A network based approach to exact and inexact graph matching

B.T. Messmer, H. Bunke
Institut für Informatik und angewandte Mathematik
Universität Bern

September 20, 1993

Abstract

In this paper a new approach to exact and inexact graph matching is introduced. We propose a compact network representation for graphs, which is capable of sharing identical subgraphs of one or several model graphs. The new matching algorithm NA works on the network and uses its compactness in order to speed up the detection process. Next, the problem of inexact graph matching is described and a distance measure based on basic graph edit operations and subgraph isomorphism is defined. We propose an inexact network algorithm INA which determines the optimal distance between an input graph and a set of model graphs along with the corresponding subgraph isomorphism. In addition to INA, a new lookahead procedure is proposed. The lookahead procedure works simultaneously over a set of model graphs and efficiently limits the size of the search space. The advantages of the new methods are studied in a theoretical complexity analysis. Finally, some experimental results with randomly generated graphs and a traditional graph matching algorithm for comparision confirm the superiority of the new methods in practice.

CR Categories and Subject Descriptors: F.2 Analysis of Algorithms and Problem Complexity; I.2.4 [Artificial Intelligence]: Knowledge representation formalisms and methods; I.2.8 [Artificial Intelligence]: Problem solving, Control Methods and Search; I.5 [Artificial Intelligence]: Pattern Recognition
General Terms: Algorithms
Additional Keywords: Graphs, subgraph isomorphism, RETE

Contents

1 Introduction

In the past few years the problem of classifying and recognizing complex structures such as visual objects or high level concepts has become more and more important. By choosing attributed relational graphs for the representation of these structures, graph matching can be used to solve the classification and recognition problem [EF86, LK89, LKG90, KU88, LRS91]. Graph matching is a very powerful technique and has been applied in the field of computer vision for some time. Other domains such as case based reasoning and machine learning received considerable impulses from the application of graph matching to certain problems.

The main drawback of graph matching stems from the fact that it belongs to the class of NP-complete problem. Therefore, the method may grow out of computational tractability in the worst case. Different approaches to graph matching have been proposed over the years. The works of Ullman, and Shapiro/Haralick mainly concentrate on the traditional backtracking tree search approach to solve the exact graph matching problem[Ull76, SH81]. To speed up the search, heuristic techniques such as forward checking, lookahead and backmarking are introduced[HE80]. Tree search methods are optimal in terms of the solutions they find, but they are prone to combinatorial explosion in time and in space. It is known that NP-complete problems can be addressed by stochastic optimization algorithms with polynomial time complexity such as simulated annealing, genetic algorithms or continuous relaxation [Hae90, KU88, HS88, KK91, CH81]. However these methods do no longer guarantee to find an optimal solution.

In practice, most applications have to deal with noisy data and therefore cannot rely solely on exact matching. The problem of inexact graph matching can be defined in various ways and the search algorithms are strongly dependent on these definitions. Bunke/Allerman[BA83], Shapiro[SH81], and Tsai/Fu[TF79] propose search procedures closely related to the $A*$ algorithm. As in the exact case, this method is guaranteed to find an optimal solution, but its search space will explode exponentially in the worst case. Lately, linear programming techiques were applied to the weighted graph matching problem, which is a special case of the general inexact graph matching, and it was shown that the problem can be solved in polynomial time at the cost of loss of optimality[AD93].

Another problem that arises when using graph matching for classification is the fact that most methods can only treat two graphs at once, matching the one to the other. Consequently, in a system with multiple models each model graph is sequentially matched to the input graph. If the models share identical substructures, much redundant work is done by this approach. To avoid this kind of redundancy, we propose a preprocessing of the model database, which results in a network representation of the models. The network nodes represent subgraphs of the model graphs. The top layer of the network consists of nodes representing single vertices. Next, always two nodes are joined in a node representing the union of the subgraphs of its parent nodes. The bottom layer contains a node for each model in the network. Identical subgraphs of different models can be represented by the same network node. Thus we arrive at a compact representation of the model database, which is less redundant and can be used to speed up the matching process considerably. The matching algorithm which is designed to work with the network representation is based on the RETE algorithm[For82] and its extension to graphs as it was proposed in [BTT91, Mes92, BM93].

In this paper we propose two algorithms. The first algorithm, NA, is for exact, and the second algorithm INA for inexact graph matching. Both algorithms are optimal in terms of the solutions they find. Consequently, their worst case time complexity is exponential. However, in practice they have proven to be very efficient and they do not tend towards the worst case behavior as fast as the tree search procedures.

The rest of the paper is organized as follows: Section 2 is dedicated to the definition of graphs and matching methods on graphs, along with transformation operations that are used for inexact matching. The algorithm A^* is given for comparison purposes and adjusted to the graph matching problem. Section 3 analyses the influence of the cost functions on the matching process. Next, section 4 introduces the network based approach to graph matching. The algorithms for compilation and subgraph isomorphism detection are given. Also, an new lookahead procedure working on the network is introduced. The lookahead will allow a back propagation of future errors after a first pass on the input data has been completed. This results in a more efficient matching procedure. Section 5 explains the main features of the network approach and compares them to the tree search procedures. Section 6 reports some experimental results on exact as well as inexact graph matching. Finally, section 7 discusses the general applicability of the network approach and future work.

2 Definitions and notations

Graphs are generally used to describe objects by representing the parts of the objects by vertices and the relations between the parts by edges of the graph. To describe the properties of the parts and the relations, symbolic labels and numeric attributes are assigned to the vertices and edges.

Definition 2.1: A *directed attributed relational graph* is a 6-tupel

$$G = (V, E, L, A, \mu, \nu)$$

where

$V = \{v_1, v_2, \dots, v_n\}$ is the set of vertices,
$E = \{e_1, e_2, \dots, e_m\} \subseteq V \times V$ is the set of edges,
L is a finite set of symbolic labels,
A is a finite set of attribute values,
$\mu : V \mapsto L \times A^k$ is the vertex labelling function,
$\nu : E \mapsto L \times A^k$ is the edge labelling function.

□

The interpretation, i.e. meaning of the attributes is usually implied by the symbolic labels. For notational convenience, directed attributed relational graphs are simply referred to as *graphs* in the rest of the paper.

Example 2.2:

$$\begin{aligned}
G &= \{V, E, L, A, \mu, \nu\} \\
V &= \{v_1, v_2\} \\
E &= \{(v_1, v_2)\} \\
L &= \{Rect, Circle\} \\
A &= \mathcal{R}, k = 2 \\
\mu(v_1) &= (Rect, 2, 3) \\
\mu(v_2) &= (Circle, 5, _) \\
\nu((v_1, v_2)) &= (inside)
\end{aligned}$$

In this example, we represent two geometric figures, a rectangle and a circle by a graph. The edge label *inside* indicates that the rectangle is inside the circle. The label *Rect* implies that the first attribute is the length and the second attribute the height of the rectangle. The label *Circle* implies that the first attribute is the radius of the circle, while the second attribute is void. □

For simplicity reasons, we do not allow that the set E of edges contains several identical elements. Hence, multiple, unidirectional edges between two vertices are not considered in the rest of the paper. This, however is not a restriction of generality because multiple edges may be replaced by a single edge using a modified edge labelling function, which incorporates the labels and the attributes of the multiple edges in a new label. For each combination of multiple edges, such a label can be created.

Furthermore, we will be using the concept of subgraphs of a graph, which are specified by a subset of vertices:

Definition 2.3: A *induced subgraph* $S(V_1)$ of $G = (V, E, L, A, \mu, \nu)$ with $V_1 \subseteq V$ is given by

$$S(V_1) = (V_1, E \cap V_1 \times V_1, L, A, \mu, \nu)$$

□

When using graphs for the representation of real world objects, we are interested in comparing different graphs and finding corresponding vertices and edges. Depending on the application domain, we intend to find instances of one graph within another allowing additional edges (graph monomorphism), finding instances with no additional edges (subgraph isomorphism), or finding a one to one correspondence between vertices and edges (graph isomorphism). More formally, when comparing two graphs G and G', we are looking for a function $f : V \mapsto V'$ which maps each vertex $v \in V$ onto a vertex $v' \in V'$ such that certain conditions are fullfilled:

Definition 2.4: A function $f : V \mapsto V'$ is a *graph monomorphism* from G to G' if

1. $\mu(v) = \mu(f(v))$ for all $v \in V$.

2. $\nu(e) = \nu(e')$ for all edges $e = (v_1, v_j) \in E$ and $e' = (f(v_i), f(v_j)) \in E'$

□

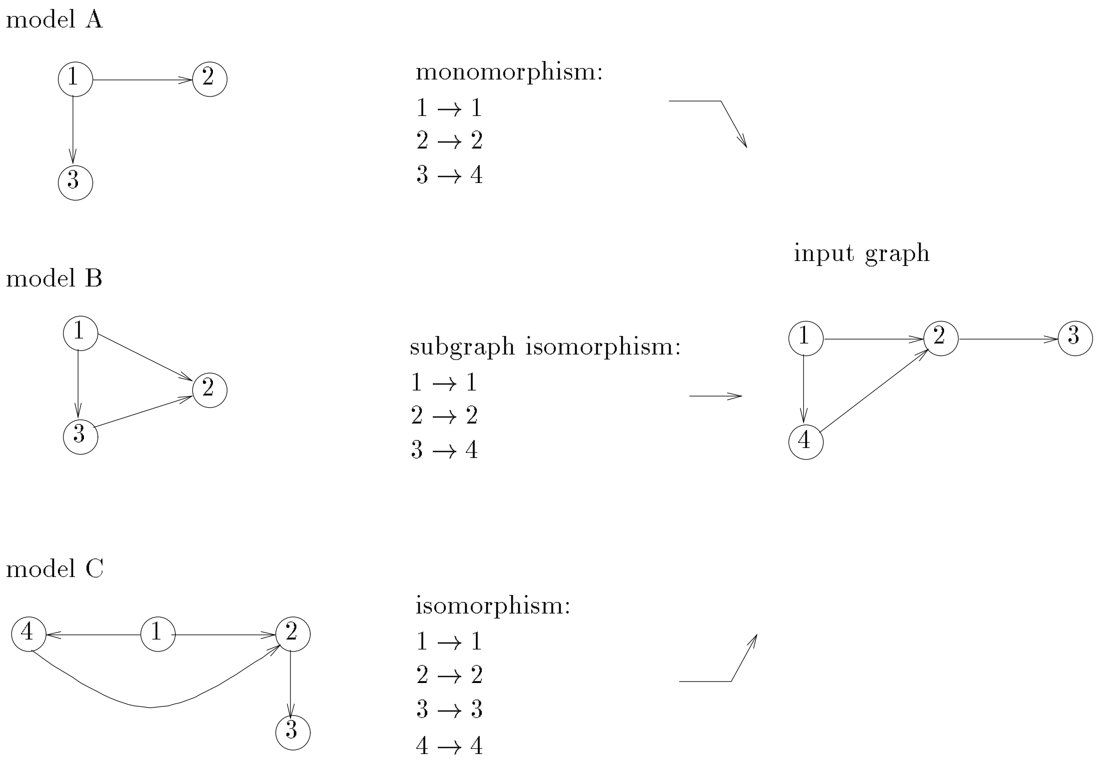


Figure 1: Three different types of morphisms

Definition 2.5: A function $f : V \mapsto V'$ is a *subgraph isomorphism* from G to G' if it is a graph monomorphism and additionally for any edge

$$e' = (v'_i, v'_j) \in E' \cap f(V) \times f(V)$$

exists an edge

$$e = (f^{-1}(v'_i), f^{-1}(v'_j)) \in E \text{ with } \nu(e') = \nu(e)$$

□

Definition 2.6: A function $f : V \mapsto V'$ is a *graph isomorphism* from G to G' if f is bijective and a subgraph isomorphism. □

Definition 2.7: Let G_1 and G_2 be two graphs with subgraphs G'_1 and G'_2, respectively. Then any isomorphism from G'_1 and G'_2 is called a *bidirectional subgraph isomorphism* from G_1 to G_2. □

Example 2.8: Fig. 1 illustrates the different types of morphism. There exists a monomorphism between model A and the input graph, but no subgraph isomorphism. Between model B and the input graph, there is a subgraph isomorphism, but no isomorphism, while model C and the input graph are isomorphic. □

In practice, there will often be a set of model graphs representing known objects and an input graph which was generated on the basis of real data. Data acquisition is often prone to errors, resulting in a distorted input graph. Therefore, none of the morphisms from the definitions 2.3, 2.4, 2.5 will hold between any of the models and the input graph. To model the error generation process, we consider three types of distortion:

1. label or attribute substitutions

2. missing vertices or edges

3. extraneous vertices or edges

In order to correct these errors, we propose to apply edit operations to the model graph until there exists a function f for the desired morphism between the edited model and the input graph. In this paper, we consider the following edit operations:

1. Substitution of a vertex or an edge. The label l and the attributes $a_1, \ldots, a_k$ of the vertex $v \in V$ are substituted by $o \in L$ and $b_1, \ldots, b_k \in A^k$. The definition for edge label substitution is analogous. Formally, a substitution is denoted by $((v, l, a_1, \ldots, a_k) \rightarrow (v, o, b_1, \ldots, b_k))$.

2. Deletion of a vertex.
   The deletion of a vertex v implies that all edges incident with v must be deleted, too. Formally, a vertex deletion is denoted by $(v \rightarrow \$)$.

3. Deletion of an edge.
   This edit operation is denoted by $(e \rightarrow \$)$.

4. Insertion of a vertex.
   This edit operation is denoted by $(\$ \rightarrow v)$.

5. Insertion of an edge.
   The insertion of edge can only be applied if the adjacent vertices exist already. Finally, an edge edit operation is denoted by $(\$ \rightarrow e = (v_i, v_j))$.

Apparently, these edit operations are powerful enough to model any combination of distortions that may occur. In order to account for the fact that certain types of distortion are more likely to occur than others, we introduce *costs* for our edit operations. Formally, $c(\delta)$ is the cost of the edit operation δ. For example, $c((v, l, a_1, a_2) \rightarrow (v, o, b_1, b_2))$ denotes the cost of a substitution, and $c(\$ \rightarrow e)$ denotes the cost of an edge insertion. The cost of each edit operation on a vertex or an edge depends on the label of the vertex or the edge. For example, the cost for the deletion $c(v \rightarrow \$)$ in fact depends on the label $\mu(v)$.

Definition 2.9: Let $\Delta = \{\delta_1, \delta_2, \ldots, \delta_l\}$ be a sequence of edit operations and G a graph. Then, $\Delta(G)$ denotes the graph resulting after $\delta_1, \delta_2 \ldots, \delta_l$ have been sequentially applied to G. No part of G must be subject to an edit operation more than once; e.g if δ_i affects the vertex $v \in V$ or the edge $e \in E$, then v or e must not be affected by any other $\delta_j, j \neq i$. Thus the order of the edit operations in Δ is irrelevant. Δ can be split up into

$$\Delta = \Delta_s + \Delta_d + \Delta_i$$

with Δ_s = sequence of substitutions, Δ_d = sequence of deletions and Δ_i = sequence of insertions. Furthermore,

$$C(\Delta) = \sum_{i=1}^{l} c(\delta_i)$$

is the total cost of Δ. □

Now we can define a distance measure between to arbitrary graphs G and G' based on the sequence Δ of edit operations that are necessary in order to establish a graph monomorphism, subgraph isomorphism or graph isomorphism between $\Delta(G)$ and G'.

Definition 2.10: Let G and G' be graphs and $\Delta = (\delta_1, \ldots, \delta_l)$ be a sequence of edit operations that transform G into $\Delta(G)$. We define three distance measures,

1. $dist_m(G, G') = MIN_\Delta\{C(\Delta)|$ there exists a monomorphism f between $\Delta(G)$ and $G'\}$
2. $dist_s(G, G') = MIN_\Delta\{C(\Delta)|$ there exists a subgraph isomorphism f between $\Delta(G)$ and $G'\}$
3. $dist_i(G, G') = MIN_\Delta\{C(\Delta)|$ there exists an isomorphism f between $\Delta(G)$ and $G'\}$

□

For two given graphs, the values of these distance measures are strongly dependent on the costs that are assigned to each edit operation. How these costs influence the distance between two graphs will be studied in the next section of this paper.

When calculating the distance between two graphs algorithmically, the edit operations and the matching function f will not be determined separately, but the edit operations will be performed while the function f is being searched. In the remainder of this paper, we will consider algorithms which detect *exact* morphisms according to the definitions 2.3, 2.4, 2.5 and *inexact* morphisms, which imply a sequence of edit operations:

Figure 2: A model graph G and a distorted input graph G'

Definition 2.11: Let G, G' and $\Delta = (\delta_1, \delta_2, \ldots, \delta_l)$ be the same as in Def. 2.6. Furthermore, let

1. $\epsilon = dist_m(G, G')$ and f is the corresponding monomorphism. Then f is called the optimal inexact monomorphisms between G and G', or ϵ-monomorphism for short.

2. $\epsilon = dist_s(G, G')$ and f the corresponding subgraph isomorphism. Then f is called the optimal inexact subgraph isomorphisms, or ϵ-subgraph isomorphism for short.

3. $\epsilon = dist_i(G, G')$ and f the corresponding isomorphism f. Then f is called the optimal inexact isomorphisms, or ϵ-isomorphism for short.

□

Note that if $\epsilon = 0$, then the sequence of edit operations is empty and we have an exact morphism between G and G'.

Example 2.12: An inexact subgraph isomorphism from graph G to graph G' as they are given in Fig. 2 exists if the following sequence of edit operations is applied to G:

$$\Delta = \{(1 \rightarrow \$), ((1,2) \rightarrow \$), (\$ \rightarrow (4,3))\}$$

If we assume that all vertices and edges are equally labelled and insertion and deletion costs do not differ, then Δ is one of four possible edit sequences with minimal costs, such that a subgraph isomorphism exists. □

For comparison reasons, we now give a short description of the standard procedure for finding an ϵ-subgraph isomorphism. Mono- and isomorphisms can be found using the same algorithm. The basic idea is to adapt the A^* algorithm[Nil80] and consider the subgraph isomorphism problem as a version of the consistent labelling problem. Assume that $(v_1, \ldots, v_n)$ is a fixed order of the vertices of the model graph $G = (V, E)$ and let $G' = (V', E')$ be the input graph. The dummy vertex \$ is added to the input graph G' to account for the fact that G may contain more vertices than G'. We start with assigning v_1 to each of the input vertices v_i', forming the first level of the search tree. If we are looking for an exact subgraph isomorphism, only pairs (v_1, v_i') with $\mu(v_1) = \mu(v_i')$ are considered. In the inexact case, each pair (v_1, v_i') gets assigned the cost of the edit operation that transforms the label of v_1 into the label of v_i'. If $v_i' = \$$ then v_1 is deleted from the model. After the first level of the search tree has been constructed, the search tree node (v_1, v_i') with the least cost is expanded and all pairs (v_2, v_j') with $v_j' \in V' - \{v_i'\}$ are generated.

Now, in addition to the labels of the vertices we must consider all edges between v_1 and v_2. Again, in the exact case, a search tree node (v_2, v'_j) is only considered, if all edges $e = (v_1, v_2)$ are identical to edges $e' = (v'_i, v'_j)$. In the inexact case, the costs for the edge transformations are added to the label costs and the costs of the parent node. This continues until the search tree contains a path from (v_1, v'_i) to (v_n, v'_k). Thus an exact, or an inexact optimal monomorphism between G and G' is found.

In the exact case it is possible to cut off certain branches in the search tree by applying a lookahead procedure [Ull76]. In the experiments described in section 6, this lookahead procedure was used in order to speed up the search. In the inexact case, however, we did not implement any future error costs estimation as it was discussed, for example, in [BA83]. The graph matching algorithm uses the following data structures:

- OPEN: a linear list with elements $(node, level, error)$, sorted in increasing order according to the third argument $error$ (costs). Insertions to OPEN are denoted by OPEN $\leftarrow (node, level, error)$.
- Model: array of vertices.
- Partial_Match: array of pairs (v, v') with $v \in V$ and $v' \in V'$.
- edgeError(Partial_Match,v, v'): cost of edge transformations that are necessary in order to satisfy the edge constraints between the subgraph of the Partial_Match and the vertices of the new pair (v, v').

```
procedure Graphmatch_A*(G, G') :
  /* OPEN is initialized with the symbolic node root */
  OPEN ←(root, 0, 0)
  Model ←array of model vertices indexed by level
  Threshold ←MAXIMUM_ERROR
  While OPEN not empty
       (node, level, error) ←OPEN.removeTop()
       If error >Threshold Then  EXIT
       Partial_Match=vertices used in the levels 1 through level
       level = level + 1
       If level > model_size Then
          /* Matching with costs error found, reset Threshold */
          Threshold = error
          output ←Partial_Match , error
       Else
          /* Try all possible substitutions from input vertices to model vertices */
          For all input vertices v
             If v ∉ Partial_Match Then
                new_error = error
                create new tree node New_Node
                new_error = new_error + c(Model[level] → v) /* Substitution */
                new_error = new_error + edgeError(Partial_Match, Model[level], v)
                OPEN ←(New_Node, level, error)
          End For
          /* Delete the model vertex at position level */
          create New_Node
          new_error = error
          new_error = new_error + c(Model[level] → $)
          new_error = new_error + edgeError(Partial_Match,Model[level], $)
          OPEN ←(New_Node, level, new_error)
       End if
  End While
```

3 Graph distance properties

The graph distance measures introduced in section 2 depend on the cost functions that are assigned to the edit operations. The definition of the cost functions for a specific application allows the application programmer to introduce heuristic knowledge into the matching process. For example, knowledge about likely and unlikely deformations may result in the assignment of low or high edit costs, respectively. In the past, the proposed cost functions were always closely linked to some application[BA83, SF83] and therefore we consider it necessary to state some general assumptions which should hold for any cost function definition. Furthermore, we will examine the relationship of the distance measures among each other and the interpretation of the distance measure values.

First, we assume without loss of generality that

$$0 \leq c(\delta) \leq \infty$$

for any edit operation δ. Negative values of c are neither allowed, nor is there a meaningful interpretation. We set the costs for an identical substitution $(v, l, a_1, \ldots, a_k) \rightarrow (v, l, a_1, \ldots, a_k)$ to zero, while any substitution between non identical labels costs more than zero. Edit operation δ which are considered impossible are given infinite costs:

$$c(\delta) = \infty.$$

Next, we assume that the following inequality holds for all substitutions:

$$\begin{aligned} c((l, a_1, \ldots, a_k) \rightarrow (o, b_1, \ldots, b_k)) \leq \quad & c((l, a_1, \ldots, a_k) \rightarrow (p, c_1, \ldots, c_k)) \\ & + c((p, c_1, \ldots, c_k) \rightarrow (o, b_1, \ldots, b_k)) \end{aligned}$$

This is a necessary assumption because otherwise any algorithm considering each part of a model graph only once will not find the minimal distance between this graph and a given input graph.

It is important to note that usually the distance measures are not symmetric, i.e.

$$dist(G, G') \neq dist(G', G)$$

for all distances $dist_i, dist_s$ and $dist_m$. Also, other metric properties, namely

$$dist(G, G') = 0 \text{ if and only if } G = G' \text{ and}$$

$$dist(G, G') \leq dist(G, G'') + dist(G'', G')$$

are usually not valid for our graph distance measure.

Based on the definition of the distance measures and the above assumptions we now introduce some theorems about the relationships between the values of the distance measures and the underlying morphisms. The following two theorems show that under certain cost functions, the computation of some of our graph distances implies the existence of a graph monomorphism, subgraph isomorphism, or graph isomorphism. The proofs are rather straightforward and are omitted. In Theorem 3.1 we start with a given morphism between two graphs and draw conclusions on the value of the distance measures.

Theorem 3.1: Let G and G' be two graphs.

1. If there exists an isomorphism between G and G' then

$$dist_i(G, G') = dist_i(G', G) = 0$$

$$dist_s(G, G') = dist_s(G', G) = 0$$

$$dist_m(G, G') = dist_m(G', G) = 0$$

2. If there exists a subgraph isomorphism from G to G', then

   (a)
$$dist_s(G, G') = 0$$
$$dist_m(G, G') = 0$$

   (b) If furthermore $c(\$ \rightarrow e) = 0$ and $c(\$ \rightarrow v) = 0$ for any vertex v and any edge e, then

$$dist_i(G, G') = 0$$

3. If there exists a bidirectional subgraph isomorphism from G to G' and if $c(v \to \$) = c(e \to \$) = 0$ for any vertex v and any edge e, then

$$dist_s(G, G') = dist_s(G', G) = 0$$

$$dist_m(G, G') = dist_m(G', G) = 0$$

4. If there exists a monomorphism from G to G', then

   (a) $$dist_m(G, G') = 0$$

   (b) If furthermore $c(\$ \to e) = 0$ for any edge, then

   $$dist_s(G, G') = 0$$

   (c) If furthermore $c(\$ \to v)$ for any vertex v, then

   $$dist_i(G, G') = 0$$

□

In Theorem 3.2 we assume that the distance measures have been calculated and determine under what circumstances the existence of a morphism can be concluded.

Theorem 3.2: Let G and G' be two graphs, and let the cost of any node or edge substitution be greater than zero.

1. If $dist_i(G, G') = 0$ and $c(\$ \to v), c(\$ \to e), c(v \to \$), c(e \to \$) > 0$ for any edge e and any vertex v then there exists an isomorphism from G to G'.
2. If $dist_i(G, G') = 0$ and $c(\$ \to e), c(v \to \$), c(e \to \$) > 0$ for any edge e and any vertex v then there exists a subgraph isomorphism from G to G'.
3. If $dist_i(G, G') = 0$ and $c(\$ \to v), c(\$ \to e), c(e \to \$) > 0$ for any edge e and any vertex v then there exists a subgraph isomorphism from G' to G.
4. If $dist_i(G, G') = 0$ and $c(v \to \$), c(e \to \$) > 0$ for any edge e and any vertex v then there exists a monomorphism from G to G'.
5. If $dist_i(G, G') = 0$ and $c(\$ \to v), c(\$ \to e) > 0$ for any edge e and any vertex v then there exists a monomorphism from G' to G.
6. if $dist_s(G, G') = 0$ and $c(\$ \to e), c(e \to \$) > 0$ then there exists a bidirectional subgraph isomorphism from G to G'
7. If $dist_s(G, G') = 0$ and $c(v \to \$), c(e \to \$), c(\$ \to e) > 0$ then there exists a subgraph isomorphism from G to G'
8. If $dist_s(G, G') = 0$ and $c(v \to \$), c(e \to \$) > 0$ then there exists a monomorphism from G to G'

9. If $dist_m(G, G') = 0$ and $c(v \to \$), c(e \to \$) > 0$ then there exists a monomorphism from G to G'

$\square$

Theorem 3.1 and 3.2 state some of the conclusions that can be drawn, when either a morphisms between G and G' exists or some of our distance measures between G and G' are equal to zero.

In practice, we are interested in the power of algorithms which solve a specific morphism problem. Assume that there is an algorithm M for the monomorphism problem, an algorithm SG for the subgraph isomorphism problem and an algorithm I for the isomorphism problem.

Theorem 3.3: Let G and G' be two graphs. If $dist'_s(G, G')$ is the distance calculated by SG under the condition that $c(\$ \to e) = 0$ and if $dist'_m(G, G')$ is the distance calculated by M under no special condition then

$$dist'_s(G, G') = dist'_m(G, G')$$

$\square$

In other words, by setting the costs for the insertion of edges to zero, we can use SG to determine the ϵ-monomorphism between G and G'. Unfortunately, such a relationship does not exist between I and SG. No setting of the cost function can ensure that I finds the optimal inexact subgraph isomorphism f_s, because edges of G' which belong to the image $f_s(G)$ of G are treated differently from edges which don't belong to the image of G.

However, it is possible to use SG in order to determine an optimal inexact isomorphism. Any algorithm SG which is applied to the graphs G and G' will determine the minimal distance $dist_s(G, G')$ and output the sequence Δ of edit operations and the subgraph isomorphism f corresponding to $dist_s(G, G')$. SG can be modified such that not only the *optimal* inexact subgraph isomorphism denoted by the pair $\{\Delta^0, f^0\}$, but *all* inexact subgraph isomorphisms $\{\Delta^k, f^k\}$ between G and G' are found. These inexact subgraph isomorphisms are generated in the order of the ascending costs $C(\Delta^k)$, starting with the optimal inexact subgraph isomorphism:

$$\begin{array}{ccccccc} \{\Delta^0, f^0\} & , & \{\Delta^1, f^1\} & , & \{\Delta^2, f^2\} & , & \dots \\ C(\Delta^0) & \leq & C(\Delta^1) & \leq & C(\Delta^2) & \leq & \dots \end{array}$$

If we now use SG to determine the ϵ-isomorphism, we need to extend the sequences Δ^k by adding the insertion operations for all vertices and edges of G', which are not part of the image $f^k(G)$. Thus, the sequence $\Delta^k + \Delta_i$, where Δ_i is the sequence of insertion operations, transforms G such that an isomorphism between $[\Delta^k + \Delta_i](G)$ and G' exists. The ϵ-isomorphism is then found by examining all pairs $\{\Delta^k, f^k\}$ generated by SG and choosing the pair $\{\Delta^m, f^m\}$ such that $C(\Delta^m + \Delta_i)$ is a minimum. From this observation it follows that an algorithm SG can be used to detect ϵ-isomorphisms as well as ϵ-monomorphisms. Therefore, in the rest of this paper, we will concentrate on the exact and inexact subgraph isomorphism problem.

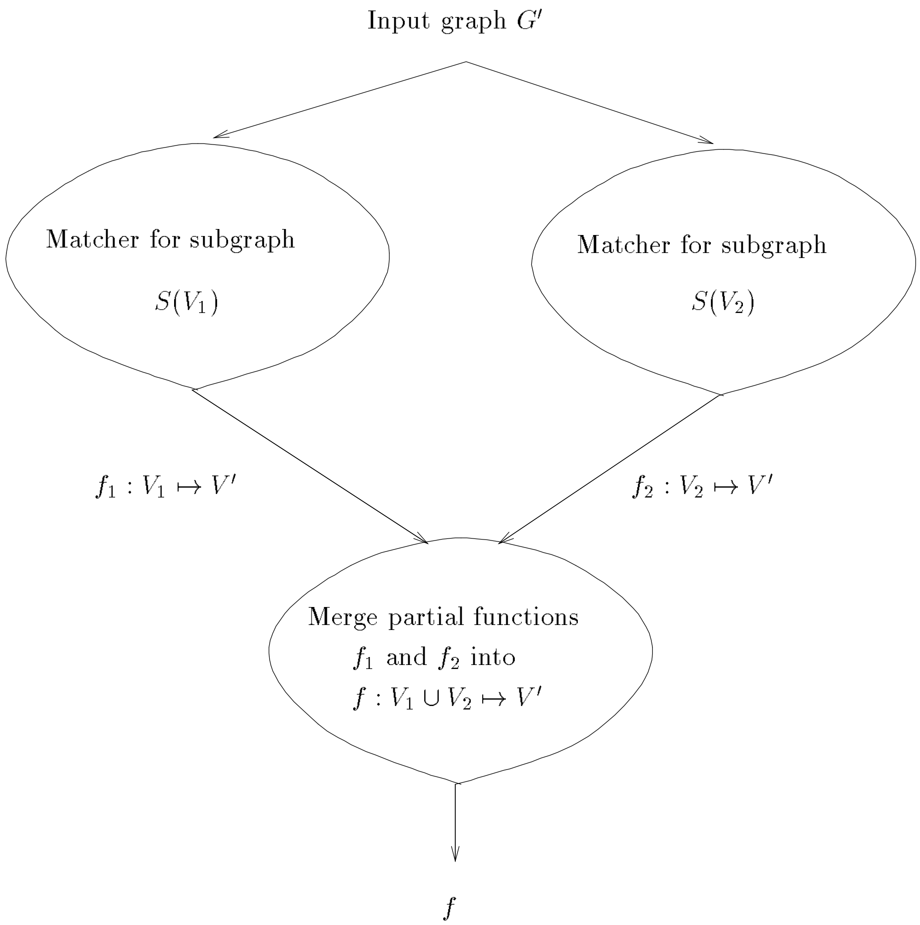


Figure 3: Basic network structure

4 A new approach to exact and inexact subgraph isomorphism

4.1 Introduction to the network approach

In the following we will be concerned with finding exact and inexact subgraph isomorphisms between a model graph and an input graph. Our approach to the problem is based on the divide and conquer idea. The model graph is partitioned into disjoint subgraphs, for which we try to find subgraph isomorphisms which can be merged in order to form a subgraph isomorphism for the full model. Starting with the complete model $G = (V, E, L, A, \mu, \nu)$ we split the set V into two disjoint sets $V = V_1 \cup V_2$. The basic setup now consists of a matching module for $S(V_1)$ and a matching module for $S(V_2)$ as it is illustrated in Fig. 3. The outputs f_1 and f_2 of the two matchers are sent to a module representing $V = V_1 \cup V_2$, where f_1 and f_2 are merged into a subgraph isomorphism for $S(V) = G$ if possible.

The segmentation of the model does not halt at this level. V_1 and V_2 are further

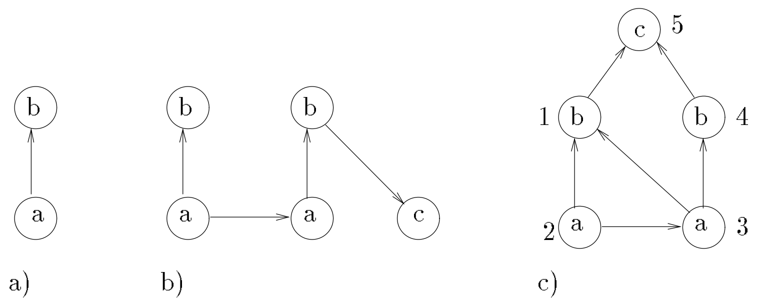


Figure 4: Two model graphs and an input graph

split up until we arrive at single vertices. Thus we get a hierarchy of partial matchers corresponding to a binary tree. The motivation behind this scheme is the fact that a matching module for a graph g can be used multiple times for the same model m if g is a subgraph of m and appears multiple times inside m. Furthermore, if g is a subgraph of several different models then the matching module for g can be used for each of the models.

We propose to organize the matching modules in a *network*. In the following, we refer to the matching modules as the nodes of the network. Each node of the network represents a graph g. Given a set of models $m_1, \ldots, m_k$ the network for the models is compiled in an off-line step. A subgraph g that is common to all the models appears exactly once in the network. Also, if a model contains several identical subgraphs, then they will be represented by the same network node. Thus we achieve a very compact representation of the model database. Fig. 5 gives an example network for the graphs in Fig. 4a and 4b, which have a subgraph in common. At run time, this network is used to find all subgraph isomorphisms between the model graph and a given input graph. In the following, we will refer to a subgraph isomorphism between a graph g and an input graph G as an *instance* of g in G. In order to detect these instances the input graph is propagated through the network. First, all instances of single model vertices which are found in the input graph, are sent down to network nodes, which represent larger graphs. Instances of these larger graphs are in turn sent further down until they arrive at a node that represents a model m_i. Each instance found by a network node is stored in a local memory before it is sent to all successor nodes. The contents of the square brackets in Fig. 5 represent the local memory contents in the network after the graph in Fig. 4c was propagated. For example, the entry $[3, 4, 5]$ in the node E represents a subgraph isomorphism from the subgraph induced by $\{3, 4, 5\}$ in the input graph to the graph corresponding to E. In other words, it represents an instance of the subgraph represented by E in the input graph.

The details of the network structure are explained in the following subsection. The recognition procedure for subgraph isomorphism is given in subsection 4.3. The extension of the network algorithm to inexact matching will be introduced in subsection 4.4, and a lookahead procedure working on the network is proposed in subsection 4.5. Finally, the off-line compilation algorithm of the network from a set of model graphs and different compilation strategies are given in subsection 4.6.

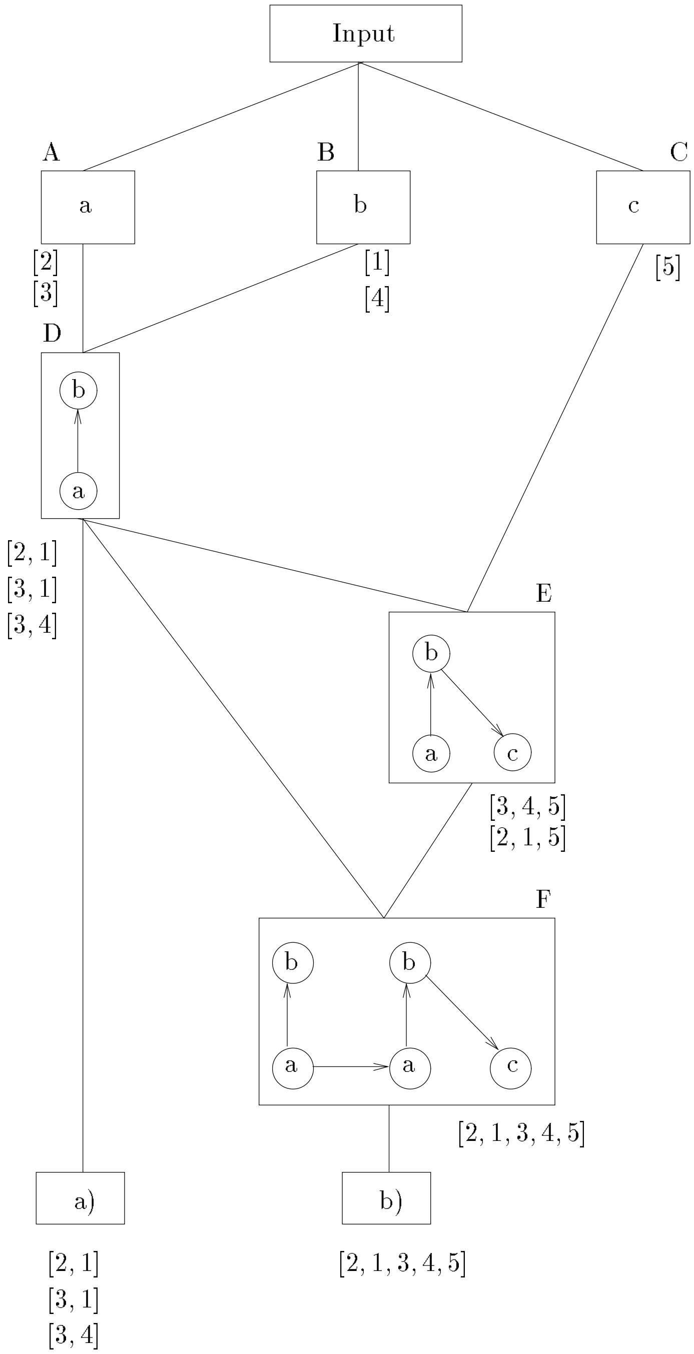


Figure 5: Network for models in Fig. 4a, 4b and local memory contents after the insertion of graph from Fig. 4c

4.2 The structure of the network

Internally, the network structure contains four types of nodes. At the top of the network there is the one and only input-node.
The **input-node** is the entrance to the network. There are one or more network edges, or n-edges for short, each leading from the input-node to a l-vertex-checker. At run time, the input graph is sent via the input-node to all l-vertex-checkers.
The **l-vertex-checkers** are the simplest matching modules in the network. Each l-vertex-checker represents a certain label l, hence any vertex v in a model graph of the database is represented by the l-vertex-checker if $\mu(v) = l$. There is exactly one incoming n-edge to each l-vertex-checker. At run time, an l-vertex-checker receives from the input-node the vertices of the input graph. If an input vertex has the label l then it is stored locally and sent to all the successor nodes. Each l-vertex-checker has one or several outgoing n-edges leading to m-model-nodes (see below) or E-subgraph-checkers.
The **E-subgraph-checkers** are the third type of nodes. They are the matching modules for graphs that consist of at least two vertices and one edge. Each E-subgraph-checker has two parent nodes of either type l-vertex-checker or E-subgraph-checker. The graph represented by the E-subgraph-checker consists of the two graphs of its parent nodes and the set of edges between them, specified by the list E, i.e. if the parent nodes n_i, n_j represent the graphs $g_i = (V_i, E_i, L, A, \mu_i, \nu_i), g_j = (V_j, E_j, L, A, \mu_j, \nu_j)$ then the graph represented in the E-subgraph-checker n_k is given by $g_k = (V_k, E_k, L, A, \mu_k, \nu_k)$ with

$$V_k = V_i \cup V_j \ \ ; \ \ E_k = E_i \cup E_j \cup E$$

where E is the set of edges attached to n_k. Any edge $e \in E$ is either an edge from g_i to g_j or an edge from g_j to g_i. If g_k is a subgraph of a model m, then g_i and g_j are two disjoint subgraphs of m and E is the set of edges connecting g_i and g_j. At run time, the E-subgraph-checker n_k will receive instances t_i of g_i and instances t_j of g_j from its parent nodes and will try to merge them into an instance t_k for g_k. Two instances can be merged if (1) they are disjoint and (2) if the edges specified in E exist between t_i and t_j and no edge exists in the input that is not specified by E. If conditions (1) and (2) are satisfied then t_i and t_j can be simply concatenated and t_k is an instance of g_k in the input graph. Each newly found instance is stored locally and sent to all the successor nodes. An E-subgraph-checker may have one or several outgoing n-edges leading to other E-subgraph-checker or to m-model-nodes.
The **m-model-nodes** are the fourth type of node. They represent the models that have been compiled into the network. Each m-model-node is connected to exactly one parent node. The graph represented in the parent node is identical to the model m. Therefore, any instance found in the parent node, is sent to the m-model-node where it is stored as an instance of the model m.

In Fig. 5 the nodes A,B and C are of the type l-vertex-checker, the nodes D,E,F are of type E-subgraph-checker while the two nodes at the bottom represent the m-model-nodes for the graphs in Fig 4a and 4b, respectively.

4.3 The exact network algorithm NA

At run time, each network node is used as a matching module and the network edges are one way channels, transporting instances from one network node to another. Given the compact network representation of the models $m_1, \ldots, m_k$ we now consider the problem of finding an exact subgraph isomorphism from each model to a given input graph G. Thus we need an algorithm which propagates the input graph through the network. We propose a propagation algorithm which is based on the RETE-algorithm and some enhancements made by Lee/Schor [LS92]. The work of Lee/Schor is relevant for non-linear networks, i.e., networks in which a node is used multiple times by the same model, and concerns the order of the instance propagation. In this paper, however, we omitted the description of the enhancements for the sake of clarity.

Each network node has a procedure attached to it, which is called whenever a new instance arrives through an incoming edge. The network propagation now works by first reading all input vertices into the network and calling the l-vertex-checker procedures.

These procedures in turn will call the procedures of their successor nodes. This calling process is repeated until every possible instance has been generated and propagated as far as possible. Note that only the vertices of the input graph are read into the network while the edges are kept in a global data structure, which is accessed directly by the E-subgraph-checkers whenever the existence of an edge $e \in E$ between two subgraphs of the input graph has to be checked. The procedure NA (<u>N</u>etwork <u>A</u>lgorithm) is the top level of the instance propagation and incorporates the input-node of the network. We pass it the input graph as a parameter:

procedure NA($G = (V, E, L, A, \mu, \nu)$)
  GE=E /* make the edges of the input graph globally accessible */
  For all vertices v in V
    For all l-vertex-checkers
      call l-vertex-checker(v)
    End For
  End For

An instance of a graph g is represented by an array t of input vertices, each position in the array denoting the assignment of a vertex of g to the input vertex that is located at this position, i.e., g is the graph repesented by a node n and $t[i] = v$ denotes the assignment of the i-th vertex of g to the input vertex v. The l-vertex-checkers produce instances t of length 1 containing a single input vertex v: $t[0] = v$.

procedure l-vertex-checker(vertex v)
  If $\mu(v) = l$ **Then**
    $t[0] = v$
    store t in the local memory
    For all successor nodes n
      If n is a E-subgraph-checker **Then**
        call E-subgraph-checker(t)
      If n is a m-model-node **Then**
        call m-model-node(t)
    End For
  End If

When the procedure E-subgraph-checker is called by one of its parents with a new instance then this instance t_p must be combined with all instances already stored in the local memory of the other parent node. Thus, the calling parent node must first be identified as left or as right parent. Then, all instances t_o stored in the opposite node are combined with the new instance t_p and if they are (1) disjoint and (2) satisfy the edge constraints, an instance $t_{new} = t_p + t_o$ is created. As we intend to find subgraph isomorphisms, any edge specified in E must exists between the subgraphs denoted by the instances t_p and t_o and no additional edges must exist[1]. It is important to note that the edges $e \in E$ are given in a special index form along with the label l of the edge, such that $e = (x, y, l) \in E$ means that there must be an edge $e' = (t_i[x], t_j[y])$ from the vertex $t_i[x]$ to the vertex $t_j[y]$ in the input graph with the label $\nu(e') = l$. The merging conditions (1) and (2) are pointed out for clarity in the following:

procedure E-subgraph-checker(instance t)
  If called by left parent node **Then**
    For all instances $t_r \in$ local memory of right parent
      If t and t_r are disjoint **And** (1)
        $\forall(i, j, l) \in E$ there exist an edge $e = (t[i], t_r[j]) \in GE$ with $\nu(e) = l$ and
        no additional edge $(t[x], t_r[y]) \in GE$ with $(x, y) \notin E$ exists **Then** (2)
        $t_{new} = t + t_r$ /* concatenate the instances */
        store t_{new} in local memory
        For all successor nodes n
          If n is a E-subgraph-checker **Then**
            call E-subgraph-checker(t_{new})
          If n is a m-model-node **Then**
            call m-model-node(t_{new})
        End For
      End If
    End For
  If called by right parent **Then**
  /* this case is analogous to the above */
  ⋮

Finally, the procedure for the m-model-nodes receives an instance of the model m and stores it locally.

procedure m-model-node(instance t)
  store t in local memory
  print the new instance t of the model m

In the following we study a simple matching example by processing each input vertex step by step. Fig. 7 displays the network for the model graph in Fig. 6a. The local memory contents are given after the input graph in Fig. 6b has been processed completely. The memory contents correspond to the last row in Table 1. The rows of Table 1 illustrate

[1] If we ingore the additional edges, then the proposed network algorithm finds graph monomorphisms

vertex	A	B	C	D
1		[1]		
2	[2]	[1]		
3	[2],[3]	[1]	[23]	
4	[2],[3]	[1],[4]	[23]	
5	[2],[3],[5]	[1],[4]	[23],[25]	[254]

Table 1: Local memory contents for algorithm NA. Total number of instances: 8

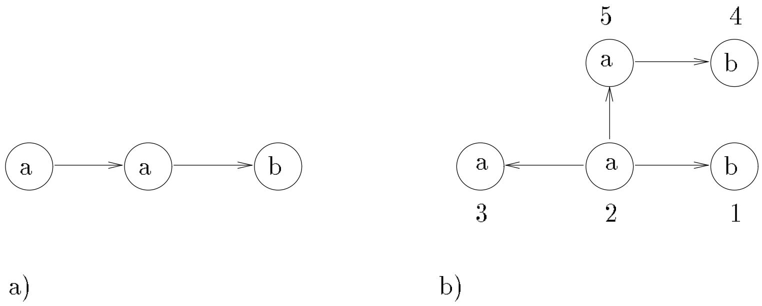


Figure 6: Two sample graphs

how the local memories are growing with every new insertion of an input vertex. For example, in row 5, the vertex 5 of the input graph with label 'a' is inserted. First vertex 5 is tested by the l-vertex-checkers A and B. Because 5 matches only the label in A, it is stored there and sent to the successor C. In the E-subgraph-checker C, 5 is combined with all instances stored in A. For each combination $[5,2],[5,3],[5,5],[2,5],[3,5]$ the procedure in C tests whether (1) the instances are vertex-disjoint, and (2) there is an edge from the first vertex to the second. Only $[2,5]$ satiesfies both conditions, thus $[2,5]$ is stored in C and sent further down to the E-subgraph-checker D. Finally, in D we find that the only valid combination is $[2,5,4]$ because only from the vertex 5 to the vertex 4 there exists an edge as it is specified in D. The instance $[2,5,4]$ is sent to the m-model-node and is printed out. In the end there is a total of 8 instances present in the network nodes.

In this subsection we introduced an network algorithm NA which uses the compact network representation in order to find instances of distinct and common subgraphs of different models and combines them to form instances for the full models. So far, the algorithm only finds exact subgraph ismorphisms. In the following subsection, we propose an extension of NA, which includes the basic edit operations from section 2.

4.4 The inexact network algorithm INA

As defined in section 2, an inexact subgraph isomorphism between a graph G and G' consists of a sequence Δ of edit operations that are applied to G and a subgraph isomorphism from $\Delta(G)$ to G'. The inexact subgraph isomorphism is optimal if there is no other sequence Δ' such that there exists a subgraph isomorphism from $\Delta'(G)$ to G' and the costs for Δ' are lower than those of Δ. If the inexact subgraph isomorphism problem is

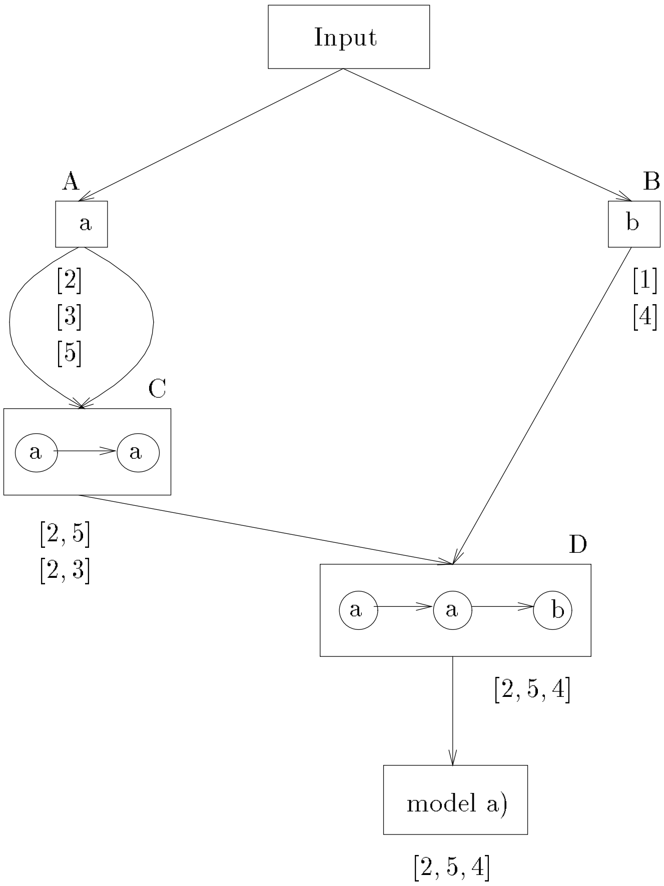


Figure 7: Network for graph in Fig. 6a and memory contents after insertion of graph in Fig. 6b

embedded into a classification problem, such that out of a set of model graphs $m_1, \ldots, m_k$ we look for the model m_i with minimal distance $dist_s(m_i, G)$ to the input graph G, then usually the distance between each model and the input graph is calculated separately and then the model with optimal distance can be determined. However, using the compact network representation and the features of NA, we propose an inexact subgraph isomorphism, which given a set of models and an input graph, determines the model with the minimal distance and the sequence of necessary edit operations *without* having to calculate the distance of each model to the input graph separately.

Clearly, we must incorporate the edit operations into the network nodes in order to find inexact subgraph isomorphisms between the models in the network and a given input graph. The tests in the network nodes must be adjusted such that for the graph g represented in a node n, not only instances denoting an exact subgraph isomorphism, but also instances which imply a sequence of edit operations are found. For example, if an l-vertex-checker n receives an input vertex v with the label k, then the instance $t[0] = v$ generated in n implies that the label l of the vertex in n was substituted by the label k. Because an instance t may now imply a sequence of edit operations, it also incorporates the costs for these edit operations. We say that $C(t)$ denotes the total costs of the edit operations that are implied by t.

From the definition of subgraph isomorphism we know that the edit operations needed in order to transform a model graph are label substitution for vertices and edges, vertex deletion, edge deletion and edge insertion. The operations on the vertices are performed in the l-vertex-checkers . In order to account for vertex deletion, we add to each input graph the dummy vertex \$. The task of the l-vertex-checkers is now adjusted in the following manner: If an input vertex v is sent to an l-vertex-checker then we create an instance $t[0] = v$. We set the costs of the instance t to the cost of the substitution of l with $\mu(v)$, $c((x,l) \rightarrow (x,l))$, where x denotes any vertex represented in the l-vertex-checker. If v is the dummy vertex \$ then the costs of the instance are set to the costs for the deletion of v, $c(v \rightarrow \$)$.

The functionality of the E-subgraph-checkers must be altered accordingly: Two instances t_l, t_r coming from the parent nodes n_l, n_r can be merged if (1) they are either vertex disjoint or they have the dummy vertex \$ in common. The edges between the two instances are no longer tested when determining whether the instances can be merged. However, if edges are non existant, mislabeled or extraneous the costs for their deletion, substitution or insertion must be added to the costs of the two instances. If t_l and t_r satisfy (1) then we generate a new instance $t = t_l + t_r$. The costs of t are initialized to

$$C(t) = C(t_l) + C(t_r),$$

because any edit operation performed in t_l and t_r must also be done in t. For any entry $(i, j, l) \in E$, we test, whether $e = (t[i], t[j], k)$ exists in the input graph. If e exists, but the label k is different from l then the costs for the substitution of l by k are added to $C(t)$. If e does not exist, then it must be deleted in the model graph and thus we add the costs for the deletion of e to $C(t)$. Finally, if there exists and an edge $e = (t[x], t[y])$ in the input graph and there is no entry $(x, y, \_)$ in E then e must be inserted and consequently the costs for the insertion are added to $C(t)$. The functionality of the l-vertex-checker and the E-subgraph-checker has now been altered in such a way that instances representing inexact subgraph isomorphisms can be found in each node.

Clearly, if the control structure and the data flow of NA is left unchanged, the resulting inexact algorithm belongs to the category of the enumeration algorithms. All possible substitutions and deletions of model vertices and all combinations of inexact instances are at one time generated and sent further through the network. The output of NA is then a collection of all possible inexact instances between the models and the input, their costs ranging from the optimal cost to the maximum cost, where all model vertices and edges are deleted and all input vertices and edges are inserted. It is obvious that an enumeration algorithm which is computationally exponential cannot be of any practical use. In order to cut down on the number of instances that are found, a control structure is needed, which works on NA and allows a more intelligent propagation of instances. In addition to the regular local storage memories for instances, which we will call GO memories from now on, we propose a sorted memory STOP for those instances representing inexact subgraph isomorphisms, whose costs exceed a certain threshold. The top element in the STOP memory is the instance with the least costs. Instead of simply sending on an instance that was just generated in a network node, we compare its costs to a given threshold. If the threshold is exceeded, then the instance is inserted into the local STOP memory and propagation is continued elsewhere, otherwise the algorithm is continued normally. First, NA processes the complete input graph. The threshold is initialized to zero. Therefore, any inexact instance generated in an l-vertex-checker or an E-subgraph-checker is inserted into the STOP memory of the respective node. When NA terminates, all the instances have been moved as far as possible and all exact subgraph isomorphisms from graphs in the network nodes to the input graph have been found. Then the STOP memory containing the cheapest instance is consulted. The threshold is reset to the costs of this instance and the instance is sent regularly through the network, until either a model instance is found or again a STOP memory must be consulted. The meaning of the STOP memories is to temporarily hide an instance from all other nodes and prevent its propagation until all instances with less costs have been considered. If an instance is in a STOP memory, it cannot be seen by the successor nodes and therefore it will not participate in any recombination with instances from an opposite node. As soon as all instances with less costs have been moved, we reinsert the instance at the position of the STOP memory by calling the procedures of all successor nodes. We say that an instance becomes *activated* if it is removed from a STOP memory and reinserted into the network.

All STOP memories containing at least one entry are organized in a sorted list OPEN. Its meaning is comparable to the sorted list in the A^* algorithm, where the tree nodes are sorted according to their suitability for expansion. Whenever the top element of a STOP memory changes, i.e. an instance with less costs than the current top element is inserted into STOP, then OPEN must be rearranged and the order of the STOP memories must be adjusted. Likewise, if an instance is removed from a STOP memory and the memory becomes empty thereafter, it must be removed from OPEN.

The procedures l-vertex-checker(...) and E-subgraph-checker(...) must be adjusted in order to generate inexact instances and manage the local memories GO and STOP correctly. Note, that the threshold $Threshold$ is not set within the node procedures, but will be initialized and set in the control structure outside NA. In the following we refer to the network algorithm using the modified version of the procedures l-vertex-checker(...) and E-subgraph-checker(...) as $\tilde{\text{N}}$A.

```
procedure l-vertex-checker(vertex v)
  If the substitution of l by μ(v) is not impossible Then
    t[0] = v
    If v = $ Then
      C(t) =costs of deletion of a vertex with label l
    Else
      C(t) =costs of substitution of l by μ(v)
    End If
    If C(t) > Threshold Then
      store t in the STOP memory
    Else
      store t in the GO memory
      For all successor nodes n
        If n is an E-subgraph-checker Then
          call E-subgraph-checker(t)
        If n is a m-model-node Then
          call m-model-node(t)
      End For
    End If
  End If

procedure E-subgraph-checker(instance t)
  If called by left parent node Then
    For all instances t_r in GO memory of right parent
      If t and t_r are disjoint or have only $ in common Then          (1)
        t_new = t + t_r /* concatenate the instances */
        call CostOfInstance(t_new, t, t_r, E) /* calculate the total cost of t_new */
        If C(t_new) > Threshold Then
          store t_new in STOP memory
        Else
          store t_new in GO memory
          For all successor nodes n
            If n is a E-subgraph-checker Then
              call E-subgraph-checker(t_new)
            If n is a m-model-node Then
              call m-model-node(t_new)
          End For
        End If
      End If
    End For
  If called by right parent Then
  /* this case is analogous to the above */
  ⋮
```

The auxiliary function $CostOfInstance$ which is called in the E-subgraph-checker procedure calculates the costs of the vertex and edge transformations implied by t_{new}. The edit operations on the vertices are already implied by the partial instances t_l, t_r including the edit operations on edges which are entirely part of either t_l or t_r. The cost of the edit operations on edges connecting t_l to t_r are calculated by looping over the entries in the list E.

procedure CostOfInstance(t_{new}, t_l, t_r, E)
  $C_{edges} = 0$
  /* calculate the costs for the edge transformations */
  For all $(i, j, l) \in E$
    If there exist an edge $e = (t_l[i], t_r[j]) \in GE$ **Then**
      If $\nu(e) \neq l$ **Then** $C_{edges} = C_{edges}$+cost of subtitution of l by $\nu(e)$.
    Else
      $C_{edges} = C_{edges}$+cost of deletion of an edge with label l
    End If
  End For
  For all edges $e = (t_l[x], t_r[y])$ in the input graph with $(x, y, \_) \notin E$
    $C_{edges} = C_{edges}$+costs for the insertion of e
  End For
  /* now add the costs C_{edges} to the costs of t_l and t_r*/
  $C(t_{new}) = C(t_l) + C(t_r) + C_{edges}$

The network algorithm ÑA is now to be embedded into a control structure that decides which instances in the network are to be moved next by consulting the list OPEN. The resulting inexact subgraph isomorphism detection algorithm is called INA, short for Inexact Network Algorithm. Some auxiliary functions concerning the list OPEN and the STOP memories must be explained in detail: OPEN takes elements of the form $(n, values)$, where n is a network node and $value$ denotes the costs of the cheapest instance in the STOP memory of the node. The elements in OPEN are sorted encreasingly according to the argument $value$, such that nodes with cheap instances in their STOP memory are at the top, while nodes with expensive instances are at the bottom of OPEN. The top element of OPEN is retrieved and removed by

$$(n, value) \leftarrow OPEN.top()$$

procedure INA(input graph $G = (V, E, L, A, \mu, \nu)$)
  $\tilde{\text{N}}$A($\tilde{G}$)   /* $\tilde{G} = (V \cup \$, E, L, A, \mu, \nu)$ */
  For all nodes n in the network
    If the STOP list of n is not empty **Then**
      insert (n,Cost of top element in STOP) into OPEN
  End For
  $(n, value) \leftarrow$OPEN.top
  $Threshold = value$
  If a model is found **Then** $Threshold = 0$
  While $Threshold \geq value$
      $t \leftarrow$remove top element of STOP memory of n
      If STOP is not empty **Then**
        insert (n,Cost of cheapest instance in STOP) into OPEN
      store t in the GO memory of n
      For all successor nodes s of the node n
        If s is an E-subgraph-checker **Then**
          call E-subgraph-checker(t)
        If s is a m-model-node **Then**
          call m-model-node(t)
      End For
      For all nodes k with modified STOP memories
        If top element t has changed **Then**
          reposition the STOP memory of k in OPEN according to $C(t)$
        If STOP is empty **Then**
          remove STOP memory of node k from OPEN
      $(n, value) \leftarrow$OPEN.top
      $Threshold = value$
      If a model is found **Then** $Threshold$ =cost of model instance
  End While

The procedure INA starts by first calling the exact network algorithm NA with the input graph G and the dummy vertex \$. When ÑA terminates, all nodes with STOP memories that are not empty are inserted into OPEN along with the costs of the cheapest instance in this STOP memory. Next, the threshold is initialized to the costs of the cheapest instance in the network, which has not yet been moved. These costs are always greater than zero. However, if during ÑA an exact subgraph isomorphism was found, then the threshold is reset to 0. Consequently with a threshold zero and *value* greater than zero the following while loop is never executed. We conclude that if there exists an exact subgraph isomorphism between a model in the network and the input graph, then INA will only call ÑA and perform the same number of steps as NA. However, ÑA will in any case generate not less instances than NA because inexact instances are also considered. If no model instance was found in ÑA, then the while loop is entered. Inside the while loop the cheapest instance in the STOP memory of the node n is moved to the GO memory. If thereafter STOP is not yet empty, we must reposition the node n in the OPEN list, because the new top instance will have cost greater than those of the previously removed instance. Now the activated instance is sent to all successor nodes by calling their attached procedures. During these and subsequent recursive procedure calls new instances are generated and top elements of STOP memories may change, such that it becomes necessary to reposition all nodes whose STOP memory have changed. Finally, we set the threshold to the cheapest instance in the network, or if a model instance was found, to the costs of this model instance.

If a model is finally found, the algorithm continues within the while loop until the costs for the best instance in a STOP memory exceed the cost of the model instance. The first model instance found represents an optimal inexact subgraph isomorphism. If the network contains several models, then INA finds the instance t for the model with the minimum distance to the input graph. No other instances of models with greater distances are generated. The instance t implies the sequence of edit operations that are necessary in order to transform the model such that a subgraph isomorphism exists. Also, the costs of the instance t, $C(t)$, are equal to the distance between the model and the input graph. Informally, we can say that INA finds the optimal inexact subgraph isomorphism because all possible edit operations are incorporated into the network. Furthermore, by always moving the cheapest instance, until some model instance is found, we will eventually find the optimal subgraph isomorphism for the model with the least distance to the input graph. In order to proof the optimality of INA formally, we first show that the following lemma holds:

Lemma 4.1: Let $G = (V, E, L, A, \mu, \nu)$ be a model and $G' = (V', E', L, A, \mu', \nu')$ an input. Furthermore, let the network node i represent the subgraph $S(V_1) \subset G, V_1 \subset V$. The first instance t in the GO memory of i represents the instance of $S(V_1)$ in G' with least cost. The distance between $S(V_1)$ and G' is

$$dist_s(S(V_1), G') = C(t)$$

□

Proof: If the threshold in the inner loop of INA is θ_0 then all instances t with costs $C(t) \leq \theta_0$ are moved. If the GO memory of node i is still empty, then there exists no subgraph isomorphism between $S(V_1)$ and G' with less or equal costs than θ_0. Next, if the instance t is activated, then the threshold is set to

$$\theta_1 = C(t).$$

If t is inserted into the GO memory of i, then $C(t)$ is the minimal distance between $S(V_1)$ and G' because any instance with costs less than θ_1 would have been moved before. □

Now we are ready to state the optimality of the algorithm INA.

Theorem 4.2: Let $G_1, \ldots, G_k$ be model graphs, which are compiled into a network and G' an input graph. Then, INA terminates and finds the optimal instance t of a model G_i such that

$$C(t) = dist_s(G_i, G') = MIN\{dist_s(G_j, G') | j = 1 \ldots k\}$$

□

Proof: If the node n represents the set V_i of vertices and thus the full model G_i, then the first instance t that is inserted into the GO memory of n represents an optimal subgraph isomorphism $f_i : V_i \mapsto V'$. Because of Lemma 4.1, it is valid that

$$dist_s(G_i, G') = C(t)$$

for the first t entering the GO memory of n. Furthermore, with G_i having the least distance from the input graph G', INA terminates after $Threshold = C(t)$ was set and all partial instances with costs less than or equal to $Threshold$ have been moved. □

Obviously, by controlling the point of action in the network and always moving the cheapest instance, a large number of the inexact instances that do not contribute to the final solution and whose costs exceed the cost of the optimal match, will never be generated. Figure 8c displays again the network for the model in Fig. 6a (identical to the graph in Fig. 8a) and a distorted input graph shown in Fig. 8b. Each node has a GO and a STOP memory, whose contents change with each new iteration of INA. We keep track of the changes in the GO and the STOP memories in Table 2. The leftmost column indicates the current state of processing. The first three steps show the insertion of the input vertices and the dummy vertex \$ into the network. After \$ is inserted, the local memories contents are identical to the ones displayed in Fig. 8c. After termination of the exact run of ÑA, the identification (A,B,C,D) of the node whose STOP list was selected for processing, is given in the leftmost column. The cost structure for substitution is defined as follows:

	a	b	c	\$
a		0.2	0.3	0.5
b	0.2		0.5	0.6
c	0.3	0.5		0.5
\$	0.5	0.6	0.5	

Insertion and deletion of an edge costs uniformly 0.1 units. For example, in the 7th row of Table 2 the instance [\$] in the STOP memory of node A has been moved to the GO memory and sent to all successor nodes. As a consequence, the STOP memory of the node C received the instances [\$1],[\$2],[1\$],[2\$],[\$\$].

INA terminates when the token [\$12] is sent from the node D to the model node. The optimal subgraph isomorphism is then found with cost 0.9.

The described inexact graph matching algorithm INA is based on ÑA. NA and ÑA differ only in that ÑA generates inexact instances and manages the STOP memories. The instance propagation however remains unchanged and therefore all the features of NA concerned with the sharing of subgraphs in the same model and among different models are also valid for ÑA. In the presented form, the algorithm terminates when an instance of a model with minimal costs has been found. However, we could also set a threshold independent of the best instance and hence find all instances that do not exceed a certain cost.

4.5 A lookahead procedure

4.5.1 Basic idea and introductory example

Although INA works definitely better than the simple enumeration version of NA, the main problem is still that too many inexact instances are sent through the network which

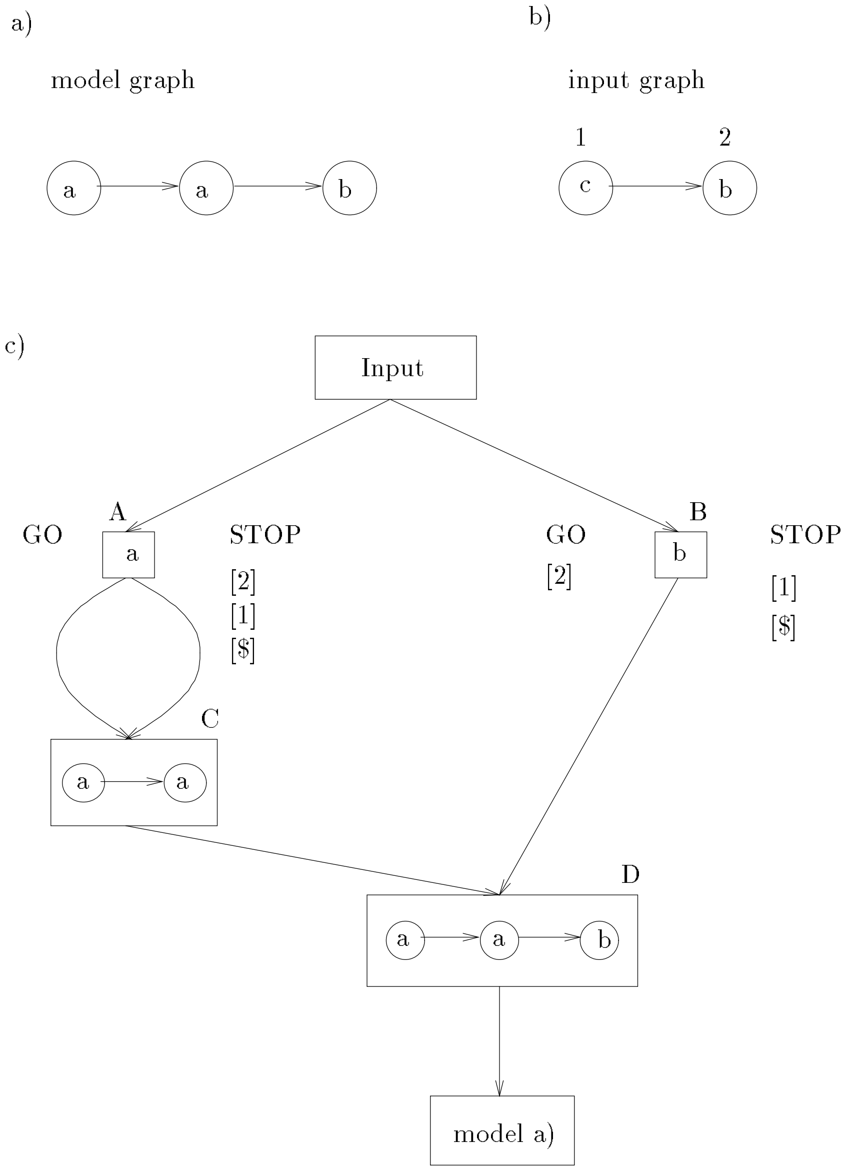


Figure 8: Network c) for model a) and local memory contents after the insertion of the distorted input graph b)

	A		B		C		D	
	GO	STOP	GO	STOP	GO	STOP	GO	STOP
1		[1]		[1]				
2		[2],[1]	[2]	[1]				
$		[2],[1],[$]	[2]	[1],[$]				
A	[2]	[1],[$]	[2]	[1],[$]				
A	[2],[1]	[$]	[2]	[1],[$]		[12],[21]		
B	[2],[1]	[$]	[2],[1]	[$]		[12],[21]		
A	[2],[1],[$]		[2],[1]	[$]		[12],[$1]..		
C	[2],[1],[$]		[2],[1]	[$]	[12]	[$1]..		[12$]
B	[2],[1],[$]		[2],[1],[$]		[12]	[$1]..		
C	[2],[1],[$]		[2],[1],[$]		[12],[$1]..	[$$]	[$12]	[1$2],[12$]..

Table 2: Local memory contents for algorithm INA. Total number of instances: 23

do not contribute to the final solution. For example, if the cost of the optimal subgraph isomorphism exceeds the cost of any basic vertex deformation, then INA will eventually remove all instances in the STOP memories of the l-vertex-checkers and send them through the network, although most of these basic deformations will never be used. Studies concerning the A^* algorithm have shown that a great reduction of the number of expanded nodes can be achieved if at each state in the tree search an estimation of the future costs is calculated on the basis of heuristics and these costs are added to the actual cost of the current partial matching[HE80]. Optimality is guaranteed as long as the estimation is a consistent lower bound of the real future costs. The future error estimation is calculated dynamically whenever a new node in the search tree is expanded. Its value depends mainly on the partial matching that is represented in the tree node. The concept of a future error estimation can be easily applied to our network algorithm in the following manner. Assume that there is a model graph G represented in the network:

1. After the input graph has been processed by NA and before the inner loop of INA is started, the network contains plenty information about partial instances of the models.

2. Based on these information, each network node representing a subgraph S is assigned an estimation $FUTerr$ of the minimal costs that are necessary in order to instantiate the rest of the model $G - S$.

3. Each instance t in the STOP memories is no longer judged individually, but its costs are added the $FUTerr$ of its node, thus anticipating the total costs that will arise if t is part of the final solution. OPEN must be rearranged, according to the new costs.

The idea behind this scheme is to identify nodes in the network which are likely to contribute to the solution. By assigning the value $FUTerr$ to the nodes, we can control the order of these nodes in the OPEN list and influence the sequence in which instances are moved by INA. For example, given an input graph G' and a model G, assume S is a subgraph of G. Then, if the distance $dist_s(G - S, G')$ is known to be very large, it can be concluded that any instance in the STOP memory of the node representing S is unlikely to contribute to the final solution and therefore it should never be activated.

The estimation procedure is not done dynamically for each instance, but statically over the whole network after ÑA has terminated. To understand the effect of the proposed lookahead procedure, consider the following example:

Example 4.3: For a given model G and a distorted input G', let ϵ be the cost of the optimal subgraph isomorphism. Assume the E-subgraph-checker i represents the subgraph $S \subset G$ and that the distance of $G - S$ to G' is large. Futhermore, the STOP memory of i contains an instance t with costs

$$C(t) = \delta, \qquad 0 < \delta < \epsilon,$$

which does not contribute to the final solution. If INA runs without lookahead, then t will be removed from the STOP memory and sent to the successors of i. However, if the lookahead succeeds in estimating the costs $FUTerr$ of $G - S$ fairly accurate, then

$$\delta + FUTerr > \epsilon.$$

Therefore, with lookahead, t will never be sent the successors of i, and the overall number of instances generated will be smaller. □

If the estimation $FUTerr$ is a lower bound of the actual costs for the rest of the model, then the optimality of INA is guaranteed, because if an instance t is part of the final solution then

$$C(t) \leq \epsilon$$

and with $FUTerr$ being a lower bound, the inequality

$$C(t) + FUTerr \leq \epsilon$$

will never be violated. Consequently, t will be activated if it is part of the final solution and the generation of the instance representing the final solution is still guaranteed.

To estimate the minimum distance between the subgraph g of the model G and the input graph G', we need to follow the segmentation that is imposed on the model by the network. Each graph g represented by a node n consists of the graphs g_l,g_r of the parent nodes n_l, n_r. If we have a minimal distance estimation for g_l and g_r, we can derive a distance estimation for g. We call the distance estimation *estimated_cost*. Based on these values, we calculate for each node n' representing a graph g' the minimal costs $FUTerr$ of the rest graph $G - g'$. If g' is a subgraph of several model $G_1, \ldots, G_k$, then we assign the minimum of the costs $FUTerr(M_i - g')$ to the node n'.

The proposed lookahead works in two phases:

phase 1 The network nodes are visited top down, each being assigned a minimal distance estimation for the graph that it represents. The estimated distances are steadily growing from the top to the bottom because the size of the graphs represented in the nodes is also growing.

phase 2 The distance estimations from phase 1 are collected bottom up and assigned as $FUTerr$ to the respective nodes. The $FUTerr$ values are steadily growing from the bottom to the top, because the smaller the represented graphs g are, the larger is the remaining graph $G - g$ and the anticipated error for $G - g$.

Phase 1 is started after the first run of ÑA with $Threshold = 0$ has terminated and the full input graph is processed. The network contains many partial instances in the GO and STOP memories of its nodes. The estimation of the distance for a subgraph represented in the network node i depends on the contents of the GO and STOP memory of i and the distance estimations for the subgraphs in the parents of i.

If the GO memory of a node i is not empty, then we must assume 0 as the minimal costs, because in the first run of ÑA no instance with costs greater than zero are inserted into GO memories. Therefore, at least one instance of the subgraph in node i with costs equal to 0 was found. On the other hand, if the GO memory is empty after the first run, then we must assume some deformation in the subgraph of node i. If node i is an l-vertex-checker, we can simply take the top entry in the STOP memory as the minimal distance between the vertex in i and the input graph. This will always be a lower bound, because it corresponds to the cheapest possible edit operation. If node i is a E-subgraph-checker, we must distinguish between several cases: First, if both parent nodes have empty GO memories, we may simply add their minimal distance estimations, because both nodes anticipate an error and an instance arriving at i will incorporate at least the sum of both these errors. Secondly, if only one parent node shows an empty GO memory, we assign its distance estimation to the node i. Finally, it is possible, that both parents of a node i have non empty GO memories, but the GO memory of i itself is empty. Clearly, we must assume that the input graph is either distorted in one of the subgraphs of the parent nodes, or in the edge structure that is tested in the node i. The minimum of the three possibilities is guaranteed to be a lower bound of the actual costs in i. Thus, even if the GO memory of a node is non empty and the minimal distance equal 0, it is necessary to keep track of the minimal deformation that might appear in a subgraph. We call the minimum deformation *hidden_cost*. The *hidden_cost* carry information about possible minimum errors associated with subgraphs. They are propagated along with the *estimated_cost* from the top to the bottom of the network.

In phase 2, we use the estimated distances in order to anticipate for each node the minimal costs $FUTerr$ that will appear in the rest of the network.

Fig. 9c illustrates some of the above cases in the calculation of the lower bound estimation. The network is compiled for the model in Fig. 9a, the memory contents are given *after* ÑA has terminated for the input graph 9b and *before* the inner loop of INA has started. The GO memories are located on the left side, the STOP memories on the right side of the network nodes. The minimum estimated distances are given below the nodes. The cost of each instance is given on the right-hand side and is based on some cost function. The distance estimations for the l-vertex-checker are straightforward: In each of the nodes A,B,C we have found one instance with costs 0, thus the minimal distance *estimated_cost* is set to 0. The node D has an empty GO memory, but its STOP memory contains 4 entries. The instance [1] has the least costs 0.3, therefore 0.3 is a lower bound for the distance of any model vertex with label 'd' to the input graph. Both parents of the node E have a minimal distance of 0, but in E itself no instance with 0 costs was found. Thus we assume that there is either a deformation in A or B or in the edge structure tested

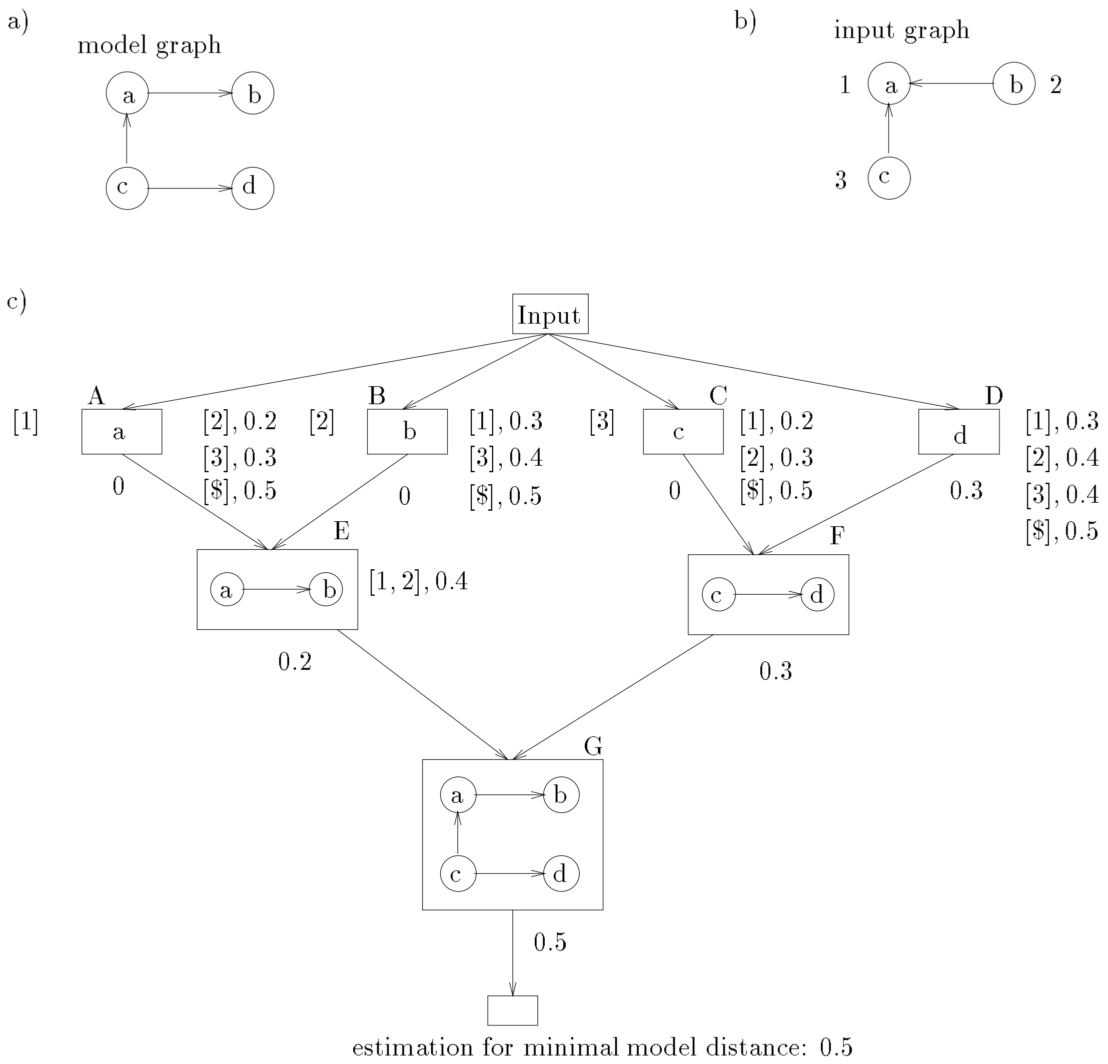


Figure 9: Distance estimation based on local memory contents

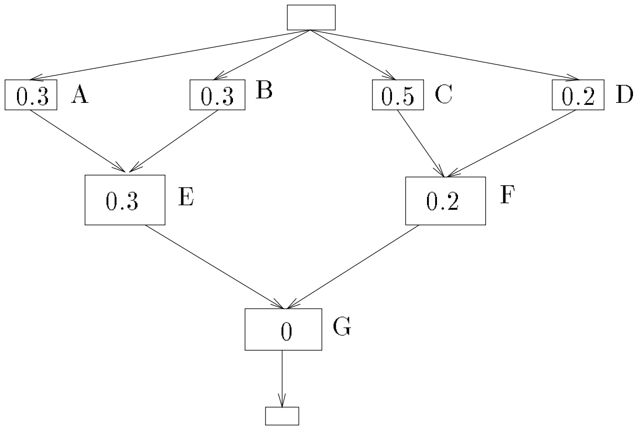


Figure 10: $FUTerr$ values of each node based on the distance estimations from Fig. 9c

in the node E. The minimal deformation in A is denoted by the top instance [2] with costs 0.2 in its STOP memory, in node B it is [1] with costs 0.3 and in E it is $[1, 2]$ with costs 0.4. The minimum of the three cost values is guaranteed to be a lower bound estimation, so we assign 0.2 to E as the *estimated_cost*. In node F only the left parent has a distance 0, while the right parent D shows a minimal distance of 0.3. Thus 0.3 is also the lower bound of the distance in node F. Finally, both parents of node G have a minimal distance greater than 0. Due to the fact that the graphs represented in E and F are disjoint, we may add their estimated distances $0.2+0.3$ and assign 0.5 as the minimal distance to node G. Hence we arrive at a distance estimation of 0.5 for the model graph after all instances with costs equal to 0 have been moved as far as possible. The meaning of the distance estimations is obvious: For example, the estimation for node F is 0.3. Therefore, if we want to establish an inexact subgraph isomorphism between the graph corresponding to F and the input graph, the sequence of edit operations will cost at least 0.3. Now, we can assign to each node the costs $FUTerr$, which denote the minimum cost that is necessary in order to comlete any instance in this node such that an instance for the model graph results. Fig. 10 shows a simplified version of the network in Fig. 9 along with the $FUTerr$ values of each node. The $FUTerr$ of the node G is of course 0, because any instance in G is already a complete instance of the model graph. In node E the $FUTerr$ is 0.3, because the minimum cost of any instance in F is 0.3, and an instance in D will be merged with an instance of F in order to form a complete instance for the model. The $FUTerr$ of the node C is 0.5 because the minimum distance of node D is 0.3 and the $FUTerr$ of the successor node F is 0.2. Clearly, any costs anticipated in F must also be anticipated in C. Therefore, we get a $FUTerr$ for C of 0.2+0.3=0.5.

4.5.2 The network lookahead algorithm NLA

The assignment of the estimated distances *estimated_cost* is done by calling the procedure $distEstimationVertexChecker$ for each l-vertex-checker and subsequently the procedure $distEstimationSubgraphChecker$ for each E-subgraph-checker.

procedure distEstimationVertexChecker(i)
  hidden_cost=Costs of cheapest instance in STOP
  If GO is not empty **Then**
    estimated_cost = 0
  Else
    estimated_cost = *hidden_cost*

The top entry in the STOP memory is equivalent to the cheapest possible substitution or insertion of a certain label. As there is no other possible deformation with less costs, the *hidden_cost* values of the l-vertex-checker are a safe lower bound in case a deformation must be assumed in this node. The lower bound estimation of the distances of a graph in an E-subgraph-checker is done by the procedure $distEstimationSubgraphChecker$.

procedure distEstimationSubgraphChecker(i)
  $left$=left parent of node i
  $right$=right parent of node i
  If $left.estimated\_cost > 0 \wedge right.estimated\_cost > 0$ **Then**
    $hidden\_cost = left.estimated\_cost + right.estimated\_cost$
  Else If $right.estimated\_cost > 0$ **Then**
    $hidden\_cost = right.estimated\_cost$
  Else If $left.estimated\_cost > 0$ **Then**
    $hidden\_cost = left.estimated\_cost$
  Else
    $hidden\_cost = MIN\{left.hidden\_cost, right.hidden\_cost, STOP.topValue\}$
  End If
  If GO is not empty **Then**
    $estimated\_cost = 0$
  Else
    $estimated\_cost = hidden\_cost$

At the end of phase 1 a first estimation of the overall costs for a model graph is reported.

In phase 2 the $FUTerr$ of a node i representing a subgraph S can now be calculated as the sum of the $FUTerr$ of the immediate successor $succ$ and the $estimated\_cost$ of the opposite node with respect to $succ$. If i has several successors, the calculation is performed for each successor and the minimum is taken. The $FUTerr$ of the model nodes are set to 0 and all $FUTerr$ of the nodes are initialized to MAX_INT.

procedure futureErrorAssignment(i, err)
  If $err < FUTerr$ **Then**
    FUTerr=err
  End If
  $ferr$ =left parent.$estimated\_cost + FUTerr$
  futureErrorAssignment(right parent,$ferr$)
  $ferr$ =right parent.$estimated\_cost + FUTerr$
  futureErrorAssignment(left parent,$ferr$)

The complete network lookahead procedure NLA starts with first applying the cost estimation procedures on all the nodes, level by level, such that when node i is treated, the parents of node i have already been assigned valid $estimated\_cost$ and $hidden\_cost$ values. Then the back propagation is recursively started at the model nodes:

```
procedure NLA
  /* top down traversal */
  For all levels l
     For all nodes i in level l
        If node i is an l-vertex-checker Then
           distEstimationVertexChecker(i)
        Else
           distEstimationSubgraphChecker(i)
  /* bottom up traversal */
  For all model nodes m
     futureErrorAssignment(parent of m,0)
```

The following procedure is a modified version of INA, which includes the lookahead NLA. For clarity, the changes in INA are numbered on the right hand side of the code segment.

```
procedure INA_with_lookahead
  ÑA(G̃) /* G̃ = (V ∪ $, E, L, A, μ, ν) */
  NLA() /* call lookahead */                                       (1)
  For all nodes n in the network
     If the STOP list of n is not empty Then
        insert (n,Cost of top element in STOP + FUTerr) into OPEN  (2)
  End For
  (n, value) ←OPEN.top
  Threshold = value
  If a model is found Then  Threshold = 0
  While Threshold ≥ value
       t ←remove top element of STOP memory of n
       If STOP is not empty Then
          insert (n,Cost of top element in STOP + FUTerr) into OPEN  (3)
       store t in the GO memory of n
       For all successor nodes s of the node n
          If s is an E-subgraph-checker Then
             call E-subgraph-checker(t)
          If s is a m-model-node Then
             call m-model-node(t)
       End For
       For all nodes n with modified STOP memories
          If top element t has changed Then
             reposition the STOP memory of n in OPEN according to C(t) + FUTerr (4)
          If STOP is empty Then
             remove STOP memory of node n from OPEN
       (n, value) ←OPEN.top
       Threshold = value
       If a model is found Then  Threshold =cost of model instance
  End While
```

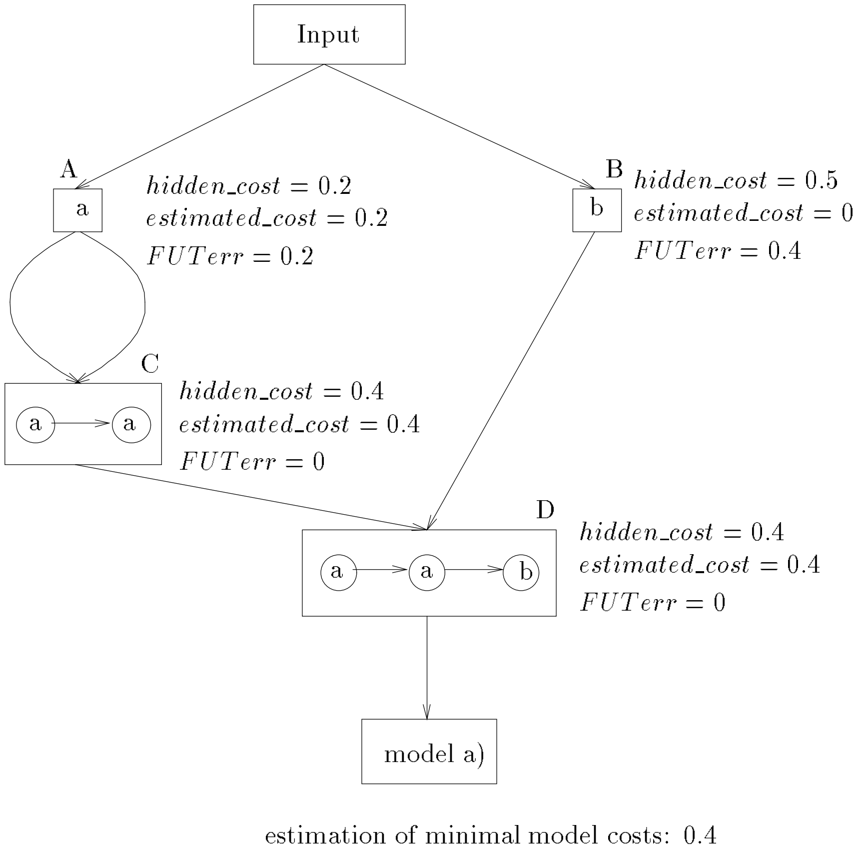


Figure 11: Assignment of the minimal costs and the future error

The lookahead procedure (1) takes its place in INA immediately after the termination of ÑA. OPEN must then be rearranged (2) according to the new cost values. Furthermore, each new insertion to OPEN (3),(4) must take into account the $FUTerr$ costs of the respective node.

The effect of the lookahead is best illustrated by going through the example in Fig. 8 of the previous section another time. Fig. 11 illustrates the assignment of the future error costs in the network from Fig. 8c based on the minimal estimated costs for each node. The tracing of the instances is given in table 3. The comparison of table 3 with table 2 shows, that the [\$] instance in node B is not removed from the STOP memory and sent to the successor node D, as it was the case in the INA without lookahead. The $FUTerr = 0.4$ of node B is added to the costs $C(\$) = 0.6$ resulting in total costs for this partial instance of 1.0. As the optimal distance between the model and the input is 0.9, the instance [\$] is never activated. As a direct consequence, the instances [12\$] and [\$1\$] in the STOP memory of D in table 2 do not appear in table 3.

4.5.3 Dynamic lookahead with updates

The main drawback of NLA as it was proposed so far is the fact that only information that was gathered in the first run of ÑA is used in order to predict the distance between a

	A		B		C		D	
	GO	STOP	GO	STOP	GO	STOP	GO	STOP
1		[1]		[1]				
2		[2],[1]	[2]	[1]				
$		[2],[1], [$]	[2]	[1],[$]				
X	$FUTerr = 0.2$		$FUTerr = 0.4$		$FUTerr = 0$		$FUTerr = 0$	
A	[2]	[1],[$]	[2]	[1],[$]				
A	[2],[1]	[$]	[2]	[1],[$]		[12],[21]		
A	[2],[1],[$]		[2]	[1],[$]		[12],[$1]..		
D	[2],[1],[$]		[2]	[1],[$]	[12]	[$1]..		
B	[2],[1],[$]		[2],[1]	[$]	[12]..	[$1]..		
D	[2],[1],[$]		[2],[1]	[$]	[12],[$1]..	[$$]	[$12]	[1$2]..

Table 3: Local memory contents for algorithm INA with lookahead. Total number of instances: 17

subgraph of a node and the input graph. From the description of the lookahead procedure we know that the *estimated_cost* of a node is greater than 0 only if the GO memory of the node is empty. However, during the inner loop of INA new instances are activated and generated and some GO memories receive their first entry. Due to Lemma 3.1 we know that the first instance t entering the GO memory of node i is equivalent to the best possible match between the subgraph in the node i and the input. Because *estimated_cost* is a lower bound of the distance between this subgraph and the input we know that $C(t) \geq estimated\_cost$ is always valid. Naturally, $C(t)$ itself is a lower bound on the distance and therefore we may update all *estimated_cost* values for the successor nodes of i. As a consequence, the $FUTerr$ values will also be updated, resulting in a more accurate estimation of the future errors for each node. Formally, the condition for an update of the estimated distances is:

Condition 4.4: Given an E-subgraph-checker n, assume that the attached procedure is called and a new instance t is generated. Then *estimated_cost* may be updated if

1. the GO memory of node n is yet empty,
2. $C(t) \leq Threshold$, therefore t is inserted into the GO memory,
3. $C(t) > estimated\_cost$

□

With the above condition satiesfied for a node n, we may set

$$estimated\_cost = C(t)$$

and call $distEstimationSubgraphChecker$ for all successor nodes of n. Then the $FUTerr$ values are adjusted by restarting the bottom up traversal.

Again, we demonstrate the effect of the lookahead with updates with the network from Fig. 8c and the input graph in Fig. 8b. The local memory contents are given in Table

	A		B		C		D	
	GO	STOP	GO	STOP	GO	STOP	GO	STOP
1		[1]		[1]				
2		[2],[1]	[2]	[1]				
$		[2],[1], [$]	[2]	[1],[$]				
X	$FUTerr = 0.2$		$FUTerr = 0.4$		$FUTerr = 0$		$FUTerr = 0$	
A	[2]	[1],[$]	[2]	[1],[$]				
A	[2],[1]	[$]	[2]	[1],[$]		[12],[21]		
A	[2],[1],[$]		[2]	[1],[$]		[12],[$1]..		
C	[2],[1],[$]		[2]	[1],[$]	[12]	[$1]..		
X			$FUTerr = 0.5$					
C	[2],[1],[$]		[2]	[1],[$]	[12],[$1]..	[$$]	[$12]	[1$2]

Table 4: Local memory contents for INA with updates. Total number of instances: 15

4. Compared to Table 3, which shows the memory contents for INA without updates, the total number of generated instances for INA with updates has shrunk to 15. The rows marked with X show at which point of INA the $FUTerr$ values are set and updated. After moving the instance [1,2] with costs 0.5 from the STOP memory of node C to the GO memory, the *estimated_cost* of C may be increased from 0.4 to 0.5. The $FUTerr$ of B is then updated to 0.5, accordingly. The instance [1] with costs 0.5 in the STOP memory of node B is now successfully kept from being moved into the GO memory, because along with the $FUTerr$ of 0.5, the costs of [1] are 0.5+0.5=1.0, and therefore exceed the costs 0.9 of the optimal solution. As a consequence, the instances [2$1] and [$21], are never generated in node D. We conclude that INA with lookahead and dynamic updates promises the best results. Therefore, in the rest of this paper we assume that INA is always implemented along with the dyamic lookahead.

The proposed lookahead procedure greatly reduces the number of generated instances in the INA and preserves optimality. Unlike the lookahead procedures for A^*, the network lookahead uses actual matching informations in order to estimate the minimal error for a certain subgraph and hence it is a fundamentally new technique. Also, it works for several models simultaneously, and is able to deliver a rough estimation for the total distance between the model and the input graph at the very beginning of the search for an inexact subgraph isomorphism. With every update that happens during the inner loop of INA, the distance estimations move steadily towards the optimal distance of the model. Most real world applications will define acceptance thresholds for the inexact subgraph isomorphism between a model and an input and thus will be able to take the distance estimations as a basis for an early cut off of the search.

The computational complexity of the network lookahead is only dependent on the number of nodes N in the network. In the beginning, each node is visited twice and OPEN must be resorted exactly once resulting in a complexity of

$$O(2 * N + N * ld(N)) = O(N * ld(N))$$

During the inner loop of INA each node may become the source of a full update cycle

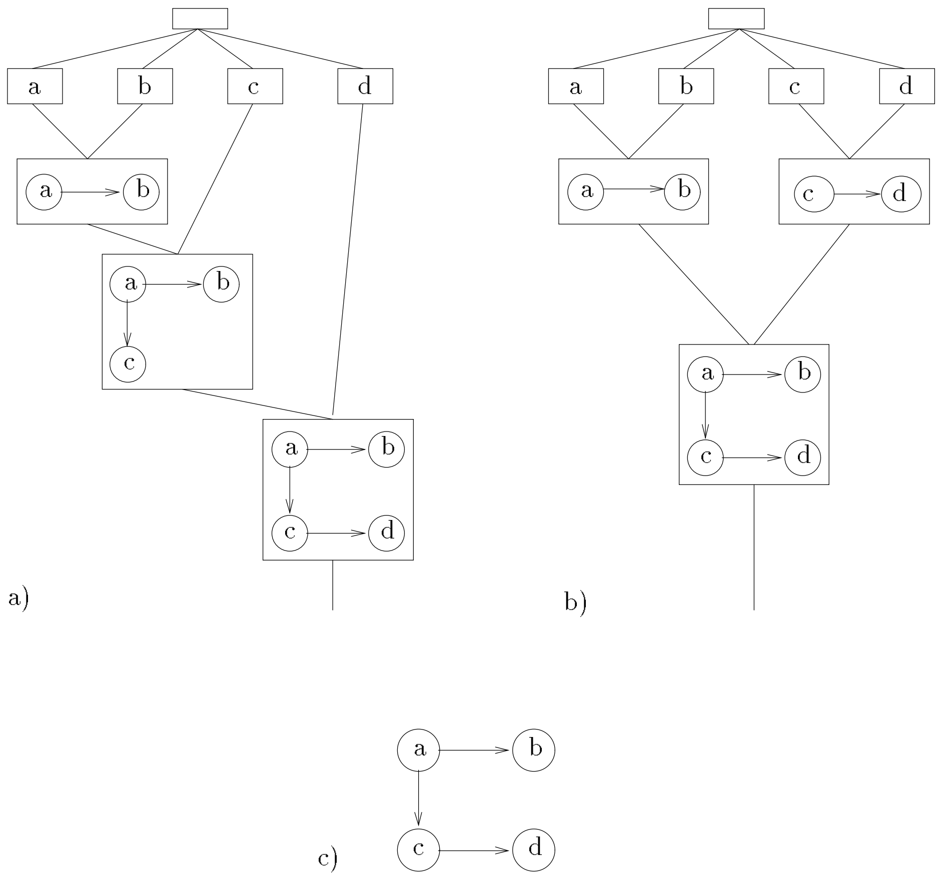


Figure 12: A *linear* a) and a *balanced* b) network for graph c)

exactly once. The overall complexity for the lookahead with updates is then

$$O(N * N * ld(N)) = O(N^2 * ld(N))$$

4.6 Compilation of the network

The compilation of a set of models into a network is not trivial. The structure of the network depends on the segmentation of the model graphs. Considering that for the first level of segmentation of a graph, where the set of n vertices is divided into two disjoint subset, there exist already $2^{n-1} - 1$ possibilities, it is obvious that the number of possible networks for a set of models is exponential in the number of vertices in the graph. In Fig. 12 two possible networks for the graph in Fig. 12c) are depicted. We call the network in a) *linear* and the network b) *balanced*. The influence of the network structure on the performance of the subgraph isomorphism algorithm is remarkable. For example, if there is an input graph identical to the graph in c) except that the vertex labeled 'a' is mislabeled 'e', then the linear network will not create any subgraph instances with more than 1 vertex, while the balanced network will still find an instance for the subgraph spanned by the vertices c and d. In general, the balanced network performs better, creates less instances and allows a more efficient lookahead in INA. If the database contains several models then we are mainly interested in sharing as much subgraphs as possible. However, there is no optimal compilation technique. Consider the problem where the database consists of three graphs G_1, G_2, G_3, where G_1 contains a subgraph A, G_2 contains a subgraph B and G_3 contains both A and B, but with A and B being not disjoint in G_3. Thus, we either create a node for A and segment G_3 into A and a rest or we create a node for B and segment G_3 into B and a rest. The decision must be left to application specific preferences and heuristics. In the following we list some compilation strategies, which try to optimize a special feature and can be used as guidelines:

- The maximum common subgraph of all the models is extracted and represented as a network node.
- The maximum common subgraph of each model *pair* is extracted and represented as a network node.
- The models are assigned priority values and compiled sequentially into the network. Each model graph tries to use as much as possible of the structures that are already represented in the network.
- A model is partitioned into subgraphs of roughly the same size. This will guarantee, that the network is fairly balanced.

For this paper we chose to compile the graphs in random order each graph using as much as possible of the exisiting network. As a second priority, we try to keep the network as balanced as possible.

We first explain the compilation of a new model graph G under the assumption that all labels appearing in the new graph are already repesented by some l-vertex-checker. The graph G is now treated as an input graph and processed by NA. After termination of NA, the local memories contain instances of subgraphs of G. Due to our assumption, each vertex of G appears in at least one instance. In the first step, we determine the node n, which represents the largest graph with respect to the vertices and has one or more instances t_i in its local memory. Out of the set of these instances $S = \{t_i\}$, we choose the largest set $SD = \{t_i | t_i \in S\}$ such that $t_i \cap t_j = \emptyset$ for all $t_i, t_j \in SD$, i.e. the instances are all disjoint. The node n therefore represents the disjoint subgraphs $t \in SD$ of the new graph G. All other instances of the node n, which do not belong to SD are deleted. Also, instances in the rest of the network containing a vertex $v \in t$ with $t \in SD$ are removed. This step is repeated until the network contains only disjoint instances and each vertex of the new model appears in exactly one instance. In the next step the nodes containing one or more instances are collected in a list and sorted according to the number of vertices of the represented graphs. By sorting them increasingly, the nodes representing small graphs come first while the nodes representing large graphs are last in the list. Now we merge the nodes into new network nodes, always choosing the nodes from the front of the sorted list. The resulting network will be more or less balanced. Two nodes n_l and n_r may become parents of a new node n if they satisfy a simple condition: There must be an instance t_l in n_l and an instance t_r in n_r such that the set E' of edges connecting t_l with t_r is not empty. The new E-subgraph-checker n is created and added to the list of successors of both parent nodes. The set of edges E' is transformed into the list E belonging to this E-subgraph-checker. Note, that the parent nodes n_l, n_r may be identical, $n_l = n_r$. Having succesfully created a new node n with parents n_l and n_r we combine all instances t_l in n_l and t_r in n_r and generate a new instance $t = t_l + t_r$ if the edges specified in E exist between t_l and t_r. Instances, that are combined succesfully are removed from their parent nodes. Thus, after a new E-subgraph-checker is created and its local memory is updated, the parent nodes may or may not contain instances. If they do, they are again considered as parents of a another new E-subgraph-checker. This step is repeated until there is only one E-subgraph-checker en in the network, which contains exactly one instance in its local memory. This node en represents the new graph G, hence we generate a G-model-node for G and add it to the list of successors of en. We call the procedure, which performs the above steps $addToNetwork$ and pass is the graph G that is to be compiled:

procedure addToNetwork(graph G)
  NA(G);
  /* Step 1 */
  sort nodes according to the size of the graph they represent, large graphs first.
  For all nodes n with non-empty memories
    Choose largest possible set of disjoint instances $SD = \{t_i\}$,
    where t_i are elements of the local memory of node n;
    remove all instances in node n, which do not belong to SD;
    remove all instances in the other nodes of the network,
    which contain a vertex v, with $v \in t$, for a $t \in SD$
  End For
  /* Step 2 */
  sort all nodes containing at least one instance according to the size of the graph
  they represent in the list L, small graphs first;
  While more than one instance is left in the network
      take first node n_l in L;
      /* let t_l be the first instance in n_l */
      For all nodes n_r in L
        For all instances t_r in n_r
          If the set of edges E' connecting t_l and t_r is not empty **Then**
            call createSubgraphChecker(n_l, n_r, E');
            Connected=TRUE;
          End If
        End For
      End For
      If Connected is FALSE **Then** ErrorExit: G is not a connected graph!
  End While
  create G-model-node for G
  add G-model-node to the list of successors of the last node in L.

procedure createSubgraphChecker(n_l, n_r, E')
  create new E-subgraph-checker n;
  make n_l and n_r parents of n;
  transform the set of edges E' into the edge specification list E;
  For all instances t_l in n_l
    For all instances t_r in n_r
      If the edges in E exists between t_l and t_r **Then**
        $t_{new} = t_l + t_r$;
        store t_{new} in the new E-subgraph-checker n;
        remove t_l and t_r from the nodes n_l and n_r, respectively
      End If
    End For
  End For
  If n_l has empty memory **Then** remove n_l from L;
  If n_r has empty memory **Then** remove n_r from L;
  insert n into L

Now we are ready to consider the compilation of a graph under no special assumptions. If we comile a graph G into an existing network and there are vertex labels in G, which do not have a corresponding l-vertex-checker, then for each such label in G, we create a new l-vertex-checker and add it to the successor list of the input-node. If, furthermore, no graph has yet been compiled and no network exists, we first create a unique input-node and then start creating the l-vertex-checkers. Afterwards we simply call the procedure $addToNetwork$.

procedure compile(graph G)
  If no network exists **Then**
    create the input-node
  For each label l in G
    If there is no l-vertex-checker **Then**
      create a new l-vertex-checker and add it the successor list of the input-node
  End For
  call addToNetwork(G)

In Fig. 13c) we follow the network compilation of model 13b) under the condition that model 13a) has already been compiled. First, an l-vertex-checker for the label 'd' is created. Then, model 2 is inserted into the network and all partial instances are stored somewhere in the network. Step 3 identifies the largest possible sets of disjoint instances $\{[1,2],[4,3]\}$ and removes all other instances. Finally, step 4 and 5 create new E-subgraph-checkers, by first merging the instances $[1,2]$ and $[4,3]$ into the instance $[1,2,4,3]$, generating an edge test, which will only merge instances if there exists an edge between the first vertex of the left instance and the second vertex of the right instance. Then $[1,2,4,3]$ is merged with the singleton $[5]$ creating a E-subgraph-checker with an edge test from the third position in the left instance to the first position in the right instance. The instance $[1,2,4,3,5]$ represents the full model 13b) and compilation ends with the creation of a m-model-node for graph in Fig. 13b.

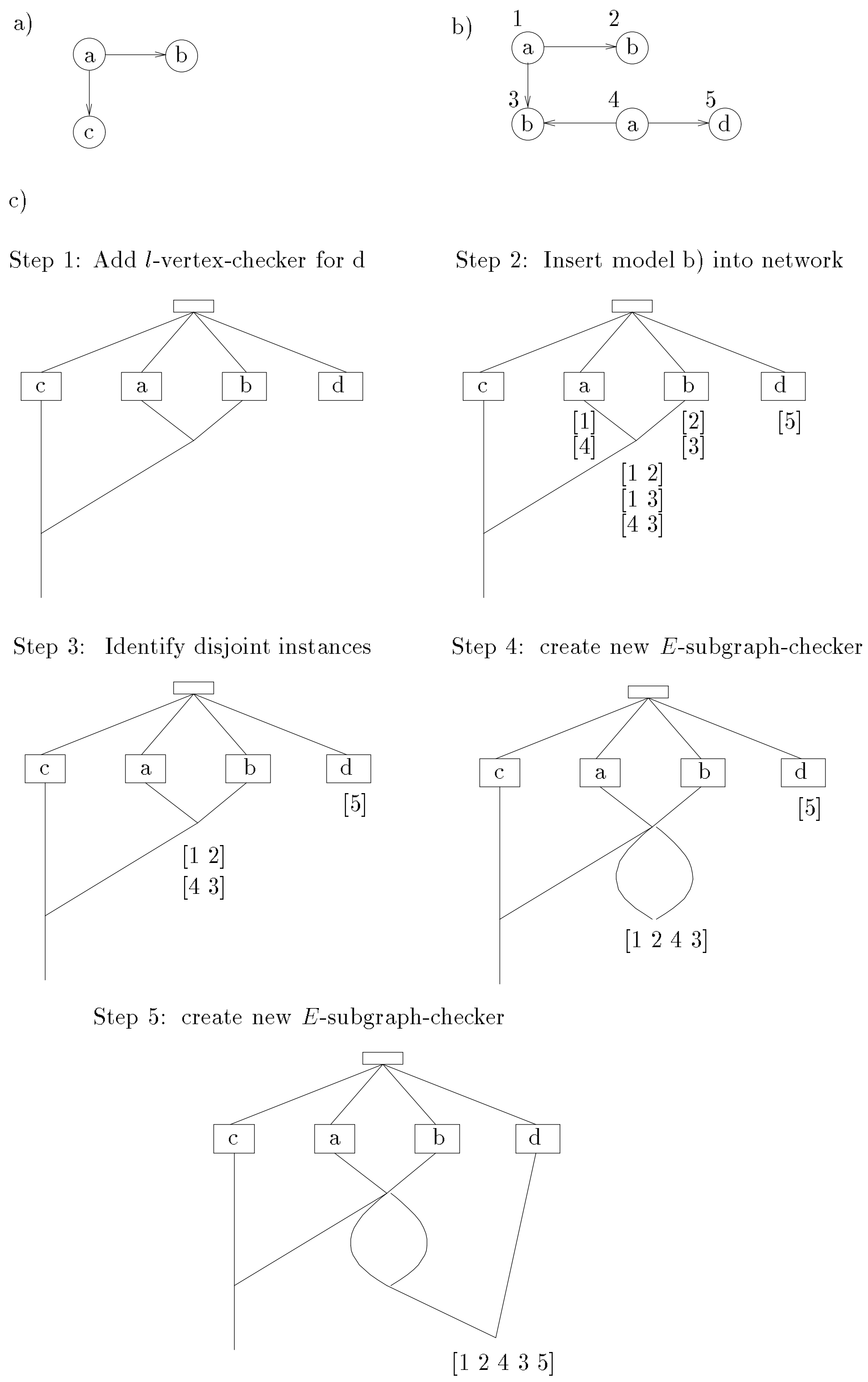


Figure 13: a) a model, b) another model, c) compilation of model b) into the network representing model a)

The compilation of the model database into a network is done off-line. Therefore we may invest as much computation time as necessary. An interesting feature of the proposed compilation algorithm is the fact that after the compilation of the first model graph, the network is in a state ready for subgraph isomorphism detection. The database of models does not have to be complete from the beginning. An application might start out with a limited set of models and whenever an unkown object is encountered, the respective graph can be added to the database and compiled into the network. Without changes to the old structures, the network can *learn* about a new model even after it has been operable for some time.

5 Comparative study on the network approach

In this section we will compare the behavior and the features of the exact and inexact network algorithm to the traditional tree search algorithms. First, we look at the complexity of the subgraph isomorphism in general and then we study the differences in the computational complexity between the traditional algorithms and the new network based algorithms.

We adjust our notation such that for a graph $G = (V, E, L, A, \mu, \nu)$, V denotes both the set of vertices and the cardinality of this set, E denotes the set of edges and its cardinality, and L is the set of labels and the cardinality of this set. The theoretical complexity of the subgraph isomorphism problem for two graphs $G = (V, E, L, A, \mu, \nu), G' = (V', E', L, A, \mu', \nu')$ is then a function

$$f_O(V, V', E, E', L).$$

It is well known that the problem is NP-complete [GJ79]. This worst case emerges, when $L = 1$, $E = V^2$, $E' = E$ and $V' = V$. Any algorithm will then require at least

$$O(V'^V)$$

steps to terminate, because each vertex in V can be assigned to each vertex in V'. On the other hand, there is a best case if the set of labels is $L = V$, i.e., each vertex in is uniquely labeled, and the set of edges is $E = V - 1$. Then a subgraph isomorphism can be detected in

$$O(V'^2)$$

steps.

Clearly, the set of edges and the set of labels are dominant factors in the complexity of the graph matching problem. Currently, it is assumed that any algorithm designed for finding an optimal subgraph isomophism between two arbitrary graphs will become computationally intractable in the worst case. Both, the network approach NA and the tree search TS belong to this class of optimal algorithms. In the worst case NA takes

$$O(EE'V'^V)$$

steps compared to

$$O(EV'^V)$$

of TS. In the best case, NA terminates in

$$O(LV' + V^2)$$

with $L \leq V$, while TS requires

$$O(V'^2)$$

steps.

In the following we briefly study the influence of changes in L and E on the performance of TS and NA. A network compiled for a model graph with L different labels in the vertices, will contain exactly L l-vertex-checkers. Thus, the tests peformed in the first level of the network are of the complexity

$$O(LV)$$

and decrease with the number of labels. On the other hand, with less labels in the graph, the number of ambiguities, i.e. the number of instances of a subgraph of G in G', increases. Many of these instances are not part of the final solution, but because NA is based on the idea of first finding instances of subgraphs of a model and merging them later, they are nevertheless generated. In the search tree of TS, many of these instances are suppressed, because there is a fixed order of assigning model vertices onto input vertices, for example, if the very first vertex cannot be assigned to any input vertex, then the search is cut off prematurely and none of the partial instances found by NA will by found by TS. The experiment 1 in section 7 will illustrate, that the reduction of the label set first results in an improvement of the performance of NA until the number of ambiguities becomes too big and the performance decreases rapidly.

The influence of E and E' on NA is similar. Clearly, the more edges there are in a model graph, the larger the number of instances of subgraphs of this model graph in the input graph. Again, this results in a increased number of instances in the network nodes, while TS reacts less sensitive, due to its fixed order of vertex assignment. Experiment 3 of section 7 will illustrate how the number of edges in a model graph influences the performance of both NA and TS.

So far, we have only studied conditions, which can be easily described by the factors V, V', E, E', L. However, there are three more features, which mainly distinguish our new approach from the traditional algorithm, namely, the sharing of parts inside the same model, the sharing of parts among different models and the new network lookahead technique. In the following subsection we study each of these features separately.

5.1 The influence of sharing identical substructures inside a model

The main problem with any tree search approach to graph matching is the fact, that partial matchings are only known to all those nodes in the search tree which are direct descendants of the node with the partial matching. In other words, if there exists an instance t of the subgraph S of model G in the input graph, then t will possibly appear multiple times inside the search tree. TS does not remember t and therefore must instantiate it every time again.

The network approach however contains exactly one structure for the subgraph S and it will find every possible instance of S in the input exactly once.

If S appears several times in G, then TS will in the worst case perform a number of steps which is far greater than that of NA. Fig. 14 illustrates the different search strategies. There are 6 possible isomorphisms between the model in Fig. 14a and the input in Fig. 14b. Both graphs consist of three identical subgraphs S, which are connected to each other by a single undirected edge. The network in Fig. 14c is given after the insertion of the input graph. The node A is a symbolic node, denoting the subnet for the graph S. There are exactly 3 instance of S found by A. The nodes B and C simply combine these instances with an expense that is quadratic in the number of vertices in S. On the other hand, the search tree in Fig. 14d that is expanded by TS, generates each of the 3 instances 5 times, resulting in an overall number of 15 instances of S. Generally, if we assume that a graph $G = (V, E, L, A, \mu, \nu)$ consists of k disjoint and isomorphic subgraphs with R vertices each, $V = k * S$, then TS will find

$$k! + \frac{k!}{2!} + \ldots + k$$

instances of S, if the input graph G' is isomorphic to G. If $f_o(S)$ denotes the computational costs of finding an instance of S in G' then TS will perform

$$O(k^k f_o(S))$$

steps. Additionally each instance must be embedded into the full model match under compliance with the edge constraints, which can be tested in quadratic time, resulting in a total complexity of

$$O_{TS} = O(k^k f_o(S) + k^k S^2)$$

NA on the other hand finds k instances of S at the expense of

$$O(k f_o(S))$$

and stores them in the network. NA will then generate all possible combinations of the partial instances resulting in a set of k^k permutations each calculated in

$$O(S^2)$$

steps. The total complexity of NA is then given by

$$O_{NA} = O(k f_o(S) + k^k S^2)$$

Both the complexities O_{TS}, O_{NA} are inherently exponential due to the factor $f_o(S)$ which in the worst case becomes $f_o(S) = O(S^S)$ and the term k^k. However, for $k > 1$ we observe:

$$O_{TS} > O_{NA}\ , \qquad k > 1$$

5.2 The influence of sharing identical structures among different models

One of the problems of databases for graphs is the redundancy that arises when each model is stored individually. Most applications will deal with objects which are not completely distinct but share certain features. Any graph matching algorithm which is designed to

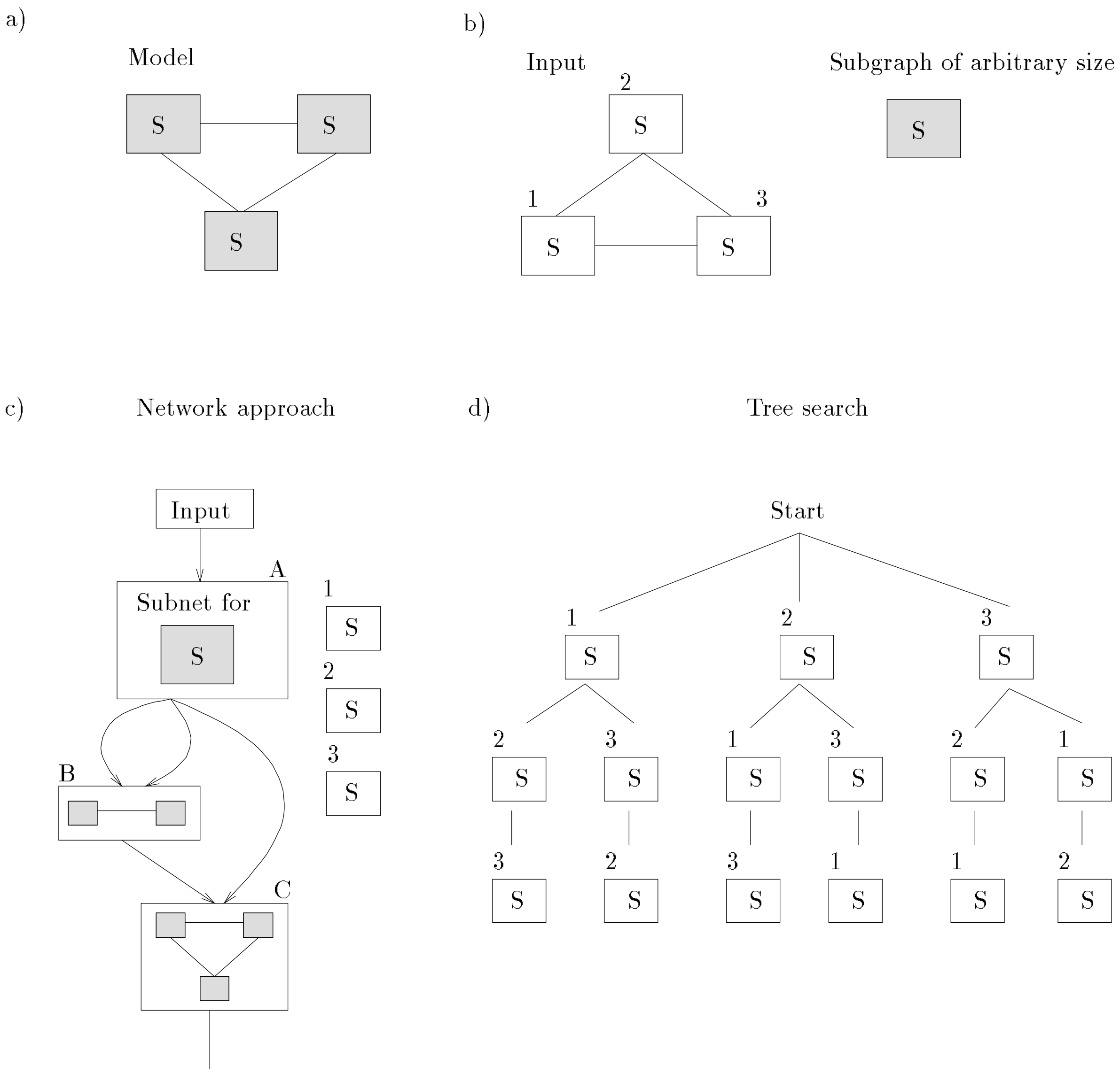


Figure 14: Instances of S generated in NA and TS

detect subgraph isomorphisms between two graphs, will have to be applied sequentially over the model database finding shared features multiple times. If the database consists of M models and each model contains the subgraph S then TS will cause

$$O(Mf_o(S))$$

steps for the detection of the subgraph S of each model in the input graph. On the other hand, the ability of the network approach to share common structures among different models prevents the subgraph S from being found M times. NA will detect S exactly once at the expense of

$$O(f_o(S))$$

and then distribute the partial match for S over the model database.

We conclude, that the compact network representation improves the time performance of the recognition process. Furthermore, if in the limit the size of the common subgraph of the models approaches the size of the models then the detection algorithm becomes independent of the number of models.

5.3 The influence of the network lookahead

The features studied so far are both valid for the exact as well as the inexact subgraph isomorphism problem. The computational complexity of the inexact subgraph isomorphism problem is in the worst case also exponential in the number of vertices of the model and the input graph. Of course, the best case results, when there is no noise in the input graph and the best case conditions of the exact subgraph isomorphism hold. Obviously, the problem can then be solved in quadratic time as well. In the following we study more closely the influence of the network lookahead on the performance of our new algorithm.

The main advantage of the network lookahead NLA compared to lookahead procedures for the tree search algorithm is its ability to estimate the minimal error of a subgraph on the grounds of an actual matching. The lookahead procedures that were proposed in the past are all based on a kind of best first matching of the vertices of the model graph on vertices of the input graph, taking the cheapest possible match as an indication for future errors. However, structural errors in a subgraph cannot be detected in this manner unless the subgraph is matched according to all edge and label constraints as it is done during the first run of $\tilde{\text{N}}$A within INA. To examine the effect of the proposed lookahead more closely consider the following problem setting:
Given a model $G = (V, E)$ and an input $G' = (V', E')$ with $E = E', |V| = |V'|$. Let the vertices be uniquely labelled from 1 to n. The cost function for substitution is defined as the absolut difference between to labels. The labels of δ input vertices are distorted and they can be matched onto exactly one model vertex with cost 1. Thus, the total cost for an inexact match between G and G' is δ.

If we assume that $(v_1, v_2, \ldots, v_n)$ is the order of model vertices which is used to construct the search tree in TS and $(v'_{n-\delta}, \ldots, v'_n)$ are the distorted vertices in the input graph, then the structure of the search tree will look like in Fig. 15. Each node in the tree gets assigned the current costs of the partial match. First, every input vertex i is assigned to the model vertex 1, with increasing costs $i - 1$. Next, below the first level, TS will expand all nodes with current costs being less than δ. If $f_\delta(V)$ is the function calculating the total

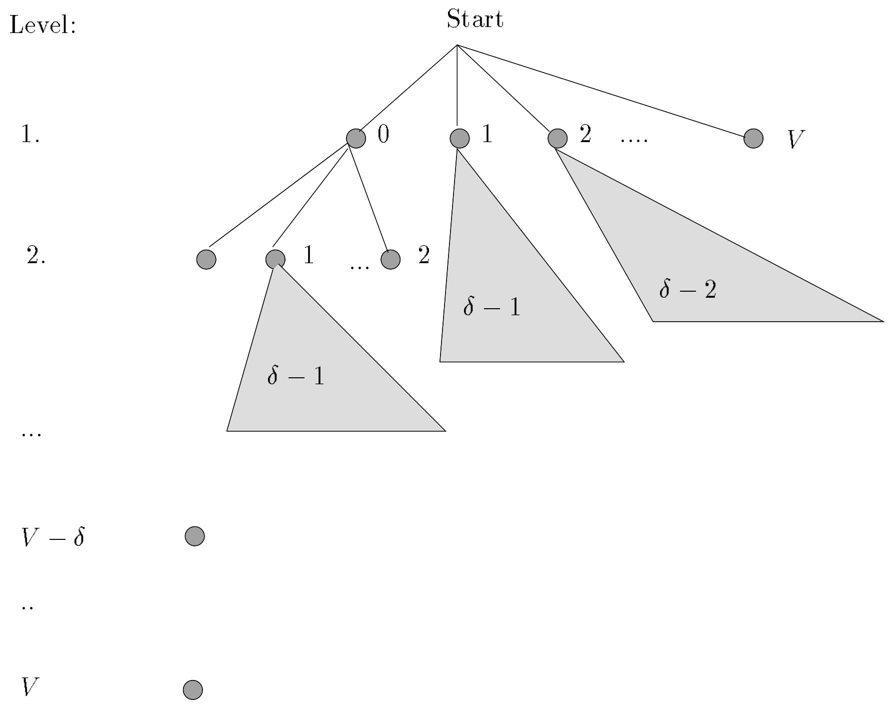


Figure 15: Subtrees expanded by TS if the minimal distance between model and input is δ

size of the search tree for error δ and input size V, then the subtree developed below the first level node with cost 1 is of size $f_{\delta-1}(V-1)$. The subtree below the node with costs 2 is of the size $f_{\delta-2}(V-1)$. On the second level, the subtree below the node with cost 1 is $f_{\delta-1}(V-2)$ and so on. The subtrees are displayed in Fig. 15 as shaded triangles. f can then be written as a recursive formula, consisting of the number of the nodes in the subtree which leads to the succesful match, $O(V^2)$, and sizes of the subtrees that are expanded on the levels 1 through $V-\delta$ for decreasing costs:

$$f_\delta(V) = O(V^2) + \sum_{j=1}^{V-\delta} \sum_{d=0}^{\delta-1} f_d(V-j)$$

For $\delta = 0$ the size of the tree is the expected $O(V^2)$, however, if $\delta = \frac{V}{2}$ the formula tends toward $O(V!)$ describing the maximum number of possible nodes.

On the other hand, the proposed network lookahead procedure is able to restrict the instance movement to the necessary minimum. In Fig. 16 the l-vertex-checkers for the vertices $v_1, \ldots, v_n$ are given. The E-subgraph-checkers are omitted for clarity. The STOP memory of the l-vertex-checkers representing $v_1, \ldots, v_{V-\delta}$ are never activated due to the future error value of δ which has been assigned to them in the second phase of the lookahead. Therefore, the partial instances, which cause TS to explode in terms of size, are never instantiated in INA. The theoretical complexity of INA with regards to the stated problem is thus independent of the number of distorted labels in the input and can be given as the sum of the best case complexity of ÑA and the lookahead complexity

$$O(V^2 + ld(V)V^2) = O(ld(V)V^2).$$

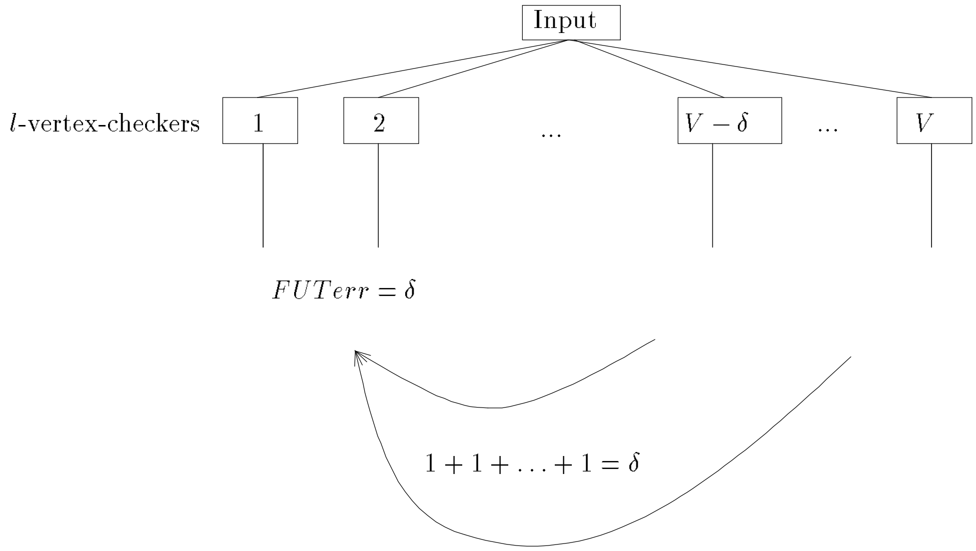


Figure 16: Lookahead cost assignment

Compared to the complexity of TS, which tends towards exponential growth with increasing noise, INA is almost independent of the amount of noise in the input under these special circumstances. Experiment 9 in section 6 illustrates this behavior.

The functionality of the lookahead is not restricted to single model applications, but works as well on a network, which incorporates several models. Fig. 17d illustrates how by looking ahead over the network certain models can be cut from the search path early. The network is compiled for the models in Fig. 17a and 17b. The GO and the STOP memories are given after the termination of ÑA. The lookahead assigns each node a FUTerr value, which appear circled in Fig. 17d. The cost for the insertion of an edge are set to 2. The cost for the substitution of a label are given in the following table:

	a	b	c	d
a		∞	1.5	1.5
b	∞		∞	∞
c	1.5	∞		1
d	1.5	∞	1	

The optimal inexact subgraph isomorphism can be established by deleting the edge between the vertices labeled 'a' in the model 17a at cost 2. Therefore, no instance with higher cost will be moved in the network. Due to the fact that the c-vertex-checker has a $FUTerr$ of 1.5, the instance [4] with costs 1.0 will not be sent to the node D, thus preventing unnecessary computations. The first STOP memory to be triggered is the one of the a-vertex-checker activating the instance [4]. Next, the STOP memory of node A containing the instance [1,2] with cost 2 is triggered and the optimal solution [1,2,3,4] with costs 2 is found. We conclude that the lookahead succesfully limits the steps in the network even if several models are involved.

Of course, INA also has some disadvantages. If the set of labels is relatively small compared to the graph size, then the model and the input graph will have many small

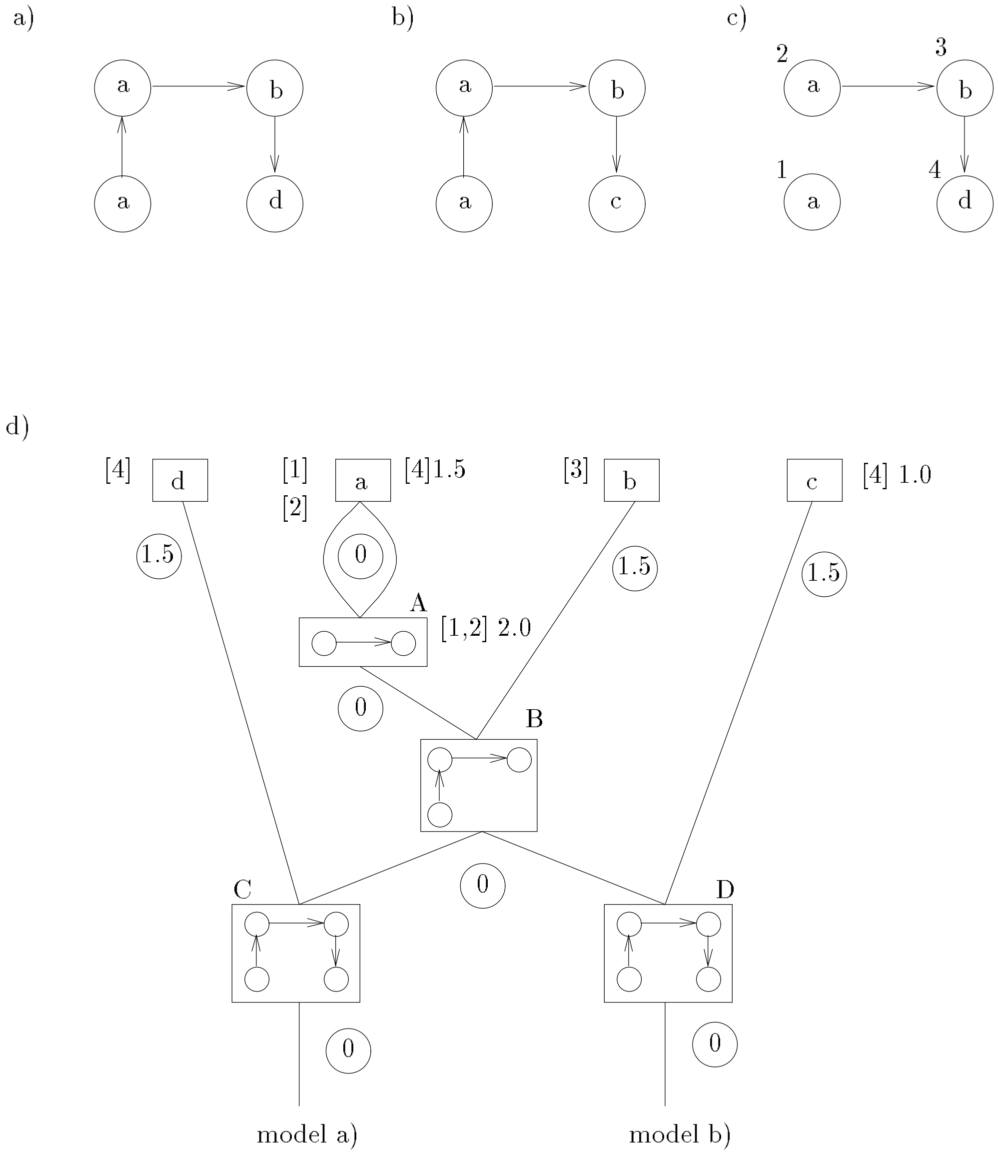


Figure 17: Lookahead over several models

subgraphs in common, even if the input is heavily distorted. The lookahead in INA will no longer be efficient, because there will be a lot of low cost partial instances, making it impossible to anticipate correctly the future errors. Another problem arises when a model graph contains many identical subgraphs. Although this is a desirable situation in the exact case, where NA performs far better than TS, in the inexact case the performance decreases rapidly, because if only one of several instances of a subgraph is existent in the input graph, then the lookahead can no longer detect a future error. Because the identical subgraphs are represented by the same network node, one single partial instance covers all actual errors and thus makes the lookahead useless.

This section has described how the three major features of the network approach can greatly reduce the computational requirements of any graph matching application. How effective their influence is in practice can be gathered from the experiments documented in the next section of the paper.

6 Experimental results

To examine more closely the practical performance of the proposed algorithms NA and INA, they have been implemented in C++ and run on a SUN Workstation. The testing environment consists of a graph generator and two traditional graph matching algorithms for comparison. In the case of the exact subgraph isomorphism problem we use Ullman's algorithm as a benchmark and in the inexact case the algorithm A^* with future costs set to zero is applied. The sample graphs are generated on the basis of the number of vertices and the edge connectivity which is assumed to be normally distributed with given average and variance. A single label with no attributes is assigned to each vertex and each edge. The labels are integers within a range that can be specified as a testing parameter. If the number of labels equals the number of vertices in the graph, then each vertex will be uniquely labelled, otherwise, if $\alpha = \frac{number of vertices}{number of labels}$ then an average of α vertices will be labelled identically.

The cost of a label substitution is defined as the absolute difference between the corresponding integer values. Insertion and deletion costs for graph vertices and edges are identical and constant. They can be set as a test parameter and are normally equal to the number of labels, i.e, if the label set contains 10 different labels, then the cost for an insertion or deletion of a vertex or an edge is 10. Thus the substitution of a vertex labelled l with a vertex labelled o is generally cheaper than the insertion or the deletion of this vertex.

Noisy input graphs are generated by first creating a model graph and then distorting its vertices and edges according to test parameters. The number of vertices whose labels are to be distorted can be varied as well as the number of vertices which are to be deleted from the input graph along with the incident edges. Label distortion is defined as adding one unit to the label integer and thus causing substitution costs of 1 for each distorted vertex in the input graph.

The results of each experiment are given by two line plots, one displaying the actual computation time in seconds and the other documenting the size of the search space in terms of expanded nodes for the tree search and the number of tokens for the network approach. While the number of expanded nodes documents how many partial instances are generated by the algorithm, it does not say anything about the total number of com-

Experiment 1:

model size: 60
connectivity: 2/1
10 repetitions per point

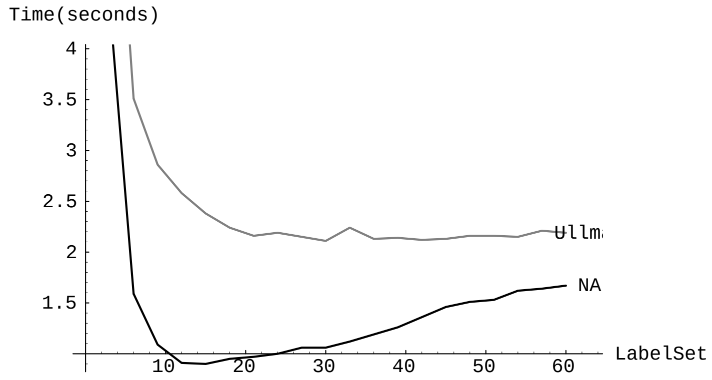


Figure 18

Experiment 1:

model size: 60
connectivity: 2/1
10 repetitions per point

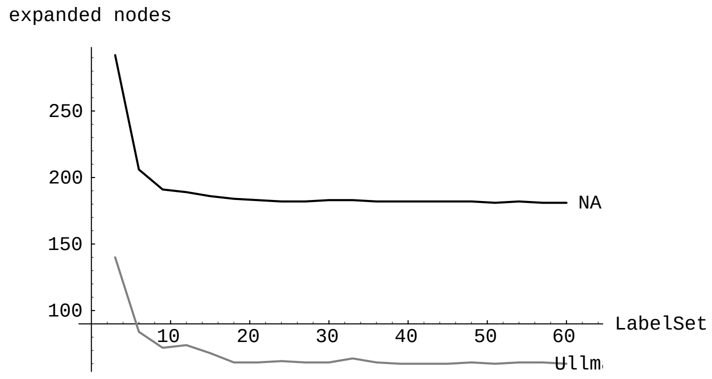


Figure 19

putation steps. Therefore, the time performance plot is more accurate and also accounts for any additional work done by the algorithm, such as Ullman's refinement procedure.

6.1 Exact subgraph isomorphism detection: NA versus Ullman's algorithm

The first experiment is to show the performance of Ullman's algorithm (UA) and NA under the influence of an increasing number of labels. We generate a single model graph containing 60 vertices. Each vertex has a degree which is normally distributed with average 2 and variance 1, therefore each vertex has an average of 2 incident edges. The input graph is identical to the model graph. The set of labels first contains only 1 label and is then increased until there are sixty labels available and each vertex is uniquely labeled. For each point in the plot, 10 samples were matched and the average time and space of the 10 runs is given. As expected, the performance of UA improves with an increasing number of labels, because less ambiguities are encountered. NA reacts slightly different, as it reaches

Experiment 2:

model size: 85
label set: 20
connectivity: 4/3
10 repetitions per point

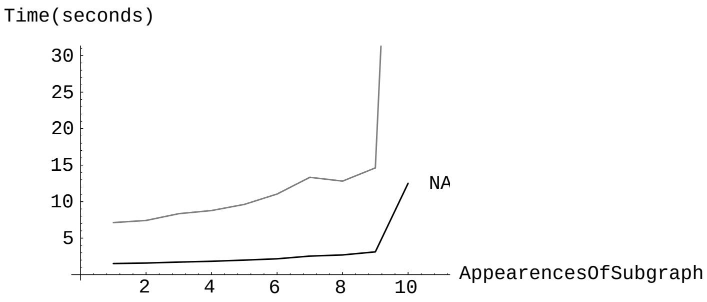


Figure 20

a minimum in time and space requirement when there are about 12 labels available and not when all vertices are uniquely labelled. This is because with everything being unique in the model, NA's feature of sharing identical parts is completely suppressed, while it shows some effect for a label set of 12.

The number of labels is a rough measure for the self-similarity of a graph and there is no real control over how many subgraphs in a model are identical. Therefore, the second experiment concentrates on directly varying the number of appearences of a subgraph in a model. We generate a single model graph containing 85 vertices and an average of 4 incident edges per vertex. There are 20 labels available. The input graph is always identical to the model graph. First, there is no subgraph which appears multiple times inside the model. Then, we generate the model such that it consists of a number of identical subgraphs of 8 vertices and 16 edges. The subgraphs are connected among each other by randomly generated edges. In the last test, the model consists of 10 identical subgraphs and 5 uniquely labeled vertices. The fact that UA does not remember previously found subgraph instances explains why its performance is getting worse with increasing number of appearences of the identical subgraph. NA however is less sensitive to the number of appearences of the identical subgraphs, because each subgraph is instantiated only once and not 10^{10} times as it may be in the worst case of UA, when the model contains 10 identical subgraphs. Again, each point represents the average value of 10 runs.

Note that in both experiment 1 and 2, the search space of UA is generally smaller than the search space of NA. This is due to the fact, that the discrete relaxation technique used in UA performs a forward checking and is able to cut off certain branches from the search tree early. However, the additional tests performed by UA need additional time, such that in general NA is faster.

As we explained in section 6, the network approach is not well suited to problems where the graphs are highly connected. In experiment 3 we increase the number of edges per vertex, starting at 1 and ending at 40. The underlying model graph consists of 50 vertices, while the label set contains 30 distinct labels. Thus the very last sample in the experiment gives the time and the space requirements for the match between two almost complete graphs. Clearly, NA is much more sensitive to the number of edges in a model graph and

Experiment 2:

model size 85
label set: 20
connectivity: 4/3
10 repetitions per point

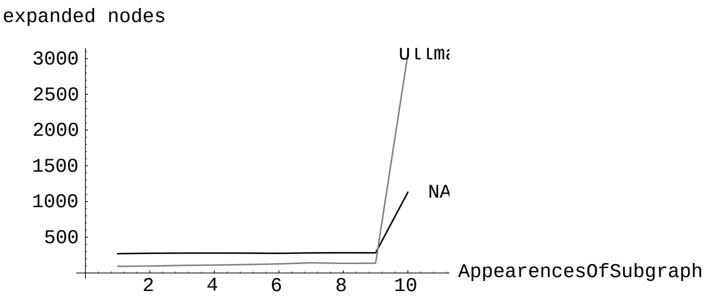


Figure 21

Experiment 3:

model size: 50
label set: 30
10 repetitions per point

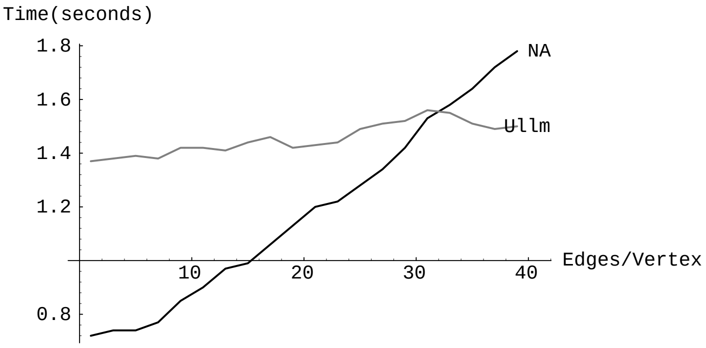


Figure 22

its time performance becomes gradually worse, while UA does not show a visible decrease in time performance.

So far, only problems concerned with a single model and an input graph have been examined. For a database with multiple models we expect NA to perform the better the larger the common subgraph of all the models becomes. In the extreme, where all model are equal, NA will be independent of the database size, while UA needs to perform a full match for each one of the models. In experiment 4, we generate a database of 15 model graphs, each containing 40 vertices. They all have a subgraph in common whose size increases from 2 to 35 vertices. Thus, the last sample will represent a database of 15 models each containing a common subgraph of 35 vertices and a unique subgraph of 5 vertices. Each of the 15 models is used as input and the average of the 15 matching processes is taken. Twice per point we generate a database according to the same parameters and give the average of the time and the space requirements in each point. While the performance of NA improves with a growing common subgraph, UA shows the inverse behavior and becomes worse.

Experiment 3:

model size 50
label set: 30
10 repetitions per point

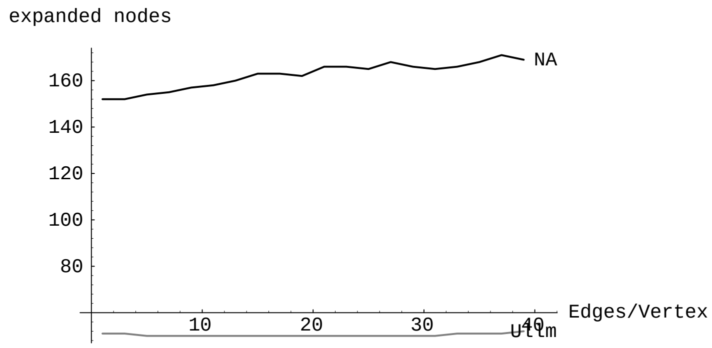


Figure 23

Experiment 4:

model size: 40
connectivity: 1/2
30 repetitions per point
database: 15

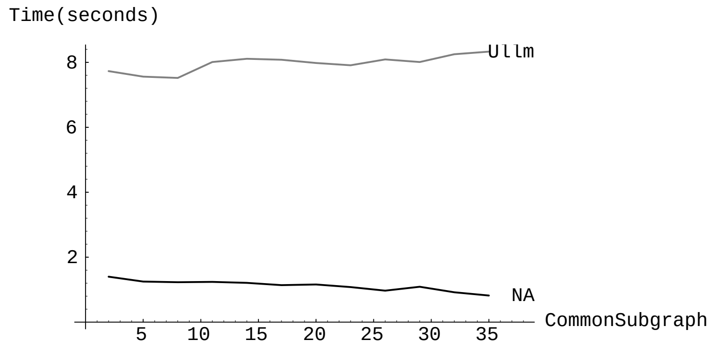


Figure 24

Experiment 4:

model size: 40
connectivity: 1/2
30 repetitions per point
database: 15

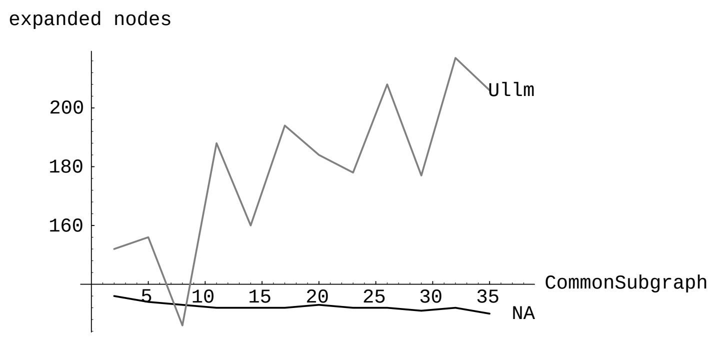


Figure 25

Experiment 5:

model size: 30
common subgraph: 20
connectivity: 2/2
1 repetitions per point

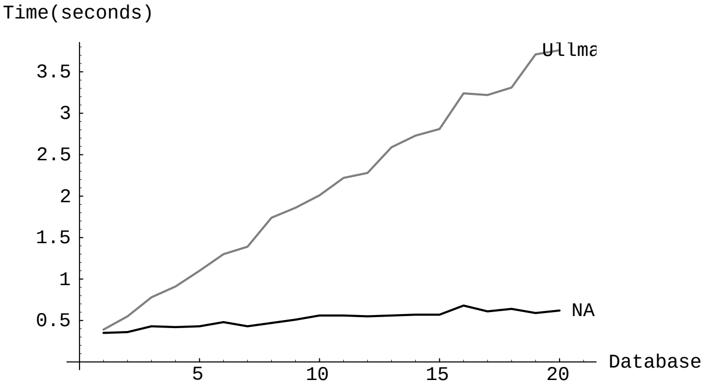


Figure 26

Experiment 5:

model size: 30
common subgraph: 20
connectivity: 2/2
1 repetitions per point

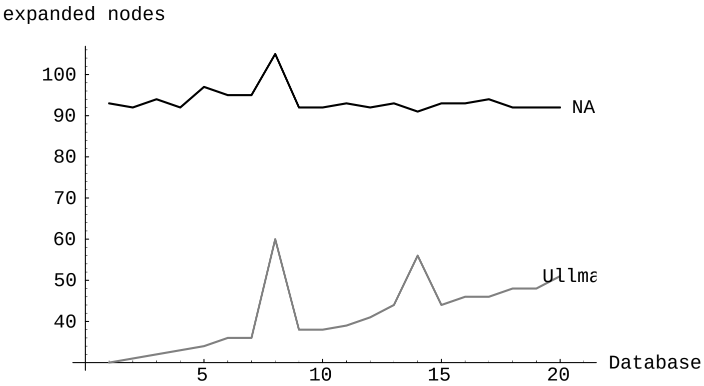


Figure 27

In experiment 5 we hold the size of the common subgraph constant at 20 vertices and the size of the models at 30 vertices. The number of models in the database is increased starting at 1 and ending at 20 models. We observe that for a database with 1 model, both UA and NA take about 0.5 seconds to terminate. With each model being added to the database the time of UA grows linearly, because the common subgraph is instantiated for each of the models in the database. NA on the other hand shows only a minor decrease in performance.

6.2 Inexact subgraph isomorphism: INA versus A*

In the following we compare the results of the inexact network algorithm INA to the A^* algorithm. The search in A^* is guided by the current costs of a partial match. We acknowledge that the number of nodes that are expanded by A^* can be reduced by applying a lookahead technique. However, most techniques for the estimation of an inexact subgraph isomorphism that have been proposed in the literature are strongly domain dependent and

Experiment 6:

model size: 30
label set: 30
connectivity: 2/2
1 repetitions per point

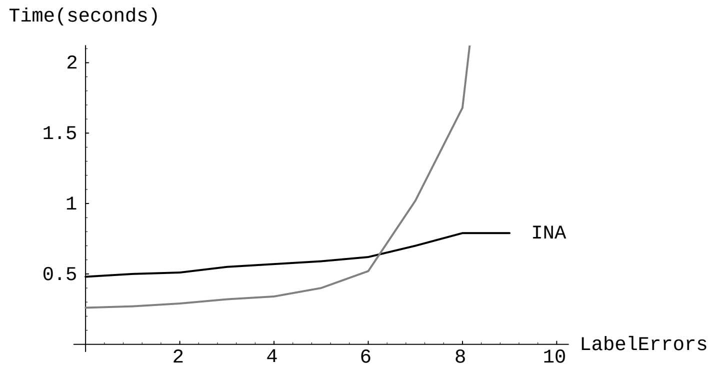


Figure 28

Experiment 6:

model size: 30
label set: 30
connectivity: 2/2
1 repetitions per point

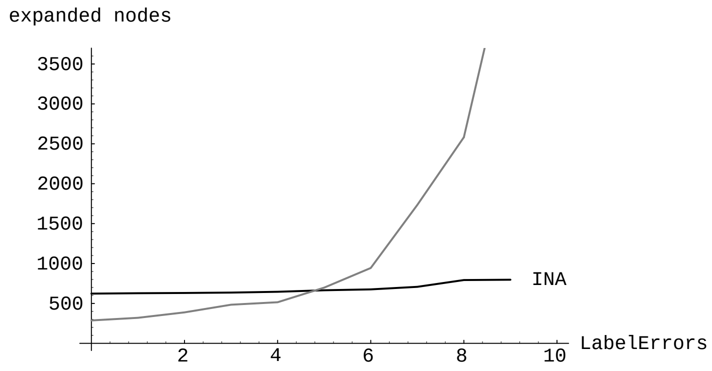


Figure 29

rely on application specific restrictions on global or local object features. Therefore we chose to set the future costs of each node in the A^* search tree to 0.

First, in experiment 6, we try to verify the theoretical result which says that for a graph with uniquely labelled vertices, A^* will be very sensitive to label errors and require an increasing amount of time, while INA is almost independent of the label noise thanks to its lookahead feature. As expected, the lookahead over the network enables INA to obtain for each network node an accurate estimation for the minimal future costs, thus restricting the total number of instances generated. In the theoretical analysis we concluded that for graph preserving distortion of labels, i.e., the underlying graph is unchanged by label distortion, INA is independent of the actual number of mislabels if each vertex is uniquely labelled. If the label set is reduced and several vertices show the same label, both algorithms will require more time and space. Experiment 7 shows, however, that for graphs where each label appears at least twice in the graph NA still performs impressively better than A^*.

We pick the parameter settings of experiment 6 at the point $labelerror = 6$ and run

Experiment 7:

model size: 30
label set: 15
connectivity: 2/2
1 repetitions per point

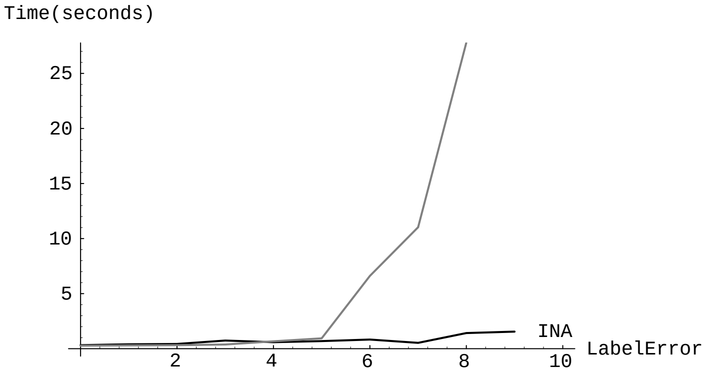


Figure 30

Experiment 7:

model size: 30
label set: 15
connectivity: 2/2
1 repetitions per point

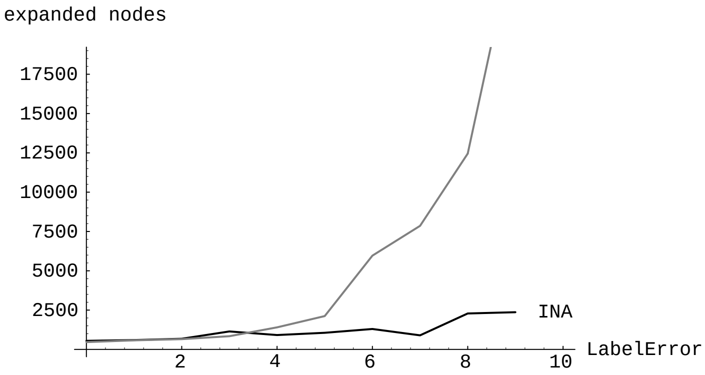


Figure 31

Experiment 8:

model size: 30
label error: 6
connectivity: 2/1
10 repetitions per point

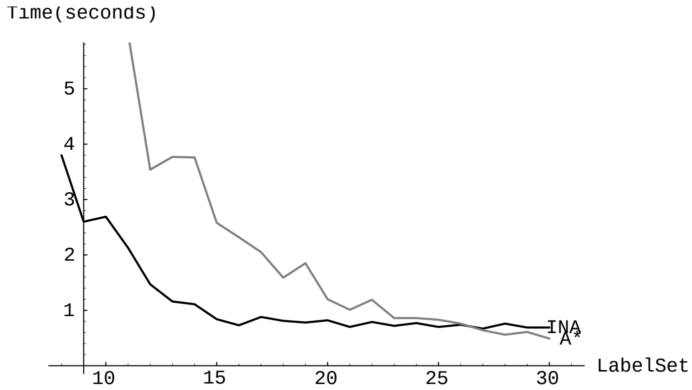


Figure 32

Experiment 8:

model size: 30
label error: 6
connectivity: 2/2
10 repetitions per point

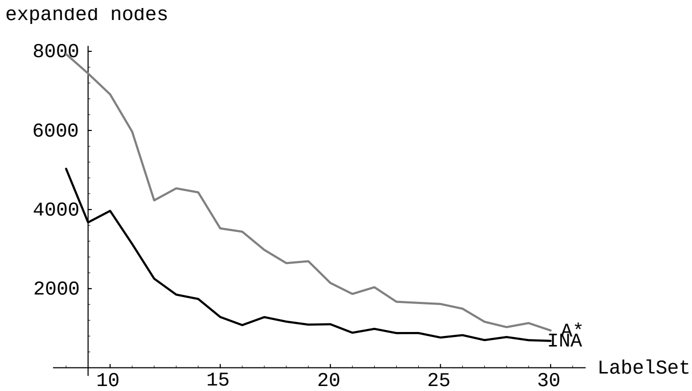


Figure 33

experiment 8, where the set of labels is made larger, starting at 8 and ending at 30, where again all vertices will be labelled unique. Just as in the exact subgraph isomorphism problem both algorithms perform better the more labels are present in the model and input graph. We observe that INA is generally faster and uses less space than A^* even if not all labels are unique.

So far only graph preserving label distortion has been studied. In experiment 9 we trace the behavior of the algorithms when the input graph is structually distorted and deletion operations on the model are necessary. In the image analysis of multiple object scenes it is often the case that large parts of the objects are occluded, thus resulting in an incomplete image graph. We simulate this situation by first generating a random model graph, which is then transformed into an input graph by deleting a given number of vertices along with the incident edges. The model is made up of 15 vertices and the degree of distortion is increased from 0 missing vertices to 14 missing vertices at which point the input graph consists only of a single vertex. We simplify the problem by declaring any substitution between different labels as impossible, thus restricting the possible matches

Experiment 9:

model size: 15
label set: 20
connectivity: 2/2
5 repetitions per point

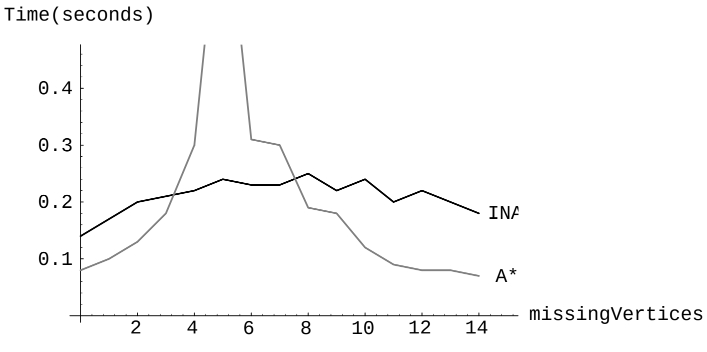


Figure 34

Experiment 9:

model size: 15
label set: 20
connectivity: 2/2
5 repetitions per point

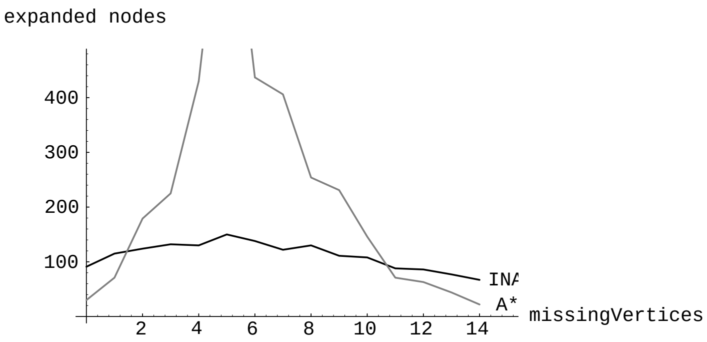


Figure 35

for each model vertex to 2, namely the exact match and the deletion. Theoretically, the search space reaches its maximum when half of the input graph vertices are occluded. The experiment illustrates, that unlike A^*, INA shows only a slight bow towards to worst case and is practically independent of the number of missing parts in the input graph.

To verify that the proposed lookahead also works over several models, in experiment 10 we generate a database containing 10 models of size 16, each sharing a common subgraph of size 8. The number of distorted labels is varied from 0 to 6. All errors appear in the common subgraph area of the input graph. The comparison of the two performance plots confirms that the sharing of common substructures and the lookahead over multiple models succesfully restrict the search space of INA. Next, experiment 11 studies the influence of the database size. The models are constructed as in the previous experiment. There are 2 missing vertices in the common subgraph. The database size is increased steadily from 1 to 20 models. As it was the case for undistorted input graphs, A^* performs worse than INA if the number of models in the database increases. At last, we investigate again the influence of the number of edges on the performance of the two algorithms. This time,

Experiment 10:

model size: 16
common subgraph size: 8
label set: 8
connectivity: 2/1
database: 10
10 repetitions per point

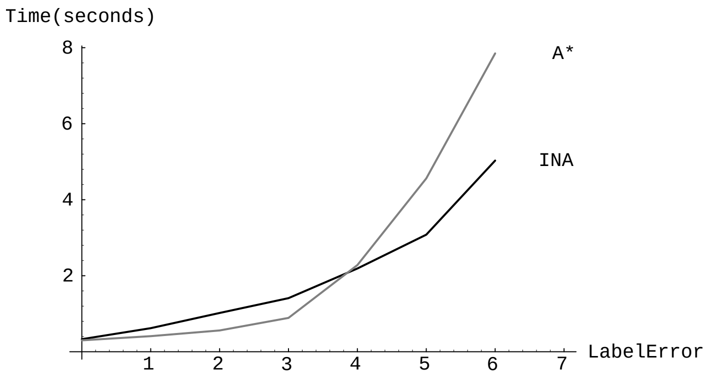


Figure 36

Experiment 10:

model size: 16
common subgraph size: 8
label set: 8
connectivity: 2/2
database: 10
10 repetitions per point

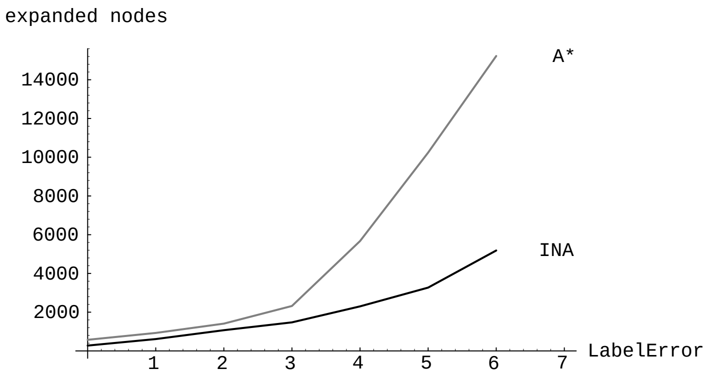


Figure 37

Experiment 11:

model size: 16
common subgraph size: 8
missing vertices: 2
connectivity: 2/2
10 repetitions per point

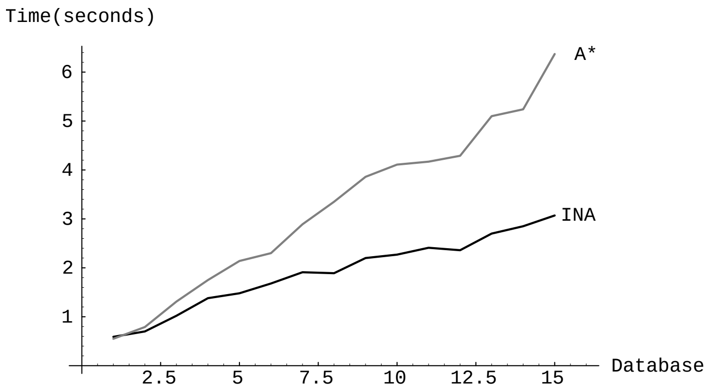


Figure 38

Experiment 11:

model size: 16
common subgraph size: 8
missing vertices: 2
connectivity: 2/2
10 repetitions per point

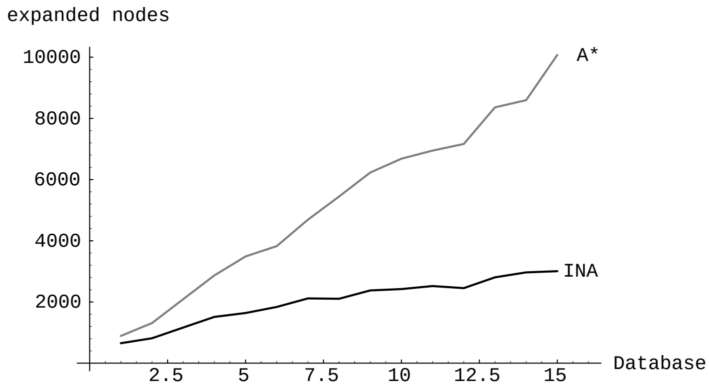


Figure 39

Experiment 12:

model size: 50
label set: 30
mislabeled vertices: 3
mislabeled edges: 3
10 repetitions per point

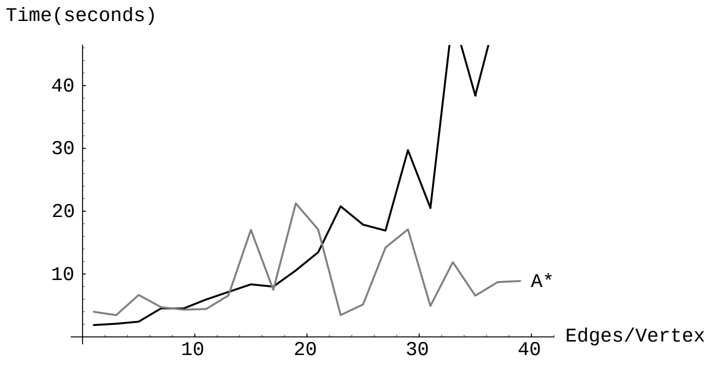


Figure 40

however, we allow for a distorted input graph. Experiment 12 documents the expected decrease in the performance of INA, which naturally suffers from the same problem as NA when the number of edges per vertex is high. The model contains 50 vertices and the connectivity per vertex is varied from 1 to 40. There are 30 labels available. Clearly, INA performs worse than A^* with an increasing number of edges and a fixed amount of noise. However, experiment 13 shows that an increase in the noise that is present in the input graph will cause the search tree expanded by A^* to grow faster than the number of instances in INA, even if the number of edges per vertex is high. The number of edges per vertex is kept at an average of 20, while the number of mislabeled vertices is varied between 1 and 8. The rest of the parameters are set to the values of the previous experiment. We observe that A^* tends toward an exponential behavior depending on the degree of noise, while INA shows only a linear dependency.

So far, we have studied the performance of INA with lookahead in comparison to the traditional A^* algorithm and demonstrated its superiority in most cases. Now we want to examine how big in practice the influence of the static and the dynamic lookahead

Experiment 12:

model size: 50
label set: 30
mislabeled vertices: 3
mislabeled edges: 3
10 repetitions per point

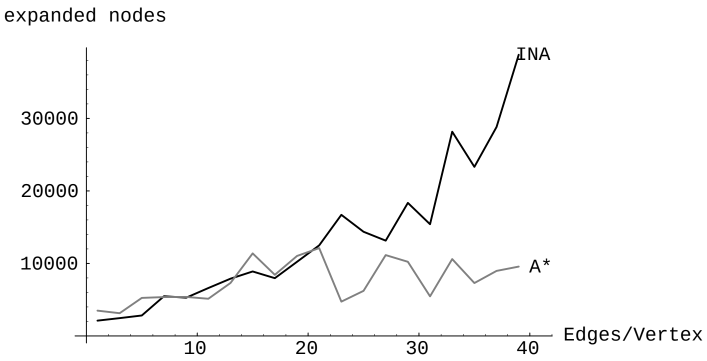


Figure 41

Experiment 13:

model size: 50
connectivity: 20/5
label set: 30
mislabeled edges: 3
10 repetitions per point

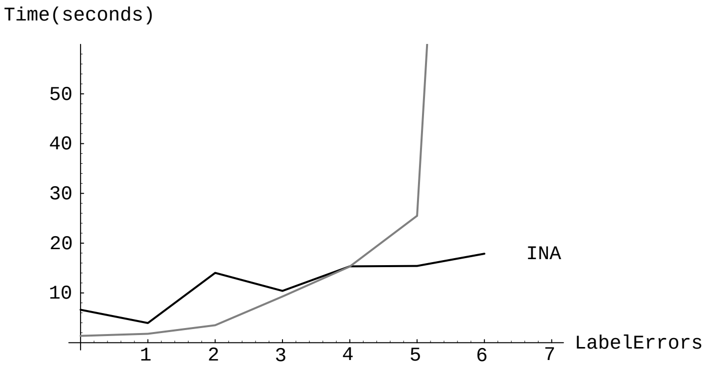


Figure 42

Experiment 13:

model size: 50
connectivity: 20/5
label set: 30
mislabeled edges: 3
10 repetitions per point

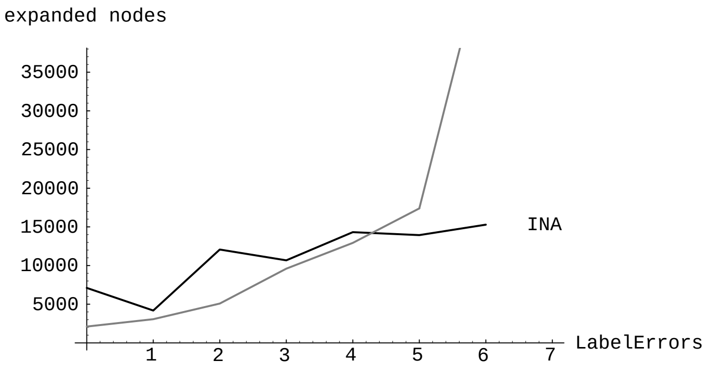
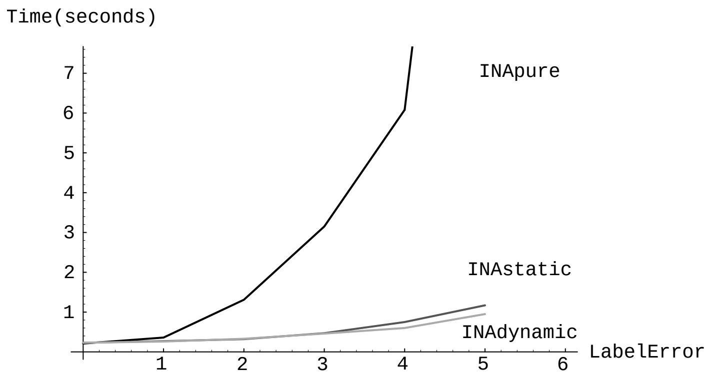


Figure 43

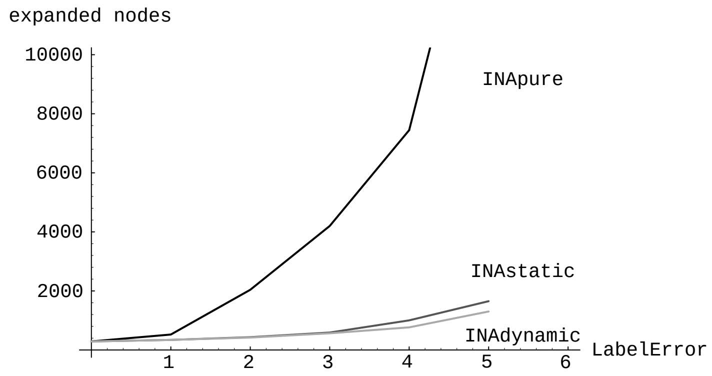
Experiment 14:

model size: 20
connectivity: 5/2
label set: 10
10 repetitions per point



Experiment 14:

model size: 20
connectivity: 5/1
label set: 10
10 repetitions per point



on the performance of INA really is. In experiment 14 we create a model containing 20 vertices and 100 edges and an identical input graph and measure the time and space needed by INApure (INA without lookahead), INAstatic (INA with static lookahead only) and INAdynamic (INA with dynamic lookahead as it was used in the previous experiments). Then we steadily increase the number of mislabeled vertices in the input graph until there are 5 mislabeled vertices. As we expected in section 4, the influence of the lookahead is enormous. INApure shows an exponential behavior even for a small number of distortions, while both INA with static and INA with dynamic lookahead show only a limited growth in their time and space requirements. If the distortion of the input graph is low, there is no additional gain from the use of the dynamic lookahead, because the estimation of the static lookahead is already very accurate. With increasing distortion, however, the dynamic lookahead further reduces the necessary time and space.

While the static lookahead estimates the distance between the model and the input graph exactly once on the grounds of a first processing, the dynamic lookahead on the other hand, adjusts the distance estimations during the subsequent search for an inexact subgraph isomorphism. In the last experiment 15 we trace the distance estimations of the lookahead during the matching process between a single model and an input graph. The model graph contains 20 vertices and 40 edges and 8 different vertex and edge labels. The input graph is a distorted version of the model, where 8 vertices and 5 edges are mislabeled resulting in a minimum distance between input and model of 13. The underlying graph

Experiment 15:

model size: 30
connectivity: 4/1
label set: 8
mislabeled vertices: 8
mislabeled edges: 3
1 sample

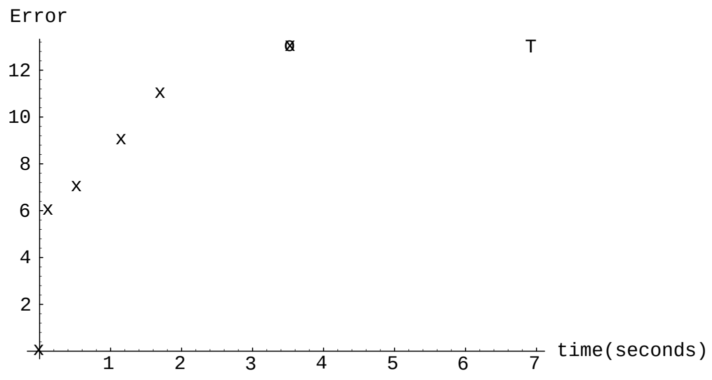


Figure 44

structure is left untouched. Therefore, no insertions or deletions will be necessary. There are x, o and T markings in the plot. The x show when a new distance estimation for the model was made. The o denotes the finding of the optimal inexact subgraph isomorphism and the T marks the termination of the algorithm. The very first x at the point 0/0 is the trivial estimation at the beginning of the algorithm. The second x is the estimation made by the static lookahead. Every subsequent x is an estimation done by the dynamic lookahead. The delay between the finding of the final solution and the termination of the program is caused by the fact that *all* inexact subgraph isomorphisms are search for. The experiment demonstrates that the proposed lookahead technique estimates the distance between the model and the input graph quite accurately after less than a third of the total run time. Thus, if a rejection threshold is specified, the distance estimation may be used to stop the search process at an early stage.

The presented experimental results provide a solid basis for judgement of the network based approach to subgraph isomorphism. If one or several of the following conditions hold for an application domain, then NA and INA will considerably speed up the detection process:

- Objects can be segmented meaningfully
- Different objects contain identical substructures
- Identical substructures appear multiple times inside the same object
- The set of labels and attributes is larger than 2.
- The connectivity of the object parts among each other is restricted
- The number of possible substitutions for each object part is restricted, preferably to a fraction of the object size

On the other hand if the problem structure is such that specific features of the objects allow the recognition or rejection of an input graph after only a few assignments, A^* will be more suited to solve the task. We acknowledge that there are numerous ways to include heuristic informations and speed up A^* considerably. However, we feel that any improvement gained by the application of heuristics can also be achieved with the network algorithm and that the fundamental ideas of sharing identical substructures and using partial instances for an intelligent lookahead, are major advantages of the network approach.

7 Discussion

In this paper the problem of exact and inexact graph morphisms was closely investigated and some results on the influence of the weighting and costs functions on the matching process were presented. A network based approach to exact and inexact subgraph isomorphism detection was proposed. The new algorithm uses an off-line preprocessing in order to speed up the on-line recognition procedure. Along with the inexact detection algorithm, we introduced a new lookahead procedure, which uses the information stored in the network in order to speed up the inexact subgraph isomorphism detection.

The suitability of the network approach was verified in extensive experiments with randomly generated graphs. Although no practical applications have yet been implemented using the new methods, possible domains include case based reasoning, computer vision and machine learning. Future work will mainly concentrate on developing a matching module along with appropriate interfaces, which could then be incorporated into applications whose tasks include graph matching.

References

[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, May 1993.

[BA83] H. Bunke and G. Allerman. A metric on graphs for structural pattern recognition. In H.W.Schüssler, editor, *SIGNAL PROCESSING II: Theories and applications*. EURASIP, Elvesier Science Publications, 1983.

[BM93] H. Bunke and B.T. Messmer. A new algorithm for efficient subgraph matching. In *San Remo Conference*. Elvesier, 1993.

[BTT91] H. Bunke, T.Glauser, and T.H. Tran. Efficient matching of dynamically changing graphs. Technical report, University of Berne, 1991.

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

[EF86] M.A. Eshera and K.S. Fu. An image understanding system using attributed symbolic representation and inexact graph-matching. *IEEE Transactions on pattern analysis and machine intelligence PAMI*, 8:604–617, 1986.

[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.

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

[Hae90] U. Haenni. Anwendung des Simulated Annealing auf Optimierungsprobleme bei der Objekterkennung in Bildern. Master's thesis, Institut für Informatik, Universität Bern, 1990.

[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.

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

[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.

[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.

[LK89] S.W. Lee and J.H. Kim. Attributed sytroke 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), 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.

[Mes92] B.T. Messmer. Inexaktes Graphenmatching mit dem Rete-Algorithmus. Master's thesis, Universität Bern, 1992.

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

[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. In *IEEE Transactions on pattern analysis and machine intelligence PAMI*, volume 3, pages 504–519. IEEE, 1981.

[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 isomorphim. *Journal of the Association for Computing Machinery*, pages 31–42, 1976.