Introducing VF3: A New Algorithm for Subgraph Isomorphism

Vincenzo Carletti(✉), Pasquale Foggia, Alessia Saggese, and Mario Vento

Department of Information Engineering, Electrical Engineering and Applied Mathematics, University of Salerno, Salerno, Italy
{vcarletti,pfoggia,asaggese,mvento}@unisa.it
http://mivia.unisa.it

Abstract. Several graph-based applications require to detect and locate occurrences of a *pattern graph* within a larger *target graph*. Subgraph isomorphism is a widely adopted formalization of this problem. While subgraph isomorphism is NP-Complete in the general case, there are algorithms that can solve it in a reasonable time on the average graphs that are encountered in specific real-world applications. In 2015 we introduced one such algorithm, VF2Plus, that was specifically designed for the large graphs encountered in bioinformatics applications. VF2Plus was an evolution of VF2, which had been considered for many years one of the fastest available algorithms. In turn, VF2Plus proved to be significantly faster than its predecessor, and among the fastest algorithms on bioinformatics graphs. In this paper we propose a further evolution, named VF3, that adds new improvements specifically targeted at enhancing the performance on graphs that are at the same time large and dense, that are currently the most problematic case for the state-of-the-art algorithms. The effectiveness of VF3 has been experimentally validated using several publicly available datasets, showing a significant speedup with respect to its predecessor and to the other most advanced state-of-the-art algorithms.

1 Introduction

A graph-based representation is commonly used in several application fields dealing with structured data, i.e. data that can be decomposed into atomic entities and relationships between entities (described using the nodes and the edges of the graph). In the last few years, in several disciplines the trend has been to use larger and larger graph structures, thanks to the increase in the available memory and computational power. Examples are the bioinformatics and chemoinformatics disciplines [2,4,5,11,12], with the obvious application to the structure of molecules or proteins, but also to less obvious information such as the protein or gene interaction networks; Social Network Analysis [19], where graphs are used to model the interactions and relations between people or organizations, in very large social networks like Facebook; semantic technologies, where huge knowledge bases (like DBPedia [13]) are encoded using the Resource Description Framework (RDF), a standardized graph-based representation.

P. Foggia et al. (Eds.): GbRPR 2017, LNCS 10310, pp. 128–139, 2017.
DOI: 10.1007/978-3-319-58961-9_12

A common problem in many graph-based applications is the search for the occurrences of a *pattern graph* within a larger *target graph*. This problem can be formalized as the search for all the *subgraph isomorphisms* between the two graphs [7]. In the general case (i.e. if no restrictive assumptions are made on the graphs) the subgraph isomorphism problem is provably NP-complete. However, many algorithms have been proposed over the years that are fast enough to be practical at least on the actual graphs commonly encountered in some applications [7,9,18]. These algorithms typically use some kind of heuristics, that take advantage of knowledge about the structure of the graphs in the common cases of the addressed applications, although they maintain a worst-case complexity that is exponential. Thus, as new and more complex applications emerge, new algorithms with more suitable heuristics are required to cope efficiently with the new cases at hand.

Most of the recently proposed algorithms follow three different approaches: Tree Search, Constraint Propagation and Graph Indexing. Algorithms based on Tree Search formulate the problem as the exploration of a *search space* (having a tree structure), composed of *states* that represent partial solutions. The search space is visited usually with a depth-first order, using heuristics to avoid exploring useless parts of the space. Two very popular algorithms based on this approach are Ullmann's algorithm [16] and VF2 [8]; this latter emerged in several benchmarks as the fastest algorithm at the time of its introduction. Other more recent algorithms in this family are RI/RI-DS [3] and VF2Plus [6], that were expressly designed to be efficient on large bioinformatics graphs.

Algorithms based on Constraint Propagation view the search for subgraph isomorphisms as a Constraint Satisfaction Problem, where the goal is to find an assignment of values to a set of variables that satisfies a set of mutual constraints. In particular, for each node of the pattern a domain of compatibility is mantained, containing the potential matching nodes in target. Local constraints (e.g. node or edge consistency) are propagated to different parts of the graphs reducing the domains, until only few candidate matchings remain, that can be explored to find the solutions. An early algorithm following this approach is McGregor's [14]; more recent proposals are by Zampelli et al. [20], Solnon et al. [15] and Ullmann [17].

The last approach, Graph Indexing, originates from graph database applications, where the goal is to retrieve, from a large set of graphs, only the ones containing the desired pattern. To this aim, an index structure is built that makes possible to quickly verify if the pattern is present or not in a target graph, usually without even requiring to load the whole target in memory, thus filtering out unfruitful targets. In general, after the index verification is passed, a more costly refinement phase is needed to actually determine if and where the pattern graph is present. GADDI [21] and TurboISO [10] are recent algorithms based on this approach.

In this paper we present a novel subgraph isomorphism algorithm called VF3. VF3 can be considered an evolution of VF2Plus [6], introduced in 2015 specifically for addressing the very large graphs that occur in several bioinformatics applications. Like its predecessor, VF3 is based on the Tree Search approach, and uses several heuristics to prune the search space. While VF2Plus is particularly effective on graphs that are large but sparse, the improvements introduced in VF3 significantly increase the performance when the graphs become more dense, without compromising the performance on sparse graphs. Thus, the new algorithm has a much broader field of applicability; in particular, it becomes the fastest algorithm on a class of graphs (simultaneously large and dense) that present serious problems for the other state-of-the-art algorithms. The effectiveness of the new algorithm has been verified experimentally with a thorough testing in comparison with VF2Plus and with other state of the art algorithms, using different publicly available databases.

2 The Base of VF3: The VF2Plus Algorithm

In this section we provide a brief introduction to VF2Plus [6], upon which VF3 is based, while next section will be devoted to the novel parts introduced in VF3.

2.1 Graph Matching and State Space Representation

Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, graph matching is the problem of finding a mapping function $M : V_1 \to V_2$ that satisfies a given set of structural constraints. In the case of the subgraph isomorphism, as detailed in [7,9,18], the function M must be injective and *structure preserving*, i.e. it must preserve both the *presence* and the *absence* of the edges between corresponding pairs of nodes.

The problem of finding a matching between two graphs is addressed by VF3, similarly to its predecessors, using a *State Space Representation* (SSR). Each *state* s of the SSR represents a partial mapping $\widetilde{M}(s) \subseteq M$ that is consistent with the matching constraints; a *goal state* is a state whose mapping is complete, i.e. when covers all the nodes in G_1. For each state s the algorithm keeps different sets: the *core sets* $\widetilde{M}_1(s)$ and $\widetilde{M}_2(s)$ containing the nodes of G_1 and G_2 that belongs to $\widetilde{M}(s)$, and two *feasibility sets* $\widetilde{\mathcal{T}}_1(s)$ and $\widetilde{\mathcal{T}}_2(s)$ containing the nodes of G_1 and G_2 connected to those in $\widetilde{M}_1(s)$ and $\widetilde{M}_2(s)$. Furthermore, we will denote as $\widetilde{V}_1(s)$ and $\widetilde{V}_2(s)$ the sets $V_1 - \widetilde{M}_1(s) - \widetilde{\mathcal{T}}_1(s)$ and $V_2 - \widetilde{M}_2(s) - \widetilde{\mathcal{T}}_2(s)$ respectively.

The algorithm starts from a state whose mapping is empty and explores the state space, using a depth-first strategy, till a goal state is reached. State transitions consist in adding a new pair of nodes (u_n, v_n) to the partial mapping of the current state s_c so as to generate a new state $s_n = s_c \cup (u_n, v_n)$, that becomes the new current state. The algorithm uses a set of rules, called *feasibility rules*, to check if the addition will generate a consistent state; if this is not the case, the new state is not explored. When no node pair remains that can be added to s_c for making a consistent state, the algorithm backtracks, i.e. it undoes the addition leading to s_c and restarts from its parent state s_p, looking for a different node pair to be added to it.

2.2 Making the SSR a Tree

The SSR is, by its nature, a graph. Indeed, if we consider a state s_c whose mapping $\widetilde{M}(s_c)$ contains k pairs, it can be generated using $k!$ different paths, involving the same pairs added in different orders. To avoid visiting the same state several times, either a memory-consuming data structure is needed to keep memory of the visited states, or the state space must be reduced to a tree, ensuring that each state is reachable from only one path. Since the order of the couples is not important to have a consistent mapping, the algorithm introduces a total order relationship over the nodes of G_1, and adds the couples so that their first component follows this order, making the state space a tree.

The algorithm defines the order relationship by computing the *node exploration sequence* N_{G_1}, that is a permutation of V_1. The order relationship is based on the idea to explore first the nodes that are more rare and constrained. As for the rareness of a node u, it is defined in terms of probability to find a node in G_2 that is suitable to generate a feasible couple. Such a probability $P_f(u)$ is obtained by combining $P_l(l)$ and $P_d(d)$ that are, respectively, the probability to find a node with label l and the probability to find a node with degree d in G_2. In the case of the subgraph isomorphism $P_f(u)$ is computed ad follows:

$$P_f(u) = P_l(\lambda_{V_1}(u)) \cdot \sum_{d' \geq d(u)} P_d(d') \tag{1}$$

where $\lambda_{V_1}(u)$ and $\lambda_{V_2}(v)$ are the labeling functions associating a label to each node. Note that the two probabilities are considered as independent instead of joint. This is a weak assumptions in some cases, but it reduces the worst-case complexity of the probability estimation from $O(N^3)$ to $O(N)$. As regards the constraints of a node u, they are computed by considering only its connections with the nodes already in the sequence N_{G_1}. To this aim, we defined the *node mapping degree* $d_M(u)$ as the number of edges connecting u to all the nodes that are already inside N_{G_1}.

Therefore, the procedure that computes N_{G_1} first uses, as the ordering criterion, d_M; 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 their degree. If also the latter are equal, the choice is done randomly.

2.3 Checking for Feasibility

As introduced in Sect. 2.1 an important issue to reduce the search space is the exploration of only consistent states, i.e. states satisfying the constraints of the subgraph isomorphism problem. In addition, a further reduction is obtained by avoiding also consistent states that surely will not be part of a solution. Therefore, before generating a new state s_n from the current state s_c, the algorithm checks the candidate couple (u_n, v_n) using the feasibility rules F_s and F_t, to check the semantic and structural feasibility respectively. F_s analyzes only the labels of the two nodes and, if present, of the edges connecting them.

F_t analyzes the structural constraints given by the neighbors of u_n and v_n. To this aim, the nodes in the two graphs are partitioned into q *equivalence classes* using a classification function $\psi : V_1 \cup V_2 \rightarrow C = c_1, \ldots, c_q$; the classification function has the only constraint that it must ensure that if two nodes can be matched in a consistent mapping, they must be in the same class. The easiest way to define a classification function is to use the node labels, but other kind of information can be used if it makes sense for the problem at hand. For each class c_i we define as $\widetilde{\mathcal{T}}_1^{c_i}(s)$ the restriction of $\widetilde{\mathcal{T}}_1(s)$ to the nodes having this class; we define similarly $\widetilde{\mathcal{T}}_2^{c_i}(s)$.

Now we can define the feasibility function F_t:

$$F_t(s_c, u_n, v_n) = F_c(s_c, u_n, v_n) \wedge F_{la1}(s_c, u_n, v_n) \wedge F_{la2}(s_c, u_n, v_n) \tag{2}$$

The rule $F_c(s_c, u_n, v_n)$ is called *core rule* and is responsible to verify the necessary and sufficient condition for the consistency. The latter verify that all the neighbors of u_n and v_n already in the mapping $\widetilde{M}(s_c)$ are mapped each other; more formally:

$$\begin{aligned} F_c(s_c, u_n, v_n) \Leftrightarrow \forall u' \in adj_1(u_n) \cap \widetilde{M_1}(s_c) \quad \exists v' = \widetilde{\mu}(s_c, u') \in adj_2(v_n) \\ \wedge\, \forall v' \in adj_2(v_n) \cap \widetilde{M_2}(s_c) \quad \exists u' = \widetilde{\mu}^{-1}(s_c, v') \in adj_1(u_n) \end{aligned} \tag{3}$$

The other two rules $F_{la1}(s_c, u_n, v_n)$ and $F_{la2}(s_c, u_n, v_n)$, called *1-level and 2-level lookahead rules* respectively, check two additional necessary but not sufficient conditions for the (sub)graph isomorphism so as to guarantee that the new state will part of a solution. In particular, the rule $F_{la1}(s_c, u_n, v_n)$ counts the number of neighbors of u_n and v_n that are in the sets $\widetilde{\mathcal{T}}_1^{c_i}(s_c)$ and $\widetilde{\mathcal{T}}_2^{c_i}(s_c)$:

$$F_{la1}(s_c, u_n, v_n) \iff F_{la1}^1(s_c, u_n, v_n) \wedge \ldots \wedge F_{la1}^q(s_c, u_n, v_n) \tag{4}$$

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

$$F_{la1}^i(s_c, u_n, v_n) \iff |adj_1(u_n) \cap \widetilde{\mathcal{T}}_1^{c_i}(s_c)| \leq |adj_2(v_n) \cap \widetilde{\mathcal{T}}_2^{c_i}(s_c)| \tag{5}$$

The rule $F_{la2}(s_c, u_n, v_n)$ counts the remaining neighbors, i.e. those are neither in $\widetilde{M}(s_c)$ nor in the feasibility sets:

$$F_{la2}(s_c, u_n, v_n) \iff F_{la2}^1(s_c, u_n, v_n) \wedge \ldots \wedge F_{la2}^q(s_c, u_n, v_n) \tag{6}$$

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

$$F_{la2}^i(s_c, u_n, v_n) \iff |adj_1(u_n) \cap \widetilde{V_1}^{c_i}(s_c)| \leq |adj_2(v_n) \cap \widetilde{V_2}^{c_i}(s_c)| \tag{7}$$

It is worth noting that the rules have been shown only for undirected graphs, but they can be easily extended to the case of directed graphs by considering incoming and outgoing edges separately.

3 The VF3 Algorithm

When it was introduced, in [6], VF2Plus brought a great performance improvement to the VF2 algorithm, especially on large graphs. VF3 optimizes and refines some of the novelties introduced by VF2Plus, so as to further improve its performances when the density and the size of the graphs increase. VF3 inherits the structure of VF2Plus and introduces two main novelties: a new procedure to pre-process the pattern graph and a new criterion to select the next candidate couples.

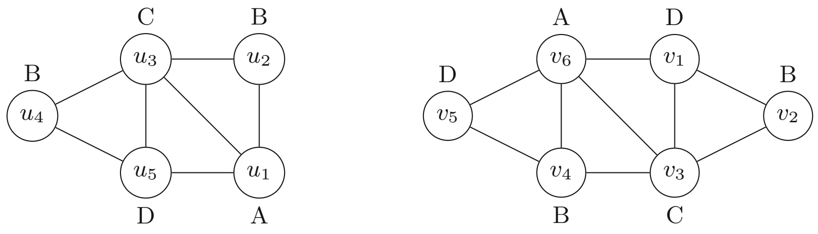


Fig. 1. Graphs, G_1 and G_2, used as an example.

3.1 State Space Precalculation

The exploration sequence N_{G_1}, provided by the sorting procedure, makes the algorithm able to pre-process the graph G_1 and compute, before starting the matching, the sets used to explore it: $\widetilde{M}_1(s)$, $\widetilde{\mathcal{T}}_1^{c_i}(s)$, for $i = 1, \ldots, q$. Furthermore, during the pre-preprocessing VF3 computes, together with the feasibility sets, a spanning tree of G_1, hereinafter the *parent tree* (see Fig. 2), that associates a parent to each node of G_1. As explained in more details below, this tree will be used during the matching process to select the next candidate node from G_2.

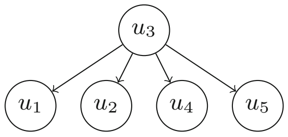


Fig. 2. Parent tree of G_1, for the example of Fig. 1.

The idea behind the pre-processing, is that the sequence N_{G_1} fixes, for each level of the depth-first search, the candidate node of G_1. So that, the algorithm is able to determine the exact composition of each feasibility set of G_1 for all the possible states. For instance, if we consider the exploration sequence $N_{G_1} = \{u_3, u_1, u_5, u_2, u_4\}$, when VF3 is exploring a state s_c that belongs to the *second level* of the SSR, the first node of the candidate couple (u_n, v_n) will always be the one is at the *third position* of N_{G_1}, i.e. u_5. Thus, the mapping $\widetilde{M}$ of all the states belonging to the third level of the SSR contains the couples (u_3, v_i), (u_1, v_j)

Table 1. Core and feasibility sets of G_1 (see Fig. 1), computed for each level of the search.

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

Algorithm 1. Procedure to preprocess the graph G_1 given the sequence N_{G_1}. The procedure computes the feasibility sets and the parent tree ($Parent(u)$ in the procedure).

```
1: function PreprocessGraph($G_1$,$N_{G_1}$)
2:     $i = 0$
3:     for all $u \in N_{G_1}$ do
4:         for all $u' \in adj_1(u)$ do
5:             $c_i = \psi(u')$
6:             Put $u'$ in $\widetilde{M}_1$ at level $i$
7:             if $u' \notin \widetilde{\mathcal{T}}_1^{c_i}$ then
8:                 Put $u'$ in $\widetilde{\mathcal{T}}_1^{c_i}$ at level $i$
9:                 $Parent(u') = u$
10:            $i = i + 1$
11:    return $Parent$
```

and (u_5, v_k). The order and the composition of these couples always follows the sequence N_{G_1}. The nodes v_i, v_j and v_k, belonging to G_2 are dynamically determined during the candidate selection step. Since $\widetilde{M}_1(s)$ is known, from G_1 and $\widetilde{M}_1(s)$ it is possible to precompute the sets $\widetilde{\mathcal{T}}_1^{c_i}(s)$ for each SSR level. The time saved by this precomputation depends on how many states are at each level of the SSR; in general it increases with the density of the graphs. Notice that with a naive encoding of the $\widetilde{\mathcal{T}}_1^{c_i}(s)$ sets, they would occupy a space that is $O(N_1^2)$ (where N_1 is the size of G_1), since there are N_1 levels, and at each level the $\widetilde{\mathcal{T}}_1^{c_i}(s)$ sets have a size that is $O(N_1)$. This would be a problem when working with very large graphs. However, we have demonstrated that for each node of G_1, the levels at which the node belongs to a given $\widetilde{\mathcal{T}}_1^{c_i}$ form a (possibly empty) interval; thus we are able to represent all the $\widetilde{\mathcal{T}}_1^{c_i}(s)$ sets with a single table that for each node reports the first and the last level at which it is in $\widetilde{\mathcal{T}}_1^{c_i}(s)$, with a space occupation that is just $O(N_1)$. Table 1 shows the result of the pre-preprocessing on the graph G_1 in Fig. 1.

3.2 Candidate Selection

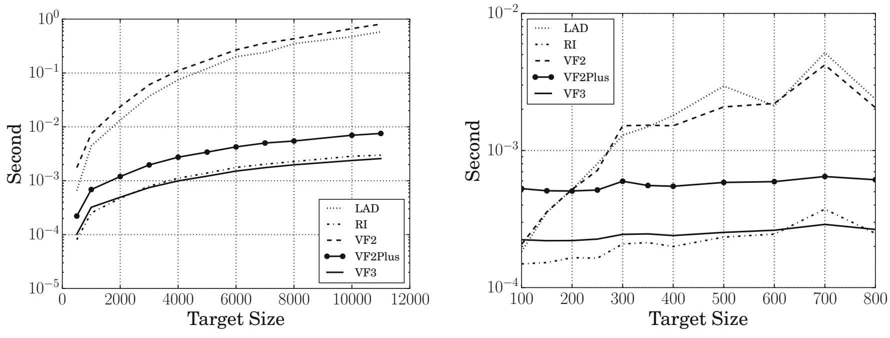
Another relevant difference between VF2Plus and VF3 is the way they select the candidate node from the graph G_2. In the previous section we have clarified that VF3 defines, before the matching begins, the candidates of G_1 for each possible state in the SSR. However, this is not possible for the graph G_2, so VF3 has to select the candidate node for each new state. Thus, being u_n, the candidate node of G_1, the algorithm analyses neighborhood of the node $\widetilde{v}$ mapped to the parent of u_n (hereinafter $Parent(u_n)$) and select the first unmapped node belonging to the same class of u_n. If the node u_n has no parent (eg. it is the first node of the sequence N_{G_1}), VF3 will select u_v from the unmapped nodes of G_2 belonging to the same class of u_n. More formally, when the u_n has not a parent VF3 will consider the set $R_2 \subset V_2$; the latter is composed of the nodes in G_2 that are not in the mapping $\widetilde{M}_2(s_c)$ of the current state s_c and belong to the same class of the node u_n.

$$R_2(s_c, \psi(u_n)) = \{v_n \in V_2 : v_n \notin \widetilde{M}_2(s_c) \wedge \psi(v_n) = \psi(u_n)\}. \tag{8}$$

In the other case, when the node $Parent(u_u)$ exists, the algorithm will consider the subset of R_2^{adj} containing only the neighbors of $\widetilde{v}$.

$$R_2^{adj}(s_c, \psi(u_n), \widetilde{v}) = \{v_n \in V_2 : v_n \in adj_2(\widetilde{v}) \cap R_2(s_c, u_n)\} \tag{9}$$

The candidate selection procedure is shown in details in Algorithm 2. As it will be shown in Sect. 4, this difference has a great impact especially on dense graphs, because it greatly reduces the number of possible candidate nodes.



(a) Proteins (b) Contact Maps

Fig. 3. Matching times for Proteins graphs (a) and Contact Map graphs (b) in the Biological dataset.

4 Experiments

The proposed approach has been tested over two different datasets: a biological dataset (hereinafter *Biological dataset*) and a synthetic dataset, composed by randomly generated graphs (hereinafter *Random dataset*).

As for the former, we have used a real dataset, recently introduced within the *International Contest on Pattern Search in Biological Databases* [4].

Algorithm 2. 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. The procedure returns a candidate couple (u_n, v_n) to be checked for the feasibility or a null couple (ϵ, ϵ) if there are no more couples to explore.

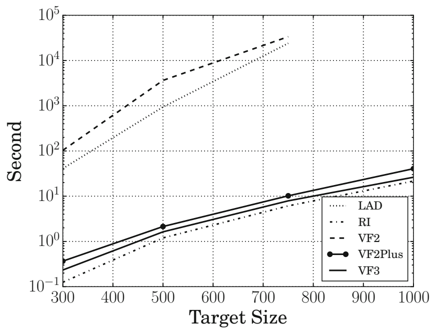
```
1: function SelectCandidate($s_c$, $(u_c, v_c)$, $N_{G_1}$, $Parent$, $G_1$, $G_2$)
2:     if $u_c = \epsilon$ then
3:         $u_n$ = GetNextInSequence($N_{G_1}$, $s_c$)
4:         if $u_n = \epsilon$ then                         ▷ The sequence is finished
5:             return $(\epsilon, \epsilon)$
6:     else
7:         $u_n = u_c$
8:     if $Parent(u_n) = \epsilon$ then                     ▷ $u_n$ has not a parent node
9:         $v_n$ = GetNextNode($v_c$, $R_2(s_c, \psi(u_n))$)
10:        return $(u_n, v_n)$
11:    else
12:        $\widetilde{v} = \widetilde{\mu}(s_c, Parent(u_n))$      ▷ Get the node matched to $Parent(u_n)$
13:        if $u_n$ in $adj_1(Parent(u_n))$ then            ▷ $u_n$ is predecessor of $Parent(u_n)$
14:            $v_n$ = GetNextNode($v_c$, $R_2^{adj}(s_c, \psi(u_n), \widetilde{v})$)
15:            return $(u', v')$
16:    return $(\epsilon, \epsilon)$                           ▷ There is not a pair for $u_n$
```

The dataset is composed of Contact Map and Protein graphs, extracted from the Protein Data Bank [1]. Protein graphs are very large and sparse: the number of nodes ranges from 500 to 10000 and the average degree is 4; contact maps graphs have a medium size (from 150 to 800 nodes) and are denser than protein graphs (their average degree is 20). The number of labels are 6 for proteins and 20 for contact maps.

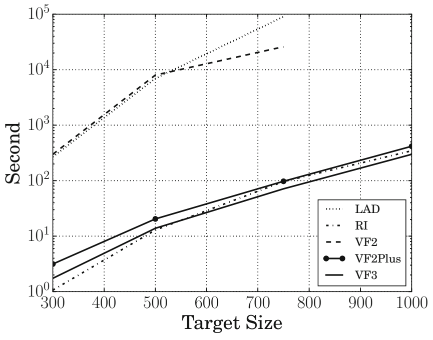
As for the latter, the choice to use a synthetic dataset has been determined by the possibility of generating a statistically significant number of graphs of any size and density, so as to analyze the performance of the proposed approach by varying these two important parameters. In more details, we generated a set of unlabeled graphs, with a size in the range $N = \{300, ..., 1000\}$ and with three different values of density, namely $\eta = 0.2, 0.3$ and 0.4. For each couple of parameters $\eta - N$ we generated 50 graphs. The times reported in the figures have been obtained by averaging over the 50 graphs sharing the same parameters.

The experimentation has been carried out on a cluster infrastructure, using identical virtual machines hosted by VMWare ESXi 5. Each virtual machine is provided with two dedicated AMD Opteron 6376 processors running at 2300 MHz, with 2 Mb of cache and 4 Gb of RAM. In order to confirm the effectiveness of the proposed approach, we have compared it with its previous versions, VF2 and VF2Plus, and with two other state of the art algorithms, namely RI and LAD. The results are reported in Figs. 3 and 4 for Biological and Random datasets, respectively.

From Fig. 3, we can note that VF3 outperforms, independently on the size of the target graphs, LAD, VF2 and VF2Plus. The most interesting comparison, however, is between VF3 and RI, the last one being the winner of the Contest on Pattern Search in Biological Databases. We can note that VF3 is particularly suited for challenging graphs, since it overcomes RI around 2000 nodes for proteins and around 500 nodes in case of contact maps.



(a) $\eta = 0.2$



(b) $\eta = 0.3$

Fig. 4. Matching times for $\eta = 0.2$ (a), $\eta = 0.3$ (b) for the graphs in the Random dataset.

This consideration is confirmed by the results obtained over random graphs, reported in Figs. 4 and 5. Indeed, we can note that VF3 improves over VF2 and LAD of around two orders of magnitude. In practice, this is a very interesting result, since it means to be able to solve, for instance, a graph with 500 nodes and having $\eta = 0.2$ in around two seconds, with respect to more than 1000 s required by VF2 and LAD over the same graph. Furthermore, VF3 improves also VF2Plus with all the η values, confirming the effectiveness of the novelties introduced in the proposed approach.

The improvement with respect to RI becomes more and more evident by increasing the size of the graphs and the η value. Indeed, the overtaking of VF3 with respect to RI is around 550 nodes for graphs having $\eta = 0.3$ and

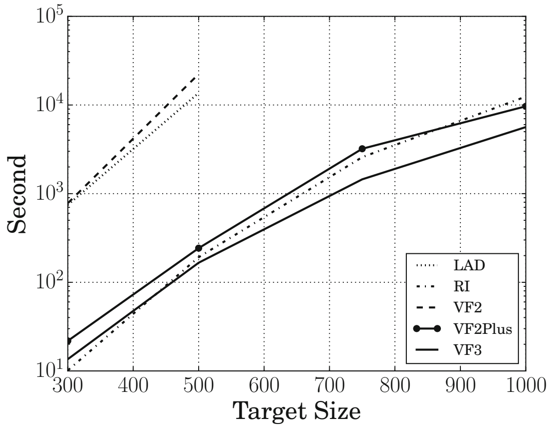


Fig. 5. Matching times for $\eta = 0.4$ (c) for the graphs in the Random dataset.

around $\eta = 450$ for $\eta = 0.4$, thus confirming that VF3 is particularly suited for challenging graphs, big and dense.

5 Conclusions

In this paper we have presented VF3, a new algorithm for (sub)graph isomorphism. VF3 extends the previously introduced VF2Plus algorithm, improving its ability to deal with larger and denser graphs. An experimental evaluation on different datasets show a consistent performance improvement, that increases as the graphs become larger or denser. The new algorithm has also been compared with two other state-of-the-art algorithm, and has shown to be the fastest one in almost all the conditions.

References

1. RCSB: Protein data bank web site (2017). http://www.rcsb.org/pdb
2. Aittokallio, T., Schwikowski, B.: Graph-based methods for analysing networks in cell biology. Brief. Bioinform. **7**(3), 243 (2006). http://dx.doi.org/10.1093/bib/bbl022
3. Bonnici, V., Giugno, R.: On the variable ordering in subgraph isomorphism algorithms. IEEE/ACM Trans. Comput. Biol. Bioinform. PP(99) (2016)
4. Carletti, V., Foggia, P., Vento, M., Jiang, X.: Report on the first contest on graph matching algorithms for pattern search in biological databases. In: Liu, C.-L., Luo, B., Kropatsch, W.G., Cheng, J. (eds.) GbRPR 2015. LNCS, vol. 9069, pp. 178–187. Springer, Cham (2015). doi:10.1007/978-3-319-18224-7_18
5. Carletti, V., Foggia, P., Vento, M.: Performance comparison of five exact graph matching algorithms on biological databases. In: Petrosino, A., Maddalena, L., Pala, P. (eds.) ICIAP 2013. LNCS, vol. 8158, pp. 409–417. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41190-8_44
6. Carletti, V., Foggia, P., Vento, M.: VF2 plus: an improved version of VF2 for biological graphs. In: Liu, C.-L., Luo, B., Kropatsch, W.G., Cheng, J. (eds.) GbRPR 2015. LNCS, vol. 9069, pp. 168–177. Springer, Cham (2015). doi:10.1007/978-3-319-18224-7_17
7. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. IJPRAI **18**(3), 265–298 (2004)
8. Cordella, L., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. **26**, 1367–1372 (2004)
9. Foggia, P., Percannella, G., Vento, M.: Graph matching and learning in pattern recognition on the last ten years. J. Pattern Recognit. **28**(1), 1450001 (2014)
10. Han, W., Lee, J.h., Lee, J.: TurboISO: towards ultrafast and robust subgraph isomorphism search in large graph databases. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 337–348 (2013)
11. Huan, J., et al.: Comparing graph representations of protein structure for mining family-specific residue-based packing motif. J. Comput. Biol. **12**(6), 657–671 (2005)
12. Lacroix, V., Fernandez, C., Sagot, M.: Motif search in graphs: application to metabolic networks. Trans. Computat. Biol. Bioinform. **4**, 360–368 (2006)

13. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - a large-scale, multilingual knowledge base extracted from Wikipedia. Semant. Web J. **6**(2), 167–195 (2015)
14. McGregor, J.: Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. Inf. Sci. **19**(3), 229–250 (1979)
15. Solnon, C.: Alldifferent-based filtering for subgraph isomorphism. Artif. Intell. **174**(12–13), 850–864 (2010)
16. Ullmann, J.R.: An algorithm for subgraph isomorphism. J. Assoc. Comput. Mach. **23**, 31–42 (1976)
17. Ullmann, J.: Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. J. Exp. Algorithm. (JEA) **15**(1) (2010)
18. Vento, M.: A long trip in the charming world of graphs for pattern recognition. Pattern Recognit. **48**(1), 11 (2014)
19. Wasserman, S., Faust, K.: Social Network Analysis: Methods and Applications, vol. 8. Cambridge University Press, Cambridge (1994)
20. Zampelli, S., Deville, Y., Solnon, C.: Solving subgraph isomorphism problems with constraint programming. Constraints **15**(3), 327–353 (2010)
21. Zhang, S., Li, S., Yang, J.: GADDI: Distance Index Based Subgraph Matching In Biological Networks. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (2009)