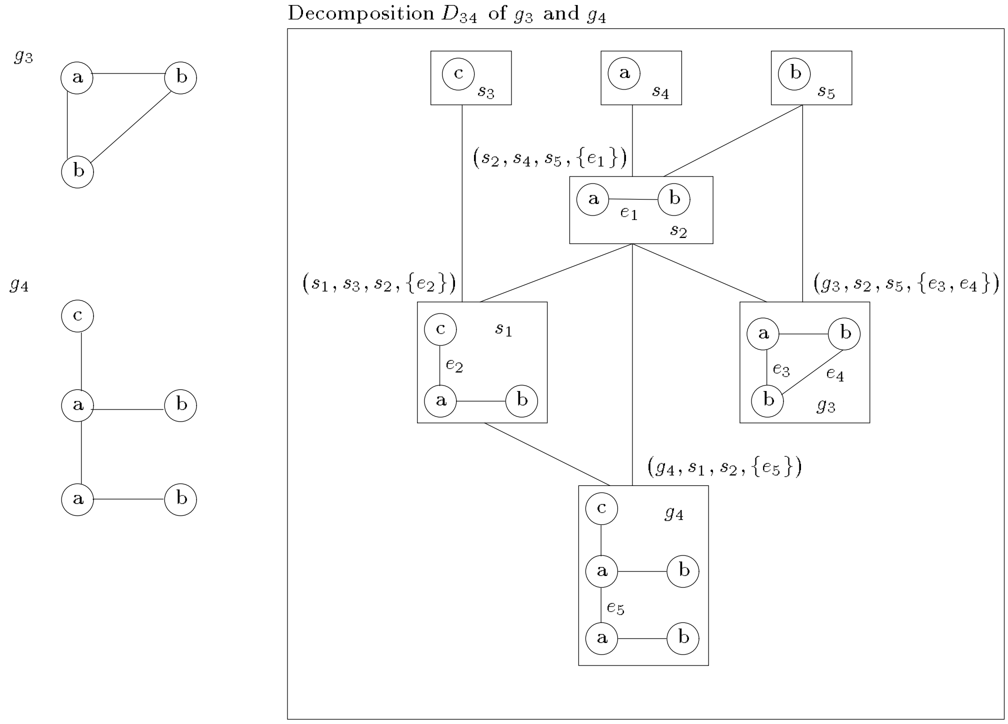

Figure 3.3: Two model graph g_3 and g_4 and a graphical representation of the corresponding decomposition D_{34}.

every model in B is decomposed, and Condition 4 implies that a decomposition is unique. By means of Condition 5 it is guaranteed that a decomposition is complete, i.e. the process of partitioning a graph into two parts is continued until individual vertices are reached. If several models $G_i, G_j, \ldots$ have a common subgraph G, or if G occurs multiple times in one model G_i, it is sufficient to represent G just by one 4-tuple (G, G', G'', E) in $D(B)$. This property not only leads to a compact representation of a set of models, B, by means of the decomposition $D(B)$, but it also is the key to an efficient matching procedure at run time.

In Fig. 3.3, an example of the representation of two graphs g_3, g_4 in a decomposition D_{34} is given. The decomposition D_{34} consists of the four 4-tuples $(g_4, s_1, s_2, \{e_5\})$, $(g_3, s_2, s_5, \{e_3, e_4\})$, $(s_1, s_3, s_2, \{e_2\})$ and $(s_2, s_4, s_5, \{e_1\})$. Notice that D_{34} is displayed as a hierarchical network. In this graphical representation, each subgraph s_1, s_2, s_3, s_4, s_5 and each graph g_3 and g_4 is represented by a network node and for each tuple $(S, S', S'', E) \in D_{34}$, there is a network edge from the node for S' to the node for S and a network edge from the node for S'' to the node for S. The network nodes are ordered from top to bottom according to the size of the subgraph they represent. At the top of the network, the nodes representing the subgraphs s_3, s_4 and s_5 are displayed. On the next level, the subgraph s_2 which consists of the subgraphs s_4 and s_5 and the connecting edge e_1 is given, followed by the subgraph s_1 and the model graph g_3. Finally, at the bottom of the network, the model graph g_4 which is decomposed into the subgraphs s_1, s_2 and the edge e_5 is displayed. Notice that the subgraph s_2 appears twice in the model graph g_4 and once in the model graph g_3, but is represented only once in the decomposition.

Clearly, a model graph can be decomposed in various ways. For example, in Fig. 3.3, the graph g_3 could also be decomposed into a subgraph consisting of a single vertex labeled a and another subgraph consisting of two vertices labeled b. As a consequence, the subgraph s_2 would not be used for the representation of g_3 and the decomposition would be less compact. Hence, there is a need for an efficient decomposition algorithm. In Chapter 2, we have presented a computationally inexpensive procedure for this problem. Although the procedure is not optimal, i.e. not all common subgraphs may be detected, the resulting decomposition is fairly compact. Moreover, the run time behavior of the algorithm is almost independent of the decomposition.

The Algorithm

The new on-line algorithm for *oec* subgraph isomorphism detection is based on the decomposition of the model graphs. Given a set of models $B = \{G_1, \ldots, G_L\}$, a decomposition $D(B)$ and an input graph G_I, the new algorithm first searches for *ec* subgraph isomorphisms for the subgraphs occurring in $D(B)$. The *ec* subgraph isomorphisms that are found are then combined to form *ec* subgraph isomorphisms for the full models $G_1, \ldots, G_L$. For example, in Fig. 3.3, the algorithm first matches the subgraphs s_3,s_4 and s_5 onto the graph G_I. Next, the algorithm tries to combine *ec* subgraph isomorphisms that were found for s_4 and s_5 in order to form *ec* subgraph isomorphisms for s_2 and so on. There are three basic problems that must be solved by the algorithm:

Problem (I) For each subgraph S in the decomposition that consists of a single vertex only, find all *ec* subgraph isomorphisms from S to G_I.

Problem (II) For each subgraph S in the decomposition that consists of more than one vertex and is decomposed into subgraphs S_1 and S_2, i.e. $(S, S_1, S_2, E) \in D(B)$, combine *ec* subgraph isomorphisms that are found for S_1 and S_2 such that *ec* subgraph isomorphisms from S to G_I result.

Problem (III) The sequence in which *ec* subgraph isomorphisms for subgraphs in the decomposition are combined must be controlled intelligently in order to avoid combinatorial explosion.

In the following, we examine each of the above problems individually and then formally describe the new algorithm for *oec* subgraph isomorphism detection.

Problem (I) Given a subgraph S that consist of only one vertex v and an input graph G_I, the *ec* subgraph isomorphisms from S to G_I can be generated by

VERTEX_MATCHING(v, l, G_I)

1. let $F = \emptyset$ and $G_I = (V_I, E_I, \mu_I, \nu_I)$
2. for all $v_I \in V_I$
   (a) generate an *ec* subgraph isomorphism $f = (\Delta, f_\Delta)$ with $f_\Delta(v) = v_I$ and $\Delta = ((l \rightarrow \mu_I(v_I))$
   (b) $F = F \cup \{f\}$
3. create $f' = (\Delta', f_{\Delta'})$ with $\Delta' = (v \rightarrow \$)$ and $f_{\Delta'} = \emptyset$ and set $F = F \cup \{f'\}$
4. output F

Figure 3.4: Algorithm *vertex_matching*.

mapping v onto each of the vertices in G_I and registering the necessary substitution operation. Additionally, the possible deletion of v must be taken into account as well. In Fig. 3.4, the procedure *vertex_matching* is displayed. It takes as arguments the single vertex v, its label l and an input graph G_I, and returns the set F of all *ec* subgraph isomorphisms from v to G_I. In the beginning, F is initialized as empty. In step 2, for each v_I in V_I an *ec* subgraph isomorphism $f = (\Delta, f_\Delta)$ is generated where Δ consist of the substitution of l by $\mu_I(v_I)$, i.e. $\Delta = ((l \rightarrow \mu(v_I)))$, and $f_\Delta(v) = v_I$. Each of these *ec* subgraph isomorphisms is collected in F. Finally, in order to account for the possible deletion of a vertex from the model graph, the *ec* subgraph isomorphism $f' = (\Delta', f_{\Delta'})$, where $\Delta' = ((v \rightarrow \$))$ denotes the deletion of v and $f_{\Delta'}$ is the null function, is generated and added to F. The set F is then outputted.

Problem (II) Given is a graph S that is decomposed into S_1 and S_2, i.e. $(S, S_1, S_2, E) \in D$. Moreover, let $f_1 = (\Delta_1, f_{\Delta_1})$ and $f_2 = (\Delta_2, f_{\Delta_2})$ be two *ec* subgraph isomorphisms from S_1 and S_2 to G_I, respectively. The problem is to find an *ec* subgraph isomorphism f from $S = S_1 \cup_E S_2$ to G_I that is based on f_1 and f_2. Clearly, f_1 and f_2 can only be combined if no two vertices in $\Delta(S_1)$ and $\Delta_2(S_2)$ are mapped onto the same input vertex. More precisely, the intersection of the images of f_{Δ_1} and f_{Δ_2} must be empty. If this is the case, then f_1 and f_2 can be combined into an *ec* subgraph isomorphism f from S to G_I. In Fig. 3.5, the procedure *combine* for the combination of *ec* subgraph isomorphisms is displayed. The input to *combine* consists of two graphs S_1, S_2, a set of edges E connecting S_1 and S_2, an input graph G_I, and two *ec* subgraph isomorphisms f_1,f_2 for S_1 and S_2, respectively. In the beginning of the procedure, it is tested whether the intersection $f_{\Delta_1}(V_1) \cap f_{\Delta_2}(V_2)$ is empty. If it is not empty, then the procedure exits immediately. Otherwise, an

COMBINE$(S_1, S_2, E, G_I, f_1, f_2)$

1. let $f_1 = (\Delta_1, f_{\Delta_1})$, $f_2 = (\Delta_2, f_{\Delta_2})$, $\Delta_1(S_1) = (V_{\Delta_1}, E_{\Delta_1}, \mu_{\Delta_1}, \nu_{\Delta_1})$, $\Delta_2(S_2) = (V_{\Delta_2}, E_{\Delta_2}, \mu_{\Delta_2}, \nu_{\Delta_2})$
2. if $f_{\Delta_1}(V_{\Delta_1}) \cap f_{\Delta_2}(V_{\Delta_3}) \neq \emptyset$ then exit
3. combine f_1 and f_2 to form an *ec* subgraph isomorphism $f = (\Delta, f_\Delta)$ from $S_1 \cup_E S_2$ to G_I, with

$$f_\Delta(v) = \begin{cases} f_{\Delta_1}(v) & \text{if } v \in V_{\Delta_1} \\ f_{\Delta_2}(v) & \text{if } v \in V_{\Delta_2} \end{cases}$$

   for all $v \in V_{\Delta_1} \cup V_{\Delta_2}$ and

$$\Delta = \Delta_1 + \Delta_2 + \Delta_E$$

   where Δ_E is the set of edit operations on the edges in E connecting S_1 and S_2 (see text).
4. output f

Figure 3.5: Algorithm *combine*.

ec subgraph isomorphism f is constructed from f_1 and f_2. The construction of an *ec* subgraph isomorphism $f = (\Delta, f_\Delta)$ from f_1 and f_2 requires that a set of edit operations Δ and a subgraph isomorphism f_Δ are generated on the basis of Δ_1, Δ_2 and f_{Δ_1}, f_{Δ_2}, respectively, such that f_Δ is a subgraph isomorphism from $\Delta(S_1 \cup_E S_2)$ to G_I. Due to the fact that the edit operations on the subgraphs S_1 and S_2 are contained in Δ_1 and Δ_2, it is easy to see that Δ is a concatenation of Δ_1, Δ_2 and the set Δ_E of edge operations on E, i.e.

$$\Delta = \Delta_1 + \Delta_2 + \Delta_E \tag{3.2}$$

For the construction of Δ_E, three possible edge edit operations must be considered for each edge specified in E. First, for each edge $e = (v_i, w_j)$ in E, $v_i \in V_1, w_j \in V_2$, there must be an edge $e_I = (f_{\Delta_1}(v_i), f_{\Delta_2}(w_j)) \in E_I$. If e_I exists and $\nu(e) \neq \nu_I(e_I)$ then, depending on the respective costs, either the label $\nu(e)$ is substituted by the label $\nu_I(e_I)$ or e is deleted and e_I is subsequently inserted. That is, if $C(\nu(e) \to \nu_I(e_I)) \leq C(e \to \$) + C(\$ \to e_I)$ then the substitution of $\nu(e)$ by $\nu_I(e_I)$ must be added to Δ_E. Otherwise, the deletion of e and the insertion of e_I must be added to Δ_E. Secondly, if e_I does not exists then the deletion of $e = (v_i, w_j)$ from the graph $S_1 \cup_E S_2$ must be added to Δ_E. Thirdly, if there is an edge $e_I = (f_{\Delta_1}(v_i), f_{\Delta_2}(w_j))$ in the input but no corresponding edge $e = (v_i, w_j)$ is specified in E, then the insertion of e into $S_1 \cup_E S_2$ must be added to Δ_E. With this, $\Delta = \Delta_1 + \Delta_2 + \Delta_E$ contains all edit operations that are necessary such that there exists a subgraph isomorphism f_Δ from $\Delta(S_1 \cup_E S_2)$ to G_I. The subgraph isomorphism f_Δ is then defined as

$$f_\Delta(v) = \begin{cases} f_{\Delta_1}(v) & \text{if } v \in V_{\Delta_1} \\ f_{\Delta_2}(v) & \text{if } v \in V_{\Delta_2} \end{cases} \tag{3.3}$$

for all $v \in V_{\Delta_1} \cup V_{\Delta_2}$. Finally, in step 4, the *ec* subgraph isomorphism $f = (\Delta, f_\Delta)$ is outputted.

Problem (III) Given a decomposition D and an input graph G_I, *ec* subgraph isomorphisms for the models in D can be found by applying the solutions to the problems I and II described above. Namely, for all subgraphs S in D that consist of a single vertex, all *ec* subgraph isomorphisms are first generated by calling the procedure *vertex_matching*. Next, all of these *ec* subgraph isomorphisms are combined to form *ec* subgraph isomorphisms for the larger subgraphs by calling the procedure *combine* until *ec* subgraph isomorphisms for the model graphs emerge. However, if all *ec* subgraph isomorphisms that are found are immediately used for combination regardless of their edit costs, their number will explode combinatorially. Thus, given a graph S that is decomposed into S_1,S_2, we propose to store the *ec* subgraph isomorphisms that are found for S_1 and S_2 locally, and then choose the *ec* subgraph isomorphisms with the least cost for combination. For this purpose, we introduce two ordered lists, *open* and *closed*, that are attached to each subgraph in D. An *ec* subgraph isomorphism f_1 that is found for some subgraph S_1 is then stored in $open(S_1)$ before it is used for further combinations. Let

$$C[open(S_1)] = \begin{cases} MIN_f\{\ C(f) \mid f \in open(S_1)\} & \text{if } open(S_1) \text{ is not empty} \\ \infty & \text{otherwise} \end{cases} \tag{3.4}$$

denote the least cost of any *ec* subgraph isomorphisms in $open(S_1)$. Now, the next *ec* subgraph isomorphism that is to be used for combination is selected according to $C[open(S_1)]$. If $C(f_1) = C[open(S_1)]$ is minimal over all subgraphs S_i in D then f_1 is removed from $open(S_1)$ and stored in the list $closed(S_1)$. Analogously to $C[open(S_1)]$, let

$$C[closed(S_1)] = \begin{cases} MIN_f\{\ C(f) \mid f \in closed(S_1)\} & \text{if } closed(S_1) \text{ is not empty} \\ \infty & \text{otherwise} \end{cases} \tag{3.5}$$

denote the least cost of any *ec* subgraph isomorphism in $closed(S_1)$. After f_1 has been removed from $open(S_1)$ and stored in $closed(S_1)$ it is combined with all *ec* subgraph isomorphisms that were found for S_2. More precisely, f_1 is combined with all *ec* subgraph isomorphisms that are already stored in $closed(S_2)$. This process guarantees that new *ec* subgraph isomorphisms are always built from *ec* subgraph isomorphisms with minimal cost.

NSG(D, G_I)

1. for all S in D with $S = (V_s, E_s, \mu_s, \nu_s)$ and $|V_s| = 1$, $\{v\} = V_s$.
   (a) F=vertex_matching($v, \mu_s(v), G_I$)
   (b) set $open(S) = open(S) \cup F$
2. choose S_1 such that $C[open(S_1)] + h(S_1) \leq T$ and minimal in D. If no such S_1 exists, goto 7.
3. choose $f_1 \in open(S_1)$ such that $C(f_1)$ is minimal in $open(S_1)$;
   remove f_1 from $open(S_1)$ and set $closed(S_1) = closed(S_1) \cup \{f_1\}$.
4. if S_1 is a model graph G_i then reset the acceptance threshold to $T = C(f_1)$.
5. for all $(S, S_1, S_2, E) \in D$ or $(S, S_2, S_1, E) \in D$ do
   (a) for all $f_2 \in\ closed(S_2)$
      i. f=combine($S_1, S_2, E, G_I, f_1, f_2$)
      ii. if f is not empty then add f to $open(S)$, i.e. $open(S) = \ open(S) \cup \{f\}$
6. goto 2.
7. for each model G_i in D, $i = 1 \ldots, L$ output $closed(G_i)$.

Figure 3.6: Algorithm *NSG*.

Based on the procedures *vertex_matching* and *combine* and the lists *open* and *closed*, the new method for *oec* subgraph isomorphism detection is now formally described in Fig. 3.6. The input to the algorithm *NSG* consists of the decomposition D which represents the set of model graphs $G_1, \ldots, G_L$ and the input graph G_I. In the beginning, $open(S)$ and $closed(S)$ are empty for each S in D. In step 1 all *ec* subgraph isomorphisms for subgraphs S consisting of a single vertex are generated and stored in $open(S)$ by calling the procedure *vertex_matching*. Next, in step 2, a subgraph S_1 is selected such that $C[open(S_1)] + h(S_1)$ is smaller than a given acceptance threshold T and minimal over all subgraphs in D. At first, the acceptance threshold T is set to the maximal cost that any *ec* subgraph isomorphism from the models to the input may have. Note that this maximal cost is strongly application dependent. It may be set to infinity. Later, in step 4, it will be reset to the cost of some *oec* subgraph isomorphism actually found for any of the models. In addition to the actual costs $C[open(S_1)]$ there is also an evaluation function $h(S_1)$ taken into account. For the moment, we assume that $h(S_1)$ is constantly set to zero for all

S, i.e. $h(S) = 0$ for all S in D. (In Section 3.4.3, a non-trivial function h will be introduced which estimates the future costs for any *ec* subgraph isomorphism f in $open(S_1)$.) After selecting a subgraph S_1 for which the condition in step 2 holds, the *ec* subgraph isomorphism $f \in open(S_1)$ with costs $C(f_1) = C[open(S_1)]$ is removed from $open(S_1)$ and added to the list $closed(S_1)$ in step 3. Next, in step 4 it is checked whether S_1 is one of the model graphs G_i. If this is the case, f_1 represents the *oec* subgraph isomorphism from G_i to G_I. This follows directly from the selection process in steps 2 and 3. By always selecting the *ec* subgraph isomorphism f_1 with the minimal costs, it is guaranteed that no other *ec* subgraph isomorphism for any of the graphs or subgraphs in the decomposition D has cost less than $C(f_1)$. Hence, the *ec* subgraph isomorphism f_1 for the model graph G_i is optimal. For some applications, it may be sufficient to find only one *oec* subgraph isomorphism. In that case, the algorithm can be terminated if the condition in step 4 is satisfied. Otherwise, if all *oec* subgraph isomorphisms with equal cost are to be found (as it is assumed here), the acceptance threshold T can be reset to $C(f_1)$ in step 4. This guarantees that the algorithm only searches for *ec* subgraph isomorphisms with costs equal to the cost of the *oec* subgraph isomorphism already found. In step 5, new *ec* subgraph isomorphisms are generated by means of combination. All graphs S which are decomposed into S_1 and some subgraph S_2, i.e. (S, S_1, S_2, E) or $(S, S_2, S_1, E) \in D$, are selected. Then, each *ec* subgraph isomorphism f_2 in $closed(S_2)$ is combined with f_1 in order to form an *ec* subgraph isomorphism f from S to G_I. For this purpose, the procedure *combine* is called with S_1, S_2, the connecting edges E, the input graph G_I and the *ec* subgraph isomorphisms f_1, f_2 as arguments. If *combine* is successful, an *ec* subgraph isomorphism f from S to G_I is returned and stored in $open(S)$. When all *ec* subgraph isomorphisms f_2 in $closed(S_2)$ have been combined with f_1, the process continues in step 2. Finally, when there are no more subgraphs S_1 with $C[open(S_1)] + h(S_1) \leq T$, all *oec* subgraph isomorphisms that were found for the models $G_1, \ldots, G_L$ are outputted in step 7 and the algorithm terminates.

It is easy to see that the algorithm *NSG* truly finds the *oec* subgraph isomorphism from a model graph G_i to an unknown input graph G_I such that $d(G_i, G_I)$ is minimal over all model graphs (see Eq. 3.1). Due to the fact that in step 2, the *ec* subgraph isomorphism with the least cost is always chosen, it is guaranteed that the very first *ec* subgraph isomorphism that is found for one of the models G_i and added to $closed(G_i)$ denotes the *oec* subgraph isomorphism from G_i to G_I. Furthermore, because the acceptance threshold T is reset to the cost of the first *oec* subgraph isomorphism that is found for a model G_i, no other, more expensive, *ec* subgraph isomorphisms will be accepted from this point on.

The main advantage of the new algorithm is that common subgraphs of different models are represented only once in the decomposition structure. Therefore, these subgraphs are matched only once with the input graph, while in the traditional algorithm the common subgraphs must be matched for each model graph individually.

3.4.3 Estimating the Future Cost

In the introductory example in Section 3.3 it was mentioned that the number of states expanded by A^* can be reduced by estimating the future error of each state. Similarly, in the new algorithm it is possible to reduce the number of generated *ec* subgraph isomorphisms by introducing a future cost estimation method. In the following, we first describe the basic idea of this future cost estimation by means of an example and then give a formal definition of the estimation function h that is used in step 2 of the algorithm NSG.

Given a model graph G, a decomposition D of G, and an input graph G_I, let G be decomposed into the subgraphs S_1 and S_2, i.e. $(G, S_1, S_2, E) \in D$. Furthermore, assume that the algorithm NSG has been called with an input graph G_I and the *open* and *closed* lists of both S_1 and S_2 are no longer empty. The costs of the *ec* subgraph isomorphisms that were found so far for S_1,S_2 are reflected in the cost

$$\begin{array}{ll} C[open(S_1)] = 4, & C[closed(S_1)] = 1, \\ C[open(S_2)] = 5, & C[closed(S_2)] = 3. \end{array}$$

Based on these costs, the algorithm NSG selects in step 2 the subgraph S for which

$$C[open(S)] + h(S) \tag{3.6}$$

is minimal over all S in D. Clearly, with $h(S) = 0$ for all S in D, the algorithm will choose S_1 as $C[open(S_1)] < C[open(S_2)]$. The *ec* subgraph isomorphism f_1 with $C(f_1) = 4$ that is removed from $open(S_1)$ will be combined with *ec* subgraph isomorphisms in $closed(S_2)$. As $C[closed(S_2)] = 3$, the cheapest *ec* subgraph isomorphism for S_2 costs at least 3 units and, therefore, the resulting *ec* subgraph isomorphism for G will cost at least 3+4=7 units. However, if S_2 were selected instead of S_1, the *ec* subgraph isomorphism $f_2 \in open(S_2)$ with $C(f_2) = 5$ would be combined with the *ec* subgraph isomorphisms in $closed(S_1)$. Because $C[closed(S_1)] = 1$, it is possible that the resulting *ec* subgraph isomorphisms for G would cost only $5 + 1 = 6$ units and therefore be cheaper than the *ec* subgraph isomorphism that is generated when S_1 is chosen. We conclude that the selection process in step 2, based on Eq. 3.6, can be improved if the contents of the *open* and *closed* lists of other subgraphs in the decomposition are also taken into account. As a consequence, the *oec* subgraph isomorphism can be found more efficiently.

We now describe a function $h(S_1)$ which estimates the future cost of any *ec* subgraph isomorphism for S_1 on the basis of *ec* subgraph isomorphisms that have been found for other subgraphs in the decomposition. Let D be a decomposition representing a set of model graphs $G_1 \ldots, G_L$ and let G_I be an input graph. Furthermore, let S_1 be a subgraph of a graph S in the decomposition, i.e. $(S, S_1, S_2, E) \in D$ and let S be itself a subgraph of the model graph G. Clearly, each *ec* subgraph isomorphism f_1 in $open(S_1)$ must be combined with some *ec* subgraph isomorphism f_2 of S_2 in order to form an *ec* subgraph isomorphism f for S. Let $g(S_2)$ denote the estimated minimal costs of any *ec* subgraph isomorphism for S_2 (see below). Then, each *ec* subgraph isomorphism f for S will cost at least $C[open(S_1)] + g(S_2)$, i.e. $C(f) \geq C[open(S_1)] + g(S_2)$. Because S is itself a subgraph of the model graph G, the future cost $h(S_1)$ of S_1 with respect to G can be recursively defined as

$$h(S_1) = g(S_2) + h(S) \tag{3.7}$$

If S_1 is a subgraph of more than one graph S, then the minimum over all future cost estimations must be taken, i.e.

$$h(S_1) = MIN_{S,S_2}\{g(S_2) + h(S) | \text{ for all } (S, S_1, S_2, E), (S, S_2, S_1, E) \in D\} \tag{3.8}$$

Note that if S_1 is a model graph then the future cost is set to zero, i.e. $h(S_1) = 0$.

The quality of the future cost estimation according to Eq. 3.8 is strongly dependent on $g(S_2)$. The purpose of $g(S_2)$ is to estimate as precisely as possible the cost of any *ec* subgraph isomorphism from S_2 to G_I. We propose to use the partial matching information that is gathered by the algorithm NSG for the computation of $g(S_2)$. Clearly, on the one hand, if *closed*(S_2) is not empty then the *ec* subgraph isomorphism for S_2 with minimal costs has already been found and

$$g(S_2) = C[closed(S_2)] \tag{3.9}$$

On the other hand, if *closed*(S_2) is empty, the value of $g(S_2)$ must be estimated. Let $n(S_2)$ denote the estimation of the minimal costs for an *ec* subgraph isomorphism of S_2 that is not yet stored in *closed*(S_2). Formally, $g(S_2)$ can now be defined as

$$g(S_2) = \begin{cases} C[closed(S_2)] & \text{if } closed(S_2) \text{ is not empty} \\ n(S_2) & \text{otherwise} \end{cases} \tag{3.10}$$

The auxiliary function $n(S_2)$ in Eq. 3.10 must consider all possible ways of creating an *ec* subgraph isomorphism for S_2 that is not yet contained in *closed*(S_2). Clearly, if S_2 consist of a single vertex then $n(S_2)$ is equal to the minimal costs of an *ec* subgraph isomorphism in *open*(S_2), i.e.

$$n(S_2) = C[open(S_2)] \tag{3.11}$$

If S_2 consist of more than one vertex and is decomposed into two subgraphs S_3, S_4, i.e. $(S_2, S_3, S_4, E_{34}) \in D$, then there are three ways of forming an *ec* subgraph isomorphism for S_2. First, if *open*(S_2) is not empty then $C[open(S_2)]$ may denote the minimal costs. Secondly, an *ec* subgraph isomorphism stored in *closed*(S_3) may be combined with an *ec* subgraph isomorphism in *open*(S_4) in order to form an *ec* subgraph isomorphism for S_2. As $g(S_3)$ denotes the minimal costs of any *ec* subgraph isomorphism for S_3, and $n(S_4)$ denotes the minimal costs of any *ec* subgraph isomorphism for S_4 that is not yet in *closed*(S_4), $g(S_3) + n(S_4)$ denotes the minimal costs of such a combination. Thirdly, the combination of an *ec* subgraph isomorphism in $open(S_3)$ with an *ec* subgraph isomorphism in $closed(S_4)$ may result in an *ec* subgraph isomorphism for S_2 with cost larger than or equal to $n(S_3)+g(S_4)$. The minimum of these three values is a lower bound estimate of any *ec* subgraph isomorphism f_2 with $f_2 \notin closed(S_2)$, i.e.

$$n(S_2) = MIN\{C[open(S_2)]\ ,\ g(S_3) + n(S_4)\ ,\ n(S_3) + g(S_4)\} \tag{3.12}$$

The equations 3.8,3.10 and 3.12 completely define the future cost estimation function h for each subgraph in a decomposition. The computation of h must be performed whenever the algorithm *NSG* arrives at step 2 because with every new *ec* subgraph isomorphism that is found in steps 3-6, the contents of the *open* and *closed* list change and, consequently, the values of g and n for subgraphs throughout the decomposition change as well. In practice, g and n are computed iteratively by starting with the smallest subgraphs and ending with complete model graphs in the decomposition. That is, $g(S)$ and $n(S)$ are first updated for all S consisting of a single vertex. Next, $g(S)$ and $n(S)$ are computed for all S consisting of two vertices, and so on. Afterwards, when g and n have been updated for all subgraphs in the decomposition, h can be computed iteratively starting with the model graphs and ending with the subgraphs that consist of single vertices. I.e., h is first computed for each model graph G_i. Then $h(S_1)$ and $h(S_2)$ are computed for each $(G_i, S_1, S_2, E_{12}) \in D$, and so on. Clearly, the total computational complexity of updating h in D is proportional to the number of subgraphs in D. As there are at most $O(M)$ possible subgraphs in the decomposition of a graph with M vertices, the number of steps performed for the computation of h is bounded by $O(M)$.

The most important property of the proposed future cost estimation function is that it is based on actual matching results that are found during the search process of *NSG*. In particular, with every new *ec* subgraph isomorphism that is found for some subgraph S in D, the future cost estimation is adjusted and improved for other subgraphs in D. We believe this estimation to be more reliable than the heuristic estimation that is computed in the lookahead procedure of the A^* algorithm. Moreover, its additional computational cost is very low.

3.4.4 An Example

In order to illustrate the new algorithm, including the future cost estimation function, we consider the graphs in Fig. 3.1 of Section 3.3. Let g_1 be the model graph and g_2 be the input. The costs of the edit operations are identical to the costs defined in Section 3.3. Furthermore, let $D = \{(g_1, s_1, s_2, \{e_1\})\ ,\ (s_1, s_3, s_4, \{e_2\})\}$ be the decomposition of g_1; e_1 is the edge between the vertices with labels a and e, and e_2 is the edge between the vertices with labels f and a. In Fig. 3.7 the decomposition D is graphically represented by means of a hierarchical network. At the top of the network, there are the subgraphs s_3, s_4, s_2, which consist of single vertices only. On the next level, the node representing the subgraph with the two vertices labeled a and f is shown. Finally, on the bottom level, the model graph g_1 is displayed.

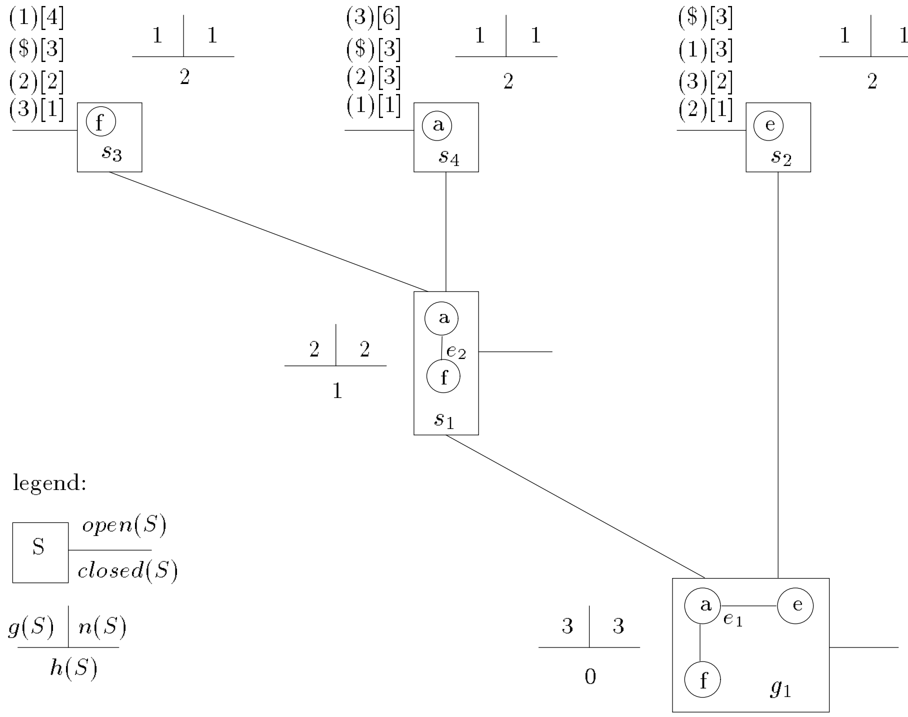

Figure 3.7: Network representation of the decomposition of the graph g_1 in Fig. 3.1. The contents of the *open* and *closed* lists are displayed after step 2 of the algorithm in Fig. 3.6 has been executed for the input graph g_2 of Fig. 3.1.

At run time, given the input graph g_2 of Fig. 3.1, the algorithm first matches the subgraphs s_2, s_3 and s_4 to g_2 and stores all *ec* subgraph isomorphisms that are found in the corresponding *open* lists. In Fig. 3.7 the contents of the *open* and *closed* lists are displayed after the first step of the algorithm has been executed. An *ec* subgraph isomorphism from a subgraph consisting of a vertex v to the vertex v_I of the input graph is denoted by $(v_I)[c]$ where c indicates the edit operation costs. For example, the *open* list of the graph s_3 contains four *ec* subgraph isomorphisms of which (3)[1] has minimum cost. Before the algorithm starts to combine *ec* subgraph isomorphisms in steps 3-6, the future cost estimation h along with the auxiliary functions g and n must be computed for s_1, s_2, s_3, s_4 and g_1. In Fig. 3.7 the values of g,n and h for each subgraph are displayed along with the corresponding network nodes (see legend). For example, the computation of

$$g(s_1) = \begin{cases} C[closed(s_1)] & \text{if } closed(s_1) \text{ is not empty} \\ n(s_1) & \text{otherwise} \end{cases}$$

according to Eq. 3.10 requires that $n(s_1)$ is computed because $closed(s_1)$ is yet empty. The computation of

$$n(s_1) = MIN\{C[open(s_1)]\ ,\ g(s_3) + n(s_4)\ ,\ n(s_3) + g(s_4)\}$$

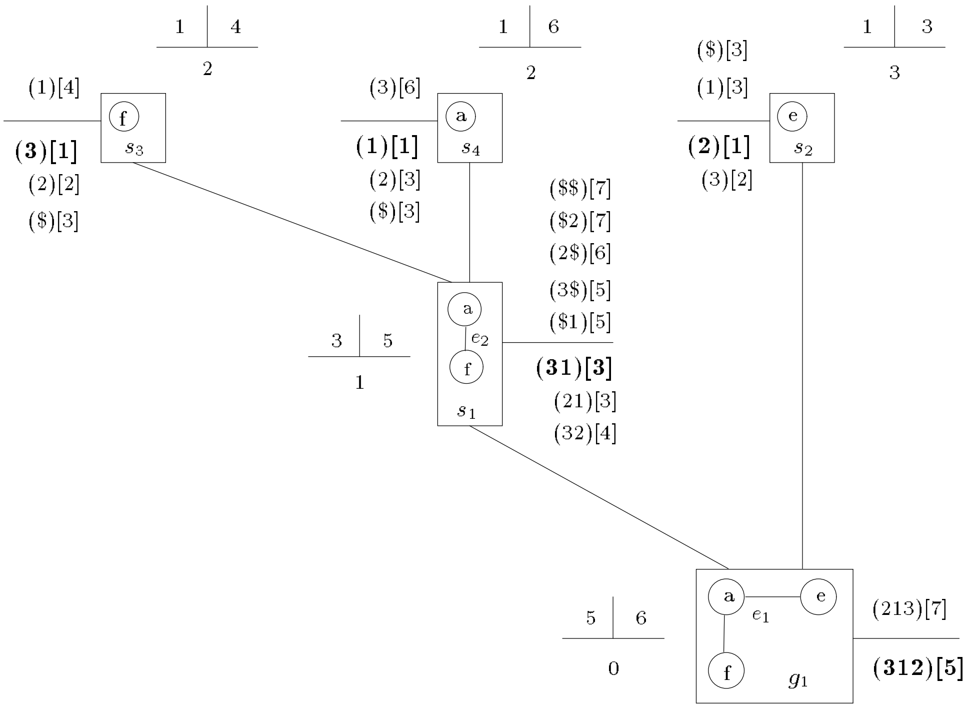

Figure 3.8: The network representation of the decomposition of g_1 in Fig. 3.1 after termination of the new algorithm.

according to Eq. 3.12 in turn requires that $g(s_3), n(s_3)$ and $g(s_4), n(s_4)$ are computed. Clearly, $g(s_3) = n(s_3) = g(s_4) = n(s_4) = 1$, because $closed(s_3)$ and $closed(s_4)$ are empty and $C[open(s_3)] = C[open(s_4)] = 1$. Consequently, the lower bound estimation for an *ec* subgraph isomorphism of s_1 is $n(s_1) = g(s_3) + n(s_4) = 2$ and, furthermore, $g(s_1) = n(s_1) = 2$. Based on these values, it is now possible to compute the future cost $h(s_2)$ of s_2, i.e. $h(s_2) = h(g_1) + g(s_1) = 0 + 2 = 2$. The future cost estimation of s_1, s_3, and s_4 can be computed analogously.

After all future cost estimations have been computed, the algorithm will execute repeatedly steps 3-6 and generate new *ec* subgraph isomorphisms. In the situation described in Fig. 3.7, $C[open(s_i)] + h(s_i)$ evaluates to 3 for each $i = 2, 3$ or 4. Thus, it must be randomly decided out of which $open(s_i)$ the minimal *ec* subgraph isomorphism is removed and added to $closed(s_i)$. The algorithm continues to combine *ec* subgraph isomorphisms and update the future cost estimations until an *oec* subgraph isomorphism for g_1 has been found in $closed(g_1)$. In Fig. 3.8 the contents of the *open* and *closed* lists after termination of the algorithm are displayed. Note that the list $closed(g_1)$ of the model graph g_1 contains exactly one *ec* subgraph isomorphism, namely (312)[5], which maps the vertex labeled f of g_1 onto the vertex labeled g of g_2, the vertex labeled a onto the vertex labeled b, and the vertex labeled e onto the vertex labeled d. Clearly, this result is identical to the *oec* subgraph isomorphism that was found by the traditional algorithm in Section 3.3. The *ec* subgraph isomorphisms which have contributed to the *oec* subgraph isomorphism are depicted in bold face in Fig. 3.8. The edit cost of the *oec* subgraph isomorphism amounts to 5 units. Hence, the algorithm terminated when for each subgraph S, $C[open(S)] + h(S)$ exceeded the threshold 5. Notice that thanks to the future cost estimation for s_2, $h(s_2) = 3$, the *ec* subgraph isomorphisms (1)[3] and ($)[3] are never removed from the $open(s_2)$ list. Consequently, the computational effort of combining them with *ec* subgraph isomorphisms in $closed(s_1)$ is avoided. Similarly, the *ec* subgraph isomorphisms (1)[4] in $open(s_3)$ and ($1)[5] in $open(s_1)$ are not used for combination because $h(s_3) = 2$ and $h(s_1) = 1$, respectively. Note that although there was only a single model graph used in this example, both the new algorithm and the future cost estimation function work equally well for databases with more than one model graph.

3.5 Computational Complexity Analysis

For comparison reasons, we examine in this section the computational complexity of both the new algorithm (Section 3.4) and the traditional algorithm (Section 3.3). The complexity analysis will be based on the following quantities:

L =the number of model graphs in the database,
n_1=the number of vertices in the common subgraph,
n_2=the number of vertices that are unique to each model graph,
n =the total number of vertices of a model graph, such that $n = n_1 + n_2$,
m =the number of vertices in the input graph.

We will study the theoretical performance of both algorithms in the best and the worst case. First, the traditional algorithm will be discussed. Given a single model graph with n vertices and an input graph with m vertices, the traditional algorithm expands exactly one state on each level of the search tree in the best case. As there are n levels and each state can be expanded into maximally m successor states, the total number of states in the search tree will be limited by $O(mn)$. In each of these states, at most $O(n)$ edge constraints will be tested. Furthermore, the lookahead procedure will be executed in each state. As mentioned in Section 3.3, the complexity of the lookahead procedure is $O(nm)$. Thus, for a single model graph, the traditional algorithm will perform at most $O(m^2n^2)$ steps. Given a database of L model graphs, each model graph will be matched individually with the input graph resulting in a best case complexity of

$$O(Lm^2n^2) \tag{3.13}$$

In the worst case, when the distance between the model and the input graph is maximal, each state in the search tree is successfully expanded into $O(m)$ successor states. As there are $O(n)$ levels, and $O(m^k)$ states on level k, the total number of states is bounded by $O(m^n n)$. Similarly to the best case, each state performs $O(n)$

edge constraint tests and $O(mn)$ computation steps in the lookahead procedure, resulting in a total complexity of $O(m^{n+1}n^2)$. For L model graphs in the database, the worst case complexity is thus bounded by

$$O(Lm^n n^3) \tag{3.14}$$

For a detailed analysis of the computational complexity of the traditional algorithm see Appendix A.3.

In order to study to new algorithm's computational complexity we will assume, without loss of generality, that the following simple decomposition scheme is used on the model graphs. Given a model graph G and a decomposition D of G, each subgraph S is decomposed into a subgraph S_v consisting of a single vertex, and the rest graph $S - S_v$. If there are several model graphs in the database, $G_1, \ldots, G_L$, and each G_i consists of the common subgraph S and a unique subgraph S_i then the model graphs are first decomposed into the common and the unique subgraph. Then, the common subgraph S and each of the unique subgraphs S_i are again decomposed according to the scheme described above. For a common subgraph of size n_1 and a unique subgraph of size n_2, the decomposition will contain $O(n_1 + n_2 L)$ subgraphs consisting of a single vertex and $O(n_1 + n_2 L)$ subgraphs consisting of more than one vertex. We now examine the run time behavior of the new algorithm based on this decomposition.

In the best case, given a single model graph with n vertices and an input graph with m vertices, the new algorithm finds $O(m/n)$ *ec* subgraph isomorphisms for each of the n subgraphs consisting of a single vertex in $O(mn)$ steps. These *ec* subgraph isomorphisms are then combined to form larger *ec* subgraph isomorphisms according to the decomposition. As there are $O(n)$ subgraphs with more than one vertex in the decomposition and there is exactly one *ec* subgraph isomorphism found for each of these subgraphs in the best case, the total number of combined *ec* subgraph isomorphisms is bounded by $O(nm/n) = O(m)$. For each of these combinations, at most $O(n)$ vertex and edge constraints are tested. Furthermore, every time a new *ec* subgraph isomorphism is found, the future cost estimation function of each subgraph in the decomposition must be updated, causing $O(n)$ computation steps. Thus, the total amount of computation steps for a single model graph is bounded by $O(mn + m(n + n)) = O(mn)$. For a database containing L model graphs, the computational effort can be categorized into three different groups of activity. First, the common subgraph is matched exactly once onto the input graph, requiring a computational effort of $O(m(n_1 + Ln_2))$ steps, where the additional factor Ln_2 denotes the number of steps for updating the future cost estimation of the unique subgraph components. Secondly, the unique subgraph of each model graph is matched with the input graph in $O(Lmn_2)$ steps and thirdly, all *ec* subgraph isomorphisms found for the common and the unique subgraphs must be combined, causing $O(Ln_1n_2)$ steps. Thus, the computational complexity of the new algorithm, given L model graphs, is bounded in the best case by

$$O(mn_1 + Ln_2(m + n_1)) \tag{3.15}$$

Note that in the extreme case of completely different model graphs, i.e. $n_1 = 0$, $n_2 = n$, Eq. 3.15 becomes $O(Lmn)$. Compared to the traditional algorithm (Eq. 3.13) the new algorithm is theoretically faster by a factor of $O(mn)$. In the other extreme case, when all model graphs are highly similar, i.e. $n_1 \to n$, $n_2 \to 0$, the size L of the database disappears from Eq. 3.15 and $O(mn)$ results. Hence, the algorithm is not only faster than the traditional algorithm by a factor of $O(mn)$ but also independent of the size of the database.

For the worst case, we again start with a single model and an input graph. We assume the same decomposition as discussed above. Now, for each subgraph in the decomposition, the complete set of possible *ec* subgraph isomorphism will be found. I.e., for a subgraph with k vertices, $O(m^k)$ *ec* subgraph isomorphisms will be found. As there are $O(n)$ subgraphs in the decomposition of the model graph and each is of a different size k with $k = 1, \ldots, n$, the total number of detected *ec* subgraphs isomorphisms is bounded by $O(m^n n)$. For each of these *ec* subgraph isomorphisms, $O(n)$ edge and vertex constraints must be tested and the future cost estimation functions must be updated in $O(n)$ steps. Thus, in the worst case, the total number of steps for a single model graph grows exponentially with n and is bounded by $O(m^n n^2)$. For a database with L model graphs, the computational effort consists again of three components. First, all *ec* subgraph isomorphisms for the common subgraph are found in $O(m^{n_1} n_1 (n_1 + L n_2))$ steps. Secondly, the subgraph isomorphisms for each of the unique subgraphs must be computed, requiring a total of $O(L m^{n_2} n_2 n)$ steps. Thirdly, the combination of the subgraph isomorphisms that were found in the previous steps requires $O(L m^n n_1 n_2)$ steps for each model graph. Hence, the computational complexity of the new algorithm is in the worst case bounded by

$$O(m^{n_1} n_1^2 \; + \; L m^{n_2} n_2 n \; + \; L m^n n_1 n_2) \tag{3.16}$$

Note that, similar to the best case, the influence of the database size depends on the size of the unique subgraphs. In the extreme case when all model graphs are completely different, i.e. $n_1 = 0, n_2 = n$, Eq. 3.16 becomes $O(L m^n n^2)$. Consequently, the new algorithm's complexity is smaller than the traditional algorithm's complexity by a factor of n when all model graphs are disjoint. In the other extreme case, when all model graphs are highly similar, i.e. $n_1 \to n, n_2 \to 0$, Eq. 3.16 becomes $O(m^n n^2)$. Thus, the new algorithm is independent of the number of model graphs in the database.

3.6 Practical Experiments

In order to study the behavior of the new algorithm in practice, we performed a number of experiments with randomly generated graphs. The new algorithm (NSG) was implemented in C++ and run on a SunSparc Workstation. For comparison reasons, we also implemented the traditional algorithm (TA) described in Section

Experiments		Parameters for the graph generation						
Nr.	Fig.	Vertices	Edges	Labels	Error	Database	C_s	C_d, C_e
1	3.9,3.10,3.11	10 - 36	15 - 48	10	1 - 10	1	1	5
2[a]	3.12,3.14,3.13	20	30	10	10	1 - 10	1	5
3	3.15,3.16	20	30	3 - 10	1 - 7	1	1	5
4[b]	3.17,3.18	20-40	30 - 60	10	1 - 10	1	1	5
5	3.19	10 - 100	15 - 150	20	5	1	1	5
6	3.20	20	30	10	5	1 - 100	1	5
7[c]	3.21,3.22	10	20	30	1 - 10	1 - 10	1	5
8[d]	3.23	15	20	10	2	1	3	1 - 5
9	3.24	15	20	10	4	1	3	1 - 5
10[e]	3.25	15	20	10	1-10	1	∞	1

[a] The size of the common subgraph of the models was increased from 5 to 18 vertices.
[b] Only the size of the input graph was varied, while the size of the model graph was fixed at 20 vertices.
[c] Only the edges were labeled in this experiment.
[d] There were two vertices missing in the input graph.
[e] The number of missing vertices was increased from 1 to 10.

Table 3.1: Parameters of the practical experiments.

3.3 along with the lookahead procedure described in [WYC90]. The graphs used in the experiments were randomly generated on the basis of the following quantities:

- the number of vertices
- the number of edges
- the number of different vertex labels (except for experiment 7, all edges were unlabeled)
- the number of graphs in the database
- the degree of label distortion
- the number of missing vertices

For each experiment, a set of labeled model graphs consisting of a specified number of vertices and edges was randomly generated. Each vertex in a model graph was randomly assigned a label out of a finite subset of the integers. From each model graph a corresponding input graph was derived by copying and permuting the vertices and edges. Then, the input graph was subject to label distortion by randomly changing the labels of a specified number of vertices. Finally, a specified number of vertices were deleted from the input graph along with their incident edges. As mentioned before, an important factor in any practical application of *ec* subgraph

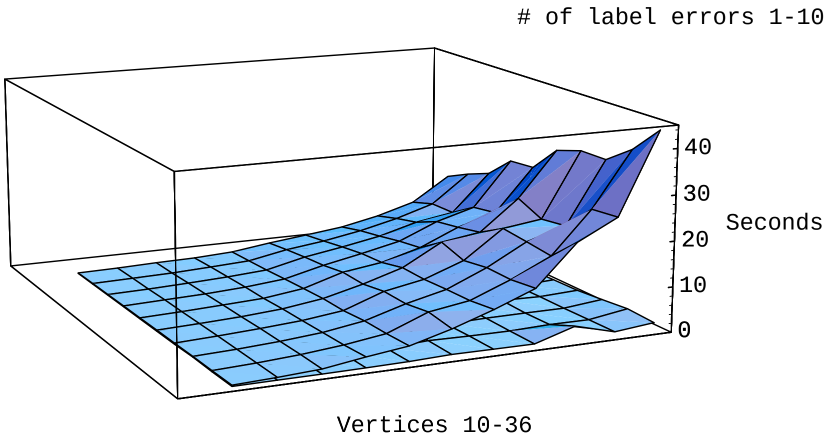

Figure 3.9: Time in seconds for an increasing number of vertices in the model graphs and an increasing number of label errors in the input graph (lower plane in the right corner denotes NSG, upper plane denotes TA).

isomorphism algorithms is the definition of the costs of the edit operations. In all our experiments, the cost for substituting a label l_1 with a label l_2 was set to $C_s|l_1 - l_2|$, where $|l_1 - l_2|$ denotes the absolute difference between l_1 and l_2, and C_s is a constant. The cost of inserting a vertex and inserting or deleting an edge was constantly set to C_d and C_e, respectively. The relationship between the costs for substituting labels, C_s, and the costs for inserting or deleting vertices and edges, C_d and C_e, strongly determines the result and the performance of the matching algorithm. Therefore, the values of C_s, C_d and C_e were also subject to study and used as parameters for the experiments.

The objective of each experiment was to measure and compare the computation times needed by NSG and TA in detecting the optimal error-correcting subgraph isomorphism from any of the model graphs to the input graph. Each measurement was repeated 20 times with different graphs and the average time was taken as the final result in order to account for the random nature of the graph generation process. In Table 3.1, an overview of the experiments is given. Notice that for the first 7 experiments, the cost C_s of substituting labels (column 8 in Table 3.1) was constantly set to 1 and the cost of inserting and deleting edges or vertices was set to 5. Consequently, in these experiments, both algorithms tried to correct at least five labels (by one unit) before the deletion of a vertex or the deletion and insertion of an edge was considered. In experiments 8,9 and 10 the influence of the relationship between the substitution cost, on the one hand, and the deletion and insertion costs, on the other hand, was directly studied.

In the first experiment, we examined the influence of the size of the model graphs and the number of label distortions on the run time performance of NSG and TA. In the beginning, a single model graph consisting of 10 vertices and 15 edges was randomly generated. From this model graph an input graph was derived by copy-

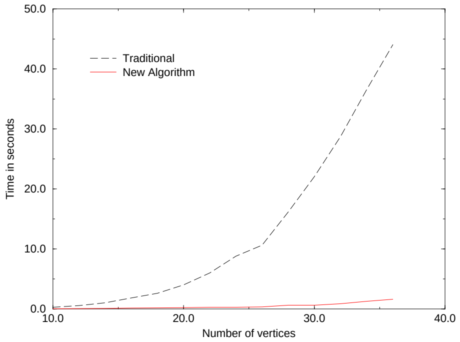

Figure 3.10: Cut through Fig. 3.9 along the vertex axis for a label error of 10 in the input graph.

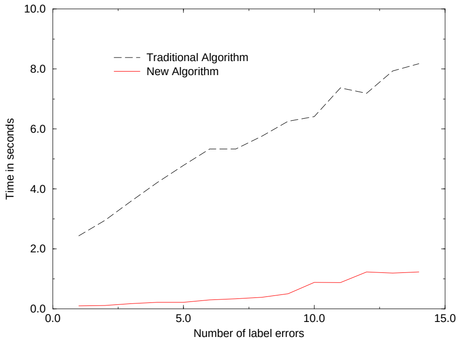

Figure 3.11: Cut through Fig. 3.9 along the label error axis for a model and input graph consisting of 20 vertices and 30 edges.

ing its vertices and edges and distorting its vertex labels according to the specified number of label errors. Note that the model and the input graph were structurally isomorphic and only the labels were distorted. The number of label errors was gradually increased from 1 to 10 and for each setting a new input graph was generated. The number of vertices in the model and input graph was also increased from 10 to 36 along with the number of edges that was increased from 15 to 48. There were a total of 10 vertex and no edge labels used in this experiment. Hence, in a graph of 30 vertices, there were on the average 3 vertices with identical labels. The result of the first experiment is displayed in Fig. 3.9. The lower (upper) plane in the right corner denotes the times of NSG (TA). Clearly, NSG outperforms TA for a growing model graph size and also for a growing number of label errors in the input graph. In Fig. 3.10 a cut through Fig. 3.9 along the vertex axis is displayed. The number of errors in the labels of the input graph was constantly set to 10. The high performance of NSG is due to the efficiency of the future cost estimation function h. As explained in Section 3.4.3, the function h uses the information that is found while matching the smaller subgraphs of the models, whereas the lookahead procedure of TA bases its estimation on tentative matchings only. Thus, the estimation provided by h is more reliable. Furthermore, the lookahead procedure of TA has a quadratic complexity while the future cost estimation of NSG is only linear in the number of model graph vertices. Therefore, NSG performs better than TA. In Fig. 3.11, a cut through Fig. 3.9 along the label error axis with the number of vertices fixed at 20, confirms that NSG is also faster than TA when the number of label errors is growing. The explanation for this behavior lies again in the efficiency of the future cost estimation of NSG.

In the experiment described above, there was only one model graph in the database. Consequently, NSG's ability to share common substructures among different model graphs could not be studied. Therefore, in the second experiment, we varied both the number of model graphs in the database and the size of the subgraph

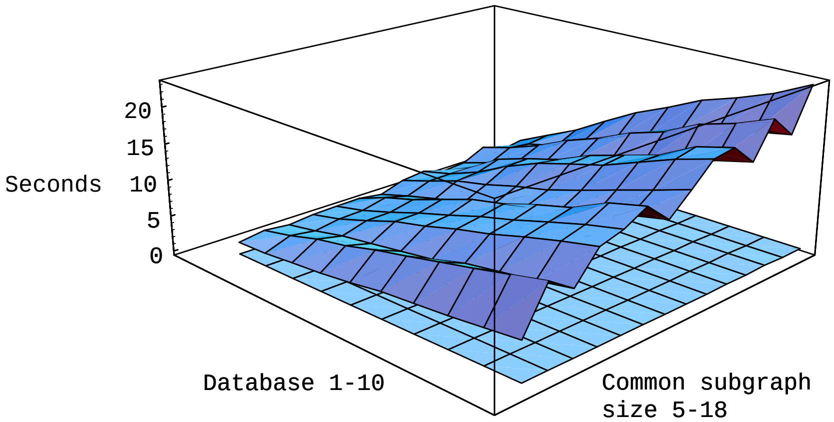

Figure 3.12: Time in seconds for a growing number of model graphs in the database and a growing common subgraph of the model graphs (lower plane in the right corner denotes the time of NSG, upper plane denotes the time of TA).

that was common to all the model graphs. Each of the model graphs consisted of 20 vertices and 30 edges. The size of the database was gradually increased from 1 to 10 model graphs. For each size of the database, we also varied the size of the common subgraph starting with 5 vertices and ending with 18 vertices. Thus for a database with 10 model graphs and a common subgraph of size 18, each model graph consisted of an identical subgraph with 18 vertices and 27 edges and a unique subgraph with 2 vertices and a single edge. The number of distorted labels was constantly set to 10. In Fig. 3.12 the average computation times for the second experiment are given. As was expected from the computational complexity analysis, TA is linearly dependent on the size of the database, while NSG has only sublinear dependency . In Fig. 3.13 a cut through Fig. 3.12 is displayed with the common subgraph constantly set to 10 vertices. Notice that NSG also requires more time for a growing database. However, it performs much better than TA. Another advantage of NSG is documented in Fig. 3.14 where a cut through Fig. 3.12 along the axis of the common subgraph size is given for 10 model graphs in the database. Clearly, TA's computation time increases for a growing common subgraph. The algorithm NSG, on the other hand, becomes slightly faster when the common subgraphs grows (not visible on this scale). The reason for this behavior is the compact representation of the model graphs in NSG. The larger the common subgraph is, the more efficient this representation becomes, and the faster NSG detects the *oec* subgraph isomorphisms with least costs from one of the model graphs to the input. The algorithm TA, however, must test each model graph individually. Consequently, for a growing common subgraph, the search space becomes larger for each model graph and the total computation time increases.

So far, the number of available labels for the model and the input graphs was constantly kept at 10. Previous research has shown that the exact version of NSG (where no graph edit operations are allowed) performs poorly when the number of

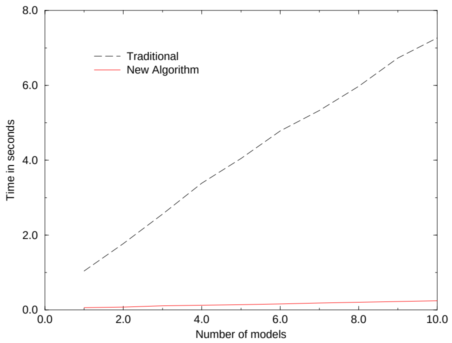

Figure 3.13: Cut through Fig. 3.12 along the database axis with a common subgraph of size 10.

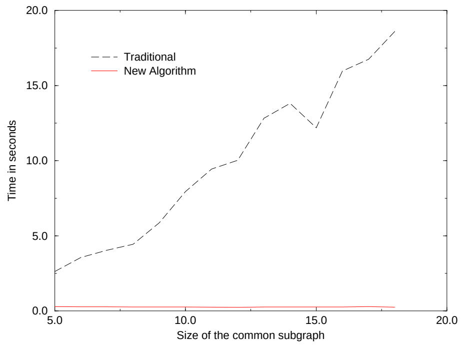

Figure 3.14: Cut through Fig. 3.12 along the common subgraph axis with 10 model graph in the database.

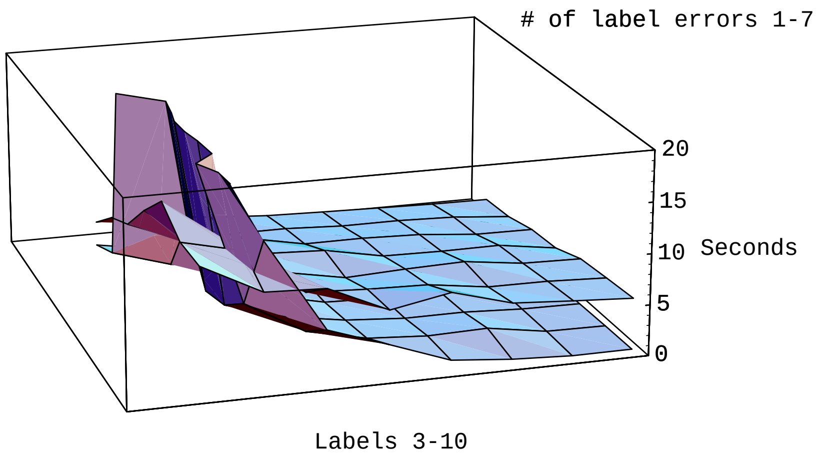

Figure 3.15: Time in seconds for a growing number of labels and a growing number of label errors (lower plane in the right corner denotes the time of NSG, upper plane in the right corner denotes the time of TA).

labels is small (see Chapter 2). Therefore, in the third experiment, we examined the influence of the number of labels on the performance of the algorithms. There was exactly one model graph consisting of 20 vertices and 30 edges in the database. The number of labels was varied between 3 and 10. Thus, in the beginning, there were on the average approximately 7 vertices in the model graph with identical labels. In the end, for 10 labels, there were on the average 2 vertices with identical labels. The edges were all unlabeled. The number of distorted vertex labels in the input graph was also varied between 1 and 7. The results of this experiment are documented in Fig. 3.15. Additionally, in Fig. 3.16 a cut through Fig. 3.15 along the axis of the number of labels is given for a constant label error of 5. For a decreasing number of labels, the performance of NSG became rapidly worse, i.e. for 4 labels NSG was already slower than TA. This behavior can be explained as follows. If

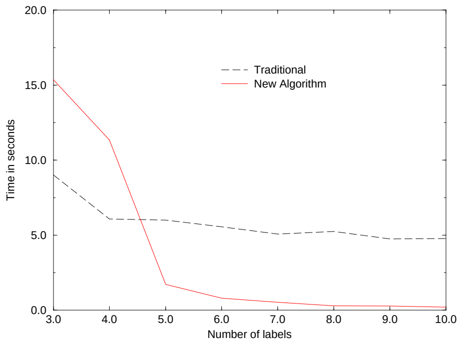

Figure 3.16: Cut through Fig. 3.15 along the label axis with a constant label error of 4.

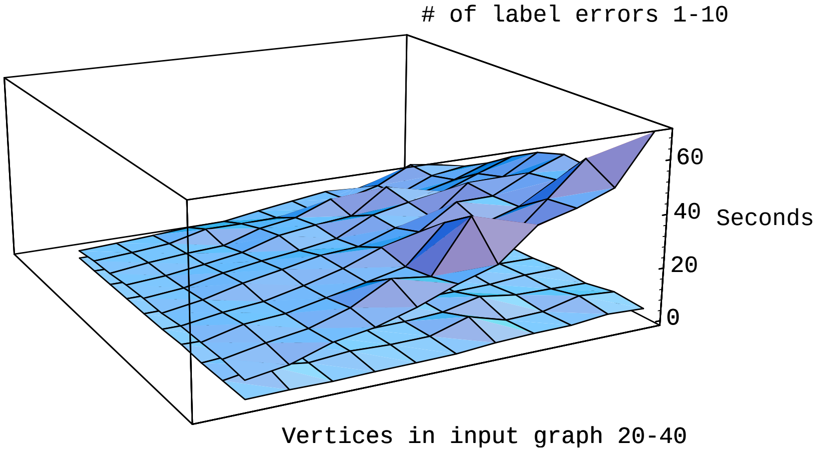

Figure 3.17: Time in seconds for a growing number of vertices in the input graph and a growing number of label errors (lower plane in the right corner denotes the time of NSG, upper plane in the right corner denotes the time of TA).

there are only few labels in a model graph, the number of isomorphic subgraphs in this model graph can be quite large and NSG will represent each of these subgraphs only once. Although this property of NSG is desirable when the database contains more than one model graph, it turns out to be a disadvantage for the future cost estimation function presented in Section 3.4.3. This function estimates the future cost by using matching information that was found for subgraphs different from the current subgraph of the model graph. If these subgraphs are, however, represented only once in the decomposition of the model graph then the future cost estimation function does not gain any new information. As a consequence, NSG requires more time and the behavior documented in Figs. 3.15 and 3.16 results.

In all of the experiments described so far, the model and the input graphs were always of equal size. Thus, we were in fact looking for *ec* graph isomorphisms. In the fourth experiment, we explicitly increased the size of the input graph from 20 to

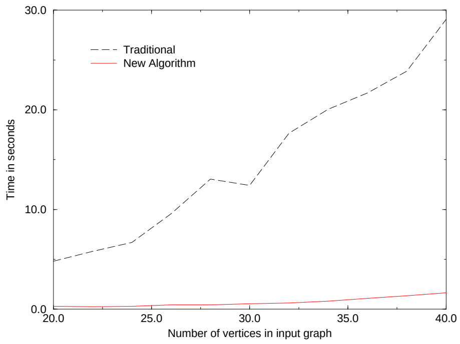

Figure 3.18: Cut through Fig. 3.17 along the input graph axis with a constant label error of 5.

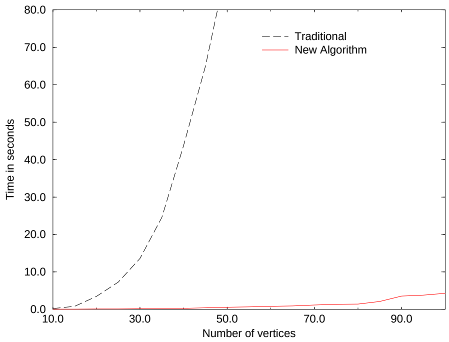

Figure 3.19: Time in seconds for an increasing number of vertices in the model graphs and a constant label error of 5.

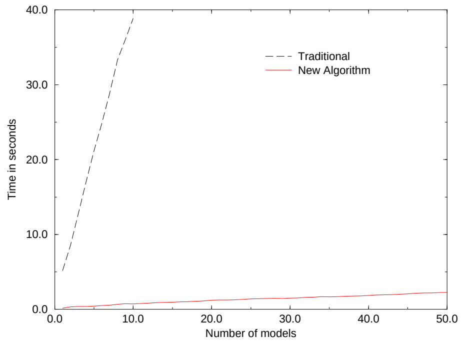

Figure 3.20: Time in seconds for an increasing number of models in the database and a constant label error of 5.

40 vertices along with the edges that were increased from 30 to 60 while the size of the model graph remained constant at 20 vertices and 30 edges. At the same time, the number of errors in the labels of the input graph was varied between 1 and 10. The results of this experiment are given in Fig. 3.17. We observe that TA again required more time than NSG for both an increasing error and an increasing input graph size. For a constant error of 5 and an input graph whose size is increased from 20 to 40 vertices TA required time that ranged from 5 to 30 seconds, while NSG took only between 0.3 and 1.64 seconds (see Fig. 3.18). Again, the main reason for the behavior of NSG is the future cost estimation function. In particular, the fact that this procedure is only linear in the number of model graph vertices and independent of the input graph size, largely contributes to the efficiency of NSG.

The previous experiments have shown that NSG is very efficient for graphs with up to 20 vertices and databases containing at most 10 model graphs. In order to explore the limits of NSG with respect to the size of the model graphs and the size of the database, we performed two more experiments. In the fifth experiment, documented in Fig. 3.19, the size of the model and input graph was increased from 10 to 100 vertices along with the edges that were increased from 15 to 150. There were 20 different vertex labels available and the labels of 5 input graph vertices were distorted by one unit. Notice that the computation time of TA is only plotted for graphs with less than 50 vertices for which it already required more than 80 seconds while NSG required in the same situation only 0.5 seconds. In the limit, when the model graph consisted of 100 vertices and 150 edges, NSG required only 5 seconds on the average.

In the sixth experiment, documented in Fig. 3.20, we tested the behavior of NSG for large databases of graphs. Similar to the second experiment (Fig. 3.14), we generated a growing database of model graphs starting at 1 and ending at 50 model graphs. Each graph consisted of 20 vertices and 30 edges. Unlike the second experiment, where a common subgraph was explicitly defined, the graphs in this experiment were generated independently of each other. However, the total number of labels was restricted to 10 such that common subgraphs of various sizes naturally evolved. As was to be expected, NSG's performance was only sublinearly dependent on the size of the database. While TA required 40 seconds for a database containing 10 models, NSG finished within 3 seconds for a database containing 50 models.

A special case of the *oec* subgraph isomorphism problem is the weighted graph matching problem in which all of the information in the graph is attached to the edges instead of the vertices. Due to the fact that NSG decomposes the model graphs into subgraphs by splitting the set of vertices and first trying to match single vertices onto the input graph, it can be said that NSG is strongly vertex oriented in its matching philosophy. Thus, NSG may not be suited for the weighted graph matching problem, where the vertices carry no semantic information. In order to better understand the behavior of NSG in this situation, we performed a seventh experiment. In this experiment, the model graphs consisted of 10 unlabeled vertices and 20 labeled edges. Both the number of distorted edge labels and the size of the database were varied. The results are given in Fig. 3.21. Obviously, for a small number of errors in the edge labels, NSG was superior to TA. Furthermore, NSG was only sublinearly dependent on the database size. However, if the number of distorted edge labels was increased, NSG performed worse than TA. In Fig. 3.22 a cut through Fig. 3.21 along the edge label error axis for a single model graph indicates that NSG's performance was heavily affected by the number of edge label errors and became slower than TA for more than 7 errors[2]. The behavior of NSG for graphs with unlabeled vertices and labeled edges is therefore completely different from its behavior for graphs with labeled vertices and unlabeled edges. As mentioned in the beginning of this paragraph NSG is strongly vertex oriented. Information about mislabeled vertices is gathered very early in the matching process and can then be used by the future cost estimation function. However, depending on the

[2]Note that the edge label error ranged from 1 to 15 in the Fig. 3.22 while in Fig. 3.21 the edge label error ranged from 1 to 10 only.

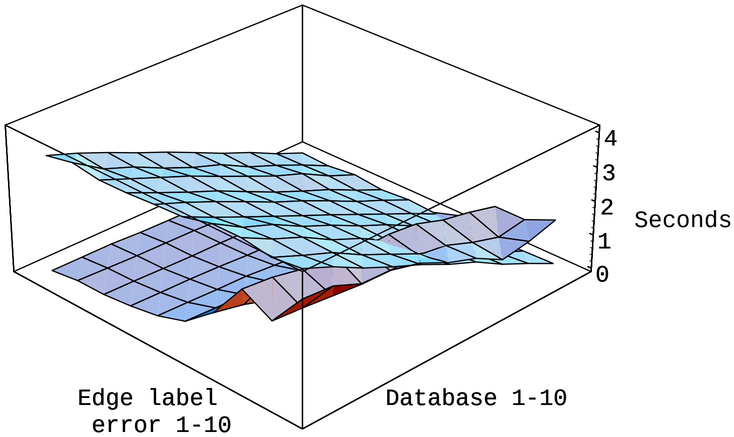

Figure 3.21: Time in seconds for a growing number of errors in the edge labels and a growing number of models in the database (lower plane in the left corner denotes the time of NSG, upper plane in the left corner denotes the time of TA).

decomposition of the model graphs, certain edges are not considered by the matching process until the very end. The lookahead procedure of TA, on the other hand, considers edges from the very beginning, thus reducing the search space in this case more efficiently. Nonetheless, due to the fact that TA is linearly and NSG only sublinearly dependent on the number of models in the database, there always exists a size of the database for which NSG finally outperforms TA.

In all of the previous experiments, the input graph that was derived from a model graph was distorted in its labels but structurally isomorphic to the model graph. In the experiments described in the following, we concentrated on structural distortions and the influence of the cost definitions on the matching process. Clearly, the application of a specific edit operation depends on its cost. For example, given a model graph G, an input graph G_I, substitution costs $C_s = 1$ and deletion costs $C_d = 5$, let three vertices of the input graph be mislabeled by one unit. Then, the graph distance $d(G, G_I)$ is 3 and consequently, any ec subgraph isomorphism that implies the deletion of a vertex or an edge is not further pursued by neither TA nor NSG. On the other hand, if the label error grows beyond 5, then both algorithms will consider the possibility of a missing vertex, a missing edge or an extraneous edge. It is easy to see that the relationship between C_d and C_s strongly influences the search space of both algorithms. In experiment 8 and 9, we studied this influence more closely.

In the eight experiment, we varied the cost of inserting a vertex, C_d, from 1 to 5, while the weight for substituting a label, C_s, was constantly kept at 3. Notice that the cost for inserting and deleting an edge, C_e, was defined to be identical to C_d, $C_e = C_d$. The database contained exactly one model graph of 15 vertices, 20 edges and 10 vertex labels. The input graph was an identical copy of the model graph except for 4 vertex labels that were distorted by one unit. Thus, the minimal

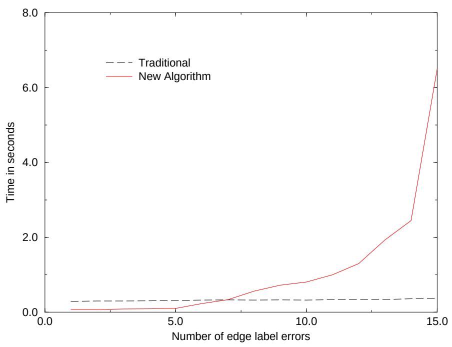

Figure 3.22: Cut through Fig. 3.21 along the axis of the edge labels for 1 model graphs in the database.

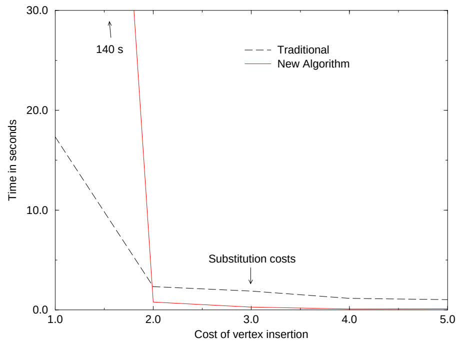

Figure 3.23: Time in seconds for growing insertion and deletion costs C_d, C_e while the substitution costs C_s were constantly set to 3 and there were 4 mislabeled vertices in the input graph.

graph distance from the model to the input graph was $4 * C_s = 4 * 3 = 12$. In Fig. 3.23, the computation times of TA and NSG are given. For $C_d = 1$, NSG required approximately 150 seconds in order to find the correct solution while TA terminated after 18 seconds. Next, for $C_d = 2, 3, 4, 5$, both algorithms finished within a couple of seconds with NSG being slightly faster. Clearly, the setting $C_d = 1$, $C_s = 3$ modeled a graph matching problem in which the deletion and insertion of graph parts was more likely than the substitution of vertex labels. Therefore, NSG first deleted all vertices before label substitutions were considered and the number of combinations for small subgraphs became very large. As soon as the cost C_d was increased, many of these combinations became too expensive and were no longer considered. Consequently, the computation time decreased. In general, it can be observed that TA's performance degenerated more gracefully than that of NSG.

For the ninth experiment documented in Fig. 3.24, the same parameter settings as in the eight experiment were made except for the fact that two vertices were randomly deleted from the input graph, while the remaining vertices were correctly labeled. In other words, the input graph was subject to structural distortions. It can be observed that for missing vertices in the input graph the computation time was growing for both algorithms when the deletion costs C_d was increased. In the limit, for $C_d = 5$, NSG performed worse than TA. The explanation for this behavior is similar to the explanation given for experiment 8. The cost definition $C_d = 5, C_s = 3$ favors errors in the vertex labels, while structural errors are considered less likely. Thus, before any vertex deletions are tried, both algorithms attempt to substitute the labels in the model graphs by the labels in the input. We can conclude from experiments 8 and 9 that NSG outperforms TA provided that the cost definition describes the graph matching problem accurately. For example, if substitution costs are low compared to the insertion costs and the input graph is only distorted in the labels, then NSG will perform better than TA. If, on the other hand, unlikely errors

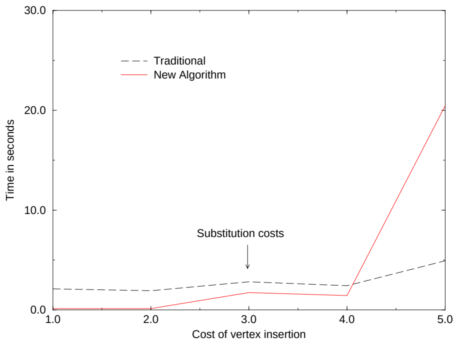

Figure 3.24: Time in seconds for growing insertion and deletion costs C_d, C_e while the substitution costs C_s were constantly set to 3 and there were 2 vertices missing in the input graph.

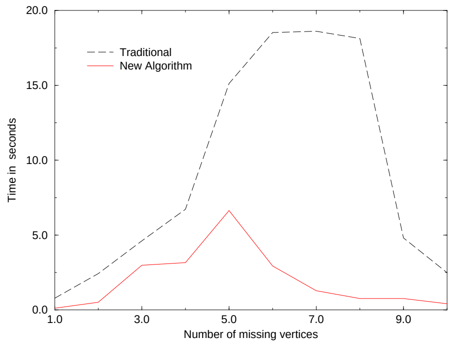

Figure 3.25: Time in seconds for a growing number of missing vertices in the input graph with deletion costs $C_d = 1$ and substitution costs $C_s = \infty$.

occur then TA will be more efficient.

At last, we performed an experiment in which only structural errors were considered. We assumed that label substitutions were highly improbable and set $C_s = \infty$. The cost for deleting a vertex (and inserting or deleting an edge) was set to $C_d = C_e = 1$. There was exactly one model graph in the database, consisting of 15 vertices, 20 edges and 10 different vertex labels. The input graph was systematically distorted by removing one vertex after the other until there were only 5 vertices left. The computation time plotted in Fig. 3.25 indicates that NSG was always more efficient for any number of missing vertices than TA. Theoretically, the worst case in this experiment arised when half of the vertices of the input graph were missing, i.e. out of the 15 vertices, 7 vertices were missing. Clearly, the worst behavior of both algorithms occurred when 5,6 or 7 vertices were missing. However, NSG successfully restricted its search space thanks to the future cost estimation function.

3.7 Advantages and Disadvantages of the New Algorithm

The results of the computational complexity analysis and the practical experiments provide a solid basis for a general analysis of the advantages and disadvantages of the new algorithm. The computational complexity analysis indicated that the new algorithm is theoretically faster than the traditional algorithm due to two major factors. First, the compact representation of the model graphs in the decomposition structure leads to a performance that is only sublinearly dependent on the database size. Secondly, the future cost estimation is based on matching information that is found by the algorithm during the regular search process. Therefore, the quality of the estimation is very good. Furthermore, the computational complexity of the future cost estimation is only linear in the number of vertices compared to the quadratic complexity of the lookahead procedure of the traditional algorithm. In the practical experiments, it was confirmed that the new algorithm is indeed more efficient than the conventional algorithm in many cases. In particular, the new algorithm outperformed the conventional algorithm in situations where the model graphs consisted of labeled vertices and the cost of the edit operations adequately modeled the actual label distortions and structural errors. However, the new algorithm was badly fitted for situations where the model graphs were unlabeled or only labeled in the edges, and where unlikely distortions occurred. Nonetheless, even in these situations its ability to compactly represent the model graphs remained an advantage such that it eventually outperformed the conventional algorithm when the database of models grew very large.

3.8 Concluding Remarks

In this chapter, we introduced a new method for the computation of optimal error-correcting subgraph isomorphisms from a set of a priori known model graphs to an input graph. The novelty of our approach is the off-line preprocessing of the database of model graphs. In this preprocessing step, the model graphs are compiled into a representation in which common subgraphs of different model graphs are stored only once. At run time, this representation is used to detect the optimal error-correcting subgraph isomorphism from any of the model graphs to the input graph in time that is only sublinearly dependent on the database size. Furthermore, the compact representation of the model graphs allows the implementation of an efficient future cost estimation function. With this function, the new algorithm outperforms conventional algorithms not only when large databases of model graphs are involved, but also for single model graphs under various scenarios.

The new method was described for unrestricted, general graphs and a set of edit operations that is powerful enough to correct any kind of graph distortion. Consequently, the adaption of the new method to specific applications is straightforward. A first application in the field of document image analysis, i.e. graphical symbol recognition and automatic symbol acquisition, is presented in Chapter 5. Furthermore, due to the principle of dividing the model graphs into smaller subgraphs, there is a potential for parallelization inherent in the new method.

Chapter 4

A Decision Tree Approach to Graph and Subgraph Isomorphism Detection

4.1 Introduction

Graph and subgraph isomorphism are concepts that have been intensively used in various applications such as analysis of chemical structures [RB79], graph grammars and semantic networks [Ehr92] or pattern recognition [LRS91, BA83, LKG90, LK89]. Furthermore, graph and subgraph isomorphism have been successfully applied in computer vision for the recognition of 3-D objects [WLR89, CK92, KK91, HS88, Won90, CH81, Won92]. It is well known that the problem of subgraph isomorphism detection is NP-complete [GJ79]. That is, any of the known algorithms requires an exponential number of computational steps in the worst case. For the problem of graph isomorphism detection, it is still an open question whether it can be solved by a polynomial algorithm. So far, all algorithms that have been proposed in the literature have an exponential time complexity in the worst case. In the following, we will give a brief overview of graph and subgraph isomorphism algorithms. For a comprehensive overview of publications on the topic of graph isomorphism before the year 1979 see [Gat79, RC77].

There are two basic approaches that past research has taken towards the problem of graph and subgraph isomorphism detection. The first approach concentrates on graph isomorphism detection only. It 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 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. Moreover, no extension of group-theoretic methods to subgraph isomorphism detection has been proposed so far.

The second approach to graph and subgraph isomorphism is more practically oriented and aims directly at developing computational procedures. Most of the algorithms are based on tree search with backtracking. One of the first publications in the field is the algorithm 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]. More recent work is described in [MLL92, FFG90] where the graph and the subgraph 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 [BM95].

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 input graph that must be tested. If the number of graphs in the database is large then the sequential matching of the input to each model graph becomes computationally very costly. As a consequence, several systems have been proposed 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 [SH82], Shapiro and Haralick propose to organize the graphs in clusters according to their mutual similarity and index each cluster by means of a representative graph. 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 method described in Chapter 3 and 4 and in [BM95]. Also based on the idea of organizing the graphs in a hierarchy is the approach described in [SB95]. Of high interest to the present chapter are the indexing approaches that are proposed in [Ike87, 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 chapter, we propose a general method for the problem of graph and subgraph isomorphism detection from an input graph to a set of a priori known model graphs. 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 integrated 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 find the subgraph isomorphisms from an input graph to the models. The main advantage of the new method is its time complexity that is only quadratic in the number of nodes of the input graph. 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 to applications where the underlying graphs are rather small, but where there is a potentially large number of model graphs and almost real time behavior is required.

The rest of this chapter is organized as follows. In Section 4.2, the basic definitions and notations are given. The basic idea and an overview of the new algorithm is then presented in Section 4.3. A more detailed description of the procedures and data structures of the new algorithm is given in Sections 4.4 and 4.5. The results of a computational complexity analysis are described in Section 4.6. 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 4.7. The practical performance of the new algorithm is studied in a number of experiments that are documented in Section 4.8. Finally, in Section 4.9, a summary and conclusions are given.

4.2 Definitions and Notations

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

Definition 4.1: 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.

□

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 4.2: 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 $V = \{v_1, v_2, \ldots, v_n\}$. Then G can also be represented by its adjacency matrix $M = (m_{ij})$, $i, j = 1, \ldots, n$, 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 4.3: A $n \times n$-matrix $P = (p_{ij})$ is called a *permutation matrix* if

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

3. $\sum_{j=1}^{n} p_{ij} = 1$ for $i = 1, \ldots, n$

$\square$

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

$$M' = PMP^T \tag{4.1}$$

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

$$M_2 = PM_1P^T \tag{4.2}$$

$\square$

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 a graph isomorphism between two graphs G_1 and G_2 is equivalent to finding a permutation matrix P for which (2) holds true.

Definition 4.5: 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. $\square$

Definition 4.6: Let $M = (m_{ij})$ be an $n \times n$ 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 $j = 1, \ldots, m$. $\square$

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 if there is an $n \times n$ permutation matrix P such that

$$M_1 = S_{m,m}(PM_2P^T). \tag{4.3}$$

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 $S_{m,m}(PM_2P^T) = S_{m,n}(P)M_2(S_{m,n}(P))^T$.

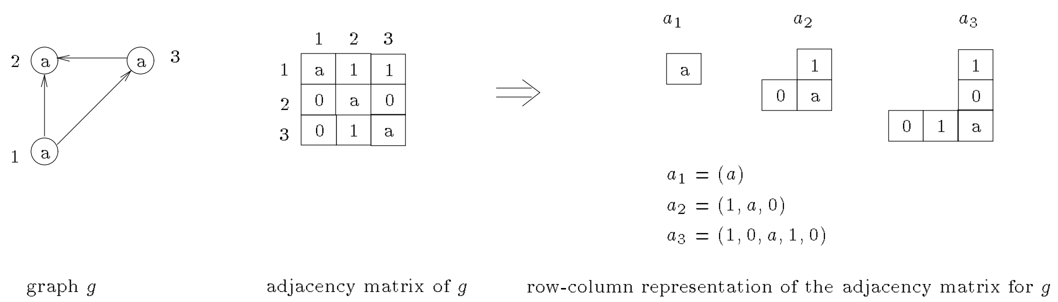

Figure 4.1: The row-column representation of an adjacency matrix.

4.3 Subgraph Isomorphism by Means of Decision Trees - The Basic Idea

We assume that there is a set of model graphs that are known a priori and an input graph that 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,

$$A(G) = \{M_P | M_P = PMP^T \text{ where } P \text{ is an } n \times n \text{ permutation matrix}\} \qquad (4.4)$$

Now the subgraph isomorphism problem can be restated in terms of $A(G)$. 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 $m \leq n$, we want to 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. $M_I = PMP^T$.

Apparently, the set $A(G)$ can be organized in a decision tree. The features that will be used for the classification process are the individual elements in the adjacency matrices, arranged in a particular way. We say that matrix M consists of an array of so-called *row-column* elements a_i, where each a_i is a vector of the form

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

The matrix can then be written as

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

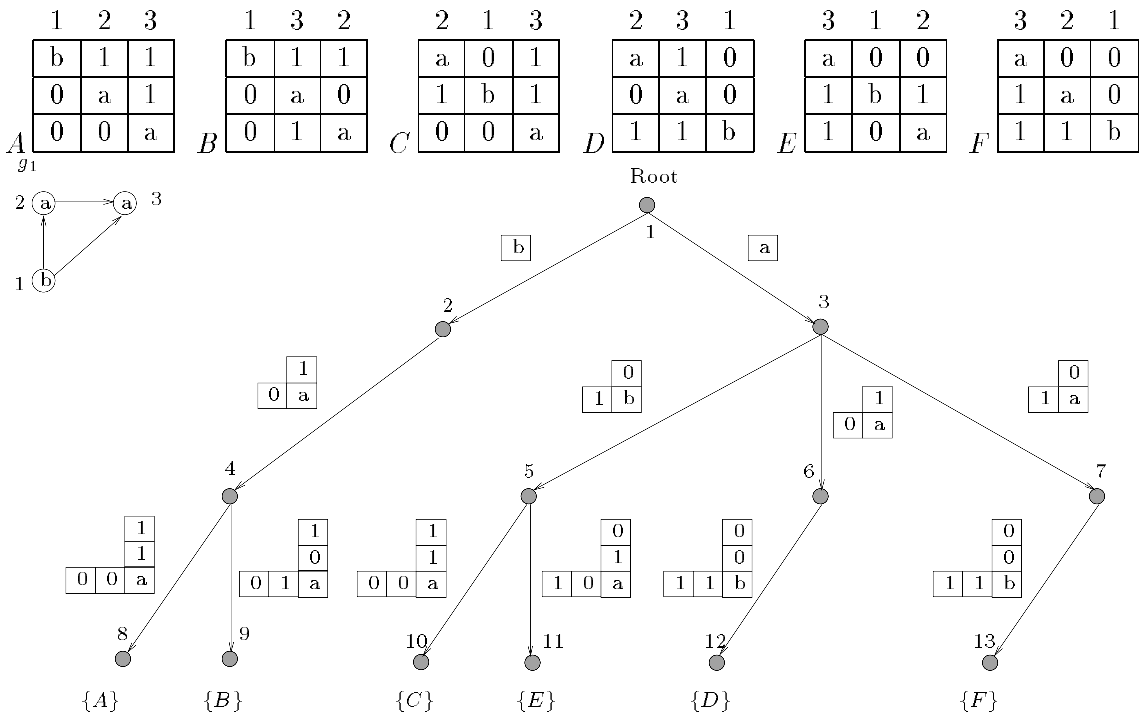

Figure 4.2: Decision tree for the classification of the adjacency matrices $A, \ldots, F$ of the graph g_1.

Fig. 4.1 illustrates the representation of an adjacency matrix M by its row-column elements. Notice that the a_i's are of different dimension, depending on i.

The decision tree for a model graph with adjacency matrix M 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 (dummy) root node. Its direct successors constitute the first level of the decision tree. On the first level, the classification is done by comparing the first row-column element of the input graph 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 4.2). Thus, on the first level of the decision tree we simply check if the label of the first vertex of the input graph matches any of the vertex labels of the model graph. Next, on the second level of the decision tree, the second row-column element a_2 of each matrix is used for the classification. This means that we check if the label of the second vertex of the input graph and the edges between the second and the first vertex have identical occurrences in any of the model graphs. This process is continued on the subsequent levels in the decision tree. With each matrix M_P that is represented by some node N on level k, the corresponding permutation matrix P is also stored. Finally, each leaf node at the bottom of the decision tree represents a

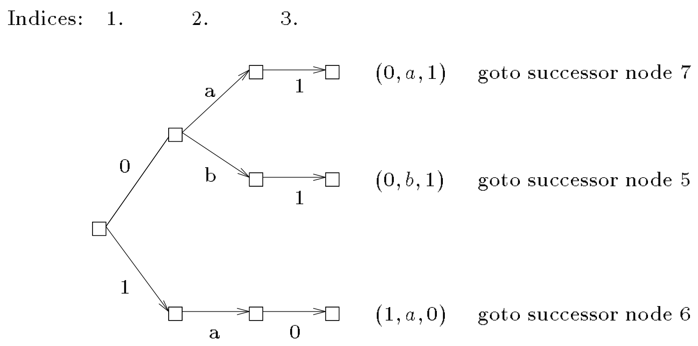

Figure 4.3: Dictionary and index for the decision tree node 3 in Fig. 4.2.

set of identical matrices $M_P \in A(G)$. The number of these matrices is equal to the number of automorphisms of G.[1] For each matrix, the corresponding permutation matrix is stored.

In Fig. 4.2, 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 a row-column element associated with it. At the top of Fig. 4.2 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, the successor which is reached via an element a_k in the decision tree must be easily decidable. 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+1} 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. As an example, consider node 3 on level 1 of the decision tree in Fig. 4.2. 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. 4.3. In this example, a three-level index structure is required.

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 of them. However, it is possible to represent several model graphs by the same decision tree. Given model graphs $G_1, G_2, \ldots G_L$, the corresponding sets $A(G_1), A(G_2), \ldots, A(G_L)$ can be treated as if they belonged to the same graph and integrated into the same decision tree. In Fig. 4.4 the decision tree for the graph g_1 of Fig. 4.2 and another

[1] An automorphism is an isomorphism of a graph to itself.

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

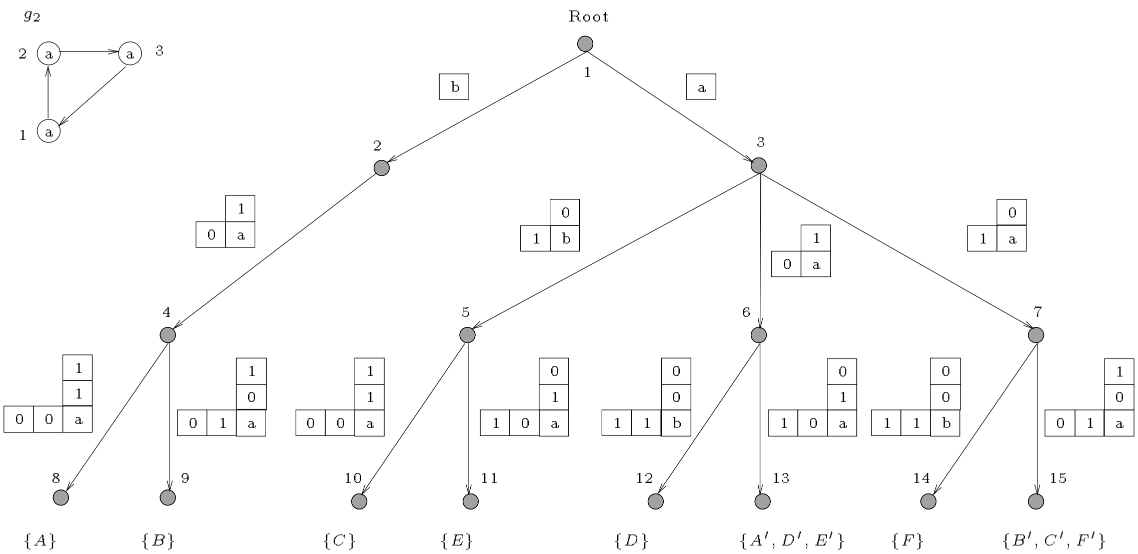

Figure 4.4: Decision tree for the graph g_1 in Fig 4.2 and another graph g_2.

graph, g_2, is displayed. The adjacency matrices in $A(g_2)$ are given at the top of Fig. 4.4. 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, nodes 13 and 15 in Fig. 4.4 each represents exactly three adjacency matrices.

At run time, a decision tree is used in order to classify the $m \times m$ adjacency matrix M_I of an 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. 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, i.e., the input graph and the model graph are of the same size, then each permutation matrix associated with N describes a graph isomorphism between the input graph and one of the models.

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 exist.
2. For $k = 1$ to $|V|$
   (a) Generate all subgraphs $S_k \subset G$ with k vertices and call merge_tree($Root, S_k$) for each of these subgraphs.

Figure 4.5: Algorithm *compile_tree*.

4.4 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, not only all subgraphs consisting of k vertices are present, but also each of these subgraphs S is represented $k!/\alpha$ times, where α denotes the number of different automorphisms of S. For example, in Fig. 4.2, the subgraph of g_1 consisting of the vertices 1 and 2 has two adjacency matrices and is therefore represented twice in the decision tree, namely, by the matrix $\begin{pmatrix} b & 1 \\ 0 & a \end{pmatrix} = S_{2,2}(A)$ in node 4 and by the matrix $\begin{pmatrix} a & 0 \\ 1 & b \end{pmatrix} = S_{2,2}(C)$ in node 5. Clearly, these representations are equivalent and 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 belonging to 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

$$M_1 = R M_2 R^T \tag{4.5}$$

R can be simply obtained as

$$R = P_1 P_2^T \tag{4.6}$$

because substituting R in Equation (4.5) 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, the adjacency matrix represented in 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 it is not necessary to create successor nodes of node N_2 in the decision tree. Instead, it is sufficient to introduce a pointer from N_2 to N_1 that

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 $M_P = (a_1, \ldots, a_k) \in A(S)$

   (a) $N = Root$.

   (b) For $i = 1$ to k

      i. If there is a successor node N_s of N for which $a_i = a_{N_s}$
         then set $N = N_s$ else
         generate a new node N_s and make N_s a direct successor of N. Insert a_i into the dictionary that is attached to N. Let N_s represent the row-column element a_i and set $N = N_s$.

3. Choose the node N on the level k 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 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 4.6).

Figure 4.6: Algorithm *merge_tree*.

indicates that, whenever decision tree node N_2 is reached, the search is to be continued with the successors of N_1. Such a pointer will be called a *redirecting branch*. Associated with a 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 decision tree nodes representing S will be used for the further decision tree traversal.

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 each subgraph 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 procedure *compile_tree* given in

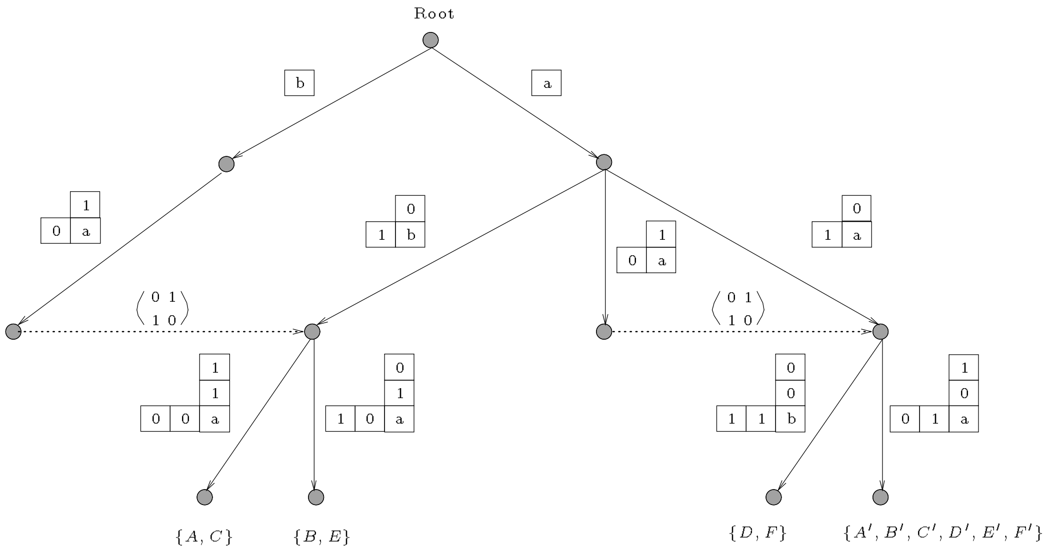

Figure 4.7: Compact decision tree for the classification of the adjacency matrices $\{A, \ldots, F\}$ and $\{A', \ldots, F'\}$ of the graphs g_1 and g_2, respectively (see also Figs. 4.2 and 4.4).

Fig. 4.5, we start by creating a single root node for the decision tree, provided that the decision tree is yet empty[2]. 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. 4.6, 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 matrix is performed in steps (2.a) and (2.b) by successively consulting each of its row-column elements. If for some node N on level i, there is no classification of the current matrix possible, i.e., there is no successor node with a row-column element that corresponds to the row-column element a_i 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_i must be inserted into the dictionary that is attached to N. When all matrices in $A(S)$ have been processed, the redirecting branches are inserted.

In Fig. 4.7, 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 $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. When this permutation matrix is applied to the 2×2 submatrices $S_{2,2}(A)$,

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

$S_{2,2}(B)$ 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 means of the redirecting branches 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, a 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.

4.5 Decision Tree Traversal

The decision tree structure that was previously described can now be used to get a very fast 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. 4.8) finds 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 successful, the classification is continued on the next level. In general, if the process is on level $k-1$ and N is the current node of the decision tree, then the successor node of N which represents a_k on the level 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_N in the dictionary that matches a_k, the process follows the branch from N to the successor node N_s that represents the row-column element a_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 the $m \times m$ 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 $M_I = R' M_I R'^T$.

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
   (a) Look up the dictionary of row-column elements that is attached to node N and find an entry a_N such that $a_N = a_k$.
   (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 model graph represented by the decision tree. Exit with failure.
   (c) If an element $a_N = a_k$ is found in the dictionary, then follow the branch marked by a_N to the node N_s.
   (d) If there is a redirecting branch from N_s to some node N' then set $M_I = R' M_I R'^T$, 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 4.8: Algorithm *decision_tree*.

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 decision tree 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, i.e., 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 important to note that the proposed 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 the adjacency matrix may fail at any stage of the decision tree.

4.6 Complexity Analysis

In this section, a computational complexity analysis is presented. It will be based on the following quantities:

L =the number of model graphs in the database,
n =the maximum number of vertices in a model graph,
m=the number of vertices in the input graph,
l_v =the number of vertex labels,
l_e =the number of edge labels.

We first consider the computational complexity of the decision tree generation step. With regards to practical applications it is important to note that the complete decision tree must be permanently present in the main memory at run time. Therefore, the size of the decision tree is of great importance. A useful measure for the size of the decision tree is the number of nodes. According to the compilation algorithm described in Section 4.4 the decision tree for a model graph is built level by level. The nodes on level k are created by isolating all subgraphs of size k of the model graph and classifying each of these subgraphs in the decision tree[3]. 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 nodes in the decision tree. Therefore, the number of nodes on each level with successor nodes is limited by $O(\binom{n}{k})$. 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} associated to a branch that originates in a node on level k 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. The total number of nodes on level $k+1$ following the level k is therefore bounded by $O(\binom{n}{k} l_e^{2k} l_v)$. The sum of the nodes over all the levels (without the root node) is then bounded by

$$O(l_v \sum_{k=0}^{n-1} \binom{n}{k} (l_e^2)^k) = O(l_v (1 + l_e^2)^n) \tag{4.7}$$

If there are several model graphs in the database, the size of the decision tree becomes linearly dependent on L. Thus, the space complexity of the decision tree is

$$O(L l_v (1 + l_e^2)^n) \tag{4.8}$$

[3]The root node is located on level 0. Subgraphs of size 1 are represented on then level 1 and so on.

This is the worst case upper bound for directed, labeled graphs. A special case is unlabeled, undirected graphs. For these graphs the size of the decision tree is bounded by[4]

$$O(L3^n) \tag{4.9}$$

The run time complexity of the new algorithm can be estimated by analyzing the algorithm *decision-tree* in Section 4.6. 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 according to the row-column element a_{k+1} requires that the row-column element a_N that matches a_{k+1} is found in any of the successor nodes of the current node. 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 spent 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 steps. Therefore, the theoretical complexity of the subgraph isomorphism algorithm is bounded by

$$O(m(2ml_e + l_v) + m^2) = O(m^2 l_e + ml_v) \tag{4.10}$$

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

$$O(m^2) \tag{4.11}$$

Notice that the complexity analysis provided above not only applies to subgraph but also to graph isomorphism detection.

For comparison reasons, we also give the computational complexity of the backtracking-based subgraph isomorphism method described in [Ull76]. (For a more detailed derivation of the following expressions, see Appendix A.2.) In the best case, its run time complexity is bounded by $O(Lmn)$. However, in the worst case, it is exponentially bounded by $O(Lm^n n^2)$. The space complexity of the backtracking-based method, on the other hand, is only $O(n^2 m)$ both in the best and the worst case because concurrent matching alternatives are tested sequentially.

It is important to note that in both the general and the special case of unlabeled graphs the computational complexity of the decision tree based algorithm is

[4] 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 $k+1$.

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, the *listing* of *all* subgraph isomorphism may take exponential time, as there may be exponentially many. 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 practical tasks, it is necessary to find ways of making the decision tree more compact.

4.7 Pruning Techniques for Decision Trees

In Section 4.4, the compilation of decision trees for the classification of adjacency matrices was described. In the following, we will refer to decision trees that are built according to this compilation procedure as *complete* decision trees. The main advantage of complete decision trees is that graph and subgraph isomorphisms can 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 a decision tree. The properties of the traversal algorithm based on a breadth-pruned decision tree are similar to those 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. That is, it is restricted to graph isomorphism. The second method aims at pruning the depth of decision trees. Depth-pruned decision trees will no longer guarantee graph isomorphism detection in polynomial time. However, they need only a fraction of the memory required by complete decision trees.

4.7.1 Breadth-Pruning of Decision Trees

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 input graph 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 remarkably reduced. In the following, we briefly outline two sets of transformations of this kind.

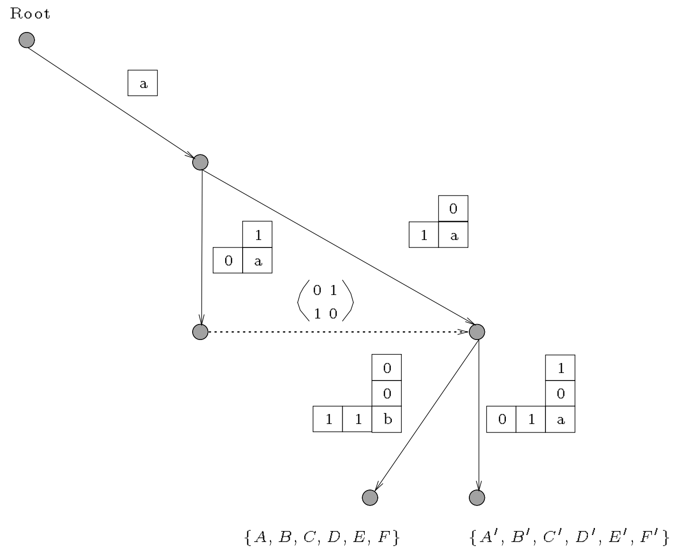

Figure 4.9: The decision tree of Fig. 4.7 (for the graph g_1 and g_2) after breadth-pruning was performed.

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 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 degree of 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.

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 and 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}. Note that there are exactly $(m-k)!$ 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 $m-k$ 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 $m-k$ last columns and rows of the input matrix M_I and try each of the $m-k$ 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 successful classification. However, with the modified version of the graph isomorphism algorithm it is no longer necessary to incorporate all $m-k$ 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 $m-k$ 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 model graph. Hence, in case of L unlabeled, undirected graphs, the size of the decision tree will be bounded by $O(L3^n/n)$ (see Equation 4.9). The computational complexity of the graph isomorphism algorithm, on the other hand, is increased by a factor of m 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 $O(m^3)$ (see Equation 4.11). 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.

There are two possibilities for the implementation of this type of breadth-pruning. The first technique directly incorporates the pruning criteria into the algorithm *merge_tree* (Fig. 4.6) such that on each level of the decision tree, breadth-pruning is immediately applied. The advantage of this implementation is that many nodes of the complete decision tree are never generated and, consequently, the breadth-pruned decision tree can represent graphs that are too large to be represented by complete decision trees. On the other hand, the resulting decision tree cannot be used for the representation of more than one model graph. For this reason, a second implementation of breadth-pruning is needed. In this implementation, breadth-pruning is performed as a postprocessing step. That is, the complete decision tree is first generated for all the model graphs in the database and then breadth-pruning is applied, removing unnecessary nodes from the complete decision tree. The drawback of this implementation, however, is that it can only handle model graphs for which the complete decision tree is not too large.

In Fig. 4.9, the decision tree for the graphs g_1 and g_2 of Fig. 4.7 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 now contains only 6 nodes instead of 11.

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

4.7.2 Depth-Pruning of Decision Trees

It can be often observed that the branching factor of a graph isomorphism search tree, such as it is used in Ullman's algorithm, is large at the top and becomes gradually smaller for deeper levels of the search tree. This is especially true for labeled graphs where there are several possible matchings only at the upper levels of the search tree, while at the lower part no backtracking is necessary anymore. Based on this observation, we propose the following pruning technique.

Instead of compiling a complete decision tree for a model graph G of size n, we only build a decision tree for all subgraphs of G up to a certain size $k < n$. The depth of the decision tree is therefore limited by k. At run time, the input graph G_I is classified using algorithm *decision_tree*. When a leaf node on level k of the decision tree is reached, a subgraph of size k of the input graph has been successfully matched onto one or several subgraphs of the model graph. To get a complete subgraph isomorphism, the search process must then continue with an algorithm based on tree search, for example Ullman's algorithm. However, the search space of the tree search algorithm will be greatly reduced as the partial matching found by the decision tree can be used as an initialization. Thus a hybrid scheme consisting of two phases results. First, there is a preclassification, or preselection, phase based on the decision tree of partial model graphs. Secondly, there is a conventional, tree search based graph matching procedure. But in this procedure, the first k levels of the decision tree are known, and concurrent alternatives have to be investigated only from level $k+1$ on. From this reduction of the search space significant savings in computation time can be expected. The first phase of the proposed hybrid matching scheme, i.e. the traversal of the partial decision tree, can be regarded an indexing procedure that selects possible candidates from a (potentially large) database.

4.8 Experimental Results

In order to examine the efficiency of the new algorithm in practice, we performed a number of experiments with randomly generated graphs. In Section 4.6, 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 many practical applications, however, it will be necessary to prune the decision tree. Thus, all the experiments documented in this section are concerned with 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 were undirected and unlabeled. The decision tree algorithm and Ullman's procedure[Ull76] were both implemented in C++ and run on a SUN Sparc10 Workstation. In order to compare the performance of the algorithms, we defined a basic computation step to be 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 executed while searching for all graph isomorphisms. Additionally, we measured the absolute computation times of both algorithms. Each experiment was repeated ten times for different permutations of the vertices of the model and the input graphs and the average number of computation steps and times were taken. The size of the decision tree in terms of the number of nodes and the required disk space were also recorded for each experiment.

In the first experiment, we wanted to investigate the run time efficiency of the new algorithm, but also its limits 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 22 vertices and 33 edges. The number of computational steps that were needed by the new and the conventional algorithm in order to find all graph isomorphisms between the model and input graph are plotted in Fig. 4.10. The corresponding computation times are displayed in Fig. 4.11. Clearly, the new algorithm is much more efficient than the backtracking algorithm, as predicted. Notice that the computation time for a graph with 22 vertices and 33 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. 4.12 the number of nodes in the breadth-pruned and the complete decision tree for a growing number of vertices in the model graphs are displayed. The largest model graphs our implementation can deal with consist of 19 vertices when the underlying decision tree is complete and 22 vertices when the decision tree is breadth-pruned.[5] The growth of the complete decision tree is clearly exponential. While there were only 174 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 requiring approximately 36 Mbyte of disk space. The breadth-pruned decision tree, on the other hand, is much smaller, i.e., for a graph consisting of 19 vertices, there were only 12'207 nodes in the decision tree. A breadth-pruned

[5] Note that in order to build a breadth-pruned decision tree for a graph with 22 vertices, it is necessary to incorporate the breadth-pruning into the compilation process as mentioned in Section 4.7.1. Thus, it is currently not possible to represent more than one model graph with 22 vertices by the same breadth-pruned decision tree.

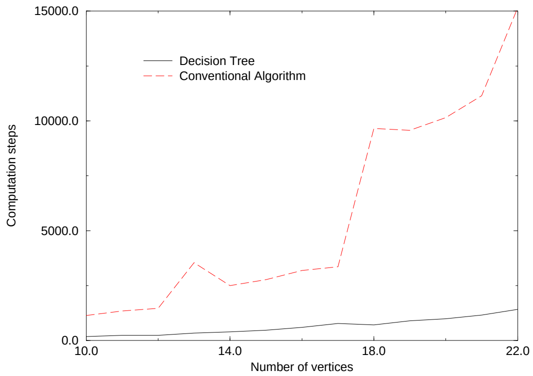

Figure 4.10: Computation steps for a growing number of vertices (first experiment).

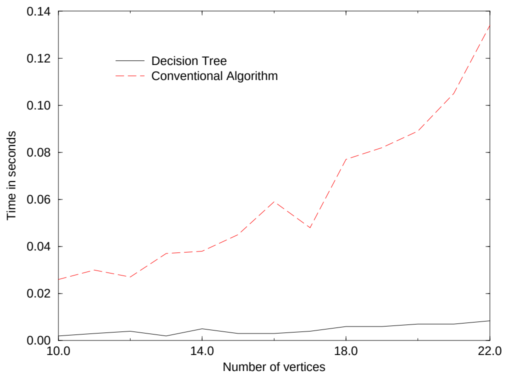

Figure 4.11: Computation time in seconds for a growing number of vertices (first experiment).

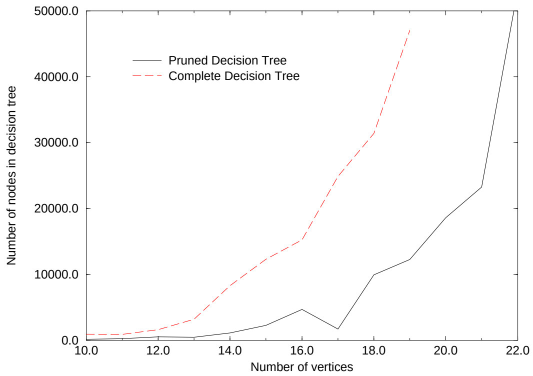

Figure 4.12: Number of nodes in the decision tree for a growing number of vertices (first experiment).

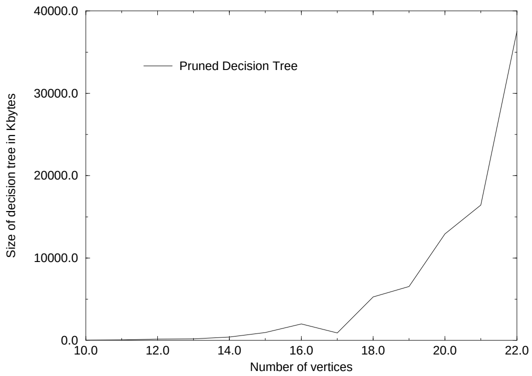

Figure 4.13: Disk space occupied by the decision tree for a growing number of vertices (first experiment).

decision tree consisting of 50'000 nodes can represent model graphs with up to 22 vertices, compared to only 19 vertices in the case of the complete decision tree. In Fig. 4.13 the disk space that is occupied by the breadth-pruned decision tree is displayed. Notice that for a graph with 22 vertices, 37.5 Mbyte of disk space are required. The compilation of a breadth-pruned decision tree for a model graph with 10 vertices took only two second, while for a model graph with 22 vertices roughly 30 minutes were needed on a Sparc10 Workstation. Due to the fact that, at run time, a decision tree which requires 40 Mbytes of disk space, occupies roughly 120 Mbytes of RAM – including the representation of various internal data structures – the application of complete (breadth-pruned) decision trees is limited to model graphs with no more than 19 (22) vertices in our current implementation.

In Section 4.7, 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

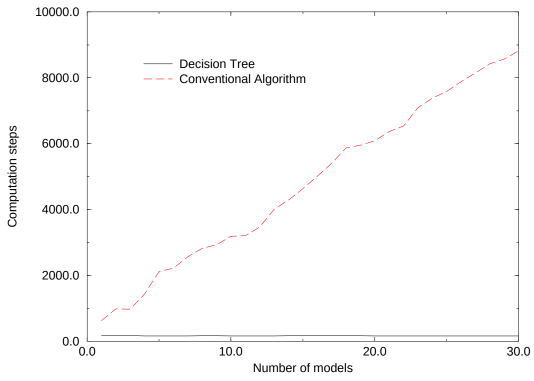

Figure 4.14: Computation steps for a growing number of model graphs (second experiment).

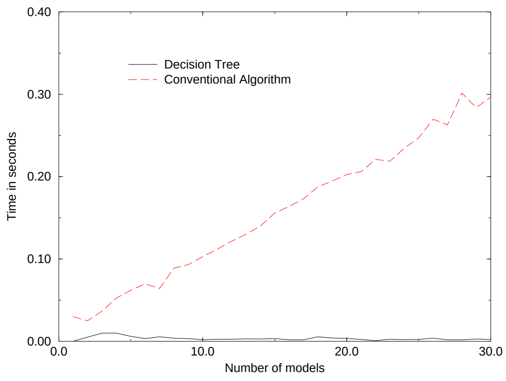

Figure 4.15: Computation time in seconds for a growing number of model graphs (second experiment).

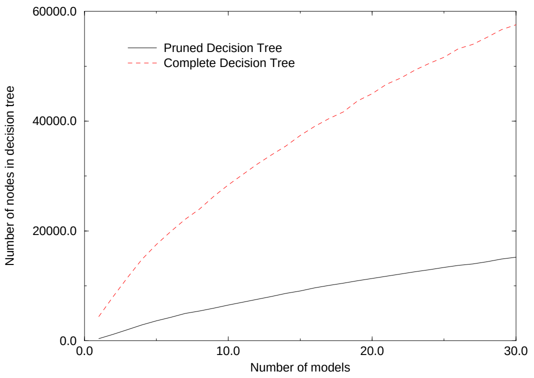

Figure 4.16: Number of nodes in the decision tree for a growing number of model graphs (second experiment).

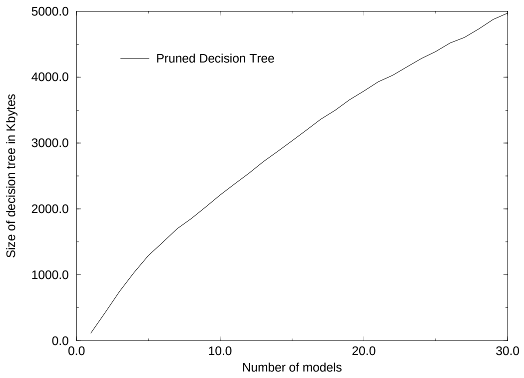

Figure 4.17: Disk space occupied by the decision for a growing number of model graphs (second experiment).

the input graph, while for a complete decision tree it is only quadratic. However, in practice, the run time is almost identical for both versions. (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 existing 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. The results of the second experiment are document in Figs. 4.14 and 4.15. As was expected from the computational complexity analysis, the decision tree algorithm's performance was

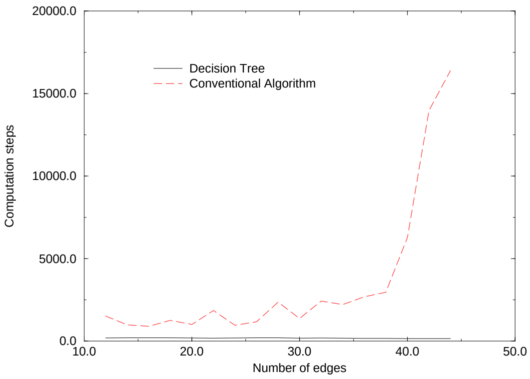

Figure 4.18: Computation steps for a growing number of edges (third experiment).

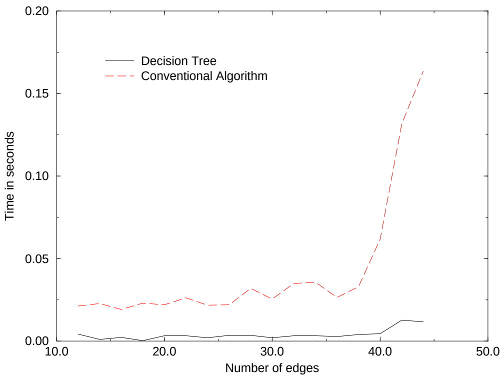

Figure 4.19: Computation time in seconds for a growing number of edges (third experiment).

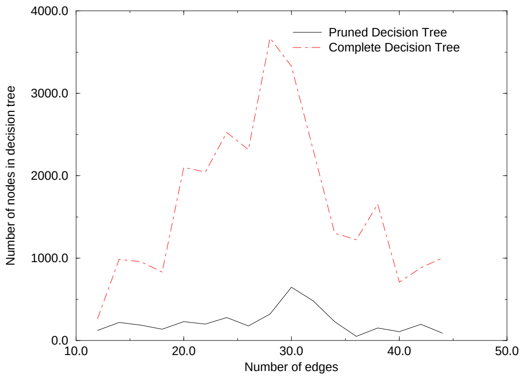

Figure 4.20: Number of nodes in the decision tree for a growing number of edges (third experiment).

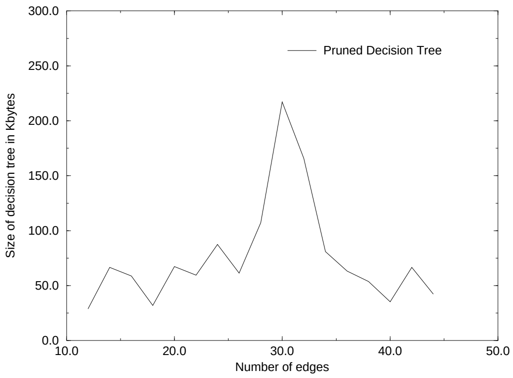

Figure 4.21: Disk space occupied by the decision tree for a growing number of edges (third experiment).

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 needed approximately 0.3 seconds for 30 model graphs.

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, however, due to the fact that common subgraphs of different model graphs are represented by the same part of the decision tree, the size is usually only sublinearly dependent on the number of model graphs. In Fig. 4.16, this effect can be clearly observed. In Fig. 4.17, 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 can expect that a database containing 100 model graphs will occupy approximately 15 Mbytes. Consequently, with the run time being independent of the number of model graphs, the new algorithm is an interesting option for applications where graphs are restricted in size but the number of models is large.

In addition to the size of the graphs and the database, the complexity of the graph isomorphism problem is also strongly dependent on the number 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 a complete graph to an isomorphic graph will take exponential time. In order to compare the performance of the new and the conventional algorithm for varying degrees of connectivity, we performed a third experiment. In this 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. 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 is displayed in Figs. 4.18 and 4.19, 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, while the conventional approach requires more computation steps and time when the number of edges grows[6]. On the other hand, the size of the decision tree depends on the number of edges in a graph. The size of the tree in terms of nodes is displayed in Fig. 4.20, while the required disk space of the breadth-pruned decision tree is given in Fig. 4.21. In Fig. 4.20 we observe that the size of the complete (and also the breadth-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 is due to the number of automorphisms. If the number of automorphisms in a graph is large, the number of 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 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. 4.20.

In the fourth experiment documented in Fig. 4.22 and 4.23 we investigated the robustness of the new algorithm with respect to different orderings of the vertices of the model and the input graphs. It is known that graph isomorphism methods based on backtracking are strongly dependent on the ordering of the model graph vertices.

[6] 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.

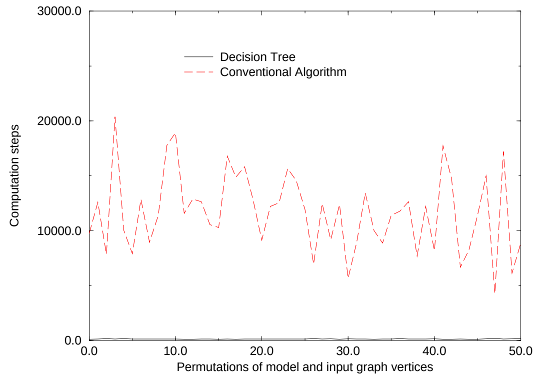

Figure 4.22: Computation steps depending on the vertex order (fourth experiment).

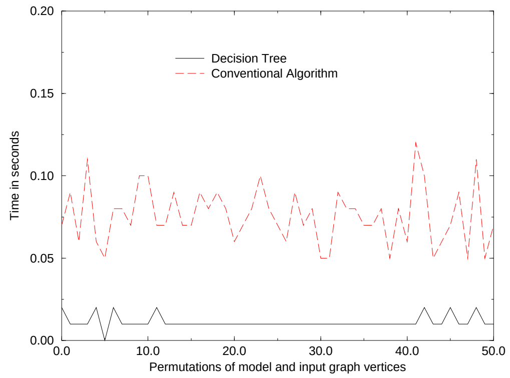

Figure 4.23: Computation time in seconds depending on the vertex order (fourth experiment).

If the vertices are ordered such that backtracking occurs at an early stage in the search process, the performance will be better than on an ordering which imposes late backtracking. For the fourth experiment, we therefore randomly generated 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. 4.22 and 4.23 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.

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.[7] 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* and the modification for depth-pruned decision trees described in Section 4.7.2. That is, when a leaf node was reached by the decision tree traversal algorithm, all the model graphs represented by this leaf node were further tested by Ullman's algorithm. However, Ullman's 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 Figs. 4.24 and 4.25 the total number of computation

[7] Note that although it is possible to create breadth-pruned decision trees for graphs with up to 22 vertices, these decision trees can only represent a single model graph at a time. Therefore, as breadth-pruning is especially interesting for large databases of graphs, the fifth experiment deals with graphs that consists of only 19 vertices. For these graphs, the complete decision tree can be generated (see Fig. 4.12).

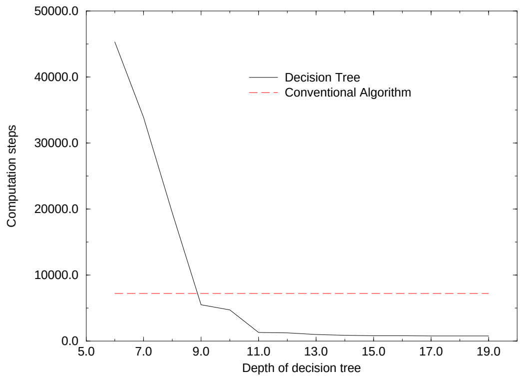

Figure 4.24: Computation steps in seconds depending on the depth of the decision tree (fifth experiment).

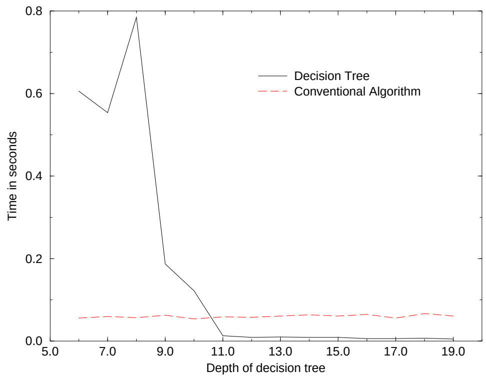

Figure 4.25: Time in seconds depending on the depth of the decision tree (fifth experiment).

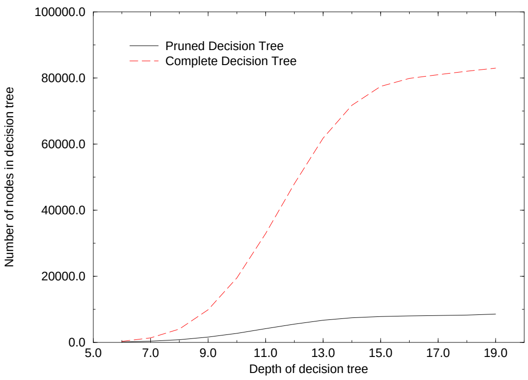

Figure 4.26: Size of the decision tree for a growing decision tree depth (fifth experiment).

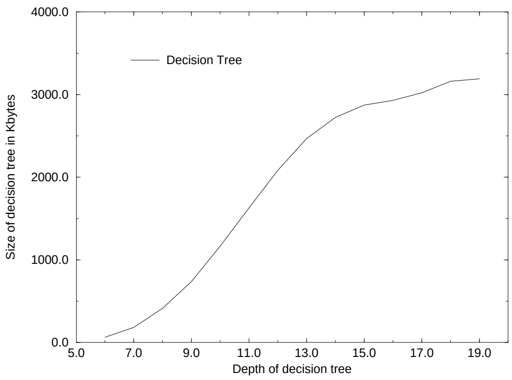

Figure 4.27: Disk space occupied by the pruned decision tree for a growing decision tree depth (fifth experiment).

steps and the time are displayed for the fifth experiment. The number of steps (computation time) is the sum of steps (time) needed by the tree traversal plus the steps (time) needed by the subsequent tree search. 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 a growing depth of the underlying decision tree. Particularly, for a decision tree of depth 11, the new algorithm was already nearly as efficient as for a tree of depth 19. In Fig. 4.26, 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 tree at depth 11 it is possible to 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. 4.27).

We conclude that both breadth- and depth-pruning are valuable techniques, which render the new algorithm applicable for practical applications. On the other hand, the depth-pruning technique only creates decision trees up to a certain depth. A depth-pruned decision tree no longer guarantees polynomial graph isomorphism. However, its main advantage lies in the efficient indexing of the database of model graphs.

4.9 Concluding Remarks

In this chapter, a new approach to graph and subgraph isomorphism detection was proposed. Given a set of model graphs, a decision tree is generated that represents, for each model graph, all possible permutations of the vertices. At run time, the isomorphisms or subgraph isomorphisms from an input graph to all model graphs can be found by a simple decision tree traversal. This approach has a time complexity that is only quadratic in the number of vertices of the input graph. Moreover, it is completely independent of the number of model graphs, and of the connectivity, i.e. the number of edges of the input and the model graphs. However, the size of the decision tree grows exponentially with the size of the model graphs.

In order to cut down the space complexity of a decision tree, different pruning techniques were discussed. The first technique is based on presorting the vertices of the input and the model graphs. Using this technique, some decision tree nodes can be discarded. As presorting can be done in quadratic time, the time complexity of the matching procedure is not increased. The second technique is based on the idea of eliminating certain vertex permutations of the model graphs from the decision tree. This method leads to a reduction of the required memory by a factor that is proportional to the size of a model graph. However, it increases the time complexity from quadratic to cubic. The first two pruning techniques, which are called breath-pruning, may still be not sufficient. For larger model graphs, a third technique – called depth-pruning – was proposed. It is based on the idea of incorporating only subgraphs of the model graphs up to a certain size in the decision tree. Under this pruning technique, the decision tree traversal procedure doesn't allow any longer to find complete isomorphisms or subgraph isomorphisms from the input to the model graphs. But it can be used as a preselection procedure in combination with conventional, tree search based graph or subgraph isomorphism detection. The result of the decision tree traversal is utilized as an initialization of the search tree. Thus, significant reductions in the size of the search space can be achieved.

The time and space complexity of the proposed method were analytically studied. All analytical complexity results could be confirmed in practical experiments with randomly generated graphs. The present implementation is able to handle breadth-pruned decision trees representing single model graphs with up to 22 vertices or up to 100 model graphs each consisting of 11 vertices. Under the depth-pruning technique, subgraphs of models with up to 22 vertices can be represented. Thus, depth-pruning allows to handle model graphs that are significantly larger than 22 vertices.

In summary, we consider the graph matching approach proposed in this chapter a contribution that is not only interesting from the theoretical point of view, but will also find its way into practical applications. For graphs of small size, the decision tree approach may be directly implemented. Its advantage over traditional, tree search based matching algorithms is its high speed, particularly in case of many models and a high degree of connectivity. Thus, the decision tree approach is interesting to applications where many model graphs of rather small size are to be processed in real or near real time. For larger graphs, depth-pruning, i.e. the combination of the decision tree approach with conventional tree search, is an interesting option. Compared to pure tree search, it is much faster. But it also allows to deal with graphs that are significantly larger than those that can be handled by complete or breadth-pruned decision trees. Furthermore, depth-pruning can be used as an efficient indexing procedure.

There are a number of open problems that deserve further research. Examples include the adaption of the method to subgraph isomorphism detection from the model graphs to the input graph, or the extension of the method to error-tolerant graph matching.

Chapter 5

Automatic Learning and Recognition of Graphical Symbols

5.1 Introduction

In the past few years the automatic processing and analysis of graphical data has become the focus of intensive research. There are numerous applications in which the detection and recognition of graphic symbols is an important task [Bun82, Pas94, Ham93, KK94, ALC$^+$93]. However, often it is not sufficient to simply recognize symbols that are already known, but it is also desirable to automatically learn unknown symbols.

The discipline of machine learning is concerned with algorithms by which a system can automatically construct models, knowledge, or skills from the outside world. Generally, there are different categories of approaches to machine learning, like learning in neural networks, symbolic learning from examples, case-based learning, explanation-based learning, or reinforcement learning. For a general introduction to the area of machine learning and an overview, the reader is referred to [Hut94, SD90, MT94].

While research on machine learning in the general artificial intelligence field has produced interesting results, machine learning in the domain of computer vision is still in its infancy. This situation is highly unsatisfactory because there are many problems in computer vision which will eventually be solved only if efficient learning schemes are available. For example, the manual construction of vision models is complicated, time consuming, and error prone. More seriously, perfect vision models can often not be built because the real world is unpredictable to some degree. Also, the type of noise or distortion that corrupt the pattern in reality is often only partly known. Therefore, it is impossible to design a computer vision or pattern recognition system that incorporates all models and knowledge that might be needed during runtime. Clearly, machine learning seems very promising to solve some of these

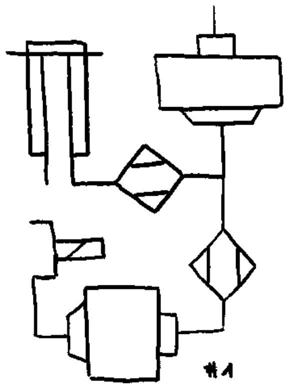

Figure 5.1: Example of an input image after binarization.

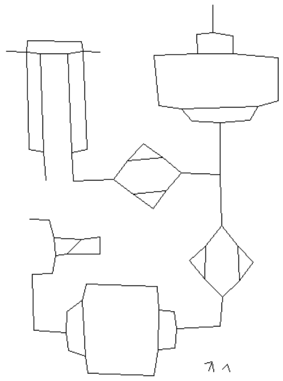

Figure 5.2: The image of Fig. 5.1 after thinning.

problems. Overviews of the state of the art of machine learning in computer vision can be found in [KM94, Bge94, SW94, MRA92].

In this chapter, we propose a system, which combines pattern recognition techniques with machine learning concepts in order to solve the problem of recognizing and learning graphical symbols in engineering drawings. The drawings and the symbols they contain are represented by attributed relational graphs. This representation is translation, scale and rotation invariant. Symbols that are known to the system are stored in a database. Instances of known symbols are detected in the drawing by determining subgraph isomorphisms from the symbol graphs in the database to the graphs representing a complete input drawing. In order to detect distorted symbols as well, a set of error correcting edit operations has been defined. The graph matching component of our system is based on the decomposition approach that was presented in Chapter 2 and 3. This approach allows a compact representation of the database of known symbols, and is very fast. The time complexity of the method is only sublinearly dependent on the size of the database. The learning component of the system is closely connected to the graph matching procedure. It makes use of the fact that the decomposition approach supports incremental updating of the database of known symbols. I.e., new symbols can be added to the database during runtime without the need to recompile the database from scratch. Given an initial database of symbols, the system tries to recognize all the symbols in an input drawing. Detected symbols are removed from the drawing and the remaining line segments are grouped into possible new symbols. Out of the set of possible new symbols, only some representatives, which are necessary in order to interpret the complete drawing, are learned and added to the database of symbols.

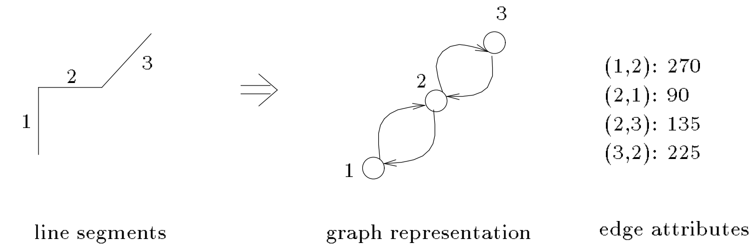

Figure 5.3: Graph representation of a simple line drawing.

5.2 Symbolic Representation

In the system described in this chapter, symbols and drawings are represented by attributed relational graphs. We allow that symbols appear at any position with varying size and orientation in a drawing. Therefore, the encoding of the symbols as attributed graphs must be translation, scale and rotation invariant. First, the drawings are given as scanned gray-level images. In a preprocessing phase, the images are binarized and thinned, and straight line segments are extracted. An example is shown in Figs. 5.1 and 5.2. For these steps, standard algorithms of image processing are applied. The current version of the system is restricted to drawings with straight line segments only, but it is straightforward to extend it so as to allow curved line segments, too.

Each line segment extracted during preprocessing is represented as a vertex of a graph. If two line segments l_1 and l_2 have an endpoint in common and v_1 and v_2 are their corresponding vertices, then the graph contains a directed edge (v_1, v_2) from v_1 to v_2 and a directed edge (v_2, v_1) from v_2 to v_1. There is an attribute associated with each of these edges. It denotes the angle that is enclosed between the two corresponding line segments. An example is shown in Fig. 5.3. Clearly, the proposed attributed graph representation is invariant with respect to translation, scaling and rotation.

Often, the drawings are corrupted by noise. Consequently, the graph representing a drawing will not always contain only exact instances of the symbol graphs. In order to correct local distortions in a drawing, we propose a number of graph edit operations. At run time, these edit operations will be applied to the graph representation of a drawing. In Fig. 5.4a a symbol is displayed; Figs. 5.4b-g show six distorted versions of it that illustrate the different types of errors that can be corrected by means of the edit operations. The first type of distortion affects the angles between two incident line segments (Fig. 5.4b). As a consequence, the attributes of the directed edges in the corresponding graph representation are not identical to the edge attributes in the original symbol. Therefore, these edge attributes must be subject to an *attribute substitution*. Secondly, line segments that are incident in the original symbol may not be incident in the drawing (Fig. 5.4c). For this reason, there will be no edges connecting the corresponding vertices in the graph represen-

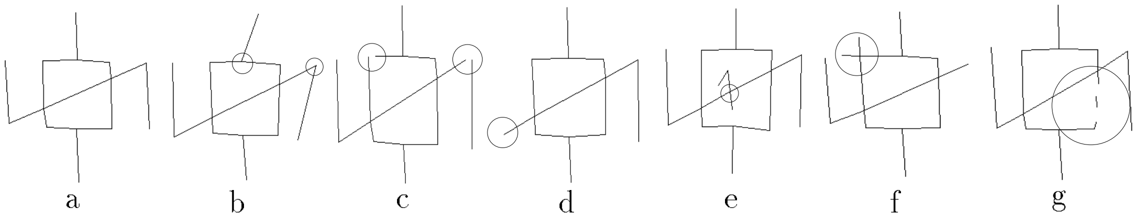

Figure 5.4: Possible distortions in input drawings. (Each distortion is indicated by a circle.)

tation. This type of error is corrected by *edge insertions* in the graph. Thirdly, line segments may be completely missing in the drawing (Fig. 5.4d). Consequently, a *vertex insertion* must be applied to the graph representation of the drawing. The edges connecting the new vertex to the rest of the graph need to be inserted, too. In the fourth case, a line segment may be erroneously intersected by some other line segment such that, instead of one vertex, there will be two vertices representing the line segment (Fig. 5.4e). In this case, a *vertex merge* must be performed to restore the original symbol. Fifth, additional line segments may appear at an intersection of original line segments (Fig. 5.4f). The graph of the drawing then contains additional vertices that are directly connected to the vertices of the symbol. For this type of distortion, no special edit operation is necessary because the matching component is based on subgraph isomorphism detection and consequently, additional vertices in the input graph are simply ignored. Finally, a line segment may be erroneously broken into several smaller segments that are separated from each other by empty space (Fig. 5.4g). In this case, the drawing graph will contain several additional vertices that are not connected to the rest of the graph via edges. To correct errors of this type, the graph matching algorithm performs a vertex insertion analogously to the operation performed for the distortion in Fig. 5.4d.

In order to model the fact that some of the distortions described above are more likely than others, each of the edit operations is associated with some cost. These costs are user-defined and must be empirically determined. The costs used in the application described in this chapter are given in Section 5.5.

The process of recognizing a particular symbol in a drawing can be formulated as the search for an error-tolerant subgraph isomorphism between the symbol graphs and the drawing graph. The objective of the error-tolerant subgraph isomorphism search is to minimize the total costs of the edit operations that are necessary in order to find an exact subgraph isomorphism between the symbol graphs and the drawing graph. We call the minimal edit cost of an error-tolerant subgraph isomorphism from a graph g_1 to another graph g_2 the *graph distance* between g_1 and g_2 and denote it by $d(g_1, g_2)$. As the database of the known symbols usually contains more than one element, the recognition task for a given drawing consists of finding all symbol graphs for which the graph distance to the drawing graph is less than a certain threshold. For this purpose, we use an efficient graph matching algorithm that we

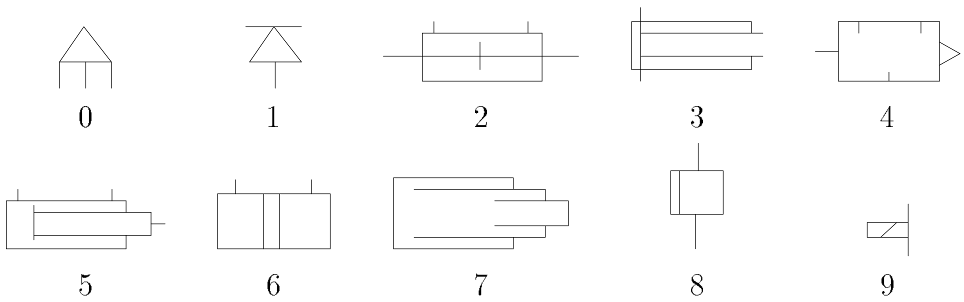

Figure 5.5: Graphic symbols denoting machine parts. (The numbers are symbol identifiers.)

describe in the next section. The algorithm is given in detail in Chapter 2 and 3.

5.3 Recognition Based on Subgraph Isomorphism

The algorithm for finding an error-tolerant subgraph isomorphism from a symbol to a drawing is based on a compact representation of the database of symbols. This representation is generated in an off-line compilation step. The basic idea of the compilation procedure is to find common substructures among the symbols, and to represent the symbols in terms of these substructures. For this purpose, the symbols are recursively split into smaller components. The smallest components taken into consideration are single line segments. For each of these components, a matching module is created, which performs the task of finding subgraph isomorphisms from the actual component to the drawing. If part of some symbol appears multiple times in the same or in different symbols, there will be only one matching module for this part. This results in a very compact representation of the set of symbols. Moreover, at runtime the search for subgraph isomorphisms for a part that occurs multiple times in one or in different symbols is done only once. Consequently, the subgraph isomorphism search is computationally very efficient.

The matching modules for the components of the symbols are organized in a network structure. At the top of this network is a matching module that represents the smallest possible part of any symbol, namely, a single straight line segment (of any orientation). Directly connected to the module on top of the network are matching modules which represent parts of symbols that consist of more than one straight line segment. These matching modules are in turn connected to modules which represent successively larger parts of symbols. Finally, at the bottom of the network there is a matching module for each of the symbols in the database. In Fig. 5.5 a database of symbols from mechanical engineering is shown [DIN80]. Fig. 5.6 illustrates the network of matching modules for the symbols 0,1,2, and 3. (The full network representing all symbols $0, 1, \ldots, 9$ can't be shown because of limited space.) On top of the network there is a matching module corresponding to single

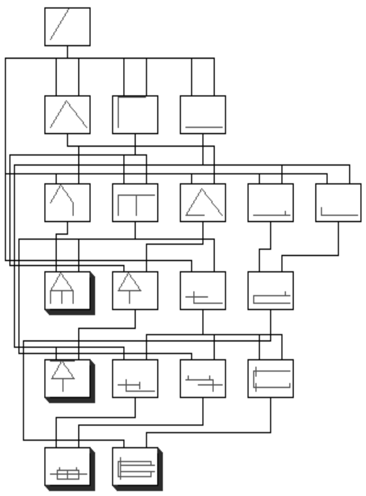

Figure 5.6: The network representation for the symbols 0,1,2 and 3 of Fig. 5.5.

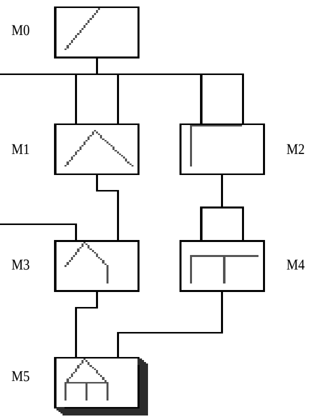

Figure 5.7: The network for the symbol 0 of Fig. 5.5.

straight line segments of any size and orientation. In the second level, there are three matching modules representing pairs of line segments that include an angle $\alpha_1 = 80°, \alpha_2 = 90°$ and $\alpha_3 = 180°$, respectively. The matching modules at the lower levels in the network represent parts of the symbols $0, 1, 2,$ and 3 that consist of three, four, ... line segments. Finally, the matching modules that are at the lowest level in the network – i.e. the terminal modules – correspond to the symbols 0,1,2, and 3. These modules are represented by shadowed rectangles in Fig. 5.6.

Note that a network like the one in Fig. 5.6 is automatically built from the graphs representing symbols like the ones shown in Fig. 5.5. A very important feature of our network compilation procedure is its incremental nature. Given a network N representing symbols $\{s_1, s_2, \ldots, s_n\}$, an extended network N' representing symbols $\{s_1, \ldots, s_n, s_{n+1}, \ldots, s_m\}$ can be built by just adding some more matching modules to N. The modules to be added represent those parts of $s_{n+1}, \ldots, s_m$ that are not yet represented by the network N. Particularly, there is no need to build N' from scratch, given N and $s_{n+1}, \ldots, s_m$. This feature of incremental extensibility is very important in the context of machine learning. If unknown symbols are discovered during the interpretation of a drawing, their symbolic representation can be easily and efficiently added to the existing network.

At runtime, all subgraph isomorphisms from a straight line segment to the input drawing are computed in the top module, and propagated to all direct successor modules. In each of these successor modules, the subgraph isomorphisms from the parent modules are combined to form subgraph isomorphisms for larger parts of the symbols. The propagation of the subgraph isomorphisms from one module to the next is controlled by the cost of the edit operations that are implied by the subgraph isomorphism. This process terminates when all subgraph isomorphisms with edit costs less than a certain threshold t_{recog} have been found. The matching modules for the symbols then contain all the subgraph isomorphisms with edit costs less than t_{recog}.

In order to illustrate the runtime behavior of the subgraph matching procedure, the part of the network of Fig. 5.6 that represents symbol 0 in Fig. 5.5 is shown in Fig. 5.7. In order to identify instances of symbol 0 in a drawing, all instances of straight line segments are recorded first by module M_0 on top of the network. Then two matching modules M_1 and M_2, are activated. M_1 extracts pairs of line segments enclosing an angle of 80°, while M_2 checks for an angle of 90°. The next module M_3, combines simple line segments from M_0 with instances recorded by M_1 to find a chain of three line segments, where the last two segments enclose an angle of 120°. On the same level, the module M_4 inspects the result of M_2 and tries to combine any two instances found by M_2 into F-shaped configurations of line segments. Finally, module M_5 detects instances of symbol 0 by combining the results of M_3 and M_4.

Notice that the graph matching procedure described in this section is completely independent of the considered application. It is applicable to any problem domain where the objects under study can be represented by attributed graphs. The experiments reported in Chapter 2 and 3 demonstrated that the method is faster than other algorithms for error-tolerant subgraph isomorphism detection that have been described in the literature before. Particularly, the theoretical time complexity is only sublinear in the size of the database.

5.4 Learning New Symbols From a Drawing

The system starts with a number of a priori known symbols in the database. (This number may be zero.) Any input drawing is first interpreted with respect to this database of symbols. All line segments in the drawing that belong to recognized symbols are removed. The remaining line segments either constitute symbol interconnections or unknown symbols.

In order to identify unknown symbols, a few heuristic rules have to be defined. These rules correspond to a generic, meta-level definition of the symbols. In other words, they encode the basic characteristics of any graphical symbol in the application domain regardless of its particular shape. Of course, the characteristics of the symbols and the corresponding rules are application specific. For the present application, i.e. the set of symbols shown in Fig. 5.5, we defined three such rules: (1) The graph of any symbol is connected. (2) The graph of any symbol contains at least one, or possibly several adjacent closed loops of line segments. (3) Line segments that intersect a closed loop of a symbol belong to the same symbol. Note that all symbols in Fig. 5.5 comply with rules (1)-(3). Given a set of line segments that could not be associated to any known symbol, the identification of potentially new symbols is started by finding all closed loops of line segments. According to rule (2), a symbol must contain at least one closed loop. If two closed loops have one or more line segments in common, then it is assumed that they belong to the same candidate symbol. Finally, all line segments that intersect with a closed loop of a candidate symbol are assumed to be part of that symbol, according to rule (3). The identification of unknown symbols terminates when no more candidate symbols can be formed.

The procedure described in the last paragraph is able to extract unknown symbols from a drawing. Under the most primitive learning strategy, one could now integrate these unknown symbols into the network that represents the database of known symbols (see Fig. 5.6). However, a problem with this approach is that the network may grow unnecessarily large. An unnecessary growth of the database occurs if there are multiple instances of the same unknown symbol in a drawing, some of which have undergone some distortions and are represented, consequently, by different graphs. In order to overcome this problem, there is a second step in our learning procedure that aims at minimizing the number of symbols that must be stored in the database. This minimization is based on the detection of similarities between the unknown symbols. First, a subset of the set of unknown symbols is searched for such that for each unknown symbol there exists a symbol in the subset with a graph distance that is less than a certain threshold t_{learn}. For this purpose, all the unknown symbols are compared to each other by means of error-tolerant subgraph isomorphism. Let $C = \{c_1, c_2, \ldots, c_n\}$ denote the set of unknown symbols that have been extracted from the drawing. The graph distance from symbol c_i to c_j is given by $d(c_i, c_j)$. Note that normally $d(c_i, c_j) \neq d(c_j, c_i)$. We now apply a hierarchical clustering procedure to the set C of unknown symbols [Eve93]. Initially, each candidate symbol c_i forms a simple cluster $U_i = \{c_i\}$. The distance of two clusters U_i,U_j is defined to be $D(U_i, U_j) = MAX\{MIN\{d(c, c'), d(c', c)\} | c \in U_i, c' \in U_j\}$. This means that we consider the minimal graph distances between any two symbols in the clusters U_i and U_j and choose the largest of these distances as the cluster distance. According to the distance measure D the two clusters with the least distance are now merged into a larger cluster. The distances between the new cluster and all the other clusters are computed and this process is continued until the distances of any two clusters exceed the threshold t_{learn}. At the end, each resulting cluster contains only candidate symbols with mutual distances that are less than the threshold t_{learn}. The size of the clusters is directly dependent on the value of t_{learn}. The larger t_{learn}, the more symbols will be merged into the same cluster and consequently less symbols will finally be learned. Out of each cluster U, a representative symbol is chosen such that the average graph distance from the representative symbol to all the other symbols in the cluster is minimized. That is, the symbol $c \in U$ for which the sum of the graph distances $\sum_{c' \in U, c' \neq c} d(c, c')$ is minimal is chosen as the representative symbol for the whole cluster U.

All the representative symbols found by the procedure described in the last paragraph are added to the database of known symbols and compiled into the network

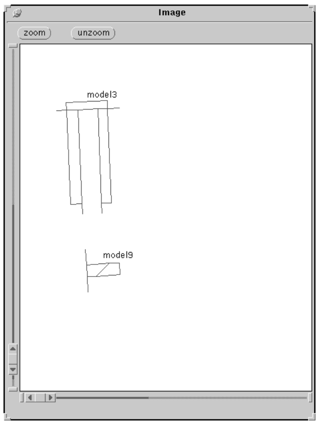

Figure 5.8: Instances of symbols 3 and 7 detected in the first drawing.

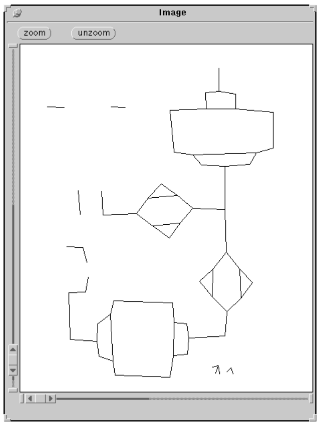

Figure 5.9: Fig. 5.2 after the symbols 3 and 7 have been removed.

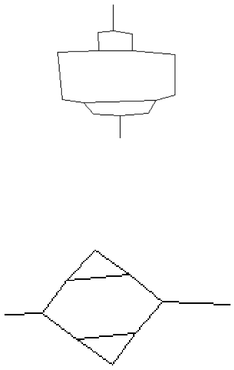

Figure 5.10: First and second symbol learned from the drawing in Fig. 5.2.

representation of the database. Note that the network grows less than linearly with each new symbol due to its ability of sharing common parts among different symbols. With the symbols that are automatically learned, it is possible to completely interpret any input drawing that contains graphical symbols that follow the rules (1)-(3). Furthermore, by learning the representative symbol from the "center" of each cluster of candidate symbols, an interpretation with minimal edit cost is guaranteed. In the following, we illustrate some of the details of the recognition and learning process by means of an example.

5.5 Experimental Results and Example

We illustrate the process of recognizing and learning symbols with an example dealing with machine layouts. Notice that our proposed procedure can be very easily adapted to other types of engineering drawings. Depending on the structure of the engineering drawings and the corresponding symbols it is necessary to empirically define the thresholds t_{recog} (Section 5.3) and t_{learn} (Section 5.4) and cost functions for the edit operations (Section 5.2). In our application, we defined the costs of the edit operations as follows (see also Fig. 5.4). We set the cost for substituting an angle α_1 with an angle α_2 to $(\alpha_1 - \alpha_2)^2$. (Note that for the cost calculation, the angles are given in radians instead of degree.) The cost for the insertion of an edge with an angle attribute α_1 between two non-intersecting line segments with distance d and enclosed angle α_2 was set to be $(\alpha_1 - \alpha_2)^2 * d$. The cost for inserting a vertex into the drawing graph was constantly set to 3. With this, we modeled the fact that substituting angles with a difference larger than 100° (1.74 radians) is highly unlikely and therefore a line segment should be inserted instead. Finally, the cost for merging two vertices that represent two collinear and touching line segments into a single vertex representing a straight line segment are constantly set to zero.

Based on these edit costs it is now possible to define the recognition threshold t_{recog} and the learning threshold t_{learn}. Because t_{recog} controls the degree of distortion that known symbols may have and t_{learn} defines the maximal distance among unknown symbols that are represented by the same model graph, there is a strong relationship between the two thresholds. If recognition and learning are to be performed on the same input drawing, it is best to set $t_{recog} = t_{learn}$. With this setting, and due to the fact that for all unknown symbols a new symbol with a graph distance of less than t_{learn} is learned, it is guaranteed that any input drawing can be completely interpreted. If some special task in the application requires that heavily distorted input drawings are interpreted, then t_{recog} may be increased empirically beyond the value of t_{learn}. In the present application, we set $t_{recog} = t_{learn} = 2$. With these thresholds and the costs of the edit operations mentioned above, it is possible, for example, to detect symbols for which two angle corrections of 60° (1 radian) are necessary.

Using the symbols shown in Fig. 5.5 as models and applying the recognition procedure to the drawing in Fig. 5.1, instances of the symbols 3 and 9 are detected and displayed at their correct locations; see Fig. 5.8. In order to be able to interpret the rest of the drawing, new symbols must be learned. For this purpose, the line segments that have been matched with the symbols 3 and 9 are removed from the drawing. The remaining line segments are given in Fig. 5.9. The learning scheme described in the previous section is then applied to these remaining line segments. In the first step of the learning procedure, four candidate symbols according to the rules stated in Section 5.4 are identified. Then these candidate symbols are clustered according to their mutual graph distances. As a result, two clusters each containing two candidate symbols are formed. In Fig. 5.10 the representative symbols that are chosen out of each cluster are displayed. These symbols are now compiled into the network representation of the other symbols in the database. With the new symbols added to the network it is possible to completely interpret the drawing in Fig. 5.1. Subsequently, the learned symbols can be detected in new input diagrams. In Fig. 5.11 (and Fig. 5.12) a drawing is given which contains distorted versions of (old) symbol 6 and one of the new symbols. Both symbols are successfully detected by the system and located at the correct positions (see Fig. 5.13). The process of recognition and learning can be continued for any number of additional input drawings.

The system described in this chapter was implemented in GNU C++ and runs on a SUN SPARC5 Workstation. It was tested on about 30 drawings with a complexity similar to those shown in Fig. 5.1 and 5.11, using the symbols shown in Fig. 5.5 and other types of symbols. With the thresholds properly set, recognition and learning worked flawless. The threshold adaption process is uncritical as there is a well-

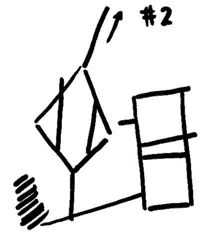

Figure 5.11: Another drawing.

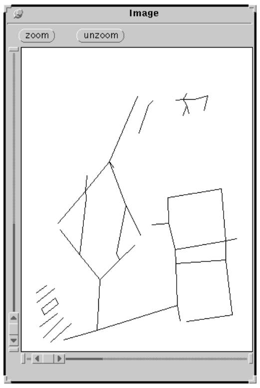

Figure 5.12: Second drawing after thinning (containing distorted versions of old and new symbols).

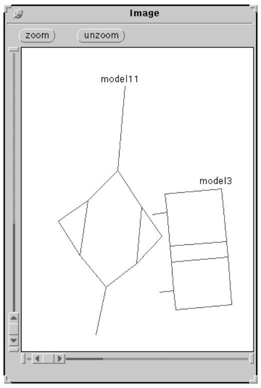

Figure 5.13: Detected and corrected symbols in second drawing.

defined meaning of each threshold, and suitable threshold ranges can be determined by manual inspection. Also, the system performance is quite stable under threshold variations.

5.6 Concluding Remarks

We have proposed a recognition and learning scheme for graphical symbols in engineering diagrams. It is based on a new and efficient error tolerant graph matching algorithm. This algorithm is used for the recognition of symbols, and in the learning procedure where possible symbols are clustered according to their mutual graph distance. Graph matching is based on a compact representation of the database, which can be incrementally updated and ensures a recognition time complexity sublinear in the size of the database. The learning and the recognition modules of the system work on general attributed graphs and can be easily adapted to other vision applications, particularly other types of engineering drawings.

Chapter 6

Summary and Conclusions

In this thesis, we studied various aspects of the graph matching problem. First, a brief overview of methods described in the literature was given. Next, it was observed that apart from requiring exponential time in the worst case, the reviewed conventional methods can only be applied to two graphs at a time. That is, for applications that are dealing with a database of model graphs, a time complexity that is linearly dependent on the size of the database results. Thus, we presented two new approaches to the problem of graph matching. For both of these approaches it was assumed that there is a set of a priori known model graphs that must be matched to an input graph.

The first approach is based on a compact representation of the model graphs. In this representation, subgraphs that are common to more than one model graph are represented only once. Consequently, at run time, the computational effort for finding these subgraphs in the input graph must be done only once. Based on this compact representation, an algorithm for exact subgraph isomorphism (Chapter 2) and another algorithm for error-correcting subgraph isomorphism detection (Chapter 3) were described. Both algorithms have a time complexity that is only sublinear in the size of the database. Furthermore, due to the integration of a new future cost estimation procedure that works on the compact representation, the error-correcting algorithm outperforms conventional algorithms not only for large databases but also for single model graphs. In order to demonstrate the usefulness of the decomposition approach in a practical application, a system for the recognition and learning of graphical symbols was described (Chapter 5). In this system, graphical symbols and technical drawings that contain these symbols are represented by attributed graphs. The recognition of symbols in a drawing is then formulated as the search for error-correcting subgraph isomorphisms from the graphs representing known symbols to the graph representing the drawing. The graph matching is performed with the decomposition approach. Furthermore, a method for the automatic acquisition of new symbols from the drawings was also presented. This learning method is again based on the decomposition approach.

exact graph matching	unlabeled	labeled
less than 20 vertices	decision tree method Ullman	decision tree method decomposition method
less than 500 vertices	Ullman	decomposition method
more than 500 vertices	continuous optimization methods	

Table 6.1: Suitability of exact graph matching methods for different graph classes.

The second approach presented in this thesis (Chapter 4) is based on a decision tree that is derived from the model graphs. This decision tree is used at run time to detect all graph and subgraph isomorphisms from an input graph to the model graphs in quadratic time. In particular, the time complexity of the decision tree method is completely independent of the number of the model graphs. However, the size of the decision tree is exponential in the number of vertices of the model graphs. Because the decision tree must be present in the main memory at run time, the application of this method is limited to small graphs. In order to reduce the size of the decision tree and make the method applicable to larger graphs as well, several pruning techniques have been proposed.

The behavior of the new methods presented in this thesis has been studied both in a theoretical complexity analysis and in a number of practical experiments with randomly generated graphs. Based on the results, we can draw the following conclusions. The decision which graph matching algorithm is best suited for a particular application depends on the number and the size of the underlying graphs, the kind of labels that occur in the graphs, and the type of matching that is required. Generally, it can be concluded that for applications that are dealing with a large number of model graphs, the new algorithms should always be taken into consideration. The final choice for an algorithm should then be based on the particular characteristics of the graphs in the application. That is, if the graphs (labeled or unlabeled) are small (up to 22 vertices) and exact graph or subgraph isomorphisms from the input to the models must be found then the decision tree method will perform best in terms of execution time. Otherwise, if the direction of matching is reversed, i.e. exact subgraph isomorphisms must be detected from the models to the input, then Ullman's algorithm or, in case of errors, an A^* based algorithm must be used (see Appendix A). If the model graphs and the input graph are of medium size (up to 500 vertices) and labeled (at least more than 5 different vertex or edge labels) then the decomposition based algorithm is expected to perform better than any conven-

error-correcting graph matching	unlabeled	labeled
less than 100 vertices	A^*	decomposition method
more than 100 vertices	continuous optimization methods	

Table 6.2: Suitability of error-correcting graph matching methods for different graph classes.

tional algorithm for both exact and error-tolerant graph and subgraph isomorphism detection from the models to the input. However, in case of unlabeled graphs and if subgraph isomorphisms must be found from the input to the models then it is again advisable to apply either Ullman's algorithm or an A^* based method. Finally, in case of very large graphs (more than 500 vertices) only approximate algorithms such as simulated annealing [HHVN90], neural networks [FLD94, XO90], genetic algorithms [JS89] or probabilistic relaxation [CKP95] will be able to solve the exact or error-tolerant graph matching problem in reasonable time, albeit at the cost of occasionally missing the optimal solution. In Table 6.1 and 6.2 a summary of the above conclusions for exact and error-correcting graph matching, respectively, is given.

Based on the above conclusions, we believe that the decomposition based approach and the decision tree approach are valuable contributions to the problem of graph matching. It is important to note that they have been described in general terms without any problem dependent assumptions or heuristics. Therefore, they can be easily adapted to various different application domains. In Appendix B, an implementation of the algorithms is presented along with a description of the various tasks that are necessary in order to solve the graph matching problem in a particular application domain. Among these tasks are the definition of an encoding scheme which transforms the objects in the application into attributed graphs and the definition of the attributes and the cost functions for the edit operations.

There are a number of issues concerning the new approaches that deserve further research. For example, certain applications in noisy environments will require other graph edit operations than the ones used in this work. Hence, it will be necessary to extend the error-correcting subgraph isomorphism algorithm based on decomposition to more arbitrary edit operations. Furthermore, there is a potential for parallelization inherent in the decomposition approach. Due to the similarity of the decomposition algorithm with the RETE network algorithm, an indication for a possible parallelization scheme can be found in [KS89]. In case of the decision tree approach, future research directions include the extension of the method to subgraph isomorphism detection from the model graphs to the input graph and, in general, the extension to error-correcting graph matching. Furthermore, the development of additional pruning techniques may render the method applicable for larger graphs as well. Finally, in [Set90] a method for the transformation of decision trees into neural networks is presented. By applying this method to a decision tree representing a set of model graphs it might be possible to construct a neural network that supports graph and subgraph isomorphism detection in constant time.

Appendix A

Conventional Graph Matching Methods

In this appendix, we give a detailed description of three traditional methods for graph matching. Each of the three methods has been previously described by several authors. First, we describe an algorithm that is based on maximal clique detection. Although this algorithm was not used for comparative purposes in any of the previous chapters, we include it here due to its general applicability. In particular, it can be used to solve not only the graph isomorphism and the subgraph isomorphism problem, but also the maximal common subgraph isomorphism problem. Next, Ullman's method for exact graph and subgraph isomorphism detection is given. Finally, the A^*-based tree search algorithm for the error-correcting graph and subgraph isomorphism problem is presented. Notice that most of the concepts and notations used in the following have been previously defined in Chapter 2, 3 and 4.

A.1 Exact Subgraph Isomorphism Detection by Clique Detection

Given a model graph $G = (V, E\mu, \nu)$ and an input graph $G_I = (V_I, E_I, \mu_I, \nu_I)$ we intend to find all subgraph isomorphisms from G to G_I. If no such subgraph isomorphisms exists, we still want to find the largest subgraph S of G such that there exists a subgraph isomorphism from S to G_I. In [Lev72] it was proposed to use the well known technique of maximal clique detection in order to solve the above problem. More recent publications also extended the method to conceptual graph matching [FFG90, MLL92]. The basic idea of graph matching by clique detection is to map each vertex $v \in V$ to each vertex $v_I \in V_I$ if $\mu(v) = \mu(v_I)$ and to record each such mapping (v, v_I) as a new vertex of a special graph, the so-called *association graph* $G_A = (V_A, E_A)$. (In order to distinguish between the model vertices and

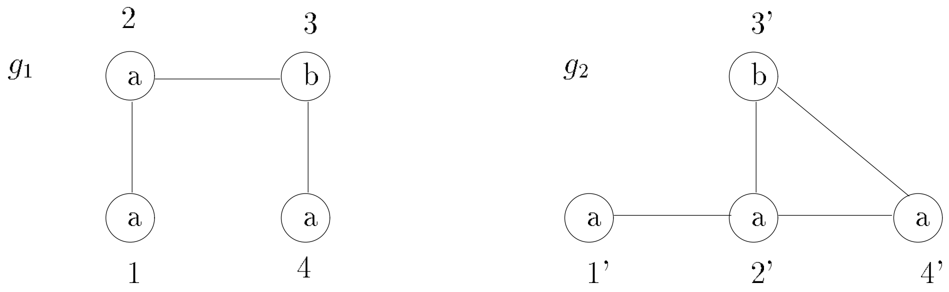

Figure A.1: Two sample graphs g_1,g_2.

edges, and the vertices and edges of the association graph, we call the latter a-vertices and a-edges.) In this association graph, the local compatibility of two vertex mappings (v, v_I) and (v', v'_I) is denoted by the existence of an edge between the a-vertex representing (v, v_I) and the a-vertex representing (v', v'_I). For example, if $v_A = (v, v_I)$ and $v'_A = (v', v'_A)$ are two a-vertices and there is an edge $e = (v, v') \in E$ and also an edge $e_I = (v_I, v'_I) \in E_I$ then the mapping (v, v_I) is compatible with the mapping (v', v'_I). I.e., the function $f : \{v, v'\} \rightarrow \{v_I, v'_I\}$ that maps v onto v_I and v' onto v_I denotes a graph isomorphism from the subgraph of G consisting of $\{v, v'\}$ to the subgraph of G_I consisting of $\{v_I, v'_I\}$. This compatibility is represented by an a-edge in the association graph connecting v_A and v'_A. Clearly, any subgraph of the association graph, in which each a-vertex is connected to each other a-vertex, defines a graph isomorphism from a subgraph in G to a subgraph in G_I. Note that a subgraph of a graph, for which each vertex is connected to each other vertex is called a *clique*. It is easy to see that the maximal clique in the association graph represents the largest common subgraph of G and G_I. Thus, the search for the maximal common subgraph and the corresponding subgraph isomorphism function can be formulated as the search for the maximal clique detection in the association graph.

In Fig. A.2, the association graph for the model graph g_1 and the input graph g_2 from Fig. A.1 is displayed. Note that there are ten possible vertex mappings from g_1 to g_2 and, consequently, the association graph consists of ten a-vertices. Apparently, there are two maximal cliques, each containing three a-vertices, namely C_1 consisting of $(1, 1'), (2, 2')$ and $(3, 3')$ and C_2 consisting of $(1, 1'), (3, 3')$ and $(4, 4')$. Both cliques are denoted by dotted lines in Fig. A.2. While C_1 represents a graph isomorphism from the subgraph of g_1 defined by the vertices $1, 2$ and 3 to the subgraph of g_2 defined by the vertices $1', 2'$ and $3'$, C_2 represents a graph isomorphism from the subgraph of g_1 with the vertices $1, 3, 4$ to the subgraph of g_2 with the vertices $1', 3'$ and $4'$. In the following, we give a formal definition of an association graph and the graph matching algorithm based on maximal clique detection.

Definition A.1: Given two graphs $G = (V, E, \mu, \nu)$, $G_I = (V_I, E_I, \mu_I, \nu_I)$, an *association graph* for G and G_I is an unlabeled graph $G_A = (V_A, E_A)$ with $V_A \subseteq$

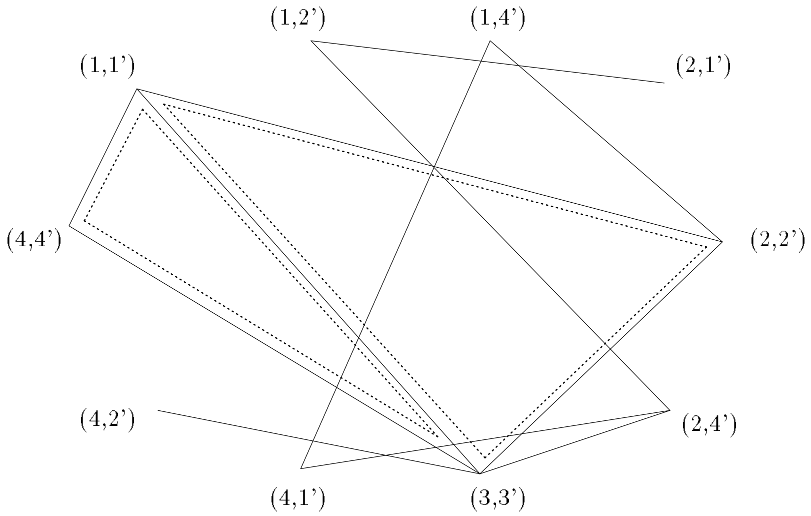

Figure A.2: Association graph for g_1 and g_2 in Fig. A.1.

$V \times V_I$, $E_A \subseteq V_A \times V_A$ and

1. $V_A = \{(v, v_I) |\ v \in V, v_I \in V_I, \text{ and } \mu(v) = \mu_I(v_I)\}$
2. E_A consists of all a-edges $e_A = (v_A, v'_A)$ with $v_A = (v, v_I)$, $v'_A = (v', v'_I)$ such that
   (a) $v \neq v'$ and $v_I \neq v'_I$
   (b) if there is an edge $e = (v, v') \in E$ then there must be an edge $e_I = (v_I, v'_I) \in E_I$ and $\nu(e) = \nu(e_I)$;
   (c) if there is no edge $(v, v') \in E$ then there must be no edge $(v_I, v'_I) \in E_I$.

□

Note that in the above definition, the graphs G and G_I are assumed to be undirected. However, the following algorithm can be easily adapted to directed graphs.

Based on the association graph G_A for two graphs G and G_I, the maximal common subgraph of G and G_I is now detected by searching for all maximal cliques in G_A. Let $C = \{v_{A_1}, v_{A_2}, .., v_{A_s}\}$ denote the set of a-vertices that constitute a clique in G_A. Then, the matching function that is implied by the a-vertices in C is a graph isomorphism from the subgraph $S \subset G$ to the subgraph $S_I \subset G_I$ with $S = \{v | (v, v_I) \in C\}$ and $S_I = \{v_I | (v, v_I) \in C\}$. Clearly, if $S = G$ then a subgraph isomorphism from G to G_I has been found. On the other hand, if $S_I = G_I$ then a subgraph isomorphism from G_I to G has been detected. The problem of finding the largest clique in a graph is well known in graph theory. In the following, we describe an algorithm which is closely related to the algorithm for detecting the largest independent set given in [Chr75]. The basic idea of the maximal clique detection algorithm is to first choose a single a-vertex v_{A_1} from the association

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

1. create an association graph $G_A = (V_A, E_A)$ from G and G_I according to Def. A.1.
2. let $i = 1$ and $C, C_{max}, CLIQUES = \emptyset$ and $L(k) = \emptyset$ for all $k = 1, \ldots, MIN\{|V|, |V_I|\}$
3. if there exists $v_A \in V_A$ with $v_A \notin L(k)$ for all $k \leq i$ then
   (a) set $L(i) = L(i) \cup \{v_A\}$
   (b) if there exists an edge $e_A = (v_A, w_A) \in E_A$ for all $w_A \in C$ then add v_A to C, $C = C \cup \{v_A\}$ and set $i = i + 1$. If $|C| > |C_{max}|$ then set $CLIQUES = \{C\}$ and $C_{max} = C$. Otherwise, if $|C| = |C_{max}|$ then set $CLIQUES = CLIQUES \cup \{C\}$.

   else

   (c) remove the $i-1$-th vertex from C.
   (d) set $L(i) = \emptyset$ and set $i = i - 1$.
4. if $i > 0$ then goto 3.
5. for each $C \in CLIQUES$
   (a) let $S = \{v | (v, v_I) \in C\}$ be the subgraph in G and $S_I = \{v_I | (v, v_I) \in C\}$ be the subgraph of G_I that are isomorphic and output the graph isomorphism function $f : S \to S_I$ that is implied by C.

Figure A.3: Algorithm *GM_by_clique_detection*.

graph and add it to a yet empty list C. Next, the list C is extended by adding another a-vertex v_{A_2} from the association graph such that there exists an a-edge connecting v_{A_2} directly to v_{A_1}. In general, a new a-vertex v_{A_i} is added to C if v_{A_i} is connected to each of the a-vertices in C. Thus, it is guaranteed that the list of a-vertices C always describes a clique of the association graph. This process stops, if no more a-vertices can be added to C. It is then necessary to backtrack one step, remove the previously added a-vertex from C and add an a-vertex that has not yet been tried. In order to avoid testing identical cliques multiple times and to keep track of the a-vertices that were already tried, a set of lists $L(i)$ is introduced, with $i = 1, \ldots, MIN\{|V|, |V_I|\}$. The purpose of the $L(i)$'s is to store for each size of the set C the a-vertices that have already been tried at some previous stage of the algorithm. In Fig. A.3, the algorithm *GM_by_clique_detection* for graph matching is formally given.

The algorithm *GM_by_clique_detection* takes as input a model graph G and an input graph G_I. In step 1, an association graph G_A is built from G and G_I based on the Def. A.1. Next, in steps 2-4, all maximal cliques in the association graph G_A are detected. For this purpose, C and $L(i)$ for all $i = 1, \ldots, MIN\{|V|, |V_I|\}$ are initialized as empty in step 2. Also, the list C_{max} which is used to store the largest maximal clique that is found at any time of the algorithm and the set $CLIQUES$ of all maximal cliques are initialized as empty. In step 3, an a-vertex $v_A \in V_A$ is selected such that v_A has not been used before. That is, if $v_A \notin L(k)$ for any $k \leq i$ then v_A is added to the current list $L(i)$ such that it will not be used again for the same clique. In step 3.b, it is then checked whether v_A is directly connected to each of the a-vertices in C such that $C \cup \{v_A\}$ forms a clique. If this is the case and the new clique $C = C \cup \{v_A\}$ is larger than any previously detected clique C_{max}, i.e. $|C_{max}| \leq |C|$, then reset the list of maximal cliques $CLIQUES = \{C\}$ and set $C_{max} = C$. On the other hand, if C and the previously detected C_{max} are of equal size, i.e. $|C| = |C_{max}|$, then add C to $CLIQUES$. The search for maximal cliques continues in step 3 until for some size i no new vertex v_A can be added to C. Consequently, in step 3.c, the algorithm backtracks from the size i to $i-1$ by removing the i-th a-vertex from C. Furthermore, the list $L(i)$ is emptied and i is decremented by one. Clearly, as long as i is larger than one, it it possible to find new cliques and the algorithm continues with step 3. However, as soon as i becomes zero, all maximal cliques have been successfully detected. Consequently, all maximal common subgraphs and the corresponding graph isomorphisms can be outputted in step 5.

It is easy to see that *GM_by_clique_detection* is indeed capable of finding either all subgraph isomorphisms from G to G_I if G is smaller than G_I, all subgraph isomorphisms from G_I to G if G is larger than G_I or all graph isomorphisms between G and G_I if they are of equal size. However, the main drawback of the graph matching algorithm based on clique detection is its computational complexity. Given two graphs G and G_I with n and m vertices, respectively, then the size of the association graph and the number of possible cliques strongly depends on the number of labels in G and G_I. For convenience, let $n \leq m$. In the best case, both G and G_I are uniquely labeled and each vertex in G matches to exactly one other vertex in G_I. Thus, the corresponding association graph contains $O(n)$ a-vertices and can be created in $O(nm)$ steps. The search for the maximal clique terminates after the initial clique is extended $O(n)$ times with $O(n)$ connectivity tests for each extension. Therefore, the total computational complexity is in the best case bounded by $O(nm + n^2)$. Clearly, if there is more than one model that must be matched with the input graph, then *GM_by_clique_detection* must be called for each model-input pair. Hence, if there are L models, the best case complexity of the algorithm is linearly dependent on L, i.e.,

$$O(Lnm) \tag{A.1}$$

In the worst case, however, all vertices of G and G_I are identically labeled and each graph is completely connected such that the corresponding association graph consists of $O(nm)$ a-vertices and each a-vertex is connected to $O(nm-n)$ other a-vertices. Clearly, at each level of the backtracking process the current clique can be extended with $O(nm)$ a-vertices. As there are $O(n)$ levels, the worst case time complexity is bounded by $O((nm)^n)$. Again, if L model graphs must be matched with one input graph, the algorithm must be applied sequentially to each pairing of two graphs, resulting in a worst case complexity of

$$O(L(nm)^n) \tag{A.2}$$

This result indicates that the clique detection based graph matching algorithm will be less efficient in the worst case than alternative conventional methods such as Ullman's method (see next section). For this reason, we did not choose the performance of the algorithm *GM_by_clique_detection* as a benchmark in this report.

A.2 Exact Subgraph Isomorphism Detection by Ullman's Method

In this section, we present a well known algorithm for the problem of graph and subgraph isomorphism detection. The algorithm is based on the principle of backtracking in combination with a forward-checking technique. It was first described in [Ull76].

Given a model graph $G=(V,E,\mu,\nu)$ and an input graph $G_I=(V_I,E_I,\mu_I,\nu_I)$, we intend to find all graph or subgraph isomorphisms from G to G_I. It is easy to imagine a straightforward enumeration procedure in which all possible mappings from vertices of G to vertices of G_I are generated and subsequently tested for the conditions of graph or subgraph isomorphism according toi Def. 2.5 and 2.6 in Chapter 2. However, the number of possible mappings is highly combinatorial and, therefore, such a procedure is not applicable in practice. In order to reduce the number of mappings that must be tested, it is better to start with a single vertex to vertex mapping and then gradually extend this mapping such that the resulting matching function always denotes a subgraph isomorphism. If, at some point, the mapping does not represent a subgraph isomorphism, then the process backtracks. That is, a previously mapped model vertex is reassigned to another input vertex and the conditions for subgraph isomorphism are again tested. Formally, let $v_1,\ldots,v_n$ and $w_1,\ldots,w_m$ denote the set of vertices of G and G_I, respectively. In the beginning, the vertex v_1 is mapped with the vertex w_{x_1} where $x_1\in[1,m]$ and it is checked whether the labels of v_1 and w_{x_1} are identical, i.e. $\mu(v)=\mu_I(w_{x_1})$. If this is the case, then the mapping denoted by $\{(v_1,w_{x_1})\}$ is a graph isomorphism from the subgraph $\{v_1\}$ of G to the subgraph $\{w_{x_1}\}$ of G_I. It can be extended by mapping the second vertex v_2 onto some input vertex w_{x_2}. In general, a mapping $\{(v_1,w_{x_1}),\ldots,(v_i,w_{x_i})\}$ that represents a subgraph isomorphism can be extended by mapping the vertex v_{i+1} onto some input graph vertex $w_{x_{i+1}}$ with $w_{x_{i+1}}\neq w_{x_k}$ for $k<i+1$ if the following two

conditions are satisfied. First, it must be true that $\mu(v_{i+1}) = \mu_I(w_{x_{i+1}})$. Secondly, for each edge $e = (v_k, v_{i+1}) \in E$ with $k \leq i$, there must be a corresponding edge $e_I = (w_{x_k}, w_{x_{i+1}}) \in E_I$ with $\nu(e) = \nu_I(e_I)$ and vice versa. Clearly, these two conditions guarantee that $\{(v_1, w_{x_1}), \ldots, (v_{i+1}, w_{x_{i+1}})\}$ represents a subgraph isomorphism from the subgraph consisting of the first $i+1$ vertices of G to G_I. If, at some level i, the vertex v_i cannot be matched onto any vertex w_{x_i}, the algorithm backtracks to the level $i-1$ and matches the vertex v_{i-1} with some vertex of the input graph that has not been tried before. If, on the other hand, a mapping $\{(v_1, w_{x_1}), \ldots, (v_n, w_{x_n})\}$ has been generated then a subgraph isomorphism from G to G_I has been found.

The algorithm described so far performs much better than the simple enumeration procedure due to the fact after each new vertex to vertex matching, a backward check for the conditions of subgraph isomorphism is made. However, in [Ull76], it was noted that the efficiency of the backtrack algorithm can be further improved by introducing a forward checking procedure. The basic idea of this procedure, is to check for each mapping (v_i, w_{x_i}) whether there exists at least one mapping for each future vertex v_j onto some vertex w_{x_j} with $j > i$ such that the conditions for subgraph isomorphism hold true. The advantage of forward checking the conditions for subgraph isomorphism at each level of the algorithm is the following. Given the situation that there is a mapping for the vertex v_i onto some vertex w_{x_i} which is consistent with all mappings (v_k, w_{x_k}) for $k < i$ but for which no consistent mapping for the vertex v_j with $j > i$ exists, forward checking will immediately reject the mapping (v_i, w_{x_i}) and the algorithm will backtracking to a previous level. Without forward checking, however, the algorithm will perform unnecessary computation steps in the intermediate levels l with $i < l < j$ before it is discovered that the vertex v_j cannot not be mapped onto the input graph due the previous mapping (v_i, w_{x_i}). For this reason, the forward checking procedure proposed by Ullman improves the efficiency of the backtracking algorithm. In the following, we give a detailed description of the backtracking algorithm and Ullman's forward checking procedure.

The algorithm *GM_Ullman* consists of the recursive subroutine *Backtrack* (Fig. A.4) and the procedure *Forward_checking*(Fig. A.5). There are two main data structures used by these procedures. First, in order to keep track of all possible vertex to vertex mappings, an $n \times m$ matrix $P = (p_{ij})$ is introduced with $p_{ij} \in \{0, 1\}$. The rows of P denote the vertices $v_1, \ldots, v_n$ of G and the columns of P denote the vertices $w_1, \ldots, w_m$ of G_I. If the vertex v_i of G can be mapped consistently with the vertex w_j of G_I then $p_{ij} = 1$. Otherwise, if the subgraph isomorphism conditions are violated by mapping v_i onto w_j, then $p_{ij} = 0$. The second data structure is a set F that is used for the representation of vertex to vertex mappings $F = \{(v_1, w_{x_i}), \ldots, (v_i, w_{x_i})\}$. The input to the algorithm *GM_Ullman* consists of a model graph G and an input graph G_I. In the first step, the matrix P is initialized by setting $p_{ij} = 1$ if the label of v_i is identical to the label of w_j, and $p_{ij} = 0$ otherwise, with $i = 1, \ldots, n$ and $j = 1, \ldots, m$. Then, in step 2, the recursive procedure *Backtrack* is called. The input to *Backtrack* is the matrix P, the current level i and the set of vertex mappings F. In the beginning, *Backtrack* is called with the matrix

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

1. let $P = (p_{ij})$ be a $n \times m$ matrix, $n = |V|$, $m = |V_I|$. Initialize P as follows. For each p_{ij}, $i = 1, \ldots, n$ and $j = 1, \ldots, m$, and $v_i \in V$, $w_j \in V_I$, if

$$\mu(v_i) = \mu_I(w_j)$$

then set $p_{ij} = 1$ else set $p_{ij} = 0$.

2. call *Backtrack*($P, 1, \emptyset$)

3. procedure Backtrack(P, i, F)

   (a) if $i > n$ then F represents a subgraph isomorphism from G to G_I. Output F and return.

   (b) for all $p_{ij} = 1$ and $j = 1$ to m

      i. set $F = F \cup \{(v_i, w_j)\}$

      ii. set $P' = P$ and $p'_{kj} = 0$ for all $k > i$

      iii. if *Forward_checking*(P', i, F) $= TRUE$ then call Backtrack($P', i + 1, F$)

      iv. remove (v_i, w_j) from F

Figure A.4: Algorithm *GM_Ullman*.

P as it was initialized in step 1, the level 1, and the empty set F. Clearly, if in step 3.a it is true that $i > n$ then all vertices of G have been successfully mapped onto G_I and F represents a subgraph isomorphism from G to G_I. On the other hand, if $i \leq n$ then v_i is the next vertex to be mapped onto G_I. For this purpose, it is checked for each vertex w_j, $j = 1, \ldots, m$, if $p_{ij} = 1$. If this is the case then v_i can be mapped with w_j and $F \cup \{(v_i, w_j)\}$ represents a subgraph isomorphism from the subgraph $\{v_1, \ldots, v_i\}$ of G to the input graph G_I. It is important to note that the conditions of subgraph isomorphism are checked in the procedure *Forward_checking* and reflected in the matrix P. Hence, with this new mapping, the matrix P must be updated. For this purpose, in step 3.b.ii, an identical copy P' of P is made such that the original matrix P remains unchanged and can be reused when backtracking occurs. In the matrix P', all entries p'_{kj} with $k > i$ are set to zero because the vertex w_j cannot not be mapped to by more than one model graph vertex. Next, in step 3.b.iii, the procedure *Forward_checking* (see below) is called in order to update the matrix P'. If it is detected in *Forward_checking* that for some future vertex v_k with $k > i$ no corresponding input vertex can be found, then FALSE is returned

FORWARD_CHECKING(P, i, F)

1. for $k = i + 1$ to n and $j = 1$ to m
   (a) if $p_{kl} = 0$ goto step 1.
   (b) check, if (v_k, w_l) is consistent with all the mappings in F. More precisely, for each $(v, w) \in F$
      i. if there is an edge $e = (v_k, v) \in E$ then there must be an edge $e_I = (w_l, w) \in E_I$ with $\nu(e) = \nu_I(e_I)$.
      ii. if there is an edge $e_I = (w_l, w) \in E_I$ then there must be an edge $e = (v_k, v) \in E$ with $\nu_I(e_I) = \nu(e)$.

      If both conditions (i) and (ii) hold true then set $p_{kl} = 1$ else set $p_{kl} = 0$.
2. if there exists a row k in P such that $p_{kl} = 0$ for $l = 1, \ldots, m$ then return FALSE else return TRUE

Figure A.5: Procedure *Forward_checking*.

and the algorithm continues directly in step 3.b.iv. Otherwise, *Backtrack* is called recursively with the updated matrix P', the level $i + 1$ and the extended set F. When *Backtrack* returns, the matching (v_i, w_j) must be removed again from F in step 3.b.iv and the next entry $p_{i(j+1)}$ is considered in step 3.b.

The procedure *Forward_checking* takes as input the matrix P, the level i and the set of mappings F. Its purpose is to test for each entry $p_{kl} = 1$ of P with $k > i$ and $l = 1, \ldots, m$ whether the mapping (v_k, w_l) is consistent with F with regard to the subgraph isomorphism conditions according to Def. 2.5 and 2.6 in Chapter 2. Hence, in step 1.b, it is checked for each $(v, w) \in F$ and for each entry $p_{kl} = 1$ if there exists an edge $e = (v, v_k) \in E$ and a corresponding edge $e_I = (w, w_l) \in E_I$ with $\nu(e) = \nu_I(e_I)$. Furthermore, it is checked if for each edge $e_I = (w, w_l) \in E_I$ there is a corresponding edge $e = (v, v_k) \in E$ with $\nu_I(e_I) = \nu(e)$. If this is the case then v_k can be matched to w_l and $p_{kl} = 1$ is set. Otherwise, v_k cannot be mapped with w_l and $p_{kl} = 0$ must be set. Clearly, if there is a row k in P such that $p_{kl} = 0$ for all $l = 1, \ldots, m$ then it can be concluded that the current set of vertex mappings F cannot be extended to form a subgraph isomorphism from G to G_I because for the vertex v_k of G no corresponding vertex in the input graph G_I exists. Consequently, the procedure *Forward_checking* returns FALSE. If, on the other hand, there is at least one possible mapping for each vertex v_k, i.e., there is at least one entry $p_{kl} = 1$ in each row k of P, then *Forward_checking* returns TRUE.

It is easy to see that the algorithm *GM_Ullman* outputs all subgraph isomorphisms from a model graph G to an input graph G_I. Due to the fact that the algorithm backtracks whenever a new subgraph isomorphism is found and investigates new vertex to vertex mappings on previous levels, no subgraph isomorphism from G to G_I is missed.

The computational complexity of the algorithm is strongly dependent on the size of the graphs and the labeling of the vertices. Given a model graph G with n vertices and an input G_I with m vertices, the best case for Ullman's algorithm arises when all the vertices of the model graph are uniquely labeled and the input graph contains an isomorphic copy of the model graph. In this situation, each vertex of the model graph can be mapped onto exactly one vertex of the input graph. The initialization of the compatibility matrix P can be done in $O(nm)$ steps. Next, the procedure *Backtrack* will be called $O(n)$ times as there is only one possible vertex to vertex mapping on each level. For each of these calls, there will be $O(n)$ steps performed in the procedure *Forward_checking*. Consequently, the computational complexity of Ullman's algorithm is in the best case bounded by $O(n^2 + nm) = O(nm)$. Clearly, the algorithm can only match two graphs at a time. Thus, if there are L model graphs in the database, the computational complexity will be linearly dependent on L, i.e.

$$O(Lmn) \tag{A.3}$$

The worst case for Ullman's algorithm arises when the vertices in the model and the input graph are unlabeled and the connectivity is very high. As a consequence, each model vertex can be mapped with each input vertex. Furthermore, due to the high connectivity, forward checking can not efficiently limit the search space. That is, on each level of the procedure *Backtrack*, there will be $O(m)$ recursive calls to *Backtrack*. As there are $O(n)$ levels for a model graph with n vertices, the number of recursive calls is bounded by $O(m^n)$. For each of these calls, the procedure *Forward_checking* is executed and $O(n^2)$ steps are performed in order to update the matrix P. Hence, the computational complexity of Ullman's algorithm is in the worst case bounded by $O(m^n n^2)$. As in the best case, given a database of L model graphs, the computational complexity is again linearly dependent on L, i.e

$$O(Lm^n n^2) \tag{A.4}$$

Although, in the worst case, the run time may be exponential in the number of the model graph vertices, the practical performance of Ullman's algorithm is very good. For this reason, Ullman's algorithm was chosen as a benchmark in this thesis.

A.3 Error-correcting Subgraph Isomorphism Detection by A^*

In this section, we present a well known algorithm for the problem of error-correcting subgraph isomorphism detection. The algorithm is based on the A^* algorithm [Nil80] and has been previously described for graph matching by various authors [SF83, Sha93, TF79]. Notice that the definitions of a graph, the edit operations and the concept of an error-correcting subgraph isomorphism can be found in Chapter 3.

Given a graph $G = (V, E, \mu, \nu)$, an input $G_I = (V_I, E_I, \mu_I, \nu_I)$ and a set of graph edit operations for which a cost function is defined, we want to find an optimal error-correcting subgraph isomorphism (Δ, f) such that there exists a subgraph isomorphism f from $\Delta(G)$ to G_I and the costs of Δ, $C(\Delta)$, are minimal. For convenience, we introduce the following notation. Let p be a mapping from the vertices of G to the vertices G_I and p consists of tuples (v, w), $v \in V$ and $w \in (V_I \cup \{\$\})$, such that there exist no two tuples $(v, w), (v', w') \in p$ with $v = v'$ or $w = w' \neq \$$. Note that the symbol \$ is used to denote missing vertices in the input graph. Furthermore, let $V^p = \{v|(v, w) \in p\}$ and $V_I^p = \{w|(v, w) \in p\}$ denote subsets of vertices of G and G_I, respectively. Clearly, a mapping p implies a set of edit operations Δ^p on the subgraph S^p of G induced by the vertices V^p and also the subgraph isomorphism f that matches $\Delta^p(S^p)$ to G_I. Formally, given a mapping p, the set Δ^p consists of the following edit operations (see Def. 3.7 in Chapter 3). For each tuple (v, w) in p

1. if $w \neq \$$ and $\mu(v) \neq \mu_I(w)$ then the substitution of $\mu(v)$ by $\mu_I(w)$ is implied by p, i.e. $(\mu(v) \rightarrow \mu_I(v_I)) \in \Delta^p$
2. if $w = \$$ then the deletion of v from G is implied by p, i.e. $(v \rightarrow \$) \in \Delta^p$

Also, Δ^p may contain edge edit operations. That is, for each pair of tuples (v, w), $(v', w') \in p$

1. if there exists an edge $e = (v, v') \in E$ and an edge $e_I = (w, w') \in E_I$ and $\nu(e) \neq \nu_I(e_I)$ then the substitution of $\nu(e)$ by $\nu_I(e_I)$ is implied by p, i.e. $(\nu(e) \rightarrow \nu_I(e_I)) \in \Delta^p$
2. if there exists an edge $e = (v, v') \in E$ and there is no edge $(w, w') \in E_I$ then the deletion of e from G is implied by p, i.e. $(e \rightarrow \$) \in \Delta^p$
3. if there exists no edge $(v, v') \in E$ but there is an edge $e_I = (w, w') \in E_I$ then the insertion of (v, v') into G is implied by p, i.e. $(\$ \rightarrow (v, v')) \in \Delta^p$

The cost $C(p)$ of a mapping p is the sum of the cost of the implied edit operations, $C(p) = C(\Delta^p)$. The subgraph isomorphism f^p from $\Delta^p(S^p)$ to G_I that is implied by p is defined as

$$f^p(v) = \begin{cases} w & \text{if } (v, w) \in p \text{ and } w \neq \$ \\ \text{undefined} & \text{otherwise} \end{cases}$$

for all v in $\Delta^p(G)$.

With this notation, the problem of error-correcting subgraph isomorphism detection can now be restated as follows. Given the graphs G and G_I and a set of edit operations, we want to find a mapping p such that $V^p = V$ and $C(p)$ is minimal over all mappings. The search for such a mapping can be performed by an algorithm that is similar to the A^* algorithm. The basic idea is to take the first vertex v_1 of the graph G and create all possible mappings p from this vertex to the vertices of G_I and the dummy vertex \$. Next, the mapping $p = \{(v_1, w_i)\}$ with minimal cost $C(p)$ is selected and expanded into a number of larger mappings. That is, for each possible mapping of the second vertex v_2 of G to a vertex w_j in $V_I - \{w_i\}$, an extended mapping $p' = \{(v_1, w_i), (v_2, w_j)\}$ is created. In general, the algorithm always selects the mapping $p = \{(v_1, w_i), \ldots, (v_k, w_j)\}$ with minimal costs $C(p)$ and expands it by adding a mapping for the next vertex $v_{k+1} \in V$. This process is continued until a mapping with minimal costs is found that represents a subgraph isomorphism from the complete graph G to the graph G_I. It is easy to see that by always expanding the mapping with the least cost, the algorithm is guaranteed to find the optimal mapping p such that Δ^p is a sequence of edit operations with minimal cost and f^p is a subgraph isomorphism from $\Delta^p(G)$ to G_I.

In Fig. A.6 the algorithm *GM_by_A** is formally described. The input to *GM_by_A** consists of the model graph G and the input graph G_I. For the organization of the mappings that are generated during the search process, a list $OPEN$ is introduced. In the beginning, $OPEN$ is initialized with all the mappings from the first vertex v_1 in V to the vertices in V_I. Next, after testing in step 2 that $OPEN$ is not empty, the mapping p with minimal costs $C(p)$ is removed from $OPEN$ and its costs are compared to the threshold *accept_threshold*. At first, *accept_threshold* denotes the highest costs that are allowed for an error-correcting subgraph isomorphism to be acceptable. Later, when a first mapping p' describing a complete subgraph isomorphism from G to G_I has been found (see below), the value of *accept_threshold* will be reset to the cost $C(p')$. Consequently, the algorithm will terminate its search when all remaining mappings have costs that exceed either the original threshold or the cost for the optimal error-correcting subgraph isomorphism. Next, in step 5, it is checked whether the current mapping p represents a subgraph isomorphism from the complete model G to the input G_I. If this is the case, then p is outputted and an optimal error-correcting subgraph isomorphism is found. Additionally, the threshold *acceptance_threshold* is reset to $C(p)$, as mentioned above and the algorithm continues with step 2. If, on the other hand, the current mapping $p = \{(v_1, w_i), \ldots, (v_k, w_j)\}$ is not yet complete, then it is expanded in steps 7a-b by mapping the next vertex v_{k+1} onto the remaining vertices of G_I. I.e., for each vertex $w \in (V_I - V_I^p) \cup \{\$\}$ where V_I^p denotes the vertices in V_I that are mapped to by p, a new mapping $p' = \{(v_1, w_i), \ldots, (v_{k+1}, w)\}$ is created. After computing the edit costs $C(p')$ that are implied by p', it is added to $OPEN$. The algorithm continues to expand and create new mappings until either the cost of the current mapping exceed the threshold (step 4) or the list of mappings $OPEN$ is empty (step 2).

It is easy to see that *GM_by_A** indeed detects all optimal error-correcting sub-

GM_BY_A*$(G = (V, E, \mu, \nu), G_I = (V_I, E_I, \mu_i, \nu_I))$

1. initialize $OPEN$: map the first vertex v_1 in V onto each vertex in V_I, i.e. for each vertex $w \in V_I \cup \{\$\}$, create a mapping $p = \{(v_1, w)\}$. Set $C(p) = C(\mu(v_1) \to \mu_I(w))$ and add p to $OPEN$, i.e. $OPEN = OPEN \cup \{p\}$.
2. if $OPEN$ is empty then exit.
3. select $p \in OPEN$ such that $C(p)$ is minimal and remove p from $OPEN$.
4. if $C(p) > accept\_threshold$ then exit.
5. if p represents a complete mapping from G to G_I then output p. Set $accept\_threshold = C(p)$. Goto 2.
6. let $p = \{(v_1, w_i), \ldots, (v_k, w_j)\}$ and $V_I^p = \{w | (v, w) \in p \text{ and } w \in V_I\}$.
7. for all $w \in (V_I - V_I^p) \cup \{\$\}$
   (a) set $p' = \{(v_1, w_i), \ldots, (v_k, w_j), (v_{k+1}, w)\}$ and compute $C(p')$ (see text)
   (b) add p' to $OPEN$, i.e. $OPEN = OPEN \cup \{p'\}$
8. goto 2.

Figure A.6: Algorithm *GM_by_A**.

graph isomorphisms from G to G_I. Clearly, given a mapping p representing an optimal error-correcting subgraph isomorphism from G to G_I, it is true that for any subset p' of p the cost of p' are less than or equal to the cost of p, i.e $C(p') \leq C(p)$. Consequently, the algorithm does not terminate before p' has been expanded. Thus, the optimal mapping p will eventually be generated from some subset p' and outputted.

It has been noted by various researchers that the algorithm's performance can be greatly improved if in addition to the edit costs $C(p)$ of a partial mapping p, the *future edit costs* of p are estimated and also taken into account. Given a graph G, an input graph G_I and a mapping p the future edit costs of p are the minimal costs of any complete mapping p' which incorporates p. Obviously, the computation of the future edit cost of p is of the same complexity as expanding the partial mapping p until a complete mapping p' is found. For this reason, the future costs of p must be estimated. An efficient estimation procedure was proposed in [Won90]. The basic idea of the procedure is to estimate for each vertex in the set $V - V^p$ the minimal cost for mapping it onto some vertex in $V_I - V_I^p$. That is, each vertex

$v \in V - V^p$ is tentatively mapped to each vertex $w \in V_I - V_I^p$ and the implied edit cost of each mapping are compared to each other. There are both vertex and edge edit operations implied by mapping a vertex v onto a vertex w. First, the label $\mu(v)$ must be substituted by the label $\mu_I(w)$ and secondly, for each mapping $(v', w') \in p$ it must be ensured that any edge $e = (v', v) \in E$ can be mapped to an edge $e_I = (w', w) \in E_I$ by means of edge edit operations. The sum of the minimal edit cost for each vertex in $V - V^p$ is a lower bound estimation of the real future costs of the mapping p. If, for each mapping p in the search space, this estimation of the future edit costs is added to the edit cost $C(p)$ of p itself, then the algorithm will be better informed. Consequently, in step 3, the selection process will be more successful in choosing the mapping that will lead to an optimal error-correcting subgraph isomorphism.

The computational complexity of the A^*-based algorithm for error-correcting subgraph isomorphism detection depends on the size and the number of the models, the size of the input graph, the number of labels and the number of errors. Apparently, the best case for the algorithm arises when the vertices of the model and the input graph are uniquely labeled and the input graph contains an isomorphic, undistorted copy of the model graph. In this case, given a single model with n vertices and an input graph with m vertices, the algorithm first generates $O(m)$ mappings of the first vertex of the model. Out of these mappings, only the one with zero cost is further extended, resulting in $O(m-1)$ new mappings. This process continues for each of the $O(n)$ vertices of the model, resulting in a total of $O(nm)$ mappings. If no future cost estimation procedure is applied, then the algorithm performs $O(n)$ edge tests for each mapping. Thus, the number of steps for matching a single model graph onto an input graph when no future cost estimation is used, is in the best case bounded by

$$O(n^2 m) \tag{A.5}$$

If, on the other hand, the future cost are estimated for each mapping, then the computational effort for the future cost estimation procedure must also be taken into account. Clearly, given a mapping p containing $O(k)$ tuples, the future cost estimation procedure as given in [Won90], tentatively maps each of the remaining vertices $O(n-k)$ onto $O(m-k)$ input vertices. Hence, the computational complexity of the future cost estimation procedure is bounded by $O(nm)$ for each mapping. Consequently, with future cost estimation, the complexity is in the best case bounded by

$$O(n^2 m^2) \tag{A.6}$$

Although it seems contradictory that the complexity of the algorithm is higher when the future cost estimation procedure is applied, it must be remembered that the number of mappings can be greatly reduced by estimating the future edit costs. However, this reduction does not show up in the computational complexity analysis.

Due to the fact that the algorithm *GM_by_A** can only be applied to two graphs at a time, its performance is linearly dependent on the number of model graphs.

That is, without future cost estimation, the computational complexity for a database containing L models, is bounded by

$$O(Ln^2m) \tag{A.7}$$

Analogously, with future cost estimation, the complexity for a database of L models is given by

$$O(Ln^2m^2) \tag{A.8}$$

The worst case for *GM_by_A** arises when the error in the input graph is very large. In this case, the edit costs for correcting the model graph become very high and, consequently, each mapping in the search space will be extended. That is, the first vertex of the model graph can be mapped to $O(m)$ vertices of the input graph. As the edit costs of the optimal mapping are very high, each of the mappings for the first vertex will be extended into $O(m(m-1))$ new mappings. Generally, for the k-th vertex of the model, there will be $O(m^k)$ mappings. Hence, the total number of mappings generated in the worst case is bounded by $O(nm^n)$. Without future cost estimation, for each mapping there are $O(n)$ edge constraints tested, resulting in worst case complexity of

$$O(n^2m^n) \tag{A.9}$$

If, on the other hand, the future cost estimation procedure is called for each mapping, then the number of steps performed in the worst case is bounded by

$$O(n^2m^{n+1}) \tag{A.10}$$

As in the best case, the performance of *GM_by_A** is linearly dependent on the size of the database. Thus, without future cost estimation, the worst case complexity for L model graphs is bounded by

$$O(Ln^2m^n) \tag{A.11}$$

Analogously, with future cost estimation, the worst case complexity for L models is bounded by

$$O(Ln^2m^{n+1}) \tag{A.12}$$

Appendix B

GUB - A Toolkit for Graph Matching

In this appendix, we present the object-oriented toolkit for graph matching *GUB* (Graph matching toolkit of the University of Berne). This guide is written for researchers and programmers with programming experience in C or C++ and some knowledge of the graph matching problem.

GUB is a collection of C++ classes for the representation of attributed graphs and various algorithms for exact and error-correcting subgraph isomorphism detection. All of the data structures and algorithms in *GUB* have been implemented in GNU C++ version 2.6 and successfully tested on SUN SPARCstations 5, 10 and 20 under the operating systems SunOS 4.1.2 and Solaris 2.0 (SunOS 5.4). The purpose of this toolkit is to provide students, researchers and application programmers with an implementation of conventional and new methods for attributed graph matching. In particular, the toolkit supports the following operations:

- defining attributed graphs
- collecting attributed graphs as models in a database
- defining cost functions for the graph edit operations
- detecting
  - exact subgraph isomorphisms from the models to an input graph
  - exact subgraph isomorphisms from an input graph to the models
  - error-correcting subgraph isomorphisms from the models to an input graph

All of the classes and methods in the toolkit are designed such that they can either be incorporated into a standalone application or split up into a client/server system where the graph matching component is run as a server providing various clients with the above services.

The following guide to the *GUB* toolkit is organized as follows. First, in Section B.1 the class describing the concept of an attributed graph is presented. Next, in Section B.2, the edit operations that are used by the toolkit for error-correcting subgraph isomorphism detection and the corresponding cost functions are described. In Section B.3 the different algorithms are shortly presented along with their advantages and disadvantages. Finally, in Section B.4, the installation of the toolkit is explained.

B.1 The Graph Class

The class `Graph` which is included in the toolkit is based on a very general definition of a graph.

Definition B.1: A graph G is a 4-tuple $G = (V, E, \mu, \nu)$ where

- V is a set of vertices,
- E is a set of edges $e = (v, v')$ with $v, v' \in V$,
- $\mu : V \rightarrow L \times A^n$ is a function assigning labels and attributes to the vertices,
- $\nu : E \rightarrow L \times A^n$ is a function assigning labels and attributes to the edges.

□

L is a set of labels and A is a set of attributes. Notice that the above definition is more general than the definitions for graphs that were given in the previous chapters. In particular, Def. B.1 allows the assignment of complex attributes to the vertices and edges. Informally, the above definition includes the following types of graphs:

- directed graphs
- undirected graphs
- graphs with multiple edges between two vertices
- graphs with loop edges (edges with identical start and end vertex)
- attributed graphs

Before graphs can be successfully used in a new application domain, an encoding scheme must be developed which defines exactly how the objects in the domain are to be represented by attributed graphs. For example, in the domain of graphical

```
#
=L=
l 1 '   d 2   t 1 1.5   w 1 0.5   i 2   c 4
=L=
l 2 '   d 3   t 1 0 0   w 3 1 2   i 3   c 3
=L=
l 3 '   d 0   t         w         i 5   c 5
=T=
1 , 2 : + 10
=T=
2 , 3 : + 5
=E=
```

Figure B.1: Example for the attribute definition file *.gm_default_attributes* in which three different labels 1,2 and 3 are defined.

symbol recognition, 2-D symbols consisting of dots, lines and curves must be detected and recognized (see also Chapter 5). In order to solve this problem by means of graph matching, the symbols must be encoded as attributed graphs. That is, for each dot, line and curve and for the spatial relationship of these elements among each other, a corresponding graph representation must be defined. Clearly, while the existence of an element or a relationship can be directly encoded as a vertex or an edge, the structure and the qualities of an element such as the length of a line or the angle between two lines can only be represented by attributes associated with the vertices and edges. Furthermore, when matching two graphs by means of subgraph isomorphism the attributes of the graphs must be compared to each other in a meaningful way, i.e., an attribute representing an angle cannot be compared to an attribute representing the length of a line. Additionally, if an error-correcting subgraph isomorphism is searched for, the attributes are compared to each other by means of a cost function which must take into account thresholds and weights that are specific to each attribute (see next section). For this reason, the class `Graph` requires that attributes and their corresponding thresholds and weights are defined for an entire application. For this purpose, the ASCII file named *.gm_defaults_attributes* is provided. In this definition file, attributes and groups of attributes are given identifying labels. The set of labels is the set of integer numbers and the set of attributes is the set of real numbers. Each label identifies exactly the number of attributes, the thresholds and the weights of attributes and the costs for inserting or deleting a vertex or edge that is assigned a particular label.

In Fig. B.1, an example for the syntax of the file *.gm_default_attributes* is given. There are three labels 1,2 and 3 defined in this file. The label 1 is defined in the third row starting with the characters `l 1 '` followed by the term `d 2` indicating

```
class Graph{
 public:

  set(int vertex, int label, double* val, int n);
  setEdge(int v1, int v2, int edge, int label, double* val,int n);

  done(int directed);

  get(int vertex, int& label, double*& val, int& n);
  getEdge(int edge, int v1 , int v2, int label, double* val, int n);
  getEdgesBetween(int v1, int v2, int& label, double*& val, int &n);

  read(char* name);
  write(char* name);
  ...
}
```

Figure B.2: Public interface of class `Graph`

that there are two attributes for this label. Next, `t 1 1.5` sets the thresholds for the attributes to 1 and 1.5, respectively, while `w 1 0.5` sets the weights of the attributes to 1 and 0.5. Finally, `i 2` defines insertion costs of 2 and `c 2` denotes deletion costs of 4 for any vertex or edge that is labeled 1. Similarly, the label 2 identifies 3 attributes while the label 3 identifies no attributes at all. The last two rows in the file are optional and set the costs for the substitution of label 1 by label 2 to 10 cost units and the costs for substituting label 2 by label 3 to 5 cost units.

Based on the above definition of the labels and attributes, the interface to the class `Graph` is now given. In Fig. B.2 the most important public methods of the class `Graph` are listed. Note that there are two methods for setting vertices and edges in an graph, an auxiliary method called `done` (see below), three methods for accessing vertices and edges in an existing graph, and two methods for reading and writing a graph from and to a file.

The creation of an instance of the class `Graph` requires that a variable `G` of type `Graph` is declared. Then, the vertices and edges in G can be set by calling `G` with

```
G.set(v1,label1,values1,n1);
```

and

```
G.setEdge(v1,v2,e,label2,values2,n2);
```

repeatedly with `v1,v2` and `e` being the identifiers of the vertices and edges and `label1,label2` being the labels along with pointers to the attributes `values1`, `values2` of dimensions `n1,n2`. The creation of a graph instance must always be terminated with a call to the function `done`. The argument to this function determines whether the final graph is directed or undirected. I.e., if `done` is called with a non-zero argument then the graph is directed, otherwise it is undirected. For an existing graph, the vertices and edges along with their labels and attributes can be accessed by calling one of the methods `get`,`getEdges` or `getEdgesBetween`.

B.2 The Edit Operations and the Cost Functions

One of the most important problems when using error-correcting subgraph isomorphism for the comparison, detection or recognition of objects that are represented by attributed graphs, is the definition of the graph edit operations and the corresponding cost functions. The toolkit *GUB* supports the following edit operations on instances of the class `Graph`:

- vertex deletion and insertion
- edge deletion and insertion
- attributes and label substitution

For each of these operations a default cost function is also provided. These default cost functions compute the costs for certain operations on the basis of the labels and attributes and the thresholds and weights defined in the file *.gm_default_attributes* (see Fig. B.1 for an example). Thus, given an entry

```
l 1 '   d 2   t 1 1.5   w 1 0.5   i 2   c 4
```

in the file *.gm_default_attributes*, the cost for inserting a vertex with the label 1 costs 2 units while the deletion of the same vertex costs 4 units. By default, the cost for the deletion and insertion of an edge are computed analogously. The default computation of the cost for substituting a label and its associated attributes with some other label and its attributes is slightly more complicated. Given a label l_1 and attributes $a_1, \ldots, a_n$ and another label l_2 and attributes $b_1, \ldots, b_m$, the cost for substituting l_1 with l_2 is computed as follows. If l_1 and l_2 are not identical then the substitution cost are set to the cost that are defined in the file *.gm_default_attributes*. If, on the other hand l_1 and l_2 are identical then the number of attributes for l_1 and l_2 is equal, i.e. $n = m$, and the substitution cost C_s are defined to be the weighted Euclidean distance between $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$, i.e.,

$$C_s = \sum_{i=1}^{n} w_i (d_{a_i b_i})^2$$

```
//example definition of a cost function

  double substitution_cost_function(int label1,double *values1,int n1,
                                    double *w1,double *t1,
                                    int label2,double *values2,int n2,
                                    double *w2,double *t2,
                                    double lw1,double lw2){

  double cost;
  // body of user-defined cost function
  ...
  // end body
  return cost;
}

// register the function

registerFunction(SUB_FUNCTION,(void*(*)())substitution_cost_function);
```

Figure B.3: Definition and registration of a cost function.

Notice that w_i is the weight assigned to the i-th attribute and $d_{a_ib_i}$ is

$$d_{a_ib_i} = \begin{cases} 0 & \text{if } |a_i - b_i| < t_i \\ a_i - b_i & \text{otherwise} \end{cases}$$

with t_i being the threshold assigned to the i-th attribute. Informally speaking, when comparing two attributes a_i and b_i, they may differ from each other within the limits of a certain threshold t_i before the difference is put to the power of two, weighted by w_i and added to the substitution cost.

In practice, the above definition of the cost functions for the edit operations have proven to be useful in many different applications and, particularly, for the development of first prototypes. However, in order to support also applications with the need for specialized cost functions, *GUB* offers the possibility to register user-defined cost functions. A user-defined cost function is a regular C-function which will be called by the toolkit instead of the default cost functions whenever an edit operations is performed during a graph matching process. For each of the edit operations it is possible to write a specific cost function. In Fig. B.3 an example of a user-defined cost function for the computation of the substitution cost is given. Notice that the argument list is mandatory, that is, the user-defined function will always be called with these arguments. Furthermore, the return value of the function must be of type `double`. The registration of the user-defined function `my_func_name` with the toolkit can be done by calling `registerFunction(OPERATION,(void* (*)()) my_func_name)` where `OPERATION` identifies the edit operation to which the registered cost function corresponds. Possible values of OPERATION are SUB_FUNCTION, INS_OF_EDGES, INS_OF_VERTICES, DEL_OF_EDGES and DEL_OF_VERTICES.

B.3 The Methods

The main component of the *GUB* toolkit is the collection of different algorithms for the problem of exact and error-correcting graph and subgraph isomorphism detection. Each of the following methods is contained in *GUB* and can be applied to instances of the class `Graph`:

- Ullman's algorithm (Appendix A.2)
- A^* algorithm (Appendix A.3)
- decomposition-based method (Chapter 2 and 3)
- decision tree method (Chapter 4)

Due to the fact that both the decomposition-based method and the decision tree method involve an intensive preprocessing step in which a special representation of the model graphs is computed, it is necessary to organize both the model graphs and the special representations persistently. For this reason, the algorithms are not implemented as simple functions, but instead they are encapsulated in a special class `KB`. In the following, we first describe the major components of the class `KB` and then we list for each of the supported graph matching methods its properties, advantages and disadvantages.

The purpose of the database class `KB` is to represent a set of model graphs and the special representations that have been compiled by either the decomposition-based method or the decision tree method. In Fig. B.4 the public interface and access methods of class `KB` are listed. Given an instance `D` of class `KB`, a new model graph `G` can be added to `D` with `addModel(G,key)` where `key` is an identifier which must be unique for each model. In order to match an input graph G' with the set of model graphs that are stored in `D` the method `recognition(G',t)` must be called. The parameter `t` is only relevant to the matching process if the applied algorithm supports error-correcting subgraph isomorphism detection. In this case, all error-correcting subgraph isomorphisms with edit costs less than t are searched for.[1] The number of detected subgraph isomorphisms – exact or error-correcting – can be accessed with `NumberOfMatches` and each of the individual solutions is

[1] A special case arises for $t = -1$ where an unrestricted search for the error-correcting subgraph isomorphisms with minimal cost is performed.

```
class KB{
 public:

  addModel(Graph *model, int key);

  recognition(Graph *input, double threshold);

  NumberOfMatches();
  InstanceData* getInstance(int index);

  setMethod(int method, int version, double parameters);
  ...

 private:
  ListOfGraphs models;
};
```

Figure B.4: Public interface of class *KB*

retrieved by `getInstance(i)` where `i` is an index into the set of solutions. The return value of `getInstance` is a pointer of the type `InstanceData`. In Fig. B.5 the structure of `InstanceData` is displayed. The purpose of an instance of the type `InstanceData` is to describe an exact or error-correcting subgraph isomorphism that was found by the graph matching process. For example, the array `model` contains the numbers of the vertices in the model graph that have been successfully mapped to the vertices of the input graph denoted by the array `image`. The remaining components of `InstanceData` provide additional information about the described subgraph isomorphism.

It is important to note that the process of adding a graph to the database of models and matching an input graph with this database is strongly dependent on the underlying graph matching method. For example, if the underlying method is Ullman's algorithm then the model graphs are simply stored in `KB` and then sequentially matched to the input graph. On the other hand, if the underlying method is decision tree based, the model graphs are first compiled into a decision tree structure. By default, at the creation of an instance of class `KB` the parameter file *.gm_defaults* is consulted and the graph matching method specified in this file is activated. In Fig. B.6 an example of the syntax of file *.gm_defaults* is displayed. Notice that only the first line in the file, namely

```
Method to use: x.y.z
```

```
struct InstanceData{
    int* model;
    int* image;
    double *error;
    int dim;
    int modelNr;
    int modelKey;
    double totalError;
    int numberOfSubstitutions;
};
```

Figure B.5: Return type of the member function `getInstance`

is relevant. The code number `x.y.z` identifies exactly the method that is to be used for graph matching. The subsequent explanations in the file provide information to the application programmer concerning the different methods and their identifying code number. In Fig. B.6 the code number `3.4.6` refers to the decision tree method and specifies that the resulting decision trees should be breadth-pruned and also pruned at depth 6. While for some applications it is sufficient to set the graph matching method only once by simply editing the file *.gm_defaults*, other applications require that the graph matching method can be changed during the run time of a program. For this purpose, the member function `setMethod(x,y,z)` has been added to the public interface of class `KB` (see Fig. B.4). The parameters of this function correspond to the individual components of the code numbers in the file *.gm_defaults*. I.e., calling `setMethod(1,3,0)` sets the active method to the A^* algorithm with future cost estimation while calling `setMethod(2,2,0)` activates the decomposition-based method for error-correcting subgraph isomorphism detection.

The graph matching methods that are provided by the toolkit *GUB* can be applied to a variety of different graphs. However, not all of the methods are equally suited to solve a specific problem. In the following, the properties, advantages and disadvantages of each method are briefly described. Based on this overview, it will be possible for application programmers to choose the method that best serves their purposes. For detailed explanations of the algorithms please refer to the Chapters 2, 3, 4 and Appendix A. Notice that the numbers in brackets preceding the name of the method correspond to the code numbers in the file *.gm_defaults*.

[1.1] Ullman's algorithm

properties: finds exact graph and subgraph isomorphisms between two graphs

advantages: performs well for unlabeled graphs

disadvantages: performance is linearly dependent on the number of model graphs

```
Method to use: 3.4.6

Explanations:
1: Tree Search
   1.1  Ullman
   1.2  A*
   1.3  A* with lookahead

2: Network based algorithm
   2.1  exact
   2.2  error-correcting

3: Polynomial Decision Tree Graph Matcher
   3.1    complete decision tree (subgraph and graph isomorphism)
   3.2.k  k-depth pruned complete decision tree
   3.3    breadth pruned decision tree (multiple model graphs)
   3.4.k  both breadth- and k-depth pruned decision tree
   3.5    breadth-pruned (single model)
   3.6.k  both breadth- and k-depth pruned decision tree
```

Figure B.6: Example for the syntax of the file *.gm_defaults*. The method specified here is the polynomial decision tree matcher where the underlying trees are breadth-pruned and also depth-pruned at depth 6.

[1.2] A*

properties: finds error-correcting graph and subgraph isomorphisms between two graphs

advantages: performs well for small graphs (up to 10 vertices) and in situations where mostly structural distortions occur (missing vertices or edges in the input graph)

disadvantages: performance is linearly dependent on the number of model graphs, performs badly for large graphs

[1.3] A* with future cost estimation

properties: finds error-correcting graph and subgraph isomorphisms between two graphs

advantages: performs well for moderately sized graphs (10 to 30 vertices)

disadvantages: performance is linearly dependent on the number of model graphs

[2.1] exact decomposition method

properties: finds exact graph and subgraph isomorphisms from a set of model graphs to an input graph advantages: sublinearly dependent on the number of model graphs, good performance for large, labeled graphs (up to 500 vertices and more than 5 different labels)

disadvantages: performs badly for unlabeled, highly connected graphs

[2.2] error-correcting decomposition method

properties: finds error-correcting graph and subgraph isomorphisms from a set of model graphs to an input graph

advantages: sublinearly dependent on the number of model graphs, good performance for large, labeled graphs (up to 100 vertices and more than 5 different labels) and also for situations where label and attribute distortions are more frequent than structural distortions (missing vertices and edges)

disadvantages: performs badly for unlabeled, highly connected graphs and in situations where structural distortions prevail

[3.1] complete decision tree method

properties: finds exact graph and subgraph isomorphisms from an input graph to a set of model graphs

advantages: performance is independent of the number and the connectivity of the model graphs, furthermore it is guaranteed to be only quadratic in the number of vertices of the input graph

disadvantages: the size of the decision tree that is compiled for the model graphs is exponential in the number of vertices of the model graphs, hence, only small model graphs (up to 19 vertices) can be handled

[3.2.k] k-depth pruned decision tree method

properties: finds exact graph and subgraph isomorphisms from an input graph to a set of model graphs in combination with Ullman's algorithm

advantages: performance is only sublinearly dependent on the number of model graphs; the size of the decision tree can be reduced by pruning its depth

disadvantages: performance decreases when the depth of the decision is pruned

[3.3] breadth-pruned decision tree method

properties: finds exact graph isomorphisms between an input graph and a set of model graphs advantages: performance is independent of the number and the connectivity of the model graphs, furthermore it is guaranteed to be only polynomial in the number of vertices of the input graph; the underlying decision tree is smaller than the complete decision tree

disadvantages: the size of the decision tree is exponential in the number of vertices of the model graphs, hence, only small model graphs (up to 19 vertices) can be handled

[3.4.k] breadth- and k-depth pruned decision tree method

properties: finds exact graph isomorphisms between an input graph and a set of model graphs in combination with Ullman's algorithm

advantages: performance is only sublinearly dependent on the number of model graphs; the size of the decision tree can be reduced by pruning its depth

disadvantages: performance decreases when the depth of the decision is pruned

[3.5] breadth-pruned decision tree method (single model)

properties: finds exact graph isomorphisms between an input graph and a model graph

advantages: performance is identical to [3.2], however, the construction of the decision tree is more efficient, thus, model graphs with up to 22 vertices can be handled

disadvantages: the decision tree can only represent one model graph at a time

[3.6.k] breadth- and k-depth pruned decision tree method (single model)

properties: finds exact graph isomorphisms between an input graph and a model graph in combination with Ullman's algorithm

advantages: performance is identical to [3.4.k], however, the construction of the decision tree is more efficient, thus, model graphs with up to 22 vertices can be handled

disadvantages: the decision tree can only represent one model graph at a time

B.4 Installation of The Toolkit *GUB*

The toolkit *GUB* can be retrieved from the anonymous FTP-server of the University of Berne at `iamftp.unibe.ch`. Access to the server is established with

```
aftp iamftp.unibe.ch
```

Next, change into the directory `/pub/incoming` and transfer the compressed toolkit package `GUB_tookit.tar.Z` to your system with

```
get GUB_tookit.tar.Z
```

In order to unpack the toolkit, create a new subdirectory, move the compressed tar file into this directory, and extract the contents of the uncompressed tar file with

```
uncompress GUB_toolkit.tar.Z
tar -xvf GUB_toolkit.tar
```

The current directory will then contain the following subdirectories and files:

sources files
The sources files of the toolkit will be placed in the `src` subdirectory

include files
The header files of the toolkit will be placed in the current directory

makefiles
The makefiles `Make_Demo`, `Make_Client` and `Make_GM_Server` will be placed in the current directory

parameter files
The parameter files `.gm_default_attributes` and `.gm_defaults` will be placed in the current directory. These two files should be copied into your home directory. If this is not possible then an environment variable `GM_HOME` must be defined and set to the current directory, i.e. `setenv GM_HOME ./` .

object files directory
An empty subdirectory `lib` will be created

sample graph files
A number of examples for ASCII files defining attributed graphs will be placed in the current directory

The toolkit includes two demonstration programs which can be used as templates for the development of domain specific applications. The first demonstration program is created with the makefile `Make_Demo`. It defines at run time two simple attributed graphs based on the attribute definitions in the file `.gm_default_attributes` and then searches for graph matches with the method that is specified in the file `.gm_defaults`. The second demonstration program is based on the client/server scheme and consists of two components. The server component `GMSrv` is created with the makefile `Make_GM_Server` and the client component `matcher` is created with the makefile `Make_Client`. The purpose of the server is to maintain an instance of the database class `KB` and accept requests from its clients. For example, the clients may request to add a new model graph to the database or to match an input graph with the graphs in the database.

Bibliography

[AD93] H.A. Almohamad and S.O. Duffuaa. A linear programming approach for the weighted graph matching problem. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI*, 5:522–525, 1993.

[ALC+93] J.F Arias, C.P. Lai, S. Chandran, R. Kasturi, and A. Chhabra. Interpretation of telephone system manhole drawings. *Proc. 2nd ICDAR,Tsukuba, Japan*, pages 365–368, 1993.

[BA83] H. Bunke and G. Allerman. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters 1*, 4:245–253, 1983.

[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.

[BC79] K.S. Booth and C.J. Colbourn. Problems polynomially equivalent to graph isomorphism. Technical Report CS-77-04, University of Waterloo, 1979.

[Bge94] B. Bhanu and T. Poggio (guest eds.). Special Section on Learning in Computer Vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI*, 16(9):865–919, 1994.

[BHS89] D.E. Brown, C.L. Huntley, and A.R. Spillane. A parallel genetic heuristic for the quadratic assignment problem. In J.D. Schaffer, editor, *Genetic Algorithms*, pages 406–415. Morgan Kaufmann, 1989.

[BL88] S. Bradtke and W.G. Lehnert. Some experiments with case-based search. In *AAAI-88: The Seventh National Conference on Artificial Intelligence*, volume 1, pages 133–138, 1988.

[Bla94] R.E. Blake. Partitioning graph matching with constraints. *Pattern Recognition*, 27(3):439–446, 1994.

[BM88] B. Bhanu and J.C. Ming. TRIPLE: A multi-strategy machine learning approach to target recognition. In *Image Understanding Workshop*, pages 537–547. Morgan Kaufmann Publishers, 1988.

[BM95] H. Bunke and B.T. Messmer. Efficient attributed graph matching and its application to image analysis. In C. Braccini, L. DeFloriani, and G. Vernazza, editors, *Lecture Notes in Computer Science 974: Image Analysis and Processing*, pages 45–55. Springer Verlag, 1995.

[BR92] J.B. Burns and E.M. Riseman. Matching complex images to multiple 3D objects using view description networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 328–334, 1992.

[Bun82] H. Bunke. Attributed programmed graph grammars and their application to schematic diagram interpretation. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI*, 4(6):574–582, 1982.

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

[CH81] J.K. Cheng and T.S. Huang. Image recognition by matching relational structures. In *IEEE Conf. on Pattern Recognition and Image Processing*, pages 542–547, 1981.

[CH94] D.J. Cook and L.B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, pages 231–255, 1994.

[Chr75] N. Christofides. *Graph Theory: An Algorithmic Approach.* Academic Press, 1975.

[CK92] C.J. Cho and J.J. Kim. Recognizing 3-D objects by forward checking constrained tree search. *Pattern Recognition Letters*, 13(8):587–597, 1992.

[CKP95] W.J. Christmas, J. Kittler, and M. Petrou. Structural matching in computer vision using probabilistic relaxation. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI*, 17(8):749–764, 1995.

[CS95] M.S. Costa and L.G. Shapiro. Analysis of scenes containing multiple non-polyhedral 3d objects. In C. Braccini, L. DeFloriani, and G. Vernazza, editors, *Lectures Notes in Computer Science 974: Image analysis and processing.* Springer Verlag, 1995.

[Deo74] N. Deo. *Graph theory with applications to engineering and computer science*, page 311. Prentice-Hall, 1974.

[DIN80] *DIN Zeichnungswesen 1*, pages 269–286. Beuth, 1980.

[EF84] M.A. Eshera and K.S. Fu. A graph distance measure for image analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, 14(3):398–408, May 1984.

[Ehr92] H. Ehrig. Introduction to graph grammers with applications to semantic networks. *Computers and Mathematics with Applications*, 23:557–572, September 1992.

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

[Eve93] B. S. Everitt. *Cluster analysis*. Edward Arnold, third edition, 1993.

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

[Fis90] D. H. Fisher. Knowledge acquisition via incremental conceptual clustering. In J.W. Shavlik and T. G. Dietterich, editors, *Readings in Machine Learning*, pages 267–283. Morgan Kaufmann, 1990.

[FLD94] J. Feng, M. Laumy, and M. Dhome. Inexact matching using neural networks. In E.S. Gelsema and L.N. Kanal, editors, *Pattern Recognition in Practice IV: Multiple Paradigms, Comparative Studies and Hybrid Systems*, pages 177–184. North-Holland, 1994.

[For82] C.L. Forgy. Rete, a fast algorithm for the many pattern / many object pattern match problem. In *Artificial Intelligence*, volume 19, pages 17 – 37. Elvesier, 1982.

[FZ91] G.P. Ford and J. Zhang. A structural graph matching approach to image understanding. *SPIE Intelligent Robots and Computer Vision X: Algorithms and Techniques*, 1607:559–569, 1991.

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

[GB90] E. Gmuer and H. Bunke. 3-D object recognition based on subgraph matching in polynomial time. In R. Mohr, T. Pavlidis, and A. Sanfeliu, editors, *Structural Pattern Analysis*, pages 131–147. World Scientific, 1990.

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

[Ham93] A.H. Hamada. A new system for the analysis of schematic diagrams. *Proc. 2nd ICDAR, Tsukuba, Japan*, pages 369–372, 1993.

[HE80] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. In *Artificial Intelligence*, volume 14, pages 263–313. Elvesier, 1980.

[HHVN90] L. Herault, R. Horaud, F. Veillon, and J.J. Niez. Symbolic image matching by simulated annealing. In *Proc. British Machine Vision Conference*, pages 319–324. Oxford, 1990.

[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.

[HS89] R. Horaud and T. Skordas. Stereo correspondence through feature grouping and maximal cliques. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI*, 11(11):1168–1180, 1989.

[Hut94] A. Hutchinson. *Algorithmic Learning.* Oxford University Press, 1994.

[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.

[JS89] K.A. De Jong and W. M. Spears. Using genetic algorithms to solve NP-complete problems. In J.D. Schaffer, editor, *Genetic Algorithms*, pages 124–132. Morgan Kaufmann, 1989.

[KCP92] J. Kittler, W. J. Christmas, and M. Petrou. Probabilistic relaxation for matching of symbolic structures. In H. Bunke, editor, *Advances in Structural and Syntactic Pattern Recognition*, pages 471–480. World Scientific, 1992.

[KK91] W.Y. Kim and A.C. Kak. 3-D object recognition using bipartite matching embedded in discrete relaxation. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI*, 13:224–251, 1991.

[KK94] S.H. Kim and J.H. Kim. Automatic input of logic diagrams by recognizing loop-symbols and rectilinear connections. *Int. Journal of Pattern Recognition*, 8(5):1113–1129, 1994.

[KM94] Y. Kodratoff and S. Moscatelli. Machine learning for object recognition and scene analysis. *Int. Journal of Pattern Recognition and Artificial Intelligence*, 8(1):259–305, 1994.

[KR79] L. Kitchen and A. Rosenfeld. Dicrete relaxation for matching relational structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(12):869–874, 1979.

[KS89] M. A. Kelly and R. E. Seviora. An evaluation of DRETE on CUPID for OPS5 matching. In *11'th International Joint Conference on Artificial Intelligence*, volume 1, pages 84–90, 1989.

[KU88] P. Kuner and B. Ueberreiter. Pattern recognition by graph matching - combinatorial versus continous optimization. *International Journal of Pattern Recognition and Artificial Intelligence*, 2(3):527–542, 1988.

[Lev72] G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo 9*, pages 341–354, 1972.

[Lev92] R. Levinson. Pattern associativity and the retrieval of semantic networks. *Computers and Mathematics with Applications*, 23:573–600, September 1992.

[LK89] S.W. Lee and J.H. Kim. Attributed stroke graph matching for seal imprint verification. *Pattern Recognition Letters*, 9:137–145, February 1989.

[LKG90] S.W. Lee, J.H. Kim, and F.C.A. Groen. Translation- rotation- and scale invariant recognition of hand-drawn symbols in schematic diagrams. *International Journal of Pattern Recognition and Artificial Intelligence*, 4(1):1–15, 1990.

[LRS91] S.W. Lu, Y. Ren, and C.Y. Suen. Hierarchical attributed graph representation and recognition of handwritten Chinese characters. *Pattern Recognition*, 24:617–632, 1991.

[LS92] H.S. Lee and M.I. Schor. Match algorithms for generalized Rete networks. *Artificial Intelligence*, pages 255–270, 1992.

[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.

[MB96] B.T. Messmer and H. Bunke. Automatic learning and recognition of graphical symbols in engineering drawings. In K. Tombre R. Kasturi, editor, *Graphics Recognition, Lecture Notes in Computer Science 1072*, pages 123–134. Springer Verlag, 1996.

[MF90] S. Mehta and L. Fulop. A neural algorithm to solve the graph matching problem. In *Proc. of Int. Neural Network Conference INNC*, volume 1, pages 262–266. Kluwer Academic Publishers, 1990.

[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.

[MRA92] R.S. Michalski, A. Rosenfeld, and Y. Aloimonos. Machine Vision and Learning: Research Issues and Directions. *Proc. NSF/ARPA Workshop on Machine Learning and Vision, Harpers Ferry, West Virigina*, 1992.

[MT94] R. Michalski and G. Tecuci. *Machine Learning: A Multistrategy Approach.* Volume IV. Morgan Kaufmann Publishers, 1994.

[Nil80] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, 1980.

[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.

[Pas94] B. Pasternak. Processing imprecise and structural distorted line drawings by an adaptable drawing interpretation kernel. In A. Dengel and L. Spitz, editors, *Proc. of IAPR Workshop on Document Analysis Systems*, pages 349–366, 1994.

[PCB94] A. Pearce, T. Caelli, and W.F. Bischof. Rulegraphs for graph matching in pattern recognition. *Pattern Recognition*, 27(9):1231–1246, 1994.

[Poo93] J. Poole. Similarity in legal case based reasoning as degree of matching in conceptual graphs. In MM. Richter, S.Wess, K.-D. Althoff, and F. Maurer, editors, *Preproceedings: First European Workshop on Case-Based Reasoning*, pages 54–58, 1993.

[RB79] D.H. Rouvray and A.T. Balaban. Chemical applications of graph theory. In R.J. Wilson and L.W. Beineke, editors, *Applications of Graph Theory*, pages 177–221. Academic Press, 1979.

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

[Rea79] R. C. Read. Algorithms in graph theory. In R. Wilson and L. Beineke, editors, *Applications of graph theory.* Academic Press, 1979.

[SB95] K. Sengupta and K.L. Boyer. Organizing large structural modelbases. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI*, 17(4), 1995.

[SD90] J.W. Shavlick and T.G. Dietterich, editors. *Readings in Machine Learning.* Morgan Kaufman, San Mateo, 1990.

[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.

[SF83] A. Sanfeliu and K.S. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:353–363, 1983.

[SH81] L.G. Shapiro and R.M. Haralick. Structural descriptions and inexact matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI*, 3:504–519, 1981.

[SH82] L.G. Shapiro and R.M. Haralick. Organization of relational models for scene analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI*, pages 595–602, 1982.

[SH92] H. Sossa and R. Horaud. Model indexing: The graph-hashing approach. In *Proceedings of IEEE Conf. on Computer Vision and Pattern Recognition*, pages 811–814, 1992.

[Sha93] L. Shapiro. Relational matching. In T.Y. Young, editor, *Handbook of Pattern Recognition and Image Processing: Computer Vision*, pages 475–496. Academic Press, 1993.

[SKP93] D.S. Seong, H.S. Kim, and K.H. Park. Incremental clustering of attributed graphs. *IEEE Transactions on System, Man, and Cybernetics*, 23(5):1399–1411, 1993.

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

[SW94] S.R. Schwartz and B.W. Wah. Machine learning of computer vision algorithms. In T.Y. Young, editor, *Handbook of Pattern Recognition and Image Processing*, pages 319–359. Academic Press, 1994.

[TF79] W.H. Tsai and K.S Fu. Error-correcting isomorphisms of attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 9:757–768, 1979.

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

[Ume88] S. Umeyama. An eigendecomposition approach to weighted graph matching problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI*, 10(5):695–703, 1988.

[WF74] R.A. Wagner and M.J. Fischer. The string-to-string correction problem. *Journal of the Association for Computing Machinery*, 21(1):168–173, 1974.

[WLR89] A.W. Wong, S.W. Lu, and M. Rioux. Recognition and shape synthesis of 3-d objects based on attributed graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI*, 11(3):279–290, 1989.

[Won90] E. K. Wong. Three-dimensional object recognition by attributed graphs. In H. Bunke and A. Sanfeliu, editors, *Syntactic and Structural Pattern Recognition- Theory and Applications*, pages 381–414. World Scientific, 1990.

[Won92] E.K. Wong. Model matching in robot vision by subgraph isomorphism. *Pattern Recognition*, 25(3):287–304, 1992.

[WYC90] A.K. Wong, M. You, and A.C. Chan. An algorithm for graph optimal monomorphism. *IEEE Transactions on Systems, Man and Cybernetics*, 20(3):628–636, 1990.

[XO90] L. Xu and E. Oja. Improved simulated annealing, Boltzmann machine, and attributed graph matching. In L. Almeida, editor, *Lecture Notes in Computer Science 412*, pages 151–161. Springer Verlag, 1990.