Challenging the Time Complexity of Exact Subgraph Isomorphism for Huge and Dense Graphs with VF3

Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento

Abstract—Graph matching is essential in several fields that use structured information, such as biology, chemistry, social networks, knowledge management, document analysis and others. Except for special classes of graphs, graph matching has in the worst-case an exponential complexity; however, there are algorithms that show an acceptable execution time, as long as the graphs are not too large and not too dense. In this paper we introduce a novel subgraph isomorphism algorithm, VF3, particularly efficient in the challenging case of graphs with thousands of nodes and a high edge density. Its performance, both in terms of time and memory, has been assessed on a large dataset of 12,700 random graphs with a size up to 10,000 nodes, made publicly available. VF3 has been compared with four other state-of-the-art algorithms, and the huge experimentation required more than two years of processing time. The results confirm that VF3 definitely outperforms the other algorithms when the graphs become huge and dense, but also has a very good performance on smaller or sparser graphs.

Index Terms—Graphs, graph matching, graph isomorphism, subgraph isomorphism, graphs dataset

1 Introduction

Structural representations are widely employed in many application fields, such as biology, chemistry, social networks, databases, knowledge discovery and so on. These representations are essential for effectively describing all those data which can be seen as composed of parts suitably connected to each other. In these cases, data can be nicely represented by means of attributed graphs, where the nodes are used to represent the different parts, and the edges the relations existing between these parts; additional information (called labels or attributes) can be attached to both nodes and edges [1], [2]. Once graphs are adopted as descriptors, applications requiring the comparison between objects must solve some kind of graph matching problem [3], [4], [5], [6]. Broadly speaking, the matching problems can be divided into *exact matching*, where a strict correspondence is required between the two graphs (or at least between their subparts), and *inexact* or *error-tolerant matching*, where some degree of structural difference between the graphs is accepted [7]. Many applications need to determine if, how many times, and where, a substructure of interest is present within the whole structure. This common kind of exact matching, called *subgraph isomorphism*, consists in finding all the possible subgraphs of a reference graph that are structurally equivalent (i.e., equal up to a permutation of the nodes) to a given graph. Unfortunately, when no assumptions are made about the structure of the graphs, subgraph isomorphism has been proven to be NP-complete [7]; consequently, in the worst case, the computational complexity is factorial. Although this limit is intrinsic and cannot be overcome, it is often possible to reduce the computational complexity in the average case. The cornerstone to achieve this goal is the introduction of heuristics that exploit some irregularities in the graphs, occurring both in their structure and/or in their nodes and edges labels. However, specific heuristics can provide advantages on some kinds of graphs but disadvantages on others; no strategy is definitely better in all the situations, so we have assisted to the birth of many algorithms aimed at addressing effectively subgraph isomorphism.

Although good heuristics may provide an acceptable performance on graphs with hundreds of nodes, the growing need of processing very large graphs, makes it necessary to improve day by day the performance of the existing algorithms, so as to make faceable some problems that a few years ago were considered practically intractable. For instance, in biological applications, protein structures easily reach tens of thousands of nodes, and genomics data are even larger. In Semantic Web, crowdsourcing projects have led to the availability of huge knowledge bases stored as graphs; consider that the DBPedia knowledge base has more than 4 millions nodes. Even larger graphs can be obtained from the interconnections present in global social networks, such as Facebook or Twitter.

The increasing interest in this research area is proved by a number of initiatives related to the assessment and comparison of graph matching algorithms: For instance, in 2014 a contest on graph matching applied to bioinformatics has been

• *The authors are with the Department of Information and Electrical Engineering and Applied Mathematics, University of Salerno, Fisciano, SA 84084, ITALY. E-mail: {vcarletti, pfoggia, asaggese, mvento}@unisa.it.*

Manuscript received 28 July 2016; revised 13 Jan. 2017; accepted 15 Apr. 2017. Date of publication 23 Apr. 2017; date of current version 13 Mar. 2018. Recommended for acceptance by M. Tistarelli.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPAMI.2017.2696940

hosted at the International Conference on Pattern Recognition (ICPR) [8], and in 2016 a special issue on Advances in Graph-based Pattern Recognition has been published [9].

Their take-home message is that even the most efficient algorithms have a practical limit of applicability, that is graphs with a few hundreds of nodes, even though the labels attached to nodes and edges would allow to significantly reduce the matching time. Neverthless, graph size is not the only factor influencing the complexity of the matching: graph density, i.e., the ratio between the number of edges of the graph and the maximum number of edges in a graph having the same number of nodes, also plays an important role. In fact, sparse graphs (i.e., graphs with a low density), are often tractable up to several tens of thousands of nodes. As an extreme case, if the degree (i.e., the number of nodes adjacent to each node) is bounded by a constant, there are algorithms that solve the problem in polynomial time [10]. On the other hand, graphs that are both large and dense still pose a significant difficulty to matching algorithms. Incidentally, a performance improvement on these kinds of graphs has a far greater practical impact than it would have on easier graphs, since the saved time can be of several hours or even days.

In this paper we introduce VF3, a new algorithm for subgraph isomorphism especially devised to deal with huge graphs, still problematic for most of the existing state-of-the-art algorithms. Although we have focused our attention on subgraph isomorphism, the structure of VF3 is general and allows it to face different exact graph matching problems.

The basic structure of VF3 is derived from its predecessors VF [11] and VF2 [12]. VF, just after its presentation, revealed to be a fast algorithm compared with the ones of its time, as shown in [13]. VF2 improved the performance and has been one of the fastest general subgraph isomorphism algorithms at the time of its introduction in 2004 and for several years; like VF2, VF3 is based on a State Space Representation (SSR) [14] of the problem, and uses depth-first search, with the help of efficient heuristic rules to reduce the search space. As detailed later, the SSR formalizes a problem as some kind of visit of a suitably defined state space, searching for goal states that represent complete solutions. Of course, the actual performance depends on how many states it needs to visit (or, equivalently, how many states it is able to prune without cutting out goal states), and on how much time it spends for visiting each state.

VF3 significantly reduces both the number of explored states and the time spent on each state with respect to VF2, by means of new fast and effective heuristics; a relevant part of this work is devoted to an extensive characterization and comparative evaluation of its performance.

For the experimentation, we have decided to adopt a synthetic database of graphs, so as to be able to control some important graph parameters (such as the density, or the presence of labels) and relate them to the time and space performance of the algorithm; for this purpose we have build and published the Mivia Large Dense Graphs (LDG) dataset, as no existing dataset contains graphs that are both large (i.e., with more than 1,000 nodes) and dense (i.e., with density > 0.1). This dataset completes the one presented in [15], successfully used for comparing the performance of graph matching algorithms on large graphs of different types. The Mivia-LDG dataset has been generated according to the Erdős-Rényi random graph model, for three reasons: First, this model does not introduce biases or constraints on the edges incident to each node (in contrast to models that limit the node degrees or impose a regular edge structure); second, it has a number of edges that grows quadratically with the nodes (in contrast to models in which the edges grow linearly, thus reducing the density when the number of nodes increases); third, the model is easy to control, since it has only two parameters, i.e., the size and the density of the graphs. The total number of graphs in the database is 12,700, organized in pairs, ranging from 300 to 10,000 nodes, with densities up to $\eta = 0.4$.

TABLE 1
The Three Main Approaches to Subgraph Isomorphism and the Relative Most Known Algorithms

Approach	Algorithms
Tree Search	Ullmann [19], VF [11], VF2 [12], BM1 [20], L2G [21], RI/RI-DS [22][23]
Constraint Programming	McGregor [24], Larrosa and Valiente[25], Zampelli [26], Solnon [27], Ullmann [28], Kotthoff [29]
Graph Indexing	GraphQL [30], QuickSI [31], GADDI [32], SPath [33], TurboIso [34]

On this dataset, we have evaluated the matching time and the memory usage of VF3 in comparison with four other state of the art algorithms, highlighting the conditions more favorable to each of them, and how the performance of an algorithm remains stable with respect to the changes in size and density of the input graphs; note that the required memory is a very important issue in the case of huge graphs, since it may become the practical limit to the use of a certain algorithm.

Given the number and the size of the graphs in the dataset, the experimental evaluation has required the impressive time of 19,749 processing hours, corresponding to about 822 days.

The paper is organized as follows: In Section 2 we present the state of the art on subgraph isomorphism. Section 3, after the problem formalization, presents the structure of VF3, the heuristic rules used to prune the search space as well as other relevant optimizations. Finally, in Section 4 we describe the dataset and the results of the experimentation.

2 State of the Art

A review of scientific literature concerning subgraph isomorphism algorithms used in Pattern Recognition can be found in [7], [16], [17], [18].

From these sources, it emerges that the exact graph matching algorithms proposed up to now are based on three main paradigms (see Table 1): Tree Search, Constraint Programming and Graph Indexing.

The Tree Search approach includes many algorithms coming from the field of artificial intelligence that deal with the problem by formulating it within a State Space Representation where each state represents a partial matching. The solution is obtained by searching inside the state space, usually by adopting a depth-first search with backtracking; the solution is determined incrementally: at each new state, a pair of nodes is added to the current solution, after

checking if the addition is consistent with the constraints of the subgraph isomorphism and with the specific heuristic used by the algorithm. If a point where no other pair can be added is reached, the algorithms backtrack, removing the previous pair and trying a new one. Depth-first search has the advantage of not requiring all the states to be kept in memory: The maximum number of allocated states is proportional to the number of nodes, and thus if the space for each state is constant, the algorithm has a linear memory complexity with respect to the number of nodes.

The most widely known tree search based algorithms are Ullmann [19], VF [11] and VF2 [12] that have been the state of the art for almost ten years. Recently, new algorithms have been proposed as BM1 [20], L2G [21] and RI/RI-DS [22]; among them, RI [22], [23] is noteworthy as it revealed to be able to process large graphs, as shown in [8] and [23].

These algorithms essentially differ in the heuristic rules used for pruning states not leading to solutions; a common trade-off is between the effectiveness of the pruning induced by the rules and the computational cost required for their evaluation. Some of the mentioned algorithms (such as RI) adopt very simple rules, that are barely more selective than the necessary conditions that ensure the correctness of the found solutions, but are very quick to be evaluated. So they are very fast to find the solutions in simple cases, even when the graphs are quite large, when the sparseness of the edges and the labels substantially reduce the number of possible states. On the other hand, for complex graphs (i.e. dense graphs, with few or no labels), it could be effective to use a more sophisticated pruning criterion, able to avoid the exploration of states even when the application of the bare correctness criteria would not have ruled them out. In these cases, although the time spent on each state is longer, the algorithm may be much more effective as the total number of visited states drops.

Another important approach is based on constraint programming, initially proposed by McGregor [24] in 1979, and improved by Larrosa and Valiente [25], Zampelli [26], Solnon [27], Ullmann [28] and Kotthoff [29]. These algorithms work by filtering out, among all the possible node pairs, the ones that are surely not contained in the solution. They preliminarily compute a *domain* of compatibility for each node of the graph and iteratively reduce the domains by propagating constraints on the structure of the mapping, until only few candidate matchings remain. The advantages of these methods derive from their ability to detect, by means of constraint propagation, incompatibilities in the assignment of nodes not immediately adjacent; furthermore, once an incompatibility has been discovered, it is stored so as to avoid its successive re-evaluation. So, in principle, these algorithms should be more effective on graphs with repetitive substructures. Their main drawback is the amount of memory required to achieve the matching; in the worst case they have a quadratic space complexity due to their need to store the domains of compatibility, while several algorithms based on tree search show a linear memory complexity.

Finally, the graph indexing approach includes algorithms that retrieve, from a database of graphs, the ones containing a given pattern. Pure graph indexing algorithms deal with the problem of testing if a pattern graph is inside a target graph, without identifying where and how many times. They compute a *graph index*, that is a vector or a tree of features representative of the structural and semantic information of a graph; successively the index is used to test the presence of the pattern inside each target graph. Of course, the more representative the index is, the higher its precision in retrieving the desired target and its computational cost is. It is important to make it evident that the index, generally, does not provide any guarantee about the isomorphism: Two not isomorphic structures may have the same index. Many algorithms generalize this approach to subgraph isomorphism. The index is used to search for all the subgraphs in the target graph that have the same index; successively the solution is refined by checking the isomorphism constraints and removing all the inconsistent matchings. These algorithms are convenient in applications where the addition of new graphs is much less frequent than the querying of existing ones, and thus the preprocessing time needed to compute the indices is well amortized. Furthermore, they are effective when the graphs are too large to be kept in memory but the indices fit the memory size: In these cases, the index based filtering can greatly reduce the accesses to the slow secondary memory. Recent algorithms are: GraphQL [30], QuickSI [31], GADDI [32], SPath [33] and TurboIso [34].

3 The Proposed Algorithm VF3

Before presenting the proposed algorithm, it is necessary to formally introduce the problem of subgraph isomorphism together with the adopted formalism.

A graph $G = (V, E)$ is composed by a set V of nodes and a set E of edges connecting these nodes, being $E \subset V \times V$. G becomes *labeled* when a set of labels is assigned to the nodes and the edges of G (L_v and L_e respectively); a node label function $\lambda_v : V \to L_v$ provides for each node the corresponding label; similarly, the edge label function $\lambda_e : E \to L_e$ assigns labels to the edges.

So, given $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, and a *mapping* $M \subset V_1 \times V_2$ as an injective correspondence between the nodes of the two graphs, we say that it satisfies a subgraph isomorphism between G_1 and G_2, if the following conditions hold, having indicated as $\mu(u)$ the node in G_2 associated to $u \in G_1$ and as $\mu^{-1}(v)$ the node in G_1 associated to $v \in G_2$

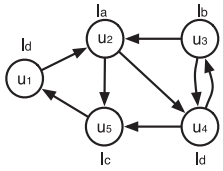
$$\forall u \in V_1 \quad \exists \mu(u) \in V_2 : (u, \mu(u)) \in M \tag{1}$$

$$\forall u, u' \in V_1 \quad u \neq u' \Rightarrow \mu(u) \neq \mu(u') \tag{2}$$

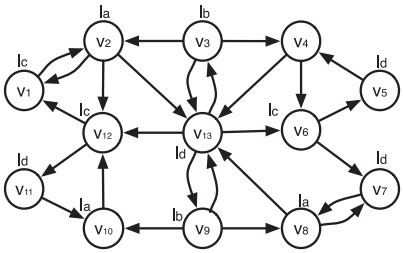
$$\forall (u, u') \in E_1 \quad \exists (\mu(u), \mu(u')) \in E_2 \tag{3}$$

$$\forall u, u' \in V_1, \quad (\mu(u), \mu(u')) \in E_2 \Rightarrow (u, u') \in E_1. \tag{4}$$

Note that the role of G_1 and G_2 in the previous definitions is not symmetric; G_1 is the smaller graph, often called *pattern graph*, while G_2 is the bigger one, also called *target graph*. Subgraph isomorphism requires that for every couple of nodes u, u' in G_1, if the edge (u, u') is in G_1, then a corresponding edge $(\mu(u), \mu(u'))$ must exist in G_2; moreover, if the edge (u, u') does not exist in G_1, then the edge $(\mu(u), \mu(u'))$ must not exist in G_2. Note that some authors [22], [27], [28] use the term subgraph isomorphism



Graph G_1 (Pattern graph)



Graph G_2 (Target graph)

Sets of G_1	
V_1	$\{u_1, u_2, u_3, u_4, u_5\}$
E_1	$\{(u_1, u_2), (u_2, u_4), (u_2, u_5), (u_3, u_2), (u_3, u_4), (u_4, u_3), (u_4, u_5), (u_5, u_1)\}$
L_{v_1}	$\{l_a, l_b, l_c, l_d\}$
λ_{v_1}	$\{(u_1, l_d), (u_2, l_a), (u_3, l_b), (u_4, l_d), (u_5, l_c)\}$

Sets of G_2	
V_2	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}\}$
E_2	$\{(v_1, v_2), (v_2, v_1), (v_2, v_{12}), (v_2, v_{13}), (v_3, v_2), (v_3, v_4), (v_3, v_{13}), (v_4, v_6), (v_9, v_8), (v_4, v_{13}), (v_5, v_4), (v_6, v_5), (v_6, v_7), (v_7, v_8), (v_8, v_7), (v_8, v_{13}), (v_9, v_{10}), (v_9, v_{13}), (v_{10}, v_{12}), (v_{11}, v_{10}), (v_{12}, v_1), (v_{12}, v_{11}), (v_{13}, v_3), (v_{13}, v_6), (v_{13}, v_9), (v_{13}, v_{12})\}$
L_{v_2}	$\{l_a, l_b, l_c, l_d\}$
λ_{v_2}	$\{(v_1, l_c), (v_2, l_a), (v_3, l_b), (v_4, l_a), (v_5, l_d), (v_6, l_c), (v_7, l_d), (v_7, l_a), (v_9, l_b), (v_{10}, l_a), (v_{11}, l_d), (v_{12}, l_c), (v_{13}, l_d)\}$

Mapping	
M	$\{(u_1, v_5), (u_2, v_4), (u_3, v_3), (u_4, v_{13}), (u_5, v_6)\}$

Fig. 1. On the left the pattern and the target graph. On the right the corresponding sets together with the functions for obtaining node and edge attributes. As edges are unlabeled, the sets L_{e_1} and L_{e_2} are not present and the functions λ_{e_1} and λ_{e_2} not defined. On the bottom right the mapping function satisfying the subgraph isomorphism.

in a slightly weaker way: Indeed, they relax the constraints by removing Equation (4). The relaxation allows the target graph G_2 to have extra edges. As in [7], we will refer to this latter matching typology as *monomorphism*; in this paper we focus our attention on subgraph isomorphism.

For attributed graphs, to the previous conditions it is necessary to add the following ones, ensuring that the mapped nodes and edges must have the same labels

$$\forall u \in V_1, \lambda_{v_1}(u) = \lambda_{v_2}(\mu(u)) \tag{5}$$

$$\forall (u, u') \in E_1, \lambda_{e_1}(u, u') = \lambda_{e_2}(\mu(u), \mu(u')). \tag{6}$$

VF3 is devised for solving the subgraph isomorphism problem. Similarly to RI [22] and VF2 [12], it uses a SSR and the depth-first strategy with backtracking, because of its efficiency in terms of space and time. In particular, the resolution process is formulated as a search over a potentially very large space of states, each representing a partial solution; states are connected to other states by means of a transition operator. In our case, a state represents a partial mapping between the two given graphs, i.e., a mapping involving a subset of the nodes of the two graphs; if the partial mapping satisfies the constraints of the subgraph isomorphism, we have a *consistent state*. All the consistent states whose mapping is complete, i.e., it uses all the nodes of the pattern graph, represent admissible solutions to the problem (*goal states*), while *dead states* are consistent states that surely will not generate consistent descendants. Therefore, the aim of the algorithm is to start from an *initial state*, representing an empty mapping, and then explore the state space to reach all the goal states, if they exist, minimizing the number of visited states.

3.1 SSR Representation of the Problem

As previously mentioned, a generic state s of the SSR represents a *partial mapping* $\widetilde{M}(s) \subseteq M$ between the graphs G_1 and G_2. We call *core set* $\widetilde{M}_1(s)$ the set of nodes of graph G_1 belonging to $\widetilde{M}(s)$, i.e., $\{u \in V_1 : \exists (u, v) \in \widetilde{M}(s), v \in V_2\}$; similarly, $\widetilde{M}_2(s)$ is the core set $\{v \in V_2 : \exists (u, v) \in \widetilde{M}(s), u \in V_1\}$. Moreover, we will use the notation $\widetilde{\mu}(s, u)$ to denote the node in V_2 matched with u in the partial mapping $\widetilde{M}(s)$, and $\widetilde{\mu}^{-1}(s, v)$ the node in V_1 matched with v.

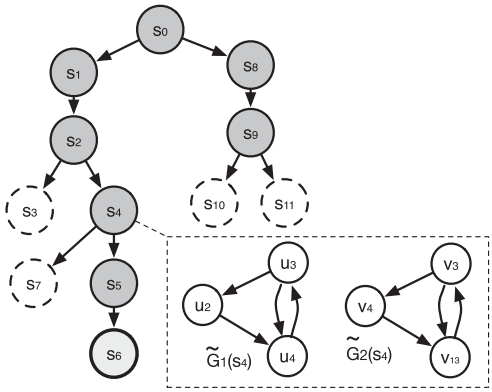
We say that a state s is *consistent* if the corresponding partial mapping $\widetilde{M}(s)$ satisfies Equations (1)-(6) when the latter are restricted to the subsets of nodes $\widetilde{M}_1(s)$ and $\widetilde{M}_2(s)$ and to the corresponding edges. In the following we will call $\widetilde{G}_1(s)$ the subgraph of G_1 containing only the nodes in $\widetilde{M}_1(s)$ and the corresponding edges. A similar definition applies to $\widetilde{G}_2(s)$. For subgraph isomorphism, consistency implies that $\widetilde{G}_1(s)$ and $\widetilde{G}_2(s)$ are isomorphic.

An example of these sets is shown in Fig. 2, which refers to the state space explored by VF3 during the matching of the graphs in Fig. 1. For instance, the state s_4 is consistent, thus the subgraphs $\widetilde{G}_1(s_4)$ and $\widetilde{G}_2(s_4)$, derived from the partial mapping $\widetilde{M}(s_4)$, are isomorphic.

3.1.1 State Space Exploration

An overview of VF3 is reported in Algorithms 1 and 2: After a preliminary phase aimed at precalculating some sets (see Sections 3.2, 3.3 and 3.4), VF3 calls the recursive Match procedure, which starts from the state s_0, that represents the empty mapping $\widetilde{M}(s_0) = \emptyset$, searching for one or more goal states. Each new state s_n is generated from a *parent state* s_c by adding a new ordered couple (u_n, v_n), such that $u_n \notin \widetilde{M}_1(s_c)$ and $v_n \notin \widetilde{M}_2(s_c)$, found by the GetNextCandidate procedure (line 8 of Algorithm 2). In practice, the state transition from s_c to s_n corresponds to the addition of the node u_n to $\widetilde{G}_1(s_c)$ and the node v_n to $\widetilde{G}_2(s_c)$ (line 12 of Algorithm 2).

Clearly, a trivial solution for finding all the goal states could be to generate all the possible states. However, due to the combinatorial nature of the problem, the computational cost of this exhaustive exploration is factorial with respect to the size of the graphs. It is possible to prove that a non consistent state s_c will not generate any successive consistent states; indeed, if $\widetilde{G}_1(s_c)$ is not isomorphic with $\widetilde{G}_2(s_c)$ then $\widetilde{G}_1(s_n)$ and $\widetilde{G}_2(s_n)$ obtained by adding u_n to $\widetilde{G}_1(s_c)$ and v_n to $\widetilde{G}_2(s_c)$ cannot be isomorphic. However, it may happen that, even if the state s_n is consistent, after a certain number



s_i	$\widetilde{M}(s_i)$
s_0	$\emptyset$
s_1	$\{(u_3, v_3)\}$
s_2	$\{(u_3, v_3), (u_4, v_{13})\}$
s_3	*inconsistent*
s_4	$\{(u_3, v_3), (u_4, v_{13}), (u_2, v_4)\}$
s_5	$\{(u_3, v_3), (u_4, v_{13}), (u_2, v_4), (u_5, v_6)\}$
$\boldsymbol{s_6}$	$\boldsymbol{\{(u_3, v_3), (u_4, v_{13}), (u_2, v_4), (u_5, v_6), (u_1, v_5)\}}$
s_7	*inconsistent*
s_8	$\{(u_3, v_9)\}$
s_9	$\{(u_3, v_9), (u_4, v_{13})\}$
s_{10}	*inconsistent*
s_{11}	*inconsistent*

Fig. 2. The state space (on the left) explored by VF3. The state s_6 is a goal state as it represents a coherent complete matching. In white, the inconsistent states s_3, s_7, s_{10}, s_{11}. They are not generated as they do not satisfy the feasibility rules (see Section 3.6). Details about the reason why s_3, s_7, s_{10} are inconsistent are given Figs. 6, 7 and 8.

of steps it cannot generate any consistent descendants and, thus, cannot be part of the path leading to a goal state. Of course, the detection of such situations would help to further reduce the number of explored states by avoiding the generation of consistent, but unfruitful, states. In order to achieve this aim, VF3 uses a set of rules which checks if the new couple (u_n, v_n) is *feasible*: Indeed, the algorithm verifies if the addition of (u_n, v_n) to s_c generates a consistent state s_n; in addition, it is able to detect k steps in advance if the state s_n will not have any consistent descendants, a k-lookahead. As we will see in the next section, the IsFeasible function combines a check for consistency with 1- and 2-lookahead. More details concerning the design and the implementation of such procedures are provided in the following sections.

Algorithm 1. VF3. The Inputs are the Graphs G_1 and G_2. The Procedure Returns the Set *Solutions* Containing the Found Solutions

```
1: function VF3(G1, G2)
2:     Solutions ← ∅
3:     Pf = ComputeProbabilities(G2, G1)
4:     N_G1 = GenerateNodeSequence(G1, P_feas)
5:     ClassifyNodes(G1, G2)
6:     (s0, Parent) = PreprocessPatternGraph(G1, N_G1)
7:     Match(s0, G2, G1, N_G1, Parent, Solutions)
8:     return Solutions
```

3.2 Feasibility Sets

Before entering into details of the proposed algorithm, we will introduce some important sets used during the matching process, in particular by the GetNextCandidate and IsFeasible procedures. We will call them *feasibility sets* and denote as $\widetilde{\mathcal{P}}_1$, $\widetilde{\mathcal{S}}_1$, $\widetilde{V}_1$, $\widetilde{\mathcal{P}}_2$, $\widetilde{\mathcal{S}}_2$ and $\widetilde{V}_2$.

Given a state s of the matching process we introduce the predecessor feasibility set $\widetilde{\mathcal{P}}_1$ as the set of all the nodes of G_1 having edges arriving in nodes of the core set $\widetilde{M}_1(s)$

$$\widetilde{\mathcal{P}}_1(s) = \{u \in V_1 - \widetilde{M}_1(s) : \exists u' \in \widetilde{M}_1(s) : (u, u') \in E_1\}. \quad (7)$$

Similar definitions are given for the successor set $\widetilde{\mathcal{S}}_1$

$$\widetilde{\mathcal{S}}_1(s) = \{u \in V_1 - \widetilde{M}_1(s) : \exists u' \in \widetilde{M}_1(s) : (u', u) \in E_1\}, \quad (8)$$

and for predecessor and successor feasibility sets of G_2, namely $\widetilde{\mathcal{P}}_2$ and $\widetilde{\mathcal{S}}_2$. Moreover, we introduce the set $\widetilde{V}_1$, representing the nodes of the graph G_1 that are not included neither in $\widetilde{M}(s)$, nor in $\widetilde{\mathcal{P}}_1$, nor in $\widetilde{\mathcal{S}}_1$

$$\widetilde{V}_1(s) = V_1 - \widetilde{M}_1(s) - \widetilde{\mathcal{P}}_1(s) - \widetilde{\mathcal{S}}_1(s). \quad (9)$$

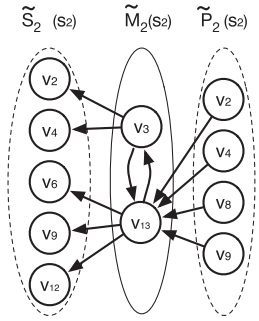
A similar definition can be given for $\widetilde{V}_2$. Fig. 3a shows the feasibility sets related to the state s_2 of the SSR expansion of Fig. 2.

Algorithm 2. The Matching Procedure Used by VF3. The Inputs are the Current State s_c, the Graphs G_1 and G_2 and the Two Structures, Namely N_{G_1} and *Parent*, Used During the Candidate Selection, that will be Described in the Following Sections. The Procedure Also Takes as Input the Set *Solutions* Where to Store the Found Solutions. The Procedure Returns *True* if at Least a Solution Exists, *False* Otherwise

```
1: function Match(s_c, G1, G2, N_G1, Parent, Solutions)
2:     if IsGoal(s_c) then
3:         Append(M̃(s_c), Solutions)
4:         return True
5:     if IsDead(s_c) then
6:         return False
7:     Set (u_c, v_c) = (ϵ, ϵ)
8:     (u_n, v_n) = GetNextCandidate(s_c, (u_c, v_c), N_G1, Parent, G1, G2)
9:     Result = False
10:    while (u_n, v_n) ≠ (ϵ, ϵ) do
11:        if IsFeasible(s_c, u_n, v_n) then
12:            s_n = s_c ∪ (u_n, v_n)
13:            if Match(s_n, G1, G2, N_G1, Parent, Solutions) is True then
14:                Result = True
15:        (u_n, v_n) = GetNextCandidate(s_c, (u_n, v_n), N_G1, Parent,
           G1, G2)
16:    return Result
```

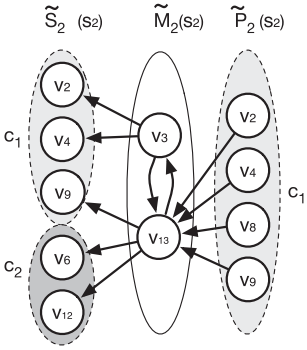
While the feasibility rules of VF2 were only based on the above sets, our experience with that algorithm has led us to improve the efficiency of those rules (i.e., reducing the number of visited states) by exploiting the structural asymmetries present in the graphs to be matched.

Partitioning the nodes of the graphs is a way for making the asymmetries evident; in particular, we partition the nodes of the two graphs using a *node classification function* ensuring



$\widetilde{\mathcal{P}}_2(s_2) = \{v_2, v_4, v_8, v_9\}$ $\widetilde{\mathcal{S}}_2(s_2) = \{v_2, v_4, v_6, v_9, v_{12}\}$

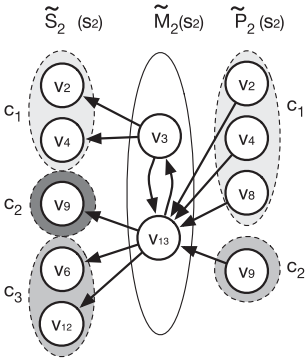
(a)



$\psi_{\mathbf{a}}(l_a) = c_1$
$\psi_{\mathbf{a}}(l_b) = c_1$
$\psi_{\mathbf{a}}(l_c) = c_2$
$\psi_{\mathbf{a}}(l_d) = c_2$

$\widetilde{\mathcal{P}}_2^{c_1}(s_2) = \{v_2, v_4, v_8, v_9\}$ $\widetilde{\mathcal{S}}_2^{c_1}(s_2) = \{v_2, v_4, v_9\}$
$\widetilde{\mathcal{P}}_2^{c_2}(s_2) = \emptyset$ $\widetilde{\mathcal{S}}_2^{c_2}(s_2) = \{v_6, v_{12}\}$

(b)



$\psi_{\mathbf{b}}(l_b) = c_1$
$\psi_{\mathbf{b}}(l_a) = c_2$
$\psi_{\mathbf{b}}(l_c) = c_3$
$\psi_{\mathbf{b}}(l_d) = c_4$

$\widetilde{\mathcal{P}}_2^{c_1}(s_2) = \{v_2, v_4, v_8\}$ $\widetilde{\mathcal{S}}_2^{c_1}(s_2) = \{v_2, v_4\}$
$\widetilde{\mathcal{P}}_2^{c_2}(s_2) = \{v_9\}$ $\widetilde{\mathcal{S}}_2^{c_2}(s_2) = \{v_9\}$
$\widetilde{\mathcal{P}}_2^{c_3}(s_2) = \emptyset$ $\widetilde{\mathcal{S}}_2^{c_3}(s_2) = \{v_6, v_{12}\}$
$\widetilde{\mathcal{P}}_2^{c_v}(s_2) = \emptyset$ $\widetilde{\mathcal{S}}_2^{c_4}(s_2) = \emptyset$

(c)

Fig. 3. The state s_2 of the SSR expansion (see Fig. 2); (a) unclassified feasibility set; b) and c) classified feasibility sets obtained by using two possible classification functions ψ_a and ψ_b defined on the graphs G_1 and G_2. The shading around the nodes identifies the class to which they belong they belong.

that if two nodes are in different partitions, they will never be matched in a consistent state. The converse may not be true: Even if two nodes are in the same partition, it does not mean that they can be matched in a consistent mapping. We propose here a redefinition of the feasibility sets incorporating these partitions, so making the algorithm able to exploit this additional information for pruning unfruitful states.

The node classification function may embody any partitioning criterion that makes sense for the problem or the application at hand. A very general criterion is to base the partitioning on the node labels, since by Equation (5) we know that nodes with different labels cannot be matched together. However this criterion can be complemented with other kinds of semantic or structural information: For example, in a chemoinformatics application we may use the fact that the node is or is not part of an aromatic ring. Formally, we define a node classification function $\psi : V_1 \cup V_2 \rightarrow C$, that assigns each node to a class $c_i \in C = \{c_1, c_2, \ldots, c_q\}$, such that $(u, v) \in M \Rightarrow \psi(u) = \psi(v)$, i.e., nodes mapped each other in any of the possible solutions M must belong to the same class.

Given an arbitrary classification function ψ, we may preliminarily introduce the sets of all the nodes $C_i(V)$ assigned to class c_i

$$C_i(V) = \{u \in V : \psi(u) = c_i, \quad i = 1, \ldots, q\}. \tag{10}$$

Successively, we define the *classified feasibility sets* starting from those obtained with unclassified nodes (i.e., $\widetilde{\mathcal{P}}_1$, $\widetilde{\mathcal{S}}_1$, $\widetilde{\mathcal{P}}_2$, $\widetilde{\mathcal{S}}_2$, $\widetilde{V}_1$, $\widetilde{V}_2$) and imposing that each set is split into a number of subsets equal to number of the different classes, as follows:

$$\widetilde{\mathcal{P}}_1^{c_i}(s) = \{u \in \widetilde{\mathcal{P}}_1(s) : \psi(u) = c_i\} \tag{11}$$

$$\widetilde{\mathcal{S}}_1^{c_i}(s) = \{u \in \widetilde{\mathcal{S}}_1(s) : \psi(u) = c_i\} \tag{12}$$

$$\widetilde{V_1^{c_i}}(s) = \{u \in \widetilde{V}_1(s) : \psi(u) = c_i\}. \tag{13}$$

Similarly, it is possible to define the corresponding sets used for the graph G_2, namely $\widetilde{\mathcal{P}}_2^{c_i}$, $\widetilde{\mathcal{S}}_2^{c_i}$ and $\widetilde{V}_2{}^{c_i}(s)$.

An important consideration for the design of the function ψ is that there is a trade-off between the time needed to compute the feasibility sets (many classes imply many sets and more time to compute them) and the improvement in the pruning ability (many classes imply more possibilities of detecting inconsistencies). Thus it may be convenient to reduce the number of classes with respect to the number made possible by the discriminant information used in ψ; for example, if ψ is based on node labels, each class may represent a group of labels, instead of a single one, in order to reduce the feasibility sets computation time.

Figs. 3b and 3c show the feasibility sets obtained by using different classification functions, ψ_a and ψ_b: ψ_a divides the nodes in two classes, one for nodes having labels l_a and l_b and another one for nodes with labels l_c and l_d; the classification function ψ_b, instead, introduces four classes, one for each different label.

3.3 Determining State Successors

It can be easily verified that, using a simplistic way of computing the state successors, each state s whose mapping $\widetilde{M}(s)$ contains k couples could be reachable from $k!$ different paths; thus the algorithm is designed so as to carefully avoid that a single state is generated more times.

The solution to this problem can be found by reshaping the state space as a tree instead of a graph. This can be obtained by defining an arbitrary total order relationship (denoted by $\prec$) over the nodes of the pattern graph and exploring the nodes with that particular order. It is evident that this strategy can be considered satisfactory as the order of the couples in M is not relevant for the correctness of the solution.

TABLE 2
Probabilities P_f of the Nodes of G_1 (see Fig. 1)

Node	$\sum \mathbf{P_d^{out}}$	$\sum \mathbf{P_d^{in}}$	$\mathbf{P_l}$	$\mathbf{P_f}$
u_1	1.000	1.000	0.308	0.308
u_2	0.615	0.692	0.308	0.131
u_3	0.615	1.000	0.154	0.095
u_4	0.615	0.692	0.308	0.131
u_5	1.000	0.692	0.231	0.160

Therefore, during the search of a new candidate couple, the algorithm must ignore any pair (u, v) if $\widetilde{M}(s)$ already contains a node u' such that $u \prec u'$.

It is possible to simply check that any arbitrary order relationship could work. VF3 introduces a novel ordering relationship of the pattern graph using suited information about the target graph, so as to obtain an optimized exploration of the search tree. The idea of using an ordering function has been also used in RI and other algorithms; however the choice of the actual ordering function significantly affects the performance of an algorithm. Differently from RI (that uses only the structural information of the pattern graph), the order relationship introduced in VF3 combines the structural and semantic information of the pattern graph with those of the target graph.

In particular, the rationale is to organize the exploration of the state space so as to give priority to the nodes having more constraints, i.e., the ones with the smallest probability of finding a match in G_2, or the ones having more connections to already matched nodes, so that those constraints can be exploited early in the search to prune the search space.

The visiting order is determined by the procedure GenerateNodeSequence, that generates the *node exploration sequence* N_{G_1}, that is a permutation of the nodes in G_1. The procedure estimates for each node $u \in G_1$ the probability $P_f(u)$ to find a node $v \in G_2$ that is compatible with u. Note that this estimate does not need to be extremely accurate: an error will affect the visiting order of the nodes, but will not have an impact on the correctness of the solution. Since the matching time and space is our primary concern, it is acceptable to have a weak estimator, if it can be computed with low temporal and spatial complexity. We have used the Maximum Likelihood Estimation method, thus assuming a uniform prior distribution, and estimating probabilities by observed frequencies. We have assumed that $P_f(u)$ depends on the label $\lambda_{v1}(u)$, and on the in- and out-degree of u

$$P_f(u) = Pr(\lambda_{v2}(v) = \lambda_{v1}(u), d^{in}(v) \geq d^{in}(u), \\ d^{out}(v) \geq d^{out}(u)), \tag{14}$$

where d^{in} and d^{out} are the in-degree and the out-degree of a node. Instead of using the above joint distribution, whose computation has a worst-case complexity of $O(N^3)$, we have assumed that the three conditions are independent of each other, thus obtaining

$$P_f(u) = P_l(\lambda_{v1}(u)) \cdot \sum_{d' \geq d^{in}(u)} P_d^{in}(d') \cdot \sum_{d' \geq d^{out}(u)} P_d^{out}(d'). \tag{15}$$

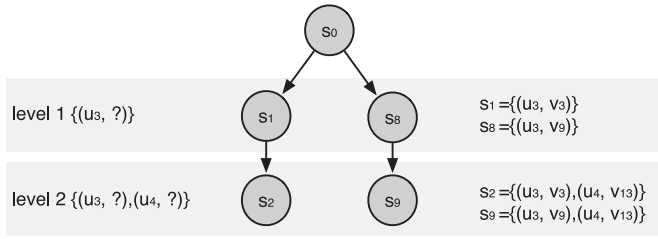


Fig. 4. The state space generated with the sequence N_{G_1}. At at each level, the nodes of G_1 are statically identified by N_{G_1}, while those of G_2 have to be dynamically selected.

where $P_l(l)$ is the probability of a node v having label l, $P_d^{in}(d)$ is the probability of having in-degree d and $P_d^{out}(d)$ of having out-degree d. Independence of these three terms is a weak assumption, but it reduces the worst-case complexity of the probability estimation to $O(N)$. Table 2 presents the values of P_f calculated for the nodes of G_1.

The procedure also takes into account the structural constraints coming from the nodes already in N_{G_1}. To this aim, we have introduced the concept of *node mapping degree*, denoted as d_M. Given a node $u \in G_1$, d_M is defined as the number of incoming and outgoing edges between u and all the nodes that are already inside N_{G_1}. So, at each step, when a new node is inserted in N_{G_1}, the procedure has to update the d_M of all the nodes $u \notin N_{G_1}$.

In more details, the GenerateNodeSequence procedure first uses d_M as the ordering criterion; if two or more nodes have the same d_M, they are sorted according to P_f; finally, if both d_M and P_f are equal, the nodes are sorted using the sum deg of their in- and out-degree. If also their degrees are equal, the choice is done randomly.

The exploration sequence generated by this procedure defines a predefined search path in the SSR over the pattern graph. As shown in Fig. 4, the states are only partially defined in advance. Indeed, they only contain nodes belonging to G_1 and need to be completed with the nodes of G_2 (the selection of the nodes of the latter will be explained in Section 3.5). In particular, each step of the sequence corresponds to a level of the depth-first search; depending on this, all the states at level i will be generated by adding to those at the previous level $i-1$, where the pattern node is fixed (it corresponds to the one at the position i in N_{G_1}) and is shared among all the states at that level.

In order to make clearer how the whole process works, Table 3 shows the steps in the computation of the node sequence N_{G_1} for the example graph G_1 from Fig. 1; the probabilities P_f are reported in Table 2. At the first step, all the nodes have the same d_M, so the procedure considers, for each node, the corresponding value of P_f; the first selected node is u_3 because has the lowest probability. Successively, the procedure selects u_4 because of its d_M. Then u_2 and u_5 for the same reason. Finally, u_1 is inserted.

3.4 State Structures Precalculation

The introduction of the sequence N_{G_1} allows VF3 to precalculate many information before and not during matching process, so as to significantly improve its performance. In particular, looking at Fig. 4, it is clear that the feasibility sets relative to graph G_1, say $\widetilde{\mathcal{P}}_1^{c_i}$, $\widetilde{\mathcal{S}}_1^{c_i}$, $\widetilde{V}_1{}^{c_i}(s)$ are equal for all the states belonging to the same level of the search tree. This

TABLE 3
The Computation of the Node Ordering of G_1 (See Fig. 1), Using the Probabilities in Table 2

Node		u_1	u_2	u_3	u_4	u_5	$\mathbf{N_{G_1}}$
$\mathbf{P_f}$		0.308	0.131	0.095	0.131	0.160	
deg		2	4	3	4	3	
				$\mathbf{d_M}$			
Steps	**0**	0	0	0	0	0	$\emptyset$
	1	0	1	–	2	0	$\{u_3\}$
	2	0	2	–	–	1	$\{u_3, u_4\}$
	3	1	–	–	–	2	$\{u_3, u_4, u_2\}$
	4	1	–	–	–	–	$\{u_3, u_4, u_2, u_5\}$
	5	1	–	–	–	–	$\{u_3, u_4, u_2, u_5, u_1\}$

property makes it possible to compute these sets in advance, only once for each level of the search tree. Furthermore, during this step, the algorithm assigns to each node u of G_1 a parent node $Parent(u)$ that is successively used by the GETNEXTCANDIDATE procedure (see Section 3.5).

In particular, the procedure PreprocessPatternGraph (see Algorithm 3) analyzes iteratively each node u in the sequence N_{G_1} and explores its neighbourhood. If the neighbor u' of u is a successor (a precessor, respectively) and it is not yet inserted in $\widetilde{\mathcal{S}}_1^{\psi(u')}$ ($\widetilde{\mathcal{P}}_1^{\psi(u')}$, respectively), then it is inserted. In case u' was not previously neither in $\widetilde{\mathcal{S}}_1^{\psi(u')}$ nor in $\widetilde{\mathcal{P}}_1^{\psi(u')}$, then u is set as $Parent(u')$. Note that the first node in N_{G_1} has not a parent. As an example, if the node u' is a successor of u and belongs to the class c_1, it will be inserted in the set $\widetilde{\mathcal{S}}_1^{c_1}$ at level i, considering that $u = N_{G_1}(i)$. Additionally, if u' has not a parent, then u will become the parent of u'.

The results of the application of the state structures precalculation to the example graph G_1 are reported in Table 4.

Algorithm 3. Procedure to Preprocess the Graph G_1 Given an Exploration Node Sequence N_{G_1}. It Returns the Associative Map *Parent*, Giving the Parent for Each Node in the Sequence

```
1: function PREPROCESSPATTERNGRAPH(G_1, N_{G_1}, T_1)
2:   i = 0
3:   for all u ∈ N_{G_1} do
4:     for all u' ∈ P_1(u) ∪ S_1(u) do
5:       c_i = ψ(u')
6:       if u' ∈ P_1(u) ∧ u' ∉ P̃_1^{c_i} then
7:         Put u' in P̃_1 at level i
8:         Parent(u') = u
9:       if u' ∈ S_1(u) ∧ u' ∉ S̃_1^{c_i} then
10:        Put u' in S̃_1 at level i
11:        Parent(u') = u
12:      i = i + 1
13:  return Parent
```

3.5 The Candidate Selection Algorithm

As previously stated, the GETNEXTCANDIDATE procedure, shown in Algorithm 4, finds out the next couple (u_n, v_n) to be explored for generating a new state s_n starting from the current state s_c, considering the last inserted couple (u_c, v_c).

As for the pattern graph, the node u_n is obtained as the one that follows u_c in the exploration sequence N_{G_1},

TABLE 4
Core and Feasibility Sets of G_1 (See Fig. 1), Precomputed for Each Level of the Search. The Parent of u_2 and u_4 is u_3, the Parent of u_5 is u_4, the Parent of u_1 is u_2.

Level	$\widetilde{\mathbf{M}_1(\mathbf{s})}$	$\widetilde{\mathcal{P}}_1^{c_1}$	$\widetilde{\mathcal{P}}_1^{c_2}$	$\widetilde{\mathcal{P}}_1^{c_3}$	$\widetilde{\mathcal{P}}_1^{c_4}$	$\widetilde{\mathcal{S}}_1^{c_1}$	$\widetilde{\mathcal{S}}_1^{c_2}$	$\widetilde{\mathcal{S}}_1^{c_3}$	$\widetilde{\mathcal{S}}_1^{c_4}$
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{u_3\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{u_4\}$	$\{u_2\}$	$\emptyset$	$\emptyset$	$\{u_4\}$
2	$\{u_3, u_4\}$	$\{u_2\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{u_2\}$	$\emptyset$	$\{u_5\}$	$\emptyset$
3	$\{u_3, u_4, u_2\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{u_1\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{u_5\}$
4	$\{u_3, u_4, u_2, u_5\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{u_1\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{u_1\}$
5	$\{u_3, u_4, u_2, u_5, u_1\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

.

providing the node which follows u_c in the sequence. As regards the target graph, the algorithm preliminary selects a set of potential candidates among all the unmatched nodes (namely the remaining nodes $R_2(s_c, \psi(u_n))$) of G_2 belonging to the same class of u_n

$$R_2(s_c, \psi(u_n)) = \{v_n \in V_2 : v_n \notin \widetilde{M}_2(s_c) \wedge \psi(v_n) = \psi(u_n)\}. \quad (16)$$

Algorithm 4. Procedure to Generate the Next Candidate Couple. The Inputs are the Current State s_c, the Last Inserted Couple (u_c, v_c), the Exploration Sequence N_{G_1}, the Set *Parent*, and the Graphs G_1 and G_2. It Returns a Candidate Couple (u_n, v_n) to be Checked for the Feasibility or a Null Couple if there are no More Couples to Explore

```
1: function GETNEXTCANDIDATE(s_c, (u_c, v_c), N_{G_1}, Parent, G_1, G_2)
2:   if u_c = ε then
3:     u_n = GETNEXTINSEQUENCE(N_{G_1}, s_c)
4:     if u_n = ε then                          ▷ The sequence is finished
5:       return (ε, ε)
6:   else
7:     u_n = u_c
8:   if Parent(u_n) = ε then                    ▷ u_n has not a parent node
9:     v_n = GETNEXTNODE(v_c, R_2(s_c, ψ(u_n)))
10:    return (u_n, v_n)
11:  else
12:    ṽ = μ̃(s_c, Parent(u_n))  ▷ Get the node matched to Parent(u_n)
13:    if u_n in P_1(Parent(u_n)) then  ▷ u_n is predecessor of Parent(u_n)
14:      v_n = GETNEXTNODE(v_c, R_2^P(s_c, ψ(u_n), ṽ))
15:      return (u_n, v_n)
16:    if u_n in S_1(Parent(u_n)) then  ▷ u_n is successor of Parent(u_n)
17:      v_n = GETNEXTNODE(v_c, R_2^S(s_c, ψ(u_n), ṽ))
18:      return (u_n, v_n)
19:  return (ε, ε)                            ▷ There is not a pair for u_n
```

The set $R_2(s_c, \psi(u_n))$ is used when the node u_n is the first explored node of the matching process (the one included in the descendants of s_0), or otherwise a node having no parent; the latter condition arises in case of graphs having a plurality of connected components, thus the algorithm is starting the analysis of a new one.

Vice versa, when the node u_n has a parent, only a small subset of R_2 is considered for obtaining v_n. In particular, if u_n is a predecessor of its parent, then $R_2^{\mathcal{P}}$ will be considered

$$R_2^{\mathcal{P}}(s_c, \psi(u_n), \widetilde{v}) = \{v_n \in V_2 : v_n \in \mathcal{P}_2(\widetilde{v}) \cap R_2(s_c, u_n)\}, \quad (17)$$

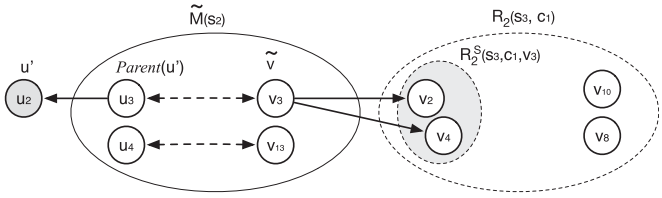
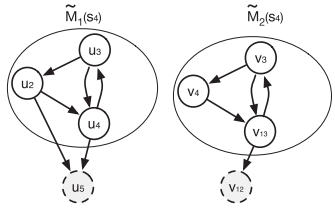


Fig. 5. The GetNextCandidate procedure applied to the state s_2 shown in Fig. 2. On the left, u_2, the current node to be mapped for G_1 and u_3, the parent of u_2. The class $\psi(u_2)$ is c_1. On the right, the set $R_2(s_2, c_1)$ of nodes having the same class of u_2 and the set $R_2^{\mathcal{S}}(s_2, c_1, v_3)$ containing the neighbors of v_3 that are candidates for u_2. v_2 is selected, being the first node of $R_2^{\mathcal{S}}(s_2, c_1, v_3)$.



$s_7 = s_4 \cup (u_5, v_{12})$

$\widetilde{M}_1(s_2) = \{u_2, u_3, u_4\}$

$\widetilde{M}_2(s_2) = \{v_3, v_4, v_{13}\}$

$\widetilde{M}(s_2) = \{(u_3, v_3), (u_4, v_{13}), (u_2, v_4)\}$

Fig. 6. Feasibility check F_c of $s_4 \cup (u_5, v_{12})$ (see Fig. 2). The couple does not satisfy the condition F_c. Indeed, u_4 is predecessor of u_5, while v_{13}, mapped with u_4, is the predecessor of v_{12} . Vice versa, u_5 has u_2 as predecessor but v_{12} has no other predecessor in $\widetilde{M}_2(s_4)$.

otherwise, if u_n is a successor of its parent, then $R_2^{\mathcal{S}}$ will be considered

$$R_2^{\mathcal{S}}(s_c, \psi(u_n), \widetilde{v}) = \{v_n \in V_2 : v_n \in \mathcal{S}_2(\widetilde{v}) \cap R_2(s_c, u_n)\}. \quad (18)$$

The nodes in the above sets R_2, $R_2^{\mathcal{P}}$ and $R_2^{\mathcal{S}}$ are ordered on the base of their index. Thus, v_u is selected as the first element in the set that has not yet been used at that level.

Fig. 5 illustrates the sets relative to the state s_2 of Fig. 2 by considering the node $u_n = u_4$.

3.6 Feasibility Rules

We briefly recall that the algorithm has to explore only consistent states and possibly avoid unfruitful consistent states by using some sort of look-ahead. The transition function, used to generate a new state s_n from a consistent state s_c, must ensure that the addition of the pair (u_n, v_n) to s_c will lead to a state s_n consistent with the subgraph isomorphism problem.

Thus, before generating a new state, VF3 analyses the candidate pair (u_n, v_n) by using a feasibility function IsFeasible that takes into account both structural and semantic information of each node; formally

$$\textsc{IsFeasible}(s_c, u_n, v_n) = F_s(s_c, u_n, v_n) \wedge F_t(s_c, u_n, v_n). \quad (19)$$

The semantic term F_s of the function depends only on the labels of the two nodes and, if present, on the labels of the edges connecting them; this term checks if the addition of (u_n, v_n) violates Equations (5) or (6).

The structural term F_t is more complex because it analyzes the neighborhood of each node of the couple to be added by considering the consistency of the core sets and the feasibility sets. It can be seen as a conjunction of three terms

$$\begin{aligned} &F_t(s_c, u_n, v_n) \\ &\quad = F_c(s_c, u_n, v_n) \wedge F_{la1}(s_c, u_n, v_n) \wedge F_{la2}(s_c, u_n, v_n). \end{aligned} \quad (20)$$

The first term F_c checks if the state obtained by adding the couple (u_n, v_n) to the state s_c is consistent; note that, to this aim, only the core sets $\widetilde{M}_1(s_c)$ and $\widetilde{M}_2(s_c)$ are analyzed.

The second term F_{la1} performs a 1-lookahead, by using a necessary but not sufficient condition. Indeed, if it is false, it guarantees that no consistent state can be reached in one step from the new state. It uses the predecessor and successor feasibility sets. Finally, the third term F_{la2} performs a 2-lookahead, a necessary but not sufficient condition which ensures that if it is false no consistent state can be reached in two steps. It uses the sets $\widetilde{V}_1^{c_i}$ and $\widetilde{V}_2^{c_i}$. Note that the algorithm would produce the correct matching solutions even if only F_c would be used, but the other two terms are essential for reducing the number of visited states.

More formally, the function F_c can be defined by applying the constraints given by Equations (3) and (4) to the new state $s_c \cup (u_n, v_n)$, under the assumption that the state s_c is already consistent

$$\begin{aligned} &F_c(s_c, u_n, v_n) \Leftrightarrow \\ &\quad \forall u' \in \mathcal{S}_1(u_n) \cap \widetilde{M}_1(s_c) \exists v' = \widetilde{\mu}(s_c, u') \in \mathcal{S}_2(v_n) \\ &\quad \wedge \forall u' \in \mathcal{P}_1(u_n) \cap \widetilde{M}_1(s_c) \exists v' = \widetilde{\mu}(s, u') \in \mathcal{P}_2(v_n) \\ &\quad \wedge \forall v' \in \mathcal{S}_2(v_n) \cap \widetilde{M}_2(s_c) \exists u' = \widetilde{\mu}^{-1}(s_c, v') \in \mathcal{S}_1(u_n) \\ &\quad \wedge \forall v' \in \mathcal{P}_2(v_n) \cap \widetilde{M}_2(s_c) \exists u' = \widetilde{\mu}^{-1}(s_c, v') \in \mathcal{P}_1(u_n), \end{aligned} \quad (21)$$

where $\mathcal{S}_1(u_n)$ denotes the set of the successors of u_n in G_1, and $\mathcal{P}_1(u_n)$ the set of the predecessors; similar notations will be used for graph G_2.

It is possible to check that starting from the state s_c, the addition of (u_n, v_n) must verify that for each successor u' of u_n belonging to the set $\widetilde{M}_1(s_c)$ should exist a successor v' of v_n belonging to $\widetilde{M}_2(s_c)$ and associated to u'. The same condition must be verified for each predecessor of u_n that belongs to $\widetilde{M}_1(s_c)$. The previous condition must be verified by symmetrically starting from v_n instead than u_n, in order to satisfy Equation (4). An example of the application of F_c is given in Fig. 6.

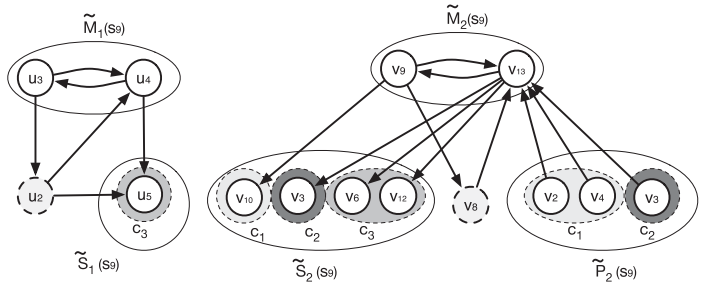
The 1-lookahead function F_{la1} checks if the number of the neighbors of u_n falling in the classified feasibility sets of G_1 are less or equal than the number of the neighbors of v_n in the classified feasibility sets of G_2. Note that the classified feasibility sets are divided in subsets, one for each class assigned to the nodes of G_1 and G_2; thus, the function F_{la1} must hold on each single subset of $\widetilde{\mathcal{P}}_1$, $\widetilde{\mathcal{S}}_1$, $\widetilde{\mathcal{P}}_2$ and $\widetilde{\mathcal{S}}_2$, considering the classes $\{c_1, \ldots, c_q\}$. Formally,

$$F_{la1}(s_c, u_n, v_n) \Leftrightarrow F_{la1}^1(s_c, u_n, v_n) \wedge \ldots \wedge F_{la1}^q(s_c, u_n, v_n), \quad (22)$$

where the functions F_{la1}^i, with $i = 1, \ldots, q$, are defined as follows:

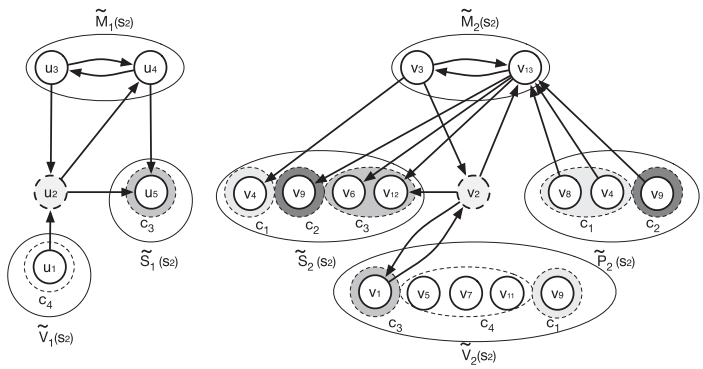
$$\begin{aligned} &F_{la1}^i(s_c, u_n, v_n) \\ &\quad \Leftrightarrow |\mathcal{P}_1(u_n) \cap \widetilde{\mathcal{P}}_1^{c_i}(s_c)| \leq |\mathcal{P}_2(v_n) \cap \widetilde{\mathcal{P}}_2^{c_i}(s_c)| \\ &\quad \wedge \quad |\mathcal{P}_1(u_n) \cap \widetilde{\mathcal{S}}_1^{c_i}(s_c)| \leq |\mathcal{P}_2(v_n) \cap \widetilde{\mathcal{S}}_2^{c_i}(s_c)| \\ &\quad \wedge \quad |\mathcal{S}_1(u_n) \cap \widetilde{\mathcal{P}}_1^{c_i}(s_c)| \leq |\mathcal{S}_2(v_n) \cap \widetilde{\mathcal{P}}_2^{c_i}(s_c)| \\ &\quad \wedge \quad |\mathcal{S}_1(u_n) \cap \widetilde{\mathcal{S}}_1^{c_i}(s_c)| \leq |\mathcal{S}_2(v_n) \cap \widetilde{\mathcal{S}}_2^{c_i}(s_c)|. \end{aligned} \quad (23)$$

An example of the application of F_{la1} is given in Fig. 7.



$$
\begin{aligned}
&s_{10} = s_9 \cup (u_2, v_8)\\
&\widetilde{M}_1(s_9) = \{u_3, u_4\}\\
&\widetilde{M}_2(s_9) = \{v_9, v_{13}\}\\
&\widetilde{M}(s_9) = \{(u_3, v_9), (u_4, v_{13})\}\\
&\widetilde{\mathcal{S}}_1^{c3}(s_8) = \{u_5\}\\
&\widetilde{\mathcal{P}}_2^{c1}(s_9) = \{v_2, v_4\}\\
&\widetilde{\mathcal{P}}_2^{c2}(s_9) = \{v_3\}\\
&\widetilde{\mathcal{S}}_2^{c1}(s_9) = \{v_{10}\}\\
&\widetilde{\mathcal{S}}_2^{c2}(s_9) = \{v_3\}\\
&\widetilde{\mathcal{S}}_2^{c3}(s_9) = \{v_6, v_{12}\}
\end{aligned}
$$

Fig. 7. Feasibility check F_{la1} of the addition of (u_2, v_8) to s_9 for obtaining the state s_{10} (see Fig. 2). The couple does not satisfy the condition F_{la1}. Indeed, the number of successors of u_2 that are in $\widetilde{\mathcal{S}}_1^{c_3}(s_9)$ is greater than the number of successors of v_8 that are in $\widetilde{\mathcal{S}}_2^{c_3}(s_9)$. No edges connect v_8 to one or more nodes in $\widetilde{\mathcal{S}}_2^{c_3}(s_9) = \{v_6, v_{12}\}$. The empty feasibility sets are not shown in the figure. The shading around the nodes identifies the class to which they belong to.



$$
\begin{aligned}
&s_3 = s_2 \cup (u_2, v_2)\\
&\widetilde{M}_1(s_2) = \{u_3, u_4\}\\
&\widetilde{M}_2(s_2) = \{v_3, v_{13}\}\\
&\widetilde{M}(s_2) = \{(u_3, v_3), (u_4, v_{13})\}\\
&\widetilde{\mathcal{S}}_1^{c3}(s_2) = \{u_5\}\\
&\widetilde{\mathcal{P}}_2^{c1}(s_2) = \{v_4, v_8\}\\
&\widetilde{\mathcal{P}}_2^{c2}(s_2) = \{v_9\}\\
&\widetilde{\mathcal{S}}_2^{c1}(s_2) = \{v_4\}\\
&\widetilde{\mathcal{S}}_2^{c2}(s_2) = \{v_9\}\\
&\widetilde{\mathcal{S}}_2^{c3}(s_2) = \{v_6, v_{12}\}\\
&\widetilde{V}_1{}^{c3}(s_2) = \{u_1\}\\
&\widetilde{V}_2{}^{c1}(s_2) = \{v_9\}\\
&\widetilde{V}_2{}^{c3}(s_2) = \{v_1\}\\
&\widetilde{V}_2{}^{c4}(s_2) = \{v_5, v_7, v_{11}\}
\end{aligned}
$$

Fig. 8. Feasibility check F_{la2} of the addition of (u_2, v_2) to s_2 for obtaining the state s_3 (see Fig. 2). The couple does not satisfy the condition F_{la2}. Indeed, the number of successors of u_2 that are in $\widetilde{V}_1{}^{c_4}(s_2)$ is greater than the number of successors of v_2 that are in $\widetilde{V}_2{}^{c_4}(s_2)$. No edges connect v_2 to one or more nodes in $\widetilde{V}_2{}^{c_4}(s_2) = \{v_5, v_7, v_{11}\}$. The empty feasibility sets are not shown in the figure. The shading around the nodes identifies the class to which they belong to.

The 2-lookahead function F_{la2} checks a condition similar to F_{la1} on the neighbors of u_n and v_n, considering the classified feasibility sets $\widetilde{V}_1{}^{c_i}$ and $\widetilde{V}_2{}^{c_i}$. Thus, the function F_{la2} can be formally defined as follows:

$$
\begin{aligned}
&F_{la2}(s_c, u_n, v_n) \Leftrightarrow\\
&\quad F^1_{la2}(s_c, u_n, v_n) \wedge \ldots \wedge F^q_{la2}(s_c, u_n, v_n),
\end{aligned}
\tag{24}
$$

where each F^i_{la2}, with $i = 1, \ldots, q$, is defined as

$$
\begin{aligned}
&F^i_{la2}(s_c, u_n, v_n) \Leftrightarrow\\
&\quad |\mathcal{P}_1(u_n) \cap \widetilde{V}_1{}^{c_i}(s_c)| \leq |\mathcal{P}_2(v_n) \cap \widetilde{V}_2{}^{c_i}(s_c)|\\
&\wedge \quad |\mathcal{S}_1(u_n) \cap \widetilde{V}_1{}^{c_i}(s_c)| \leq |\mathcal{S}_2(v_n) \cap \widetilde{V}_2{}^{c_i}(s_c)|.
\end{aligned}
\tag{25}
$$

An example of application of F_{la2} is given in Fig. 8.

Table 5 reports a comparison in terms of features among VF3 and the other tree-search based algorithms, VF2 and RI, the most similar ones as for their general structure.

4 Experiments

In order to assess the performance of the proposed algorithm, we have decided to use a synthetic dataset made of graphs generated according to a stochastic model. At the best of our knowledge, the Mivia graphs dataset [35] [1] has been one of the most used datasets in the last ten years; it contains graphs of five different typologies; although it is very useful for comparing performance of graph matching algorithms, it can not be used in the current experimentations. In fact, the maximum size of the graphs is 1,000 nodes (an appropriate size for the applications and the resources available at the date of the dataset creation). Moreover, their

TABLE 5
Feature Comparison Among VF2, VF3 and RI

Feature	VF2	VF3	RI
Node Ordering	No	Based on both labels and structure of two input graphs	Based only on the structure of the pattern graph
Data structures required in each state	Computed at each state	Partially precomputed before starting the matching process	Computed at each state
Choice of the next candidate node pair	Based on Feasibility sets	Based on a restriction of Feasibility sets (R-sets)	Based on the node pair neighborhood
Feasibility rules	Based on Feasibility sets	Based on Feasibility sets and node classification	Based on adjacency relations

1. http://mivia.unisa.it/datasets/graph-database/arg-database/

maximum density is 0.1. Thus, we decided to generate a new synthetic dataset, composed of bigger and denser graphs, so as to obtain a representative benchmark for assessing graph matching algorithms in harsh conditions, at the current edge of scientific maturity.

Using a synthetic dataset, we have the possibility of generating a statistically significant number of graphs for any choice of the parameters; in this case we can use a much wider set of graphs if compared with the ones coming from real applications. Moreover, at the same time, we can use graphs of any size and density so as to better study the dependency of the performance on these crucial parameters.

The choice of the stochastic model has taken into account the goal of stretching the algorithm to its limits with graphs that are not so easy to be matched. For this reason, we have excluded generation models in which the valence of the nodes (i.e., the number of edges connected to each node) is bounded, since for such graphs there are special purpose algorithms that can solve the matching problem in polynomial time. We have also excluded models in which the number of edges grows linearly with the number of nodes (and not quadratically, as it is possible); this is because the number of edges usually plays an essential role in determining both the matching time and the memory occupation of an algorithm.

According to the previous considerations, a natural choice has been the random graph model proposed by Erdős and Rényi in 1959 [36]. It generates graphs of N nodes, where each node has an edge probability of η; it means that, in the average, each node has $\eta \cdot (N-1)$ edges connecting it to other nodes of the graph. As the edges are randomly generated, this model produces the so called random graphs, i.e., graphs having a random topology.

There are two variants of the model: the version by Erdős and Rényi, in which the number of edges is exactly given (for directed graphs) by $E = \eta \cdot N \cdot (N-1)$ but the edges are not probabilistically independent of each other, and the version proposed by Gilbert, where each edge is independent of the others and E in the equation above is the *expected* number of edges, which may not coincide with the actual number.

In our experiments, we have used the second version of the model because it is more realistic and does not contain an a priori information about the number of edges contained in the graph. If the parameter η is fixed independently of N, the number of edges grows quadratically with respect to the size of the graphs. Also, η corresponds to the *density* of the generated graphs. We can expect that the larger the value of η is, the more difficult the matching problem becomes.

In the construction of our database, we have used three different values for η: 0.2, 0.3 and 0.4, so ranging from a medium-low density value to a medium-high density. In fact, three values should be sufficient to appreciate any trend in the dependency of the matching time on graph density. The experimentation has been stopped at $\eta = 0.4$ because larger values of η would require several more years of computation time; in fact note that for $\eta = 0.4$ and $N = 1{,}500$ the average matching time for each pair of graphs is more than two days. Larger problems would not be indicative of applications that are practically tractable on commonly available workstations at this time.

TABLE 6
The Proposed Dataset, Containing Unlabeled and Labeled Graphs with Different Values of η

η	Unlabeled	Labeled
0.2	300; 500; 750; 1,000; 1,250; 1,500; 2,000; 2,500; **3,000**; **3,500**; **4,000**; **4,500**; **5,000**	300; 500; 750; 1,000; 1,250; 1,500; 2,000; 2,500; 3,000; 3,500; 4,000; 4,500; 5,000; 5,500; 6,000; 6,500; 7,000; 7,500; 8,000; 9,000; 10,000
0.3	300; 500; 750; 1,000; 1,250; 1,500; **2,000**; **2,500**; **3,000**	300; 500; 750; 1,000; 1,250; 1,500; 2,000; 2,500; 3,000; 3,500; 4,000; 4,500; 5,000; 5,500; 6,000; 6,500; 7,000; 7,500; 8,000; 9,000
0.4	300; 500; 750; 1,000; **1,250**; **1,500**	300; 500; 750; 1,000; 1,250; 1,500; 2,000; 2,500; 3,000; 3,500; 4,000; 4,500; 5,000; 5,500; 6,000; 6,500; 7,000

The cells report the size of the generated graphs. For each size, fifty graphs are generated, except for those reported in bold, which refer to 10 graphs. The total number of graphs is 12,700.

In order to ensure that at least one subgraph isomorphism exists between the pattern and the target graph, we have built the dataset according to the following criteria: (i) for each generated target graph, we have extracted a pattern subgraph with a fixed fraction of the target nodes (in our experiments 20% of size of the target graph), with the constraint that the subgraph had to be connected; (ii) we have applied a random permutation to the nodes of the extracted pattern subgraphs, ensuring that the order of the nodes does not provide information that could help the matching process.

Since label information (usually on the nodes, sometimes on the edges) is present in many applications, it is important to assess the ability of the algorithms to exploit this information to reduce the matching time. For this reason, we have also generated labeled graphs, having eight different numeric labels attached to the nodes. The labels have been assigned using two probability distributions: One is uniform, while the other is non uniform, in order to represent cases where some of the labels are much more frequent than others. In more details, the size of the graphs varies in the range [300 - 5,000] in case of unlabeled graphs and in the range [300 - 10,000] for labeled graphs, as shown in Table 6. For every combination of N and η, we have generated different graphs so as to average the matching time over them. We have made the dataset publicly available, obtainable at [37].

VF3 has been compared with four state of the art algorithms, namely VF2 [12], RI [22], L2G [21] and LAD [27]. The choice of these algorithms has been done by considering the following criteria. First, we have selected algorithms able to solve the subgraph isomorphism and whose implementation is officially provided by the authors, so as to expect a highly optimized implementation. Note that, although both RI and LAD in their original papers deal with the monomorphism problem, the authors also provide an optimized version for the subgraph isomorphism, that has been used in our experiments. Unfortunately, this is not true for other recent algorithms (such as, for instance, FocusSearch [28]) which deal with monomorphism. Furthermore, the selected algorithms cover the different approaches to the subgraph isomorphism. In more details,

VF2, RI and L2G are representative for the tree search approaches with backtracking, while LAD is representative for the constraint programming approaches and provides a useful term of comparison with respect to algorithms based on the tree search paradigm. Since constraint programming techniques are able to filter out inconsistencies even when they are not directly related to the nodes being matched (and thus they behave as if they had a larger look-ahead), an algorithm like LAD should have some advantage when dealing with very symmetric or repetitive structures, and should be less sensitive to the density of the graphs.

Graph indexing approaches have been excluded since they typically are not able to find all the solutions, but only to test the presence of the matching. Furthermore, up to our knowledge, the source code of these methods is not publicly available.

Our choice to use RI, L2G and LAD is also strengthened by the fact that they are the most promising algorithm nowadays available in the literature. Indeed they scored at the first three positions in the ICPR contest on graph matching for biological databases [8]. The executables for the algorithms can be obtained by requesting them using our web site [38].

In our experiments we have measured both the matching time and the memory usage of each algorithm for obtaining all the solutions; even though for most graph pairs there was only one solution, the algorithms had been configured so as they did not stop after the first solution was found, thus completing the exploration of the whole search space. Hence, the reported measurements are indicative also for those applications where many instances of the pattern graph may be contained in the target. For the time measurements, we have computed for each value of the parameters N and η both the average and the maximum matching time.

Time measurements have been performed on a cluster infrastructure, using identical virtual machines hosted by VMWare ESXi 5. Each virtual machine had two dedicated AMD Opteron 6,376 processors running at 2,300 MHz, with 2 Mb of cache and 4 Gb of RAM. The virtual machines were configured so as to avoid phenomena like *time drifting* or *lost clocks* that could introduce a bias in the measure.

On these machines, the total time required to carry out all the reported experiments involving 6,350 pairs of graphs has been about 19,749 hours, that correspond to 822 days, more than two years. This impressive figure is, to the best of our knowledge, unprecedented in the literature, also because it has been focused on hard cases instead of simple instances of the problem. We consider this significant experimentation another important contribution of the research described in this paper.

Memory usage has been measured by using the information provided by the operating system (Linux) about the memory occupation of the process. Since the Linux operating system has an aggressive allocation strategy, in the sense that it tends to anticipate the memory requirements by allocating some pages before they are actually required, this measure is only partially accurate for graphs with a small number of nodes; however, when graphs get large, this method can be considered highly accurate for the task.

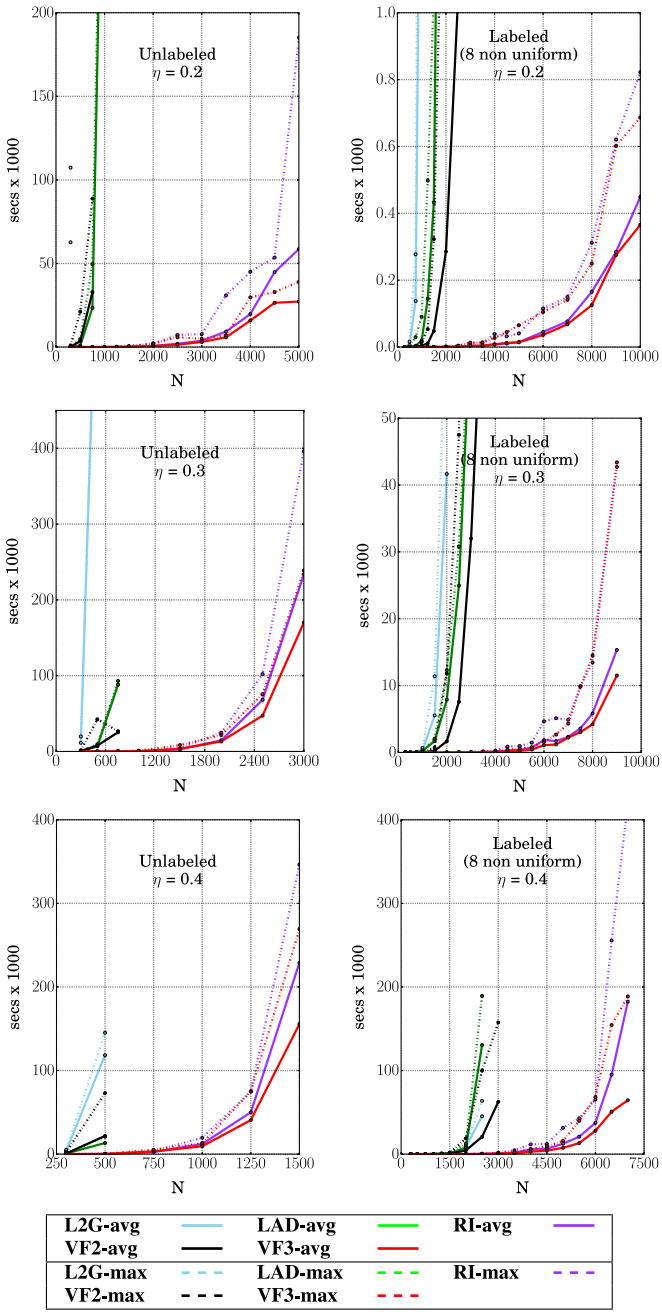


Fig. 9. The average (solid lines) and the maximum (dotted lines) matching times for $\eta = 0.2$, $\eta = 0.3$ and $\eta = 0.4$ for unlabeled and non uniform labeled graphs.

4.1 Results

We start presenting the matching times for the considered algorithms. Note that the number of classes considered in our experiments is equal to the number of labels. Fig. 9 shows the average and the maximum matching time, with respect to graphs having $\eta = 0.2$, $\eta = 0.3$ and $\eta = 0.4$, and for unlabeled and non uniformly labeled graphs. Note that the experimentation has been also extended to uniformly labeled graphs and the relative results are available in [38].

The first thing that we can notice is that the performance of VF2, L2G and LAD are rather similar to each other and significantly lower than those obtained by VF3 and RI: In particular, this happens for any value of η and for both labeled and unlabeled graphs; this difference of

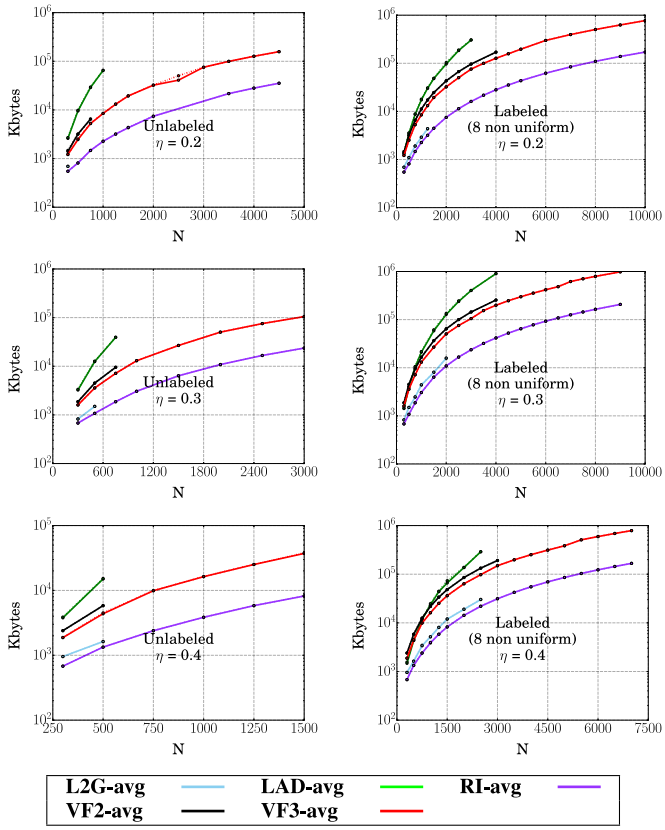


Fig. 10. The memory usage for $\eta = 0.2$, 0.3 and 0.4 for unlabeled graphs and labeled with non uniform distribution.

performance is so high that VF2, L2G and LAD are practically able to achieve the solutions only for graphs whose size is under 1,000 nodes for unlabeled graphs and 4,000 for labeled ones. In more details, it can be seen that when the value of η increases, the maximum size of processable graphs drops significantly: For example, in case of unlabeled graphs, it passes from $N \simeq 1,000$ for $\eta = 0.2$ to $N \simeq 400$ for $\eta = 0.4$. A similar situation occurs for labeled graphs. This consideration makes it evident that VF3 is definitively faster than VF2, L2G and LAD and suggests to better analyze what happens in the comparison between VF3 and RI. To this concern, it is clear that in case of unlabeled graphs and for relatively small values of N, the two algorithms have a similar average matching time: In particular, below $N = 2{,}000$ for $\eta = 0.2$ and $\eta = 0.3$ and $N = 1{,}000$ for $\eta = 0.4$. Beyond these values of N, VF3 becomes faster and faster with respect to RI: The advantage becomes more evident for higher values of η. The absolute value of the difference of the matching time of the two algorithms is practically very significant. For instance, at $N = 3{,}000$ nodes and $\eta = 0.2$, VF3 is about 16 minutes faster than RI, the time saved for $N = 3{,}000$ and $\eta = 0.3$ is 16 hours, and arrives to 20 hours for $N = 1{,}500$ and $\eta = 0.4$.

For the maximum matching times, we can see that with the increase of η the improvement of the advantage of VF3 with respect to the other algorithms, and in particular to RI, follows the same trend of the average value. In fact, while for $\eta = 0.2$ the improvement starts around 1,900 nodes, for $\eta = 0.3$ VF3 improves the maximum time over RI for graphs larger than 1,700 nodes, and for $\eta = 0.4$ the improvement starts already at about 900 nodes.

TABLE 7
Comparison Between VF2 and VF3 in Terms of Measured Computational Complexity

Dataset	VF2		VF3	
	Labeled	Unlabeled	Labeled	Unlabeled
$\eta = 0.2$	$\theta(N^{5.7})$	$\theta(N^{6.3})$	$\theta(N^{3.5})$	$\theta(N^{3.6})$
$\eta = 0.3$	$\theta(N^{5.7})$	$\theta(N^{6.4})$	$\theta(N^{3.7})$	$\theta(N^{3.7})$
$\eta = 0.4$	$\theta(N^{5.7})$	$\theta(N^{6.6})$	$\theta(N^{4.5})$	$\theta(N^{4.7})$

It is noteworthy that even in the case of unlabeled graphs, where some of the VF3 heuristics cannot be used (namely, the node classification function), VF3 is consistently faster than RI for very large and dense graphs: for $\eta = 0.2$ and $\eta = 0.3$ it even happens that the maximum time for VF3 is less than the average time of RI; for $\eta = 0.4$ the difference of the maximum between VF3 and RI is around 22 hours. In the case of labeled graphs, while the maximum times of the two algorithms are comparable for $\eta = 0.2$ and $\eta = 0.3$, the advantage of VF3 is up to 72 hours for $\eta = 0.4$.

Now let us consider the results relative to the memory usage of the algorithms. Fig. 10 shows the maximum memory usage for the different values of η and for different typologies of graphs, both unlabeled and labeled.

The first thing that can be noticed is that for all the considered algorithms the memory usage does not depend on graph density. Even if the curves in the three figures did not arrive to the same number of nodes (because for the higher densities the matching time was too high for the slower algorithms) we can see that for the points present in all the figures the memory requirement of each algorithm does not change significantly. Of the considered algorithms, RI and L2G appear to be the ones with the best memory usage. VF3 is just after, closely followed by VF2. Notice that VF3 has a slight improvement of the memory with respect to VF2, due to a better usage of the data structures, even though this aspect was not considered among the objectives of the modifications introduced in this algorithm. The LAD algorithm is the one that performs worse with respect to memory, requiring more space than the older VF2 algorithm.

To summarize the experimental results, we can say that VF3 improves greatly with respect to VF2, with a difference of up to four orders of magnitude. This improvement is consistently observed independently of the graph size, graph density and presence of labels; thus it is advisable to replace VF2 with VF3 in all the applications; even though VF3 needs more complex data structures than VF2, it has a smaller memory usage, in the average of about 30 percent. In more details, Table 7 shows an experimental comparison between VF2 and VF3 in terms of computational complexity; note that as for the unlabeled graphs only few measures are available for VF2.

VF3 proves to be the fastest algorithm for large and dense graphs; however, even in small or sparse cases where it is overtaken by RI, its matching time is close to the best. Furthermore, the VF3 algorithm shows a reduced variance in matching time with respect to other algorithms, in particular RI. Consequently, it can be a safe choice for applications where the characteristics of the graphs are not completely known in advance.

5 Conclusion

In this paper we have introduced a novel subgraph isomorphism algorithm, named VF3, which has been especially tailored for the complex case of large, dense graphs. We have described in detail the algorithm, highlighting the different heuristics and optimizations in its implementation. We have performed an extensive experimental evaluation, with a database of large and dense graphs that has been made public, comparing both the memory usage and the execution time of VF3 and other recent matching algorithms. The experimentation has required the impressive figure of more than 2 years of machine time in order to be completed. The results of the experimentation confirm that VF3 has a good memory usage, that allows it to work on very large graphs, and its execution time has several orders of magnitude of improvement with respect to its predecessor. Furthermore, when the graph size and density grow, VF3 is the fastest existing algorithm.

References

[1] H. Bunke and M. Vento, "Benchmarking of graph matching algorithms proc. 2nd IAPR- TC15 workshop on graph-based representations," in *Proc. 2nd IAPR-TC-15 Workshop Graph-Based Representations*, 1999, pp. 109–114.

[2] M. Pelillo, "Replicator equations, maximal cliques, and graph isomorphism," *Neural Comput.*, vol. 11, no. 8, pp. 1933–1955, 1999.

[3] F. Serratosa, "Computation of graph edit distance: Reasoning about optimality and speed-up," *Image Vis. Comput.*, vol. 40, pp. 38–48, 2015.

[4] B. Luo and E. R. Hancock, "Structural graph matching using the em algorithm and singular value decomposition," *IEEE Trans. Pattern Anal. Mach. Intell. Graph - Algorithms Comput. Vis.*, vol. 23, no. 10, pp. 1120–1136, Oct. 2001.

[5] F. Serratosa, "Fast computation of bipartite graph matching," *Pattern Recog. Lett.*, vol. 45, pp. 244–250, Aug. 2014.

[6] K. Riesen and H. Bunke, "Approximate graph edit distance computation by means of bipartite graph matching," *Image Vis. Comput.*, vol. 27, no. 7, pp. 950–959, Jun. 2009.

[7] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in Pattern Recognition," *Int. J. Pattern Recogn.*, vol. 18, no. 3, pp. 265–298, 2004.

[8] V. Carletti, P. Foggia, M. Vento, and X. Jiang, "Report on the first contest on graph matching algorithms for pattern search in biological databases," in *Proc. Graph-Based Representations Pattern Recog.*, 2015, pp. 178–187.

[9] C.-L. Liu, B. Luo, and W. Kropatsch. "Special issue Advances in graph-based pattern recognition," *Pattern Recog. Lett.*, vol. 87, pp. 1–3, 2017, doi: https://doi.org/10.1016/j.patrec.2016.10.008.

[10] E. Luks, "Isomorphism of bounded valence can be tested in polynomial time," *J. Comput. Syst. Sci.*, vol. 25, pp. 42–65, 1982.

[11] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "An improved algorithm for matching large graphs," in *Proc. 3rd IAPR-TC15 Workshop Graph-Based Representations Pattern Recog.*, 2001, pp. 149–159.

[12] L. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Trans. Pattern Anal.*, vol. 26, no. 10, pp. 1367–1372, Oct. 2004.

[13] P. Foggia, C. Sansone, and M. Vento, "A performance comparison of five algorithms for graph isomorphism," in *Proc. 3rd IAPR-TC15 Workshop Graph-Based Representations Pattern Recog.*, 2001, pp. 188–199.

[14] N. J. Nilsson, *Principles of Artificial Intelligence*. Berlin, Germany: Springer, 1982.

[15] P. Foggia, C. Sansone, and M. Vento, "A database of graphs for isomorphism and sub-graph isomorphism benchmarking," in *Proc. 3rd IAPR-TC15 Workshop Graph-Based Representations Pattern Recog.*, 2001, pp. 176–187.

[16] M. Vento, "A long trip in the charming world of graphs for Pattern Recognition," *Pattern Recog.*, vol. 48, pp. 1–11, Jan. 2014.

[17] P. Foggia, G. Percannella, and M. Vento, "Graph Matching and Learning in Pattern Recognition on the last ten years," *Int J. Pattern Recog. Artif. Intell.*, vol. 28, no. 1, 2014.

[18] L. Livi and A. Rizzi, "The graph matching problem," *Pattern Anal. Appl.*, vol. 16, no. 3, pp. 253–283, Aug. 2013.

[19] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. Assoc. Comput. Mach.*, vol. 23, pp. 31–42, 1976.

[20] R. Battiti and F. Mascia, *An Algorithm Portfolio for the Sub-Graph Isomorphism Problem*. Berlin, Germany: Springer, 2007, pp. 106–120.

[21] I. Almasri, X. Gao, and N. Fedoroff, "Quick mining of isomorphic exact large patterns from large graphs," in *Proc. IEEE Int. Conf. Data Mining Workshop*, Dec 2014, pp. 517–524.

[22] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro, "A subgraph isomorphism algorithm and its application to biochemical data," *BMC Bioinf.*, vol. 14, 2013, Art. no. S13.

[23] V. Bonnici and R. Giugno, "On the variable ordering in subgraph isomorphism algorithms," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 14, no. 1, pp. 193–203, Jan./Feb. 2017.

[24] J. McGregor, "Relational consistency algorithms and their application in finding subgraph and graph isomorphisms," *Inf. Sci.*, vol. 19, no. 3, pp. 229–250, 1979.

[25] J. Larrosa and G. Valiente, "Constraint satisfaction algorithms for graph pattern matching," *Math. Structures Comput. Sci.*, vol. 12, pp. 403–422, 2002.

[26] S. Zampelli, Y. Deville, and C. Solnon, "Solving subgraph isomorphism problems with constraint programming," *Constraints*, vol. 15, no. 3, pp. 327–353, 2010.

[27] C. Solnon, "Alldifferent-based filtering for subgraph isomorphism," *Artif. Intell.*, vol. 174, no. 12/13, pp. 850–864, 2010.

[28] J. Ullmann, "Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism," *J. Exp. Algorithmics*, vol. 15, 2010, Art. no. 1–6.

[29] L. Kotthoff, C. McCreesh, and C. Solnon, "Portfolios of subgraph isomorphism algorithms," in *Proc. Learn. Intell. Optimization Conf.*, 2016, pp. 107–122.

[30] H. He and A. Singh, "Graphs-at-a-time: Query language and access methods for graph databases," *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 405–417.

[31] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: An efficient algorithm for testing subgraph isomorphism," *Proc. VLDB Endowment.*, vol. 1, no. 1, 2008, pp. 364–375.

[32] S. Zhang, S. Li, and J. Yang, "GADDI: Distance index based subgraph matching in biological networks," in *Proc. 12th Int. Conf. Extending Database Technol. Adv. Database Technol.*, 2009, pp. 192–203.

[33] P. Zhao and J. Han, "On graph query optimization in large networks," *Proc. VLDB Endowment*, vol. 3, no. 1–2, 2010, pp. 340–351.

[34] W. Han, J.-h. Lee, and J. Lee, "Turbo iso: Towards ultrafast and robust subgraph isomorphism search in large graph databases," *Proc. ACM SIGMOD Int. Conf. Manage. Data*, pp. 337–348, 2013.

[35] M. De Santo, P. Foggia, C. Sansone, and M. Vento, "A large database of graphs and its use for benchmarking graph isomorphism algorithms," *Pattern Recogn. Lett.*, vol. 24, pp. 1067–1079, 2003.

[36] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D. Hwang, "Complex networks: Structure and dynamics," *Phys. Rep.*, vol. 424, no. 4–5, pp. 175–308, Feb. 2006.

[37] "Mivia large dense graphs dataset," 2017. [Online]. Available: http://mivia.unisa.it/datasets/graph-database/

[38] "VF3: Experimental results," 2017. [Online]. Available: http://mivia.unisa.it/datasets/graph-database/vf3-library/vf3-experimental-results/

Vincenzo Carletti received the PhD degree with European label in computer engineering in 2016, from the University of Salerno, Fisciano, Italy, where is currently a research fellow. His research activity is focused on exact and inexact graph matching for structural pattern recognition.

Pasquale Foggia received the PhD degree in electronic and computer engineering from University of Naples Federico II, Italy, in 1999. Since 2008, he is an associate professor of computer science at the University of Salerno, Italy. His research interests include basic methodologies and applications in the fields of computer vision and pattern recognition. In 2016 he is elected chairman of the IAPR technical committee 15 on graph-based representations in pattern recognition.

Alessia Saggese received the PhD degree in computer engineering from University of Salerno, Italy, and University of Caen, France, in 2014. She is currently an assistant professor at the University of Salerno. Her research interests include basic methodologies and applications in computer vision and pattern recognition. She is a member of the International Association for Pattern Recognition Technical Committee 15 on Graph-Based Representations in Pattern Recognition since 2012.

Mario Vento received the PhD degree in computer engineering, in 1989 from the University of Napoli "Federico II". He is currently a full professor of Computer Science and Artificial Intelligence with the University of Salerno, Fisciano, Italy, where he is the Coordinator of the Artificial Vision Laboratory and the Dean of Department of Information and Electrical Engineering and Applied Mathematics. His research activities cover real-time video analysis and interpretation for video surveillance applications, classification techniques, either statistical, syntactic and structural, exact and inexact graph matching, multiexpert classification, and learning methodologies for structural descriptions. He served as the Chairman of the IAPR Technical Committee 15 on Graph-Based Representation in Pattern Recognition from 2002 to 2006. He is the fellow scientist of the International Association Pattern Recognition (IAPR).