An Improved Algorithm for Matching Large Graphs

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

Dipartimento di Informatica e Sistemistica
Università degli Studi di Napoli "Federico II"
Via Claudio, 21 – 80125 Napoli ITALY
{cordel,foggiapa,carlosan,vento}@nina.it

Abstract
In this paper an improved version of a graph matching algorithm is presented, which is able to efficiently solve the graph isomorphism and graph-subgraph isomorphism problems on Attributed Relational Graphs. This version is particularly suited to work with very large graphs, since its memory requirements are quite smaller than those of other algorithms of the same kind. After a detailed description of the algorithm, an experimental comparison is made against both the previous version (developed by the same authors) and the Ullmann's algorithm.

1. Introduction

Graphs are data structures endowed with such an expressive power to make their use profitable in the most disparate areas, including Pattern Recognition and Computer Vision. The most natural way of exploiting graphs in this field is that of using them for providing structural descriptions, but graphs are also employed in low-level image representations [11], that are generally made of thousands of nodes.
In this paper, the attention will be mainly devoted to the problem known as exact graph matching, which is one of the ways to perform graph comparison. As it is well known, among the different types of exact matching (monomorphism, isomorphism, graph-subgraph isomorphism) subgraph isomorphism is a NP-complete problem [9], while it is still an open question if also graph isomorphism is a NP-complete problem. As a consequence, worst-case time requirements of matching algorithms increase exponentially with the size of the input graphs, restricting the applicability of many graph based techniques to very small graphs (tens of nodes).
Low complexity algorithms suited for matching large graphs have been a subject of research during the last three decades. Some of the proposed algorithms reduce the computational complexity by imposing topological restrictions on the graphs (e.g. planar graphs [10], trees [1] or bounded valence graphs [12]). Other algorithms, such as [5], transform the input graphs into a representation more convenient for matching; however, such algorithms, which usually deal only with isomorphism, often cannot take advantage, during the matching process, of the semantic information provided by Attributed Relational Graphs (ARG's).
A widely known matching algorithm is Ullmann's [17], based on a backtracking

procedure with an effective look-ahead function to reduce the search space. This algorithm is devised for both graph isomorphism and graph-subgraph isomorphism and, albeit rather old, is still today one of the most commonly used for exact graph matching [14] because of its generality and effectiveness. In a more recent method [6], the matching problem is reduced to the clique detection problem, which can be solved efficiently but has quite larger space requirements. Another recent method [15] attempts to reduce the overall computational cost when matching a sample graph against a large set of prototypes, with an impressive matching time, but at the cost of an expensive preprocessing and of an exponential memory requirement. An interesting algorithm dealing only with the graph isomorphism problem is that of the *Nauty* library, described in [13], which is claimed to be the fastest available software for isomorphism testing; the adopted algorithm is based on group theory.

In this paper, we propose an improved release of a deterministic matching method for verifying both isomorphism and graph-subgraph isomorphism. The algorithm has general validity, since no constraints are imposed on the topology of the graphs to be matched, and can exploit semantic information if available.

The basic algorithm, developed by the same authors, is described in detail in [2, 3, 4]. The major improvement presented in this paper is that the exploration of the search space is organized in such a way to significantly reduce memory requirements, making the algorithm suitable for matching graphs with thousands of nodes and branches, but also faster on medium sized graphs.

In the rest of the paper, after a brief description of the algorithm, we present the new data structures used for the search, and discuss their spatial complexity. Finally, the experimental comparison of the new algorithm with both its predecessor and the Ullmann's algorithm is illustrated. It is shown that the obtained results confirm the effectiveness of the improvements made.

2. The Algorithm

2.1 Overview

Given two graphs $G_1 = (N_1, B_1)$ and $G_2 = (N_2, B_2)$, a mapping $M \subset N_1 \times N_2$ is said to be an isomorphism iff it is a bijective function that preservers the branch structure of the two graphs, that is, M maps each branch of G_1 onto a branch of G_2 and vice versa. M is said to be a graph-subgraph isomorphism iff M is an isomorphism between G_2 and a subgraph of G_1. In this papers we will assume that the graphs involved are *directed graphs*, i. e. a branch (i, j) is to be considered different from (j, i). The extension of the algorithm to undirected graphs is however trivial.

The matching process can be suitably described by means of a State Space Representation (SSR) [16]. Each state s of the matching process can be associated to a partial mapping solution $M(s)$, which contains only a subset of the components of the mapping function M. A partial mapping solution univocally identifies two subgraphs of G_1 and G_2, say $G_1(s)$ and $G_2(s)$, obtained by selecting from G_1 and G_2

only the nodes included in the components of $M(s)$, and the branches connecting them. In the following we will denote by $M_1(s)$ and $M_2(s)$ the projection of $M(s)$ onto N_1 and N_2 respectively, while the sets of the branches of $G_1(s)$ and $G_2(s)$ will be denoted by $B_1(s)$ and $B_2(s)$ respectively.
A high-level description of the matching algorithm can be outlined at this point:

```
PROCEDURE Match(s)
    INPUT:  an intermediate state s; the initial state s0 has M(s0)=∅
    OUTPUT: the mappings between the two graphs

    IF M(s) covers all the nodes of G2 THEN
      OUTPUT M(s)
    ELSE
      Compute the set P(s) of the pairs candidate for inclusion in M(s)
      FOREACH (n, m)∈P(s)
        IF F(s, n, m) THEN
          Compute the state s´ obtained by adding (n, m) to M(s)
          CALL Match(s´)
        END IF
      END FOREACH
       Restore data structures
    END IF
END PROCEDURE
```

where $F(s, n, m)$ is a boolean function (called *feasibility function*) that is used to prune the search tree. If its value is *true*, it is guaranteed that the state $s´$ obtained adding (n, m) to s is a partial isomorphism if s is; hence the final state is either an isomorphism between G_1 and G_2, or a graph-subgraph isomorphism between a subgraph of G_1 and G_2. Moreover, F will also prune some states that, albeit corresponding to an isomorphism between $G_1(s)$ and $G_2(s)$, would not lead to a complete matching solution.

2.2 Definition of the set $P(s)$ and of the feasibility function $F(s, n, m)$

Before detailing the construction of $P(s)$ and the computation of $F(s, n, m)$, we have to introduce some more notations. Given a graph G and one of its nodes n, we call Pred(G, n) (the *predecessors* of n) the set of nodes of G from which a branch originates that ends in n. Similarly, we call Succ(G, n) (the *successors* of n) the set of nodes of G that are the destination of a branch starting from n. We define the *out-terminal set* $T^{out}_1(s)$ as the set of nodes of G_1 that are not in $M_1(s)$ but are successors of a node in $M_1(s)$, and define the *in-terminal set* $T^{in}_1(s)$ as the set of nodes that are not in $M_1(s)$ but are predecessors of a node in $M_1(s)$. Analogously we define $T^{out}_2(s)$ and $T^{in}_2(s)$.

The set $P(s)$ is constructed as follows: if both $T^{out}_1(s)$ and $T^{out}_2(s)$ are not empty, then

$$P(s) = T_1^{out}(s) \times \left\{\min T_2^{out}(s)\right\}$$

where the min refers to the node in $T^{out}_2(s)$ which has the smallest label (actually, any other total ordering criterion could be used). If instead both $T^{out}_1(s)$ and $T^{out}_2(s)$ are empty, and both $T^{in}_1(s)$ and $T^{in}_2(s)$ are not, then

$$P(s) = T_1^{in}(s) \times \left\{\min T_2^{in}(s)\right\}$$

Finally, if all the four terminal sets are empty, then

$$P(s) = (N_1 - M_1(s)) \times \left\{\min\left(N_2 - M_2(s)\right)\right\}$$

In case that only one of the in-terminal sets or only one of the out-terminal sets is empty, it can be demonstrated that the state s cannot be part of a matching, and it is not further explored. It can be shown that this definition of $P(s)$ ensures that the search algorithm never visit the same state twice.

Now let us turn our attention to the feasibility function. Its expression is dependent on the desired type of mapping , but the rationale that lies behind the expression is the same for all cases. In order to evaluate $F(s, n, m)$ the algorithm examines all the nodes connected to n and m; if such nodes are in the current partial mapping (i.e. they are in $M_1(s)$ and $M_2(s)$), the algorithm checks if each branch from or to n has a corresponding branch from or to m and vice versa. Otherwise, the algorithm counts how many nodes are in $T^{in}_i(s)$, $T^{out}_i(s)$ and $(N_i\text{-}M_i(s)\text{-}T^{in}_i(s)\text{-}T^{out}_i(s))$; for the isomorphism these counts must be equals for n and m, while for the graph-subgraph isomorphism, the count relative to the small graph must be less than or equal to the count for the large graph. More details can be found in [3].

If the nodes and the branches of the graphs being matched also carry semantic attributes, another condition must also hold for $F(s, n, m)$ to be true; namely the attributes of the nodes and of the branches being paired must be *compatible*, in a sense that must be defined by the application; in the stricter case, attribute equality is needed, but there can be cases where a looser meaning of compatibility can be more appropriate.

2.3 Data structures and other implementation issues

In order to make the algorithm run with an acceptable time and space complexity also on large graphs, it is important to employ well devised data structures for performing the computation of $P(s)$ and of $F(s, n, m)$. In our new implementation of the algorithm, the following data structures are used (besides the ones needed to store the graphs being matched):

- two vectors `core_1` and `core_2`, whose dimensions correspond to the number of nodes in G_1 and G_2 respectively, containing the current mapping; in particular, `core_1`[n] contains the index of the node paired with n, if n is in $M_1(s)$, and the distinguished value `NULL_NODE` otherwise; the same encoding is used for `core_2`.

- four vectors `in_1`, `out_1`, `in_2`, `out_2`, whose dimensions are equal to the number of nodes in the corresponding graphs, describing the membership of the terminal sets. In particular, `in_1`[*n*] is non-zero if *n* is either in $M_1(s)$ or in $T_1^{in}(s)$; similar definitions hold for the other three vectors. The actual value stored in the vectors is the depth in the SSR tree of the state in which the node entered the corresponding set.

The above arrays are shared among all the states, hence the storage required by the algorithm is proportional to the number of nodes of the two graphs. Besides these vectors, some scalar variables are used, which are duplicated for each state *s*: the current depth of the state (which is also the number of pairs in the current mapping), the number of nodes in each of the terminal sets, and the pair of nodes that were added to the current state with respect to its direct ancestor.

Using the vectors described above, the tests for the membership of the various sets require a constant time; for example, to check whether node *n* is in $T_1^{in}(s)$, the algorithm has to test whether `in_1[n]>0` and `core_1[n]==NULL_NODE` It follows that the computation of *P*(*s*) can be done in a time in the worst case proportional to $|N_1|+|N_2|$, while the computation of *F*(*s*, *n*, *m*) can be performed in a time proportional to the number of the branches involving *n* and *m*.

It is important to note that all the vectors have the following property: if an element is non-null in a state *s* (where non-null means different from `NULL_NODE` for `core_1` and `core_2` and different from zero for the other vectors), it will remain non-null in all the states descending from *s*. This property, together with the depth-first strategy of the search, is used to avoid the need to store a different copy of the vectors for each state: when the algorithm backtracks, it restores the previous value of the vectors, using the variables holding the last added pair for `core_1` and `core_2`, and using the depth for the other vectors. This operation can be performed in a time proportional to the number of branches connected to the last pair of nodes. These clean-up operations corresponds to the step "*Restore data structures*" in the outline of the algorithm presented in subsect. 2.1; as a matter of facts, this step has been added in this release, since no special restoring action was needed in the previous version of the algorithm.

The memory requirement, with respect to the number of nodes *N*, is quite lower than in other similar algorithms. In fact, except for the six vectors shared among the states, each state need a constant (and small) amount of memory, and the depth-first search strategy ensures that there can be at most *N* states in memory at a time. Since the size of the vectors is *N*, it follows that the memory required is O(*N*), with a small constant factor. For comparison, the previous release of the algorithm has a memory requirement which is $O(N^2)$ [3], and Ullmann's algorithm [17] has a requirement which is $O(N^3)$. This fact constitutes a double advantage: larger graphs can be dealt with, and for medium sized graphs the smaller memory footprint allows a more proficient use of cache memories.

3. Experimental results

In order to verify the effectiveness of the proposed algorithm, a test has been performed on a subset of a large graph database which is described in [7]. In particular, about 3000 pairs of graphs have been selected, for which a graph/subgraph isomorphism exists. In the selected pairs, the subgraph always contain 20% of the nodes of the complete graph; however, tests on subgraphs with different percentages of nodes have shown similar results.

On these graphs we have measured the time required for finding all the graph-subgraph isomorphisms using the new version of our algorithm, which will be denoted from now on as **VF2**. These computation times have been compared with the ones of our previous version (called **VF**, see [3, 4]), and the ones of Ullmann's algorithm.

This latter has been chosen for comparison because it is well known and widely used for graph-subgraph isomorphism, and shows a quite good performance with respect to other techniques [14], especially when little or no search space pruning can be done using node/branch attributes. In [8] a more extensive benchmarking is presented for graph isomorphism, using a larger number of algorithms (including VF and VF2) available for that problem, and graphs of up to 1000 nodes.

In fig. 1 the results obtained on randomly generated graphs are presented, using two distinct values for the parameter h described in [3, 7], which characterizes the density of the graphs. For each pair of graphs, we have evaluated the computation time ratios between either the Ullmann's or the VF algorithms, and the VF2 algorithm.

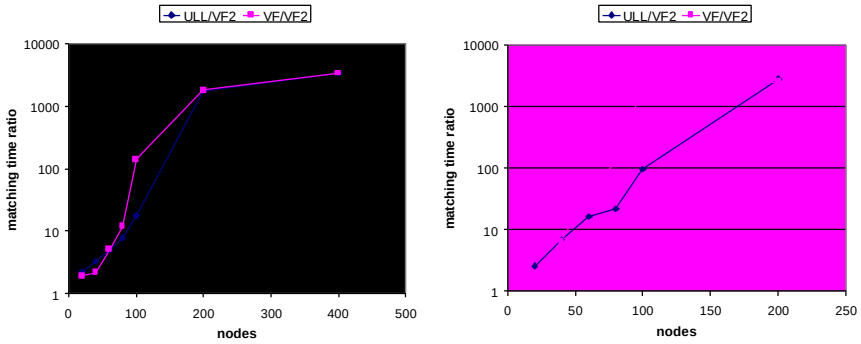

Fig. 1: Matching time ratios between Ullmann's algorithm and VF2 and between VF and VF2, for randomly generated graphs with h=0.005 (on the left) and h=0.01 (on the right). The vertical scale is logarithmic.

The plots show the average value of the time ratios, as a function of the number of nodes. Notice that, for graphs of 200 nodes or more, both Ullmann's and the VF algorithm are more than 1000 times slower than VF2.

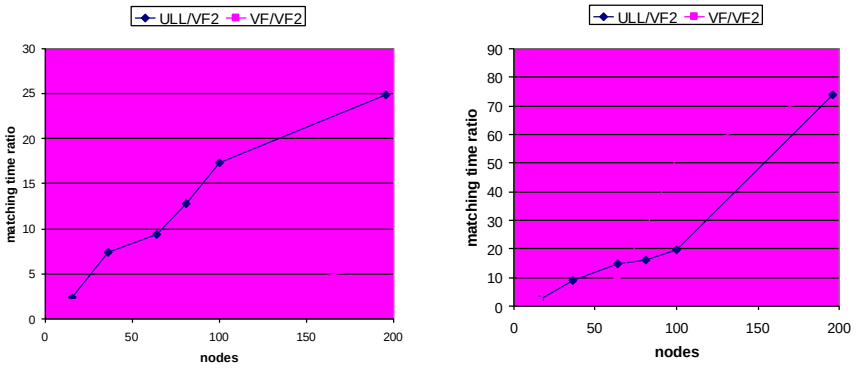

Fig. 2: Matching time ratios between Ullmann's algorithm and VF2 and between VF and VF2, for irregular 2-D meshes with r=0.4 (on the left) and r=0.6 (on the right).

Fig. 2 shows the same information for a different kind of graphs, which have been generated according to a model described in [3, 7] as *irregular* 2*D meshes*, using two distinct values for the parameter r, which characterizes the degree of irregularity of the meshes. These graphs are usually harder for matching algorithms than unconstrained randomly generated graphs, since they are more regular, and present more different subgraph isomorphisms. It can be seen that, although the speed improvement of VF2 is not so impressive as in the former case, VF2 is still several times faster than its competitors.

4. Concluding remarks

In this paper an improved matching algorithm that can be used for both isomorphism and graph-subgraph isomorphism has been presented. The algorithm can exploit semantic information attached to nodes and branches, when available. A remarkable feature of the algorithm is the small memory requirement, which makes it particularly suited for working with large graphs. The results obtained in a preliminary comparative test confirmed the effectiveness of the proposed approach.
The code implementing the new algorithm and Ullmann's, together with the graph database, is available on Internet at the site: http://amalfi.dis.unina.it/graph .

References

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

[2] L.P. Cordella, P. Foggia, C. Sansone, M. Vento, Subgraph Transformations for the Inexact Matching of ARG, *Computing* suppl. 12, pp. 43-52, 1998.

[3] L.P. Cordella, P. Foggia, C. Sansone, M. Vento, Performance evaluation of the VF Graph Matching Algoritmh, Proc. of the 10th ICIAP, IEEE Computer Society Press, pp. 1172-1177, 1999.

[4] L.P. Cordella, P. Foggia, C. Sansone, M. Vento, Fast Graph Matching for Detecting CAD Image Components, Proc. of the 15th Int. Conf. on Pattern Recognition, IEEE Computer Society Press, vol. 2, pp. 1038-1041, 2000.

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

[6] B. Falkenhainer, K.D. Forbus, D. Gentner, The structure-mapping engine: algorithms and examples, *Artificial Intelligence*, vol. 41 pp. 1-63, 1989/90.

[7] P. Foggia, C. Sansone, M. Vento, A Database of Graphs for Isomorphism and Sub-Graph Isomorphism Benchmarking, Proceedings of the 3rd IAPR-TC15 Workshop on Graph based Representation (GbR2001), Italy, 2001.

[8] P. Foggia, C. Sansone, M. Vento, A performance comparison of five algorithms for graph isomorphism, Proceedings of the 3rd IAPR-TC15 Workshop on Graph based Representation (GbR2001), Italy, 2001.

[9] M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman & co., New York, 1979.

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

[11] W. G. Kropatsch, M. Burge, S. Ben Yacoub, N Selmaoui, Dual Graph Contraction with LEDA, *Computing* suppl. 12, pp. 101-110, 1998.

[12] E. M. Luks, Isomorphism of Graphs of bounded valence can be tested in polynomial time, *Journal of Computer System Science*, pp. 42-65, 1982.

[13] B. D. McKay, Practical Graph Isomorphism, *Congressus Numerantium*, vol. 30, pp. 45-87, 1981.

[14] B. T. Messmer, *Efficient Graph Matching Algorithms for Preprocessed Model Graphs*, Ph.D. Thesis, Inst. of Comp. Sci. and Applied Mathematics, University of Bern, 1996.

[15] B.T. Messmer, H. Bunke, A decision tree approach to graph and subgraph isomorphism detection, *Pattern Recognition*, vol. 32, pp. 1979-1998,1999.

[16] N.J. Nilsson, *Principles of Artificial Intelligence*, Springer-Verlag, 1982.

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