Performance Evaluation of the VF Graph Matching Algorithm

L.P. Cordella, P. Foggia, C. Sansone, M. Vento
Dipartimento di Informatica e Sistemistica – Via Claudio, 21 - I-80125 Napoli (Italy)
E-mail: {cordel,foggiapa,carlosan,vento}@unina.it – http://amalfi.dis.unina.it

Abstract

The paper discusses the performance of a graph matching algorithm tailored for dealing with large graphs without using information about the topology of the graphs to be matched. The algorithm, presented in more details in other papers (and publicly available on the WEB as VF), is now discussed with reference to its computational complexity and memory requirements.

The performance analysis is carried out by theoretically characterizing the matching time and the required memory in the best and worst cases. The theoretical analysis is completed by tests on a database of graphs randomly generated.

The algorithm is compared with the one proposed by J.R. Ullmann: experimental results confirmed the theoretical expectations, highlighting the overall efficiency of the algorithm.

Some results obtained by researchers who recently used the algorithm in application domains requiring a massive use of graph matching techniques are finally reported.

1. Introduction

Graphs are data structures widely used for representing information both in low-level and high-level vision tasks. We will consider Attributed Relational Graphs (ARG's) since they represent a very general type of graph and structural descriptions of visual patterns can be conveniently put in form of ARG's [1]. ARG's provide both syntactic information, held by the layout of unlabeled nodes and branches, respectively identifying the structural primitive components of the pattern and their relations, and semantic information consisting of the attributes associated to nodes and branches.

One of the most relevant problems, dealing with graphs, is that of matching a sample graph against a reference graph; to this concern, a main issue consists in limiting the computational cost of the algorithms.

Purpose of this paper is to discuss the performance, in terms of computation time and memory requirements, of a ARG matching algorithm and to compare it with the performance of one of the most commonly used algorithms performing the same task. The algorithm considered for comparison was first described by J. R. Ullmann in [2].

Both our algorithm and the Ullmann's algorithm do not rely on any special topological property of the graphs and allow us to find all graph or graph-subgraph isomorphisms between two given graphs.

It should be noted that the literature reports plenty of graph matching algorithms which reduce the overall computational complexity of the matching process by taking into account some peculiar topological properties of the graphs to be matched. For instance, algorithms for special classes of graphs, like planar graphs [3] or trees [4] have been proposed in the past, but, despite their efficiency, their applicability is restricted only to domains that satisfy the hypotheses on which these algorithms are based.

Still other algorithms, like the well known method by Cornell and Gottlieb [5], perform suitable transformations on the graphs, in order to find a different representation for which the matching is easier. These algorithms, although in some cases are faster than Ullmann's algorithm, do not fit well with our intended application, since they do not take advantage in the matching process of the semantic information provided by ARG's.

An interesting method, presented in [6], is aimed to reduce the overall computational cost when matching a sample graph against a large set of prototypes. The method allows us to perform in linear time the matching of a graph against a database of prototype graphs; a remarkable feature of the algorithm is that the time does not depend on the number of prototypes. It uses a preprocessing phase in which a decision tree is built from the prototypes, and then employed for the matching. The main drawback of this algorithm is that the memory required to store the decision tree grows exponentially with the dimension of the graphs, making this approach suitable only for small graphs.

In the next section we briefly summarize a graph matching algorithm [7], referred as the VF algorithm, tailored for working with large graphs (up to 2–3000 nodes), which, using a set of feasibility rules, allows us to significantly reduce the computational cost of the matching process. The considered feasibility rules, defined for solving the problem of graph isomorphism,

0-7695-0040-4/99 $10.00 © 1999 IEEE

allow us to significantly prune the search space, so obtaining a very efficient matching process. In Section 3, a theoretical analysis of the overall efficiency of the VF algorithm, in the best and in the worst case, in terms of computational and spatial complexity is presented. Section 4 is devoted to present an experimental analysis of the VF algorithm with reference to a database of randomly generated graphs, and to a comparison analysis with the Ullmann's algorithm.

2. The algorithm

A matching process between two graphs $G_1 = (N_1, B_1)$ and $G_2 = (N_2, B_2)$ consists in the determination of a mapping M which associates nodes of the graph G_1 to nodes of G_2 and vice versa. As it is well known, different constraints can be imposed to M and consequently different mapping types can be obtained: monomorphism, strict isomorphism and graph-subgraph isomorphism are the most frequently used.

Generally, the mapping M is expressed as the set of ordered pairs (n,m) (with $n \in G_1$ and $m \in G_2$) each representing the mapping of a node n of G_1 with a node m of G_2:

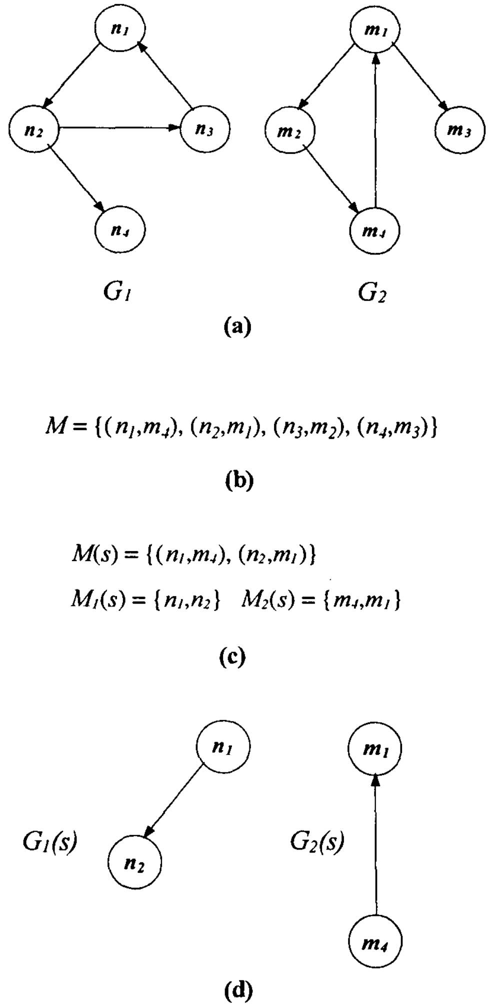
$$M = \{(n,m) \in N_1 \times N_2 \mid n \text{ is mapped onto } m\} \qquad (1)$$

The State Space Representation (from now on SSR) can be effectively used to describe a graph matching process, if each state s of the matching process represents a partial mapping solution. A partial mapping solution $M(s)$ is a subset of M, i.e. contains only some components of M. In Fig. 1 two graphs, a mapping M and a partial mapping solution are presented. In the following we will denote as $M_1(s)$ and $M_2(s)$ the projection of $M(s)$ onto N_1 and N_2 respectively.

In the adopted SSR representation a transition between two states corresponds to the addition of a new pair of matched nodes (see Fig. 2).

In principle, the solutions to the matching problem could be obtained by computing all the possible partial solutions and selecting the ones satisfying the wanted mapping type (Brute Force approach). In order to reduce the number of paths to be explored during the search, for each state on the path from s_0 to a goal state, we impose that the corresponding partial solution verify some coherence conditions, depending on the desired mapping type. For example, to have an isomorphism or a graph-subgraph isomorphism it is necessary that the partial mappings are isomorphisms between the corresponding subgraphs. If the addition of a node pair to the partial mapping produces a state that does not meet the coherence condition, further exploration of that path can be avoided, since it certainly cannot lead to a goal state.

The rationale of our algorithm is that of introducing, given a state s, criteria for foreseeing if s has no coherent successors after a certain number of steps. It is clear that these criteria (feasibility rules) should allow us to detect as soon as possible conditions leading to incoherence; in particular, we say that a rule implements a k-look-ahead if, given a state s and a pair (n,m) to be included in s (so obtaining a state s') it allows us to establish if all the states reachable from s' in k steps are incoherent. Therefore states which don't satisfy a feasibility rule can be discarded from further expansions.



$$M = \{(n_1,m_4), (n_2,m_1), (n_3,m_2), (n_4,m_3)\}$$

(b)

$$M(s) = \{(n_1,m_4), (n_2,m_1)\}$$

$$M_1(s) = \{n_1,n_2\} \quad M_2(s) = \{m_4,m_1\}$$

(c)

Fig. 1: (a) Two graphs G_1 and G_2, (b) the only possible mapping M, (c) a partial mapping solution $M(s)$ and (d) the corresponding subgraphs $G_1(s)$ and $G_2(s)$.

In Fig. 3 the proposed algorithm is outlined. It employs a depth-first search strategy implemented in a recursive fashion. For each intermediate state s the algorithm considers the set $P(s)$ of the node pairs that can be added to the current state; the definition of $P(s)$ will be provided later.

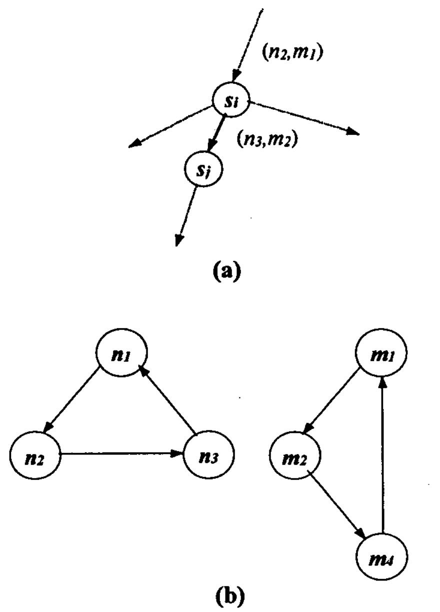


Fig. 2: (a) The addition of the pair (n_3,m_2) to the state s_i gives rise to the state s_j, corresponding to the partial mapping reported in (b).

```
PROCEDURE Match($s$)
  INPUT:  an intermediate state $s$; the
          initial state $s_0$ has $M(s_0) = \varnothing$
  OUTPUT: the mappings between the two
          graphs

  IF $M(s)$ covers all the nodes THEN
    OUTPUT $M(s)$
  ELSE
    Compute the set $P(s)$ of the pairs
    candidate for inclusion in $M(s)$
    FOREACH $p \in P(s)$
      IF the feasibility rules succeed for
         the inclusion of $p$ in $M(s)$
      THEN
        Compute the state $s'$ obtained by
        adding $p$ to $M(s)$
        CALL Match($s'$)
      END IF
    END FOREACH
  END IF
END PROCEDURE
```

Fig. 3: The VF matching algorithm.

Look-Ahead	Rule	Condition
0	R_pred	*Iff for each predecessor n′ of n in the partial mapping, the corresponding node m′ is a predecessor of m, and vice versa*
	R_succ	*Iff for each successor n′ of n in the partial mapping, the corresponding node m′ is a successor of m, and vice versa*
1	R_termin	*Iff the number of predecessors (successors) of n that are in $T^{in}_1(s)$ is equal to the number of predecessors (successors) of m that are in $T^{in}_2(s)$*
	R_termout	*Iff the number of predecessors (successors) of n that are in $T^{out}_1(s)$ is equal to the number of predecessors (successors) of m that are in $T^{out}_2(s)$*
2	R_new	*Iff the number of predecessors (successors) of n that are neither in $M_1(s)$ nor in $T_1(s)$ (new nodes) is equal to the number of predecessors (successors) of m that are neither in $M_2(s)$ nor in $T_2(s)$*

Tab. 1: The feasibility rules for exact graph isomorphism.

There are two kinds of feasibility rules, respectively regarding the syntax and the semantics of the graphs. The syntactic feasibility rules defined for an exact isomorphism are shown in Tab. 1.

It is worth noting that the 0-look-ahead rules (R_pred and R_succ) ensure necessary and sufficient conditions for the coherence of the successive partial solutions, while the others provide necessary (but not sufficient) conditions and are mainly used for pruning the search graph.

Note that the rules are evaluated with reference to some sets which depend on the considered state s. In particular we have denoted as $T^{out}_1(s)$ ($T^{in}_1(s)$) and $T^{out}_2(s)$ ($T^{in}_2(s)$) the sets of outgoing (ongoing) nodes from the two subgraphs $G_1(s)$ and $G_2(s)$, and $T_1(s) = T^{out}_1(s) \cup T^{in}_1(s)$, $T_2(s) = T^{out}_2(s) \cup T^{in}_2(s)$.

The above defined sets are also used to construct the set $P(s)$ of node pairs candidate for inclusion in the state s. If $T^{out}_1(s)$ and $T^{out}_2(s)$ are not empty, $P(s)$ is constituted by all the pairs (n,m) where n is the node in $T^{out}_1(s)$ with the smallest label and m is a node belonging to $T^{out}_2(s)$. If $T^{out}_1(s)$ and $T^{out}_2(s)$ are empty, $T^{in}_1(s)$ and $T^{in}_2(s)$ are used instead, and if these latter sets are empty too, N_1-$M_1(s)$ and N_2-$M_2(s)$ are employed. It can be easily demonstrated that with this definition of $P(s)$ each state can be explored at most once during the search process.

In Fig. 4, a state s and the corresponding sets $T^{in}_1(s)$ and $T^{out}_1(s)$ are given.

Semantic compatibility can be introduced very easily in the matching process: each time a node of the sample is compared to a node of the prototype to determine if a new pair can be added to the current partial solution, the attributes of the two nodes and of the branches linking them to the nodes already in s are tested for semantic compatibility. Obviously semantic compatibility has to be defined with reference to the specific application domain.

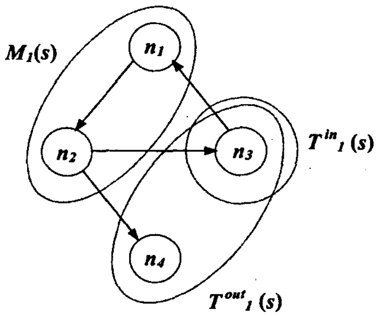


Fig. 4: The sets $T^{in}_1(s)$ and $T^{out}_1(s)$ relative to the state in which $M_1(s)=\{n_1,n_2\}$.

3. Computational complexity evaluation

In this Section we will address the problem of evaluating the memory requirements and the computational complexity of the VF algorithm considering both the best and the worst case. In order to quantitatively assess the performance in the average case, some experimental results will be presented in the next Section with reference to a database of randomly generated graphs.

The computational complexity of the matching algorithm depends on two factors: the number of the SSR states currently visited and the time needed for visiting each state. We will begin from the latter factor, as it is simpler to be analytically evaluated.

The cost of exploring a state obtained by adding a pair (n,m) to a state s can be decomposed into three terms:

- the cost needed to verify if the new state satisfies the feasibility rules;
- the cost needed to calculate the sets (T^{in}_1, T^{in}_2, etc.) associated to the new state;
- the cost needed to generate the sets of the pairs candidate for inclusion in the current state.

The first two terms have a cost proportional to the number of branches having n or m as an endpoint. In fact, for each branch outgoing from n or ongoing into n it must be verified if the other endpoint n' is in $M_1(s)$, $T^{in}_1(s)$ or $T^{out}_1(s)$, and, if n' is in $M_1(s)$, it must be also checked if a branch linking m with the node of $M_2(s)$ associated to n' exists. In a similar way, the branches ongoing or outgoing from m must be checked. As the operations needed for each branch can be performed in a constant time, the whole time will be proportional to the number of branches. If we denote this quantity with b, the cost for the first two terms will be $\Theta(b)$.

The third term requires a number of operations which is at least proportional to the number of the nodes of the two graphs. In fact, in order to find all the pairs (n,m) candidate to the inclusion in the current state it is necessary to determine the node of $T^{out}_1(s)$ with the smallest label (spending a time proportional to the number of nodes contained in G_1) and to examine all the nodes of G_2 to verify if they belong to $T^{out}_2(s)$ (spending a time proportional to the number of nodes in G_2). If we suppose that the two graphs have the same number N of the nodes, the total cost for this term will be $\Theta(N)$.

In order to evaluate the computational complexity in the average case we have to define a probabilistic model of the distribution of the branches within the graphs. A model often used in literature [2] fixes the value η of the probability that a branch is present between two distinct nodes n and n'. In this case, the average number of branches in the graph will be equal to $\eta N(N-1)$ and the average value of b will be $2\eta(N-1)$, and thus $\Theta(b)= \Theta(N)$. It follows that the cost for the exploration of a single state is $\Theta(N)$.

Now we will try to evaluate the number of states explored by the algorithm, starting from the best case. The best case happens when in each state only one of the potential successors satisfies the feasibility predicate (in the hypothesis that an isomorphism exists). In this situation, the number of explored states is equal to N, and thus the computational complexity in the best case is $\Theta(N^2)$. For comparison, Ullmann's algorithm has a best case complexity of $\Theta(N^3)$.

In the worst case, in each state the predicate will not be able to avoid the visit of any of the successors, and the algorithm will have to explore all the states before reaching a solution. This situation may occur if the graphs exhibit strong symmetries, for example if they are almost completely connected. In such a case, the number of states can be computed as follows: each state at level i of the search tree will have $N-i$ successors which have to be explored; hence, the number of states at level $i+1$ can be computed by multiplying by $N-i$ the number of states at level i. It follows that, by adding up the states at each level, the total number of states will be:

$$\begin{aligned}&1+N+N(N-1)+N(N-1)(N-2)+...+\\&+N(N-1)(N-2)\cdot...\cdot 3+N!\end{aligned} \tag{2}$$

This sum can be rewritten as:

$$1+N!\sum_{i=1}^{N-1}\frac{1}{i!} \tag{3}$$

and it can be easily shown that the summation is bounded by a constant smaller than 2. Hence, the total number of states is proportional to $N!$ and the computational complexity in the worst case is $\Theta(N!N)$. For comparison, the computational complexity of Ullmann's algorithm is in the worst case $\Theta(N!N^3)$.

We can also easily evaluate the spatial complexity of our algorithm, which, if an isomorphism exists, depends only on the number N of nodes of the graphs. In fact, from the analysis of the algorithm presented in Fig. 3, it follows that the maximum number of states simultaneously present in memory is N. In order to evaluate the space required by a single state, we can consider that each state holds a constant amount of information for each node, namely which is the corresponding node in the other graph under the current mapping, whether the node belongs to T^{out}_1, T^{in}_1 and so on. Hence, the space required by each state is $\Theta(N)$, and the overall spatial complexity is $\Theta(N^2)$. Table 2 summarizes the computational complexity of the VF algorithm compared with the one of the Ullmann's algorithm.

	VF Algorithm		**Ullmann's Algorithm**	
Complexity	***Best Case***	***Worst Case***	***Best Case***	***Worst Case***
Temporal	$\Theta(N^2)$	$\Theta(N!N)$	$\Theta(N^3)$	$\Theta(N!N^3)$
Spatial	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N^3)$	$\Theta(N^3)$

Tab. 2: Spatial and temporal complexity of the proposed algorithm and of the Ullmann's algorithm in the best case and in the worst case.

4. Experimental results

In order to estimate the computational complexity in the average case, we have to evaluate the reduction of the number of visited states, with respect to the worst case, due to the algorithm and to the feasibility rules described in Section 2.

First we will try to evaluate the reduction due to the fact that the algorithm depicted in Fig. 3 chooses the pairs to be added by using the set $P(s)$. More precisely, the number of successors of a state s is equal to $|P(s)|$, and thus is smaller than $N-i$ for a state at the i-th level of the search tree. In order to determine how much smaller, we will make the assumption that the graphs are connected; in this case for $i>0$ $|P(s)|$ is equal to either $|T^{out}_2(s)|$ or $|T^{in}_2(s)|$; hence, we have to estimate the average cardinality of these sets.

It follows by the definitions given in the previous Section that a node m of G_2 is in $T^{out}_2(s)$ if there is a branch (m',m) with m' in $M_2(s)$. Since at level i of the search tree $|M_2(s)|=i$, there are i possible branches connecting m to $M_2(s)$; in the adopted graph model each of these branches has a probability η of being actually part of G_2, and the branches are independent. Hence, the probability that m does not belong to $T^{out}_2(s)$ is:

$$\Pr\{m \notin T_2^{out}(s)\} = (1-\eta)^i \qquad (4)$$

From eq. (4) it follows that:

$$\Pr\{m \in T_2^{out}(s)\} = 1-(1-\eta)^i \qquad (5)$$

The same probabilities hold for $T^{in}_2(s)$.

Taking into account all the levels of the search tree, the average reduction of the number of visited states is:

$$R_{f_t}(N) = \prod_{i=1}^{N} 1-(1-\eta)^i \qquad (6)$$

Figure 5 shows the behavior of this function for some values of η and N. As it can be seen, the value of R_{f_t} for $N \to \infty$ converges very quickly to a constant which depends on η.

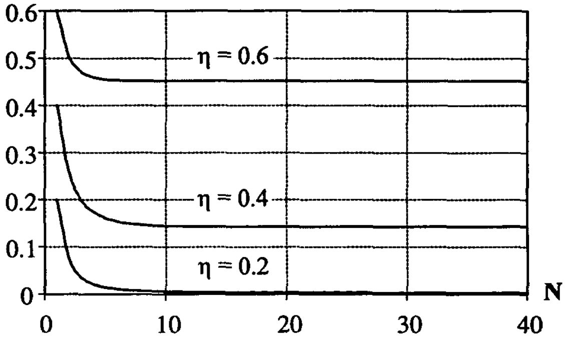


Fig. 5: The trend of $R_{f_t}(N)$ for some values of η and N.

Therefore, we may draw the conclusion that the use of the set $P(s)$ does not modify in the average case the computational complexity, but introduces a constant reduction factor which can be quite effective if the graph is sparse (i.e. η has a small value).

The analytical estimation of the reduction in the number of states due to the feasibility predicate is not a simple task. For this reason, we have performed an experiment aimed at evaluating the average number of explored states. We have randomly generated pairs of isomorphic graphs according to the graph model previously described, and we have measured the number of states visited by both our algorithm and the Ullmann's algorithm in the search for the isomorphism.

Figure 6 shows for several numbers of nodes the average number of states visited in excess with respect to the minimum. It can be noted that this number is quite small, meaning that in the average case the behavior of both the algorithms is close to the best case. The value used for η is 0.5, hence the graphs can be considered "dense", even though they are not close to the worst case, i.e. the complete graph. We have also tested the algorithms with smaller values of η, without appreciable variations in the number of states. The behavior of Ullmann's algorithm appears slightly better than ours with respect to the number of explored states, but the greater number of states is adequately compensated by the smaller time required to explore each state.

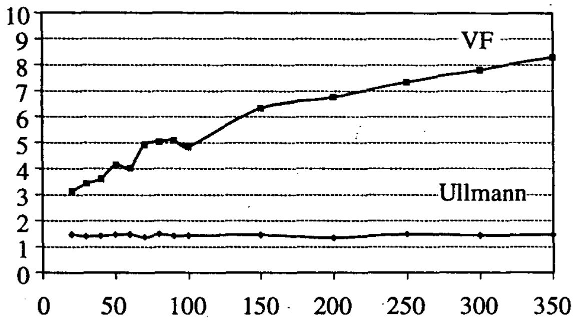


Fig. 6: Average number of visited states, in excess with respect to the minimum number, for VF and Ullmann's algorithms as a function of the number of nodes (in case of η equal to 0.5).

To this concern, Fig. 7 shows the average time required by the matching with respect to the number of nodes of the graphs. Tests have been carried out on a Pentium II 333 MHz with 128 MB of RAM under Linux 2.0.36. The software used has been developed in C++, using the GNU egcs 1.1 compiler. It can be noted that the curves corresponding to the Ullmann's algorithm stop at a smaller number of nodes. This fact is due to the greater spatial complexity of this algorithm, which for graphs of more than 300-350 nodes requires the use of virtual memory, with a drastic performance decrease.

5. Conclusions and final notes

The characteristics of the proposed graph matching algorithm are very interesting, making it efficiently usable especially in application domains requiring large graphs (more than 1000 nodes).

Since November 1997 a preliminary release of the source code of our VF implementation has been made publicly available through World Wide Web (http://amalfi.dis.unina.it/graph). Up to now, it has been downloaded by over 250 users from both universities and private companies; some of them have sent us a short description of the use they made of VF. In particular, it has been employed by a company working on knowledge extraction from large chemical databases as a component of a software for searching and comparing chemical compounds. Also, a modified version of VF is being used in a project at Moscow State University to detect duplicates in a database of randomly generated large molecular graphs (up to 10000 nodes). Another use of VF that has been reported to us is in the context of solid models retrieval from a database.

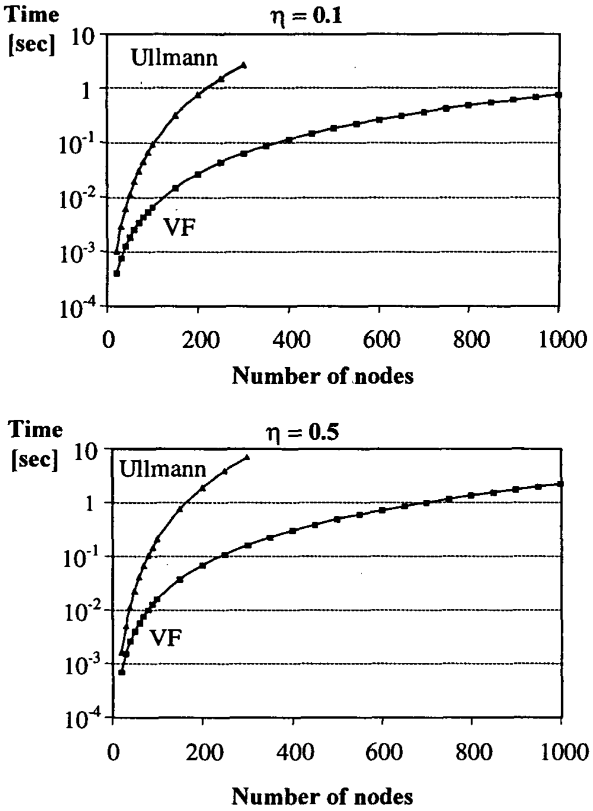


Fig. 7: Average matching times as a function of the number of nodes for η equal to 0.1 and to 0.5. The times are expressed in a logarithmic scale.

References

[1] L.G. Shapiro, R.M. Haralick, "Structural Description and Inexact Matching", IEEE Trans. on Pattern Analysis and Machine Intelligence, vol. 3, pp. 505-519, 1981.

[2] J.R. Ullmann, "An Algorithm for Subgraph Isomorphism", Journal of the Association for Computing Machinery, vol. 23, pp. 31-42, 1976.

[3] J. Hopcroft, J. Wong, "Linear time algorithm for isomorphism of planar graphs", in Proc. 6th Annual ACM Symp. Theory of Computing, pp. 172-184, 1974.

[4] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The design and analysis of computer algorithms, Addison Wesley, 1974.

[5] D.G. Corneil, C.C. Gotlieb, "An efficient algorithm for graph isomorphism", Journal of the Association for Computing Machinery, vol. 17, pp. 51-64, 1970.

[6] H. Bunke, B.T. Messmer, "Efficient Attributed Graph Matching and its Application to Image Analysis", in C. Braccini, L. De Floriani, G. Vernazza (eds.), Image Analysis and Processing, Berlin Heidelberg: Springer (Lecture Notes in Computer Science, vol. 974), pp. 45-55, 1995.

[7] L.P. Cordella, P. Foggia, C. Sansone, M. Vento, "Subgraph Transformations for the Inexact Matching of Attributed Relational Graphs", Computing, vol. 12, pp. 43-52, 1998.