Graph Matching: a Fast Algorithm and its Evaluation

L.P. Cordella, P. Foggia, C. Sansone, F. Tortorella, M. Vento

Dipartimento di Informatica e Sistemistica Università di Napoli "Federico II"

Via Claudio 21, I-80125 Napoli (Italy) Phone: +39 81 768 3185 Fax: +39 81 768 3186

E-mail: {cordel,foggiapa,carlosan,ftortor,vento}@unina.it - Web Home Page: <http://amalfi.dis.unina.it>

Abstract

A graph matching algorithm is illustrated and its performance compared with that of a well known algorithm performing the same task. According to the proposed algorithm, the matching process is carried out by using a State Space Representation: a state represents a partial solution of the matching between two graphs, and a transition between states corresponds to the addition of a new pair of matched nodes. A set of feasibility rules is introduced for pruning states corresponding to partial matching solutions not satisfying the required graph morphism. Results outlining the computational cost reduction achieved by the method are given with reference to a set of randomly generated graphs.

1. Introduction

Graphs are data structures widely used for representing information both in low-level and high-level vision tasks. One of the problems of interest, with graphs, is matching a sample graph against a reference graph. Depending on the nature of the considered vision task and on the characteristics of the graphs, either exact or inexact matching may be required. In both cases, different types of morphisms are possible for the matching: we will mainly consider graph isomorphism.

A relevant problem when matching graphs is that of limiting the computational cost of the process. Purpose of this paper is to illustrate a novel graph matching algorithm of Attributed Relational Graphs and to compare its performance with that of one of the most commonly used algorithms performing the same task by taking into account the problem of reducing the matching time. The algorithm considered for comparison was first described by J. R. Ullman in [1]. It does not rely on any special property of the graphs and allows to find all graph or graph-subgraph isomorphisms between two given graphs. Several other algorithms have been proposed for special classes of graphs, like planar graphs [2] or trees [3]: we have not considered them for the comparison since our method does

not impose restrictions on the graphs to be matched.

We will consider Attributed Relational Graphs (ARG's) [4] which provide both syntactic information, held by the layout of unlabeled nodes and branches, and semantic information consisting of the attributes associated to nodes and branches. In real pattern recognition applications, the variability of the samples is such that they are seldom identical to prototypes. The variations may be reflected by both syntactic and semantic deformations of the representative graphs, so that pattern recognition can only be achieved by inexact graph matching methods able to manage both these possibilities.

The quite few approaches to inexact graph matching proposed in the literature, try to extend the applicability of exact matching methods, by introducing criteria allowing matching in presence of syntactic and/or semantic deformations. In [5], a pattern deformational model is proposed, while a generalization of the method, including the possibility of deleting nodes and branches, is discussed in [6]. The algorithm, though powerful enough for some practical applications, is not effective when large variations among the members of a same class may exist. In these cases inexact matching approaches based on the definition of a distance measure between graphs, seem more appropriate [4,7].

In the following sections we will define and evaluate an ARG matching algorithm which, using a set of feasibility rules [8], allows to reduce the computational cost of the matching process. The extension of the method to inexact matching, taking into account deformations on syntactic and semantic parts of the graphs, will be also discussed.

2. The Matching 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. Generally, the mapping 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 . Each pair is here denoted as component m_i of the mapping M .

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. it contains only some components of M . In the adopted SSR representation a transition between two states corresponds to the addition of a new pair of matched nodes.

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 verifies some coherence conditions, depending on the desired mapping type. 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 to detect as soon as possible conditions leading to incoherence. States which don't satisfy a feasibility rule can be discarded from further expansions.

```

INPUT: The graphs  $G_1$  and  $G_2$ 
OUTPUT: A mapping  $M$ 
BEGIN
 $M(s_0) := \emptyset$ ;  $S := \{s_0\}$ ; (* Initialization *)
REPEAT
  FOREACH state  $s$  contained in  $S$ 
    Remove  $s$  from  $S$ 
    Determine the set  $P(s)$  of all the
    pairs candidate to be included in
     $M(s)$ .
    FOREACH candidate pair  $p \in P(s)$ 
      Evaluate the feasibility rules
      IF the feasibility rules succeed
      for the inclusion of  $p$  in  $M(s)$ 
      THEN add to  $S$  the state obtained
      including  $p$  into  $M(s)$ 
    ENDFOR
  ENDFOR
UNTIL all the nodes of  $G_2$  have been
considered OR  $S$  is empty
END.

```

Fig. 1: The matching algorithm.

In Fig. 1 the proposed algorithm is outlined. At each iteration of the outer loop, the algorithm considers the set $P(s)$ of node pairs that can be added to the state s , discarding those pairs which does not satisfy the feasibility rules.

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 has been described in [8]. Semantic compatibilities 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 (which obviously has to be defined with reference to the specific application domain).

The exact matching algorithm can be extended by considering both transformations on the structure of the graph and on nodes and branches attributes. The considered syntactic transformations are the split of a node into a subgraph, the merge of a subgraph into a node and the insertion or deletion of a branch.

Syntactic transformations are taken into account, during the expansion process of the search graph, by generating new states. When examining a state in the search process, the algorithm checks if there is a syntactic transformation that can be applied to it. For each applicable transformation, a new state s is added to the SSR graph. The nodes involved in a syntactic transformation are marked, in order to avoid reconsidering them in successive transformations. In this way we avoid the possible generation of infinite length paths in the search graph and prevent the possibility that a prototype is matched with a too different sample, as a consequence of the repeated application of some transformation. The conditions a state s has to meet in order that a transformation can be applied, depend on the transformation type. For each type of transformation, only the nodes in $P(s)$ are considered as candidates so as to ensure that, in the next step of the algorithm, the nodes generated by the transformation (or at least some of them) will be tested using the feasibility rules. In this way, if the new paths are fruitless, they will be pruned as soon as possible.

3. Performance Evaluation and Discussion

Before presenting the experimental results illustrating the performance of our matching algorithm, the one we have selected for comparison will be briefly recalled. The Ullman's algorithm [1] is based on a backtracking search enhanced with a look-ahead function (called *forward checking*) to reduce the number of alternatives to be checked. In this respect it is similar to our method: in fact a backtracking search algorithm can be viewed as a SSR search in which the states are explored in a depth first order. In order to keep track of the allowed node pairings, the algorithm computes, at each step s , a matrix C^s such that $c_{ij}^s = 1$ if it is possible to match node i of the first graph with node j of the second graph. A sketch of the algorithm, described with the language used in [1], is reported below:

- 1) $i := 0$; initialize C^0
- 2) $i := i + 1$
- 3) find a new j such that $c_{ij}^{i-1} = 1$; if no

```

    such j exists go to 7
4) if i is the last node of first graph, a
   matching has been found; report the
   matching and go to 3
5) compute  $C^i$  from  $C^{i-1}$  and perform the
   forward checking
6) if the forward checking fails, go to 3;
   else go to 2
7) if  $i=1$  exit;
   else  $i:=i-1$ , go to 3

```

The initialization of step 1 and the computation of the new matrix in step 5 depend on the desired matching type. In particular, for our case (directed graph isomorphism), c_{ij}^0 is set to 1 if the in-degrees and the out-degrees of the two nodes i and j are equal. The new matrix C^i is derived by C^{i-1} by setting to 0 all the elements corresponding to future node pairings that does not satisfy the desired matching type. In this way, all the elements of column j are set to zero and for each pair (p, q) with $p > i$, c_{pq} is set to 0 if there is a pair (u, v) with $c_{uv} = 1$ such that there is an edge between p and u and no edge between q and v , or vice versa. After the update of the matrix, a *forward checking* is performed, to ensure that there is no node in the first graph which will not have a correspondent node in the second graph; this condition can be verified by looking for rows of the matrix containing only 0's. This check, although not necessary to find an isomorphism, in many cases can reduce considerably the number of iterations.

Note that the memory required by Ullman's algorithm, which is essentially used for the matrices C^s , is proportional to n^3 , where n is the number of nodes in the graphs, while our algorithm requires an amount of memory proportional to n^2 .

The comparison of the two algorithms has been carried out by considering an exact isomorphism problem, without taking into account semantic information. It is worth pointing out that semantic information is dependent on the considered application domain and its use generally allows to significantly reduce the search space. Consequently, the performance obtained by considering only the syntactic feasibility rules represents an upper bound.

A set of connected graphs with a number N of nodes varying from 10 to 500, and an average number of

branches equal to ηN^2 , with $\eta \in [0.1, 0.25, 0.5]$, have been randomly generated. Figure 2 reports, in logarithmic scale, the matching times obtained implementing our algorithm and the Ullman's algorithm in C++ under Linux, on a Pentium 120 MHz, with 32 Mb of RAM. Note that the results of some tests with the Ullman's algorithm are not reported in Fig. 2 since memory requirements significantly increased with the number of nodes, overcoming the memory available on the adopted system. Both the algorithms show an exponential trend, but the coefficient of the exponential is lower for our algorithm. The absolute matching times are up to two orders of magnitude lower in our case. The C++ code of both algorithms is available on Internet at Web page <http://amalfi.dis.unina.it/graph>.

Future investigations will be devoted to evaluate the performance of the method in case of the graph-subgraph isomorphism, and to characterize its computational complexity in case of both exact and inexact matching.

References

- [1] Ullman, J.R.: An Algorithm for Subgraph Isomorphism. Journal of the ACM 23, 31-42 (1976).
- [2] Hopcroft, J., Wong, J.: Linear time algorithm for isomorphism of planar graphs, Proc. 6th Annual ACM Symp. on Theory of Computing, 172-184, (1974)
- [3] Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The design and analysis of computer algorithms, Addison Wesley, (1974)
- [4] Shapiro, L.G., Haralick, R.M.: Structural Description and Inexact Matching. IEEE Trans. on PAMI 3, 505-519 (1981).
- [5] Tsai, W.H., Fu, K.S.: Error-Correcting Isomorphisms of Attributed Relational Graphs for Pattern Analysis. IEEE Trans. on SMC 9, 757-768 (1979).
- [6] Tsai, W.H., Fu, K.S.: Subgraph Error-Correcting Isomorphisms for Syntactic Pattern Recognition. IEEE Trans. on SMC 13, 48-62 (1983).
- [7] Sanfeliu, A., Fu, K.S.: A distance measure between Attributed Relational Graphs for Pattern Recognition. IEEE Trans. on SMC 13, 353-362 (1983).
- [8] Cordella, L. P., Foggia, P., Sansone, C., Vento, M.: An Efficient Algorithm for the Inexact Matching of ARG Graphs Using a Contextual Transformational Model. In: Proc. 13th ICPR, vol. III, 180-184 (1996).

Fig. 2: Matching time versus number of nodes with times expressed in seconds in a logarithmic scale.