Discrete Applied Mathematics 242 (2018) 69–81

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

VF2++—An improved subgraph isomorphism algorithm

Alpár Jüttner [a,b,*], Péter Madarasi [b]

[a] *MTA-ELTE Egerváry Research Group, Budapest, Hungary*
[b] *Department of Operations Research, Eötvös Loránd University, Budapest, Hungary*

ARTICLE INFO

Article history:
Received 30 November 2016
Received in revised form 20 January 2018
Accepted 23 February 2018
Available online 29 March 2018

Keywords:
Computational biology
Subgraph isomorphism problem

ABSTRACT

This paper presents a largely improved version of the VF2 algorithm for the *Subgraph Isomorphism Problem*. The improvements are twofold. Firstly, it is based on a new approach for determining the matching order of the nodes, and secondly, more efficient — nevertheless easier to compute — cutting rules are proposed. They together reduce the search space significantly.

In addition to the usual *Subgraph Isomorphism Problem*, the paper also presents specialized algorithms for the *Induced Subgraph Isomorphism* and for the *Graph Isomorphism Problems*.

Finally, an extensive experimental evaluation is provided using a wide range of inputs, including both real-life biological and chemical datasets and standard randomly generated graph series. The results show major and consistent running time improvements over the other known methods.

The C++ implementations of the algorithms are available open-source as part of the LEMON graph and network optimization library.

© 2018 Published by Elsevier B.V.

1. Introduction

In the last decades, combinatorial structures, and especially graphs have been considered with ever increasing interest, and applied to the solution of several new and revised questions. The expressiveness, the simplicity and the deep theoretical background of graphs make them one of the most useful modeling tool and appears constantly in several seemingly independent fields, such as bioinformatics and chemistry.

Complex biological systems arise from the interaction and cooperation of plenty of molecular components. Getting acquainted with the structure of such systems at the molecular level is of primary importance, since protein–protein interaction, DNA–protein interaction, metabolic interaction, transcription factor binding, neuronal networks, and hormone signaling networks can be understood this way.

Many chemical and biological structures can easily be modeled as graphs, for instance, a molecular structure can be considered as a graph, whose nodes correspond to atoms and whose edges to chemical bonds. The similarity and dissimilarity of objects corresponding to nodes are incorporated to the model by *node labels*. Understanding such networks basically requires finding specific subgraphs, thus it calls for efficient subgraph isomorphism algorithms.

Other real-world fields related to some variants of subgraph isomorphism include pattern recognition and machine vision [3], symbol recognition [9], and face identification [13].

Subgraph and induced subgraph isomorphism problems are known to be NP-Complete [6], while the graph isomorphism problem is one of the few problems in NP neither known to be in P nor NP-Complete. Although polynomial-time isomorphism

* Corresponding author at: Department of Operations Research, Eötvös Loránd University, Budapest, Hungary.
E-mail addresses: alpar@cs.elte.hu (A. Jüttner), madarasip@caesar.elte.hu (P. Madarasi).

https://doi.org/10.1016/j.dam.2018.02.018
0166-218X/© 2018 Published by Elsevier B.V.

algorithms are known for various graph classes, like trees and planar graphs [11], bounded valence graphs [15], interval graphs [14] or permutation graphs [5]. Furthermore, an FPT algorithm has also been presented for the colored hypergraph isomorphism problem in [1].

In the following, some algorithms which do not need any restrictions on the graphs are summarized. Even though, an overall polynomial behavior is not expectable from such an alternative, they may often have good practical performance, in fact, they might be the best choice in practice even on a graph class for which polynomial algorithm is known.

The first practically usable approach was due to *Ullmann* [20], which is a commonly used algorithm based on depth-first search with a complex heuristic for reducing the number of visited states. A major problem is its $\Theta(n^3)$ space complexity, which makes it impractical in the case of big sparse graphs.

In a recent paper, Ullmann [21] presents an improved version of this algorithm based on a bit-vector solution for the binary Constraint Satisfaction Problem.

The *Nauty* algorithm [16] transforms the two graphs to a canonical form before starting to look for an isomorphism. It has been considered as one of the fastest graph isomorphism algorithms, although graph categories were shown in which it takes exponentially many steps. This algorithm handles only the graph isomorphism problem.

The *LAD* algorithm [19] uses a depth-first search strategy and formulates the isomorphism as a Constraint Satisfaction Problem to prune the search tree. The constraints are that the mapping has to be one-to-one and edge-preserving, hence it is possible to handle new isomorphism types as well.

The *RI* algorithm [2] and its variations are based on a state space representation. After reordering the nodes of the graphs, it uses some fast executable heuristic checks without using any complex pruning rules. It seems to run really efficiently on graphs coming from biology, and won the International Contest on Pattern Search in Biological Databases [22].

Currently, the most commonly used algorithm is the *VF2* [8], an improved version of *VF* [7], which was designed for solving pattern matching and computer vision problems, and has been one of the best overall algorithms for more than a decade. Although, it is not as fast as some of the new specialized algorithms, it is still widely used due to its simplicity and space efficiency. VF2 uses a state space representation and checks specific conditions in each state to prune the search tree.

Meanwhile, another variant called *VF2 Plus* [4] has been published. It is considered to be as efficient as the RI algorithm and has a strictly better behavior on large graphs. The main idea of VF2 Plus is to precompute a heuristic node order of the graph to be embedded, on which VF2 works more efficiently.

This paper introduces *VF2++*, a new further improved algorithm for the graph and (induced) subgraph isomorphism problems. It is based on efficient cutting rules and determines a node order in which VF2 runs significantly faster on practical inputs.

The rest of the paper is structured as follows. Section 2 defines the exact problems to be solved, Section 3 provides a description of VF2. Based on that, Section 4 introduces VF2++. Some technical details necessary for an efficient implementation are discussed in Section 5. Finally, Section 6 provides a detailed experimental evaluation of VF2++ and its comparison to the state-of-the-art algorithm.

It is also worth mentioning that the C++ implementations of the algorithms have been made publicly available for evaluation and use under an open-source license as a part of LEMON [10,12] graph library.[1]

2. Problem statement

This section provides a formal description of the problems to be solved.

2.1. Definitions

Throughout the paper $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ denote two undirected graphs.

Definition 2.1.1. $\mathcal{L} : (V_1 \cup V_2) \longrightarrow K$ is a **node label function**, where K is an arbitrary set. The elements in K are the **node labels**. Two nodes, u and v are said to be **equivalent** if $\mathcal{L}(u) = \mathcal{L}(v)$.

For the sake of simplicity, the graph, subgraph and induced subgraph isomorphisms are defined in a more general way.

Definition 2.1.2. G_1 and G_2 are **isomorphic** (by the node label $\mathcal{L}$) if $\exists\mathfrak{m} : V_1 \longrightarrow V_2$ bijection, for which the following is true:

$\forall u \in V_1 : \mathcal{L}(u) = \mathcal{L}(\mathfrak{m}(u))$ and
$\forall u, v \in V_1 : (u, v) \in E_1 \Leftrightarrow (\mathfrak{m}(u), \mathfrak{m}(v)) \in E_2$

Definition 2.1.3. G_1 is a **subgraph** of G_2 (by the node label $\mathcal{L}$) if $\exists\mathfrak{m} : V_1 \longrightarrow V_2$ injection, for which the following is true:

$\forall u \in V_1 : \mathcal{L}(u) = \mathcal{L}(\mathfrak{m}(u))$ and
$\forall u, v \in V_1 : (u, v) \in E_1 \Rightarrow (\mathfrak{m}(u), \mathfrak{m}(v)) \in E_2$

[1] The source code repository of the implementation is available at http://lemon.cs.elte.hu/hg/lemon-vf2. It is inclusion to the official LEMON library has also been proposed, see http://lemon.cs.elte.hu/trac/lemon/ticket/597.

Definition 2.1.4. G_1 is an **induced subgraph** of G_2 (by the node label $\mathcal{L}$) if $\exists \mathfrak{m} : V_1 \longrightarrow V_2$ injection, for which the following is true:

$\forall u \in V_1 : \mathcal{L}(u) = \mathcal{L}(\mathfrak{m}(u))$ and
$\forall u, v \in V_1 : (u, v) \in E_1 \Leftrightarrow (\mathfrak{m}(u), \mathfrak{m}(v)) \in E_2$

2.2. Common problems

The focus of this paper is on the following problems appearing in many applications.

The **subgraph isomorphism problem** is the following: is G_1 isomorphic to any subgraph of G_2 by a given node label?

The **induced subgraph isomorphism problem** asks the same about the existence of an induced subgraph.

The **graph isomorphism problem** can be defined as induced subgraph isomorphism problem where the sizes of the two graphs are equal.

In addition, one may either want to find a **single** embedding or **enumerate** all of them.

Finally, let us mention two possible further variants of the problems above. Firstly, *edge labels* and edge label preserving versions of the above problems could also be defined in an analogous way. Secondly, one may want to find (subgraph) isomorphism between *directed graphs*.

In fact, it is straightforward to extend the proposed algorithm to handle both the above variants. For the sake of simplicity, we omit these technical details from the discussion.

3. The VF2 algorithm

This algorithm is the basis of both the VF2++ and the VF2 Plus. VF2 is able to handle all the variations mentioned in Section 2.2. Although it can also handle directed graphs, for the sake of simplicity, only the undirected case is discussed.

3.1. Common notations

Assume G_1 is searched in G_2. The following definitions and notations are used throughout this paper.

Definition 3.1.1. An injection $\mathfrak{m} : D \longrightarrow V_2$ is called (partial) **mapping**, where $D \subseteq V_1$.

Notation 3.1.1. *$\mathfrak{D}(f)$ and $\mathfrak{R}(f)$ denote the domain and the range of a function f, respectively.*

Definition 3.1.2. Mapping $\mathfrak{m}$ **covers** a node $u \in V_1 \cup V_2$ if $u \in \mathfrak{D}(\mathfrak{m}) \cup \mathfrak{R}(\mathfrak{m})$.

Definition 3.1.3. A mapping $\mathfrak{m}$ is **whole mapping** if $\mathfrak{m}$ covers all the nodes of V_1, i.e. $\mathfrak{D}(\mathfrak{m}) = V_1$.

Definition 3.1.4. Let **extend**$(\mathfrak{m}, (u, v))$ denote the function $f : \mathfrak{D}(\mathfrak{m}) \cup \{u\} \longrightarrow \mathfrak{R}(\mathfrak{m}) \cup \{v\}$, for which $\forall w \in \mathfrak{D}(\mathfrak{m}) : f(w) = \mathfrak{m}(w)$ and $f(u) = v$ holds, where $u \in V_1 \setminus \mathfrak{D}(\mathfrak{m})$ and $v \in V_2 \setminus \mathfrak{R}(\mathfrak{m})$; otherwise *extend*$(\mathfrak{m}, (u, v))$ is undefined.

Notation 3.1.2. *Throughout the paper,* **PT** *denotes a generic problem type which can be substituted by any of the* **SUB**, **IND** *and* **ISO** *problems, which stand for the problems mentioned in* Section 2.2 *respectively.*

Definition 3.1.5. Let $\mathfrak{m}$ be a mapping. The **consistency function for PT** is a logical function $\mathbf{Cons_{PT}}(\mathfrak{m})$ for which $\mathbf{Cons_{PT}}(\mathfrak{m})$ is true if and only if $\mathfrak{m}$ satisfies the requirements of **PT** considering the subgraphs of G_1 and G_2 induced by $\mathfrak{D}(\mathfrak{m})$ and $\mathfrak{R}(\mathfrak{m})$, respectively.

Definition 3.1.6. Let $\mathfrak{m}$ be a mapping. A logical function $\mathbf{Cut_{PT}}$ is a **cutting function for PT** if the following holds. $\mathbf{Cut_{PT}}(\mathfrak{m})$ is false if there exists a sequence of extend operations which results in a whole mapping satisfying the requirements of *PT*.

Definition 3.1.7. A mapping $\mathfrak{m}$ is said to be **consistent mapping by PT** if $Cons_{PT}(\mathfrak{m})$ is true.

$Cons_{PT}$ and Cut_{PT} will often be used in the following form.

Notation 3.1.3. *Let* $\mathbf{Cons_{PT}(p, \mathfrak{m})} := Cons_{PT}(extend(\mathfrak{m}, p))$*, and* $\mathbf{Cut_{PT}(p, \mathfrak{m})} := Cut_{PT}(extend(\mathfrak{m}, p))$*, where* $p \in V_1 \setminus \mathfrak{D}(\mathfrak{m}) \times V_2 \setminus \mathfrak{R}(\mathfrak{m})$.

$Cons_{PT}$ will be used to check the consistency of the already covered nodes, while Cut_{PT} is for looking ahead to recognize if no whole consistent mapping can contain the current mapping.

Algorithm 1 *A high level description of VF2*

```
1: procedure VF2(Mapping $\mathfrak{m}$, ProblemType $PT$)
2:     if $\mathfrak{m}$ covers $V_1$ then
3:         Output($\mathfrak{m}$)
4:     else
5:         Compute the set $P_{\mathfrak{m}}$ of the candidate pairs for extending $\mathfrak{m}$
6:         for all $p \in P_{\mathfrak{m}}$ do
7:             if $\mathrm{Cons}_{PT}(p, \mathfrak{m}) \wedge \neg\mathrm{Cut}_{PT}(p, \mathfrak{m})$ then
8:                 call VF2($extend(\mathfrak{m}, p)$, $PT$)
```

3.2. Overview of the algorithm

VF2 begins with an empty mapping and gradually extends it with respect to the consistency and cutting functions until a whole mapping is reached.

Algorithm 1 is a high-level description of the VF2 algorithm. Each state of the matching process can be associated with a mapping $\mathfrak{m}$. The initial state is associated with a mapping $\mathfrak{m}$, for which $\mathfrak{D}(\mathfrak{m}) = \emptyset$, i.e. it starts with an empty mapping.

For the current mapping $\mathfrak{m}$, the algorithm computes $P_{\mathfrak{m}}$, the set of candidate node pairs for extending the current mapping $\mathfrak{m}$.

For each pair p in $P_{\mathfrak{m}}$, $Cons_{PT}(p, \mathfrak{m})$ and $Cut_{PT}(p, \mathfrak{m})$ are evaluated. If the former is true and the latter is false, the whole process is recursively applied to $extend(\mathfrak{m}, p)$. Otherwise, $extend(\mathfrak{m}, p)$ is not consistent by PT, or it can be proved that $\mathfrak{m}$ cannot be extended to a whole mapping.

The correctness of the procedure follows from the claim below.

Claim 3.2.1. *Through consistent mappings, only consistent whole mappings can be reached, and all the consistent whole mappings are reachable through consistent mappings.*

Note that a mapping may be reached in exponentially many different ways, since the order of extensions does not influence the nascent mapping.

However, one may make the following observations.

Definition 3.2.1. A total order $(u_{\sigma(1)}, u_{\sigma(2)}, \dots, u_{\sigma(|V_1|)})$ of V_1 is **matching order** if VF2 can cover $u_{\sigma(d)}$ on the dth level for all $d \in \{1, \dots, |V_1|\}$.

Claim 3.2.2. *If VF2 is prescribed to cover the nodes of G_1 according to a matching order, then no mapping can be reached more than once and each whole mapping remains reachable.*

Note that the cornerstone of the improvements to VF2 is to choose a proper matching order.

3.3. The candidate set

Let $P_{\mathfrak{m}}$ be the set of the candidate pairs for inclusion in $\mathfrak{m}$.

Notation 3.3.1. *Let* $\mathbf{T_1(\mathfrak{m})} := \{u \in V_1 \setminus \mathfrak{D}(\mathfrak{m}) : \exists \tilde{u} \in \mathfrak{D}(\mathfrak{m}) : (u, \tilde{u}) \in E_1\}$, *and* $\mathbf{T_2(\mathfrak{m})} := \{v \in V_2 \setminus \mathfrak{R}(\mathfrak{m}) : \exists \tilde{v} \in \mathfrak{R}(\mathfrak{m}) : (v, \tilde{v}) \in E_2\}$.

The set $P_{\mathfrak{m}}$ contains the pairs of uncovered neighbors of covered nodes, and if there is not such a node pair, all the pairs containing two uncovered nodes are added. Formally, let

$$P_{\mathfrak{m}} = \begin{cases} T_1(\mathfrak{m}) \times T_2(\mathfrak{m}) & \text{if } T_1(\mathfrak{m}) \neq \emptyset \text{ and } T_2(\mathfrak{m}) \neq \emptyset, \\ (V_1 \setminus \mathfrak{D}(\mathfrak{m})) \times (V_2 \setminus \mathfrak{R}(\mathfrak{m})) & \text{otherwise.} \end{cases}$$

3.4. Consistency

Let $p = (u, v) \in V_1 \times V_2$, and suppose $\mathfrak{m}$ is a consistent mapping by PT. $Cons_{PT}(p, \mathfrak{m})$ checks whether adding pair p into $\mathfrak{m}$ leads to a consistent mapping by PT.

For example, the consistency function of the induced subgraph isomorphism problem is the following.

Notation 3.4.1. *Let* $\mathbf{\Gamma_1(u)} := \{\tilde{u} \in V_1 : (u, \tilde{u}) \in E_1\}$, *and* $\mathbf{\Gamma_2(v)} := \{\tilde{v} \in V_2 : (v, \tilde{v}) \in E_2\}$, *where* $u \in V_1$ *and* $v \in V_2$. *That is,* $\mathbf{\Gamma_i(w)}$ *denotes the set of neighbors of node* w *in* G_i $(i = 1, 2)$.

Claim 3.4.1. *extend*$(\mathfrak{m}, (u, v))$ *is a consistent mapping by IND if and only if* $\mathfrak{m}$ *is consistent and* $(\forall \tilde{u} \in \mathfrak{D}(\mathfrak{m}) : (u, \tilde{u}) \in E_1 \Leftrightarrow (v, \mathfrak{m}(\tilde{u})) \in E_2)$.

The following formulation gives an efficient way of calculating $Cons_{IND}$.

Claim 3.4.2. $Cons_{IND}((u, v), \mathfrak{m}) := Cons_{IND}(\mathfrak{m}) \wedge \mathcal{L}(u) = \mathcal{L}(v) \wedge (\forall \tilde{v} \in \Gamma_2(v) \cap \mathfrak{R}(\mathfrak{m}) : (u, \mathfrak{m}^{-1}(\tilde{v})) \in E_1) \wedge (\forall \tilde{u} \in \Gamma_1(u) \cap \mathfrak{D}(\mathfrak{m}) : (v, \mathfrak{m}(\tilde{u})) \in E_2)$ *is the consistency function for IND.*

3.5. Cutting rules

$Cut_{PT}(p, \mathfrak{m})$ is defined by a collection of efficiently verifiable conditions. The requirement is that $Cut_{PT}(p, \mathfrak{m})$ can be true only if it is impossible to extend *extend*$(\mathfrak{m}, p)$ to a whole mapping.

As an example, a cutting function of induced subgraph isomorphism problem is presented.

Notation 3.5.1. *Let* $\tilde{\mathbf{T}}_\mathbf{1}(\mathfrak{m}) := (V_1 \setminus \mathfrak{D}(\mathfrak{m})) \setminus T_1(\mathfrak{m})$, *and* $\tilde{\mathbf{T}}_\mathbf{2}(\mathfrak{m}) := (V_2 \setminus \mathfrak{R}(\mathfrak{m})) \setminus T_2(\mathfrak{m})$.

Claim 3.5.1. $Cut_{IND}((u, v), \mathfrak{m}) := |\Gamma_2(v) \cap T_2(\mathfrak{m})| < |\Gamma_1(u) \cap T_1(\mathfrak{m})| \vee |\Gamma_2(v) \cap \tilde{T}_2(\mathfrak{m})| < |\Gamma_1(u) \cap \tilde{T}_1(\mathfrak{m})|$ *is a cutting function for IND.*

4. The VF2++ algorithm

Although any matching order makes the search space of VF2 a tree, its choice turns out to dramatically influence the number of visited states. The goal is to determine an efficient one as quickly as possible.

The main reason for the superiority of VF2++ over VF2 is twofold. Firstly, taking into account the structure and the node labeling of the graph, VF2++ determines a matching order in which most of the unfruitful branches of the search space can be pruned immediately. Secondly, introducing more efficient — nevertheless still easier to compute — cutting rules reduces the chance of going astray even further.

In addition to the usual subgraph isomorphism problem, specialized versions for induced subgraph and graph isomorphism problems have also been designed.

Note that a weaker version of the cutting rules of VF2++ and an efficient candidate set calculation method were described in [4].

The basic ideas and the detailed description of VF2++ are provided in the following.

The goal is to find a matching order in which the algorithm is able to recognize inconsistency or prune the infeasible branches on the highest levels and goes deep only if it is needed.

Notation 4.0.1. *Let* $\mathbf{Conn_H(u)} := |\Gamma_1(u) \cap H|$, *that is the number of neighbors of u which are in H, where* $u \in V_1$ *and* $H \subseteq V_1$.

The principal question is the following. Suppose a mapping $\mathfrak{m}$ is given. For which node of $T_1(\mathfrak{m})$ is the hardest to find a consistent pair in G_2? The more covered neighbors a node in $T_1(\mathfrak{m})$ has — i.e. the largest $Conn_{\mathfrak{D}(\mathfrak{m})}$ it has —, the more rare-to-satisfy consistency constraints for its pair are given.

Most of the graphs of biological and chemical structures are sparse, thus several nodes in $T_1(\mathfrak{m})$ may have the same $Conn_{\mathfrak{D}(\mathfrak{m})}$, which makes reasonable to define a secondary and a tertiary order between them. The observation above proves itself to be as determining, that the secondary ordering prefers nodes with the most uncovered neighbors among which have the same $Conn_{\mathfrak{D}(\mathfrak{m})}$ to increase $Conn_{\mathfrak{D}(\mathfrak{m})}$ of uncovered nodes as much, as possible. The tertiary ordering prefers nodes having the rarest uncovered labels in G_2.

Note that the secondary ordering is the same as ordering by degrees, which is a static data in front of the above used.

These rules can easily result in a matching order which contains the nodes of a long path successively, whose nodes may have low *Conn* and is easy to match into G_2. To try to avoid that, a Breadth-First-Search order is used, and on each of its levels, the ordering procedure described above is applied.

In the following, examples are shown, demonstrating that VF2 may be slow, even though a matching can be found easily by using a proper matching order.

Example 4.0.1. Suppose G_1 can be mapped into G_2 in many ways without node labels. Let $u \in V_1$ and $v \in V_2$.

$\mathcal{L}(u) :=$ *black*

$\mathcal{L}(v) :=$ *black*

$\mathcal{L}(\tilde{u}) :=$ *red* $\forall \tilde{u} \in V_1 \setminus \{u\}$

$\mathcal{L}(\tilde{v}) :=$ *red* $\forall \tilde{v} \in V_2 \setminus \{v\}$

Now, any mapping by $\mathcal{L}$ must contain (u, v), since u is black and no node in V_2 has a black label except v. If unfortunately u were the last node which will get covered, VF2 would check only in the last steps, whether u can be matched to v.

However, had u been the first matched node, u would have been matched immediately to v, so all the mappings would have been precluded in which node labels cannot correspond.

Example 4.0.2. Suppose there is no node label given, and G_1 is a small graph that cannot be mapped into G_2 and $u \in V_1$.

Let $G'_1 := (V_1 \cup \{u'_1, u'_2, \dots, u'_k\}, E_1 \cup \{(u, u'_1), (u'_1, u'_2), \dots, (u'_{k-1}, u'_k)\})$, that is, G'_1 is $G_1 \cup \{a\ k$ long path, which is disjoint from G_1 and one of its starting points is connected to $u \in V_1\}$.

If, unfortunately, the nodes of the path were the first k nodes in the matching order, the algorithm would iterate through all the possible k long paths in G_2, and it would recognize that no path can be extended to G'_1.

However, had it started by the matching of G_1, it would not have matched any nodes of the path.

These examples may look artificial, but the same problems also appear in real-world instances, even though in a less obvious way.

4.1. Matching order

Notation 4.1.1. *Let* $\mathbf{F}_{\mathcal{M}}(\boldsymbol{l}) := |\{v \in V_2 : l = \mathcal{L}(v)\}| - |\{u \in \mathcal{M} : l = \mathcal{L}(u)\}|$, *where l is a label and* $\mathcal{M} \subseteq V_1$.

Definition 4.1.1. Let $\textbf{arg max}_f(S) := \{u \in S : f(u) = max_{v\in S}\{f(v)\}\}$ and $\textbf{arg min}_f(S) := arg\ max_{(-f)}(S)$, where S is a finite set and $f : S \longrightarrow \mathbb{R}$.

Notation 4.1.2. *Let $deg(v)$ denote the degree of node v.*

Algorithm 2 *The method of $VF2++$ for determining the node order*

```
1: procedure VF2++ORDER
2:     $\mathcal{M} := \emptyset$                                        ▷ matching order
3:     while $V_1 \backslash \mathcal{M} \neq \emptyset$ do
4:         $r \in$ arg max$_{deg}$ (arg min$_{F_\mathcal{M} \circ \mathcal{L}}(V_1 \backslash \mathcal{M})$)
5:         Compute $T$, a BFS tree with root node $r$.
6:         for $d = 0, 1, \dots, depth(T)$ do
7:             $V_d$:=nodes of the $d$th level
8:             Process $V_d$                                  ▷ See Algorithm 3
```

Algorithm 3 *The method for processing a level of the BFS tree*

```
1: procedure VF2++PROCESSLEVEL($V_d$)
2:     while $V_d \neq \emptyset$ do
3:         $m \in$ arg min$_{F_\mathcal{M} \circ\, \mathcal{L}}$( arg max$_{deg}$(arg max$_{Conn_\mathcal{M}}(V_d)$))
4:         $V_d := V_d \backslash m$
5:         Append node $m$ to the end of $\mathcal{M}$
6:         Refresh $F_\mathcal{M}$
```

Algorithm 2 is a high-level description of the matching order procedure of VF2++. It computes a BFS tree for each component in ascending order of their rarest node labels and largest deg, whose root vertex is the minimal node of its component. Algorithm 3 is a method to process a level of the BFS tree, which appends the nodes of the current level in descending lexicographic order by $(Conn_\mathcal{M}, deg, -F_\mathcal{M})$ separately to $\mathcal{M}$, and refreshes $F_\mathcal{M}$ immediately.

Claim 4.1.1. *Algorithm 2 provides a matching order.*

4.2. Cutting rules

This section presents the cutting rules of VF2++, which are improved by using extra information coming from the node labels.

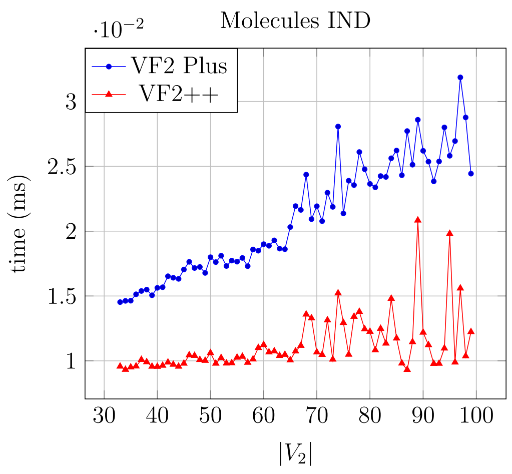
Notation 4.2.1. *Let* $\boldsymbol{\Gamma_1^l(u)} := \{\tilde{u} : \mathcal{L}(\tilde{u}) = l \wedge \tilde{u} \in \Gamma_1(u)\}$ *and* $\boldsymbol{\Gamma_2^l(v)} := \{\tilde{v} : \mathcal{L}(\tilde{v}) = l \wedge \tilde{v} \in \Gamma_2(v)\}$, *where $u \in V_1$, $v \in V_2$ and l is a label.*

Claim 4.2.1 (*Cutting Function for ISO*).

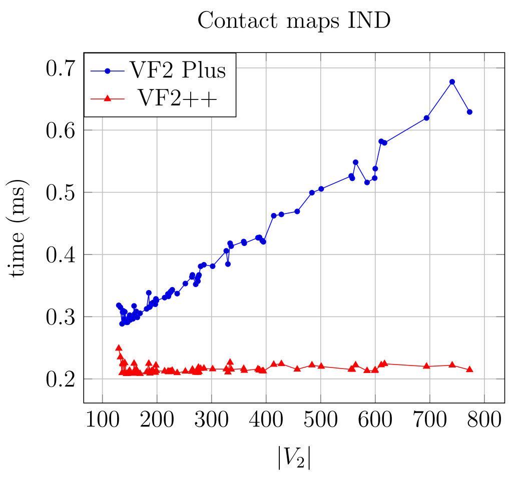
$$LabCut_{ISO}((u, v), \mathfrak{m}) := \bigvee_{l \text{ is label}} |\Gamma_2^l(v) \cap T_2(\mathfrak{m})| \neq |\Gamma_1^l(u) \cap T_1(\mathfrak{m})| \vee$$

$$\bigvee_{l \text{ is label}} |\Gamma_2^l(v) \cap \tilde{T}_2(\mathfrak{m})| \neq |\Gamma_1^l(u) \cap \tilde{T}_1(\mathfrak{m})|$$

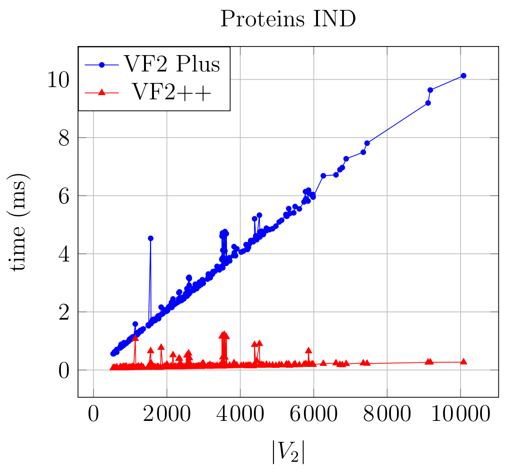
is a cutting function for ISO.



(a) In the case of molecules, the algorithms have similar behavior, but VF2++ is almost two times faster even on such small graphs.



(b) On contact maps, VF2++ runs almost in constant time, while VF2 Plus has a near-linear behavior.



(c) Both of the algorithms have linear behavior on protein graphs. VF2++ is more than 10 times faster than VF2 Plus.

Fig. 1. Induced subgraph isomorphism problem on biological graphs.

Claim 4.2.2 (*Cutting Function for IND*).

$$LabCut_{IND}((u,v),\mathfrak{m}) := \bigvee_{l \text{ is label}} |\Gamma_2^l(v)\cap T_2(\mathfrak{m})| < |\Gamma_1^l(u)\cap T_1(\mathfrak{m})| \vee$$
$$\bigvee_{l \text{ is label}} |\Gamma_2^l(v)\cap \tilde{T}_2(\mathfrak{m})| < |\Gamma_1^l(u)\cap \tilde{T}_1(\mathfrak{m})|$$

is a cutting function for IND.

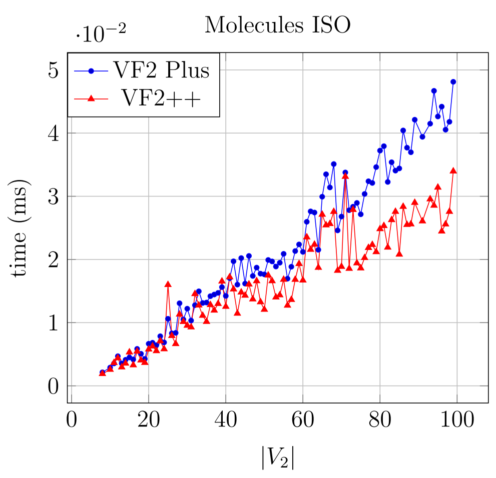
Claim 4.2.3 (*Cutting Function for SUB*).

$$LabCut_{SUB}((u,v),\mathfrak{m}) := \bigvee_{l \text{ is label}} |\Gamma_2^l(v)\cap T_2(\mathfrak{m})| < |\Gamma_1^l(u)\cap T_1(\mathfrak{m})|$$

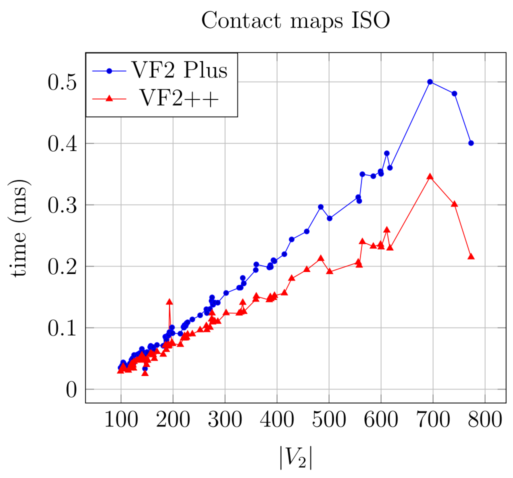
is a cutting function for SUB.

5. Implementation details

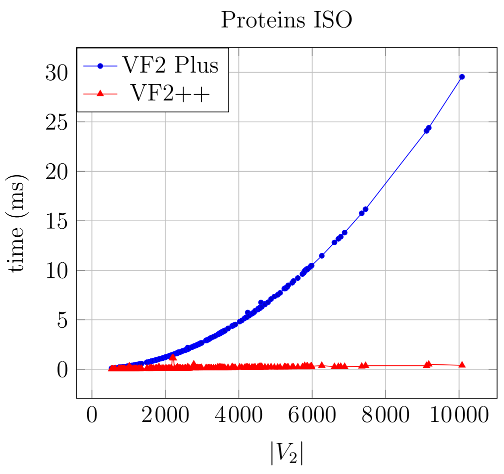
This section provides a detailed summary of an efficient implementation of VF2++.



(a) The results are close to each other on molecules, but VF2++ seems to be slightly faster as the number of nodes increases.



(b) In the case of contact maps, there is no significant difference, but VF2++ performs consistently better.



(c) On protein graphs, VF2 Plus has a super linear time complexity, while VF2++ runs in near constant time. The difference is about two orders of magnitude on large graphs.

Fig. 2. Graph isomorphism problem on biological graphs.

Notation 5.0.1. *Let Δ_1 and Δ_2 denote the largest degree in G_1 and G_2, respectively, and let* $\Delta = \max\{\Delta_1, \Delta_2\}$.

5.1. *Storing a mapping*

After fixing an arbitrary node order $(u_0, u_1, .., u_{|V_1|-1})$ of G_1, an array M can be used to store the current mapping in the following way.

$$M[i] = \begin{cases} v & \textit{if } (u_i, v) \textit{ is in the mapping} \\ \textit{INVALID} & \textit{if no node has been mapped to } u_i, \end{cases}$$

where $i \in \{0, 1, .., |V_1| - 1\}$, $v \in V_2$ and *INVALID* means “no node”.

5.2. *Avoiding the recurrence*

The recursion of Algorithm 1 can be realized as a *while loop*, which has a loop counter *depth* denoting the current depth of the recursion. Fixing a matching order, let M denote the array storing the current mapping. Observe that M is *INVALID* from

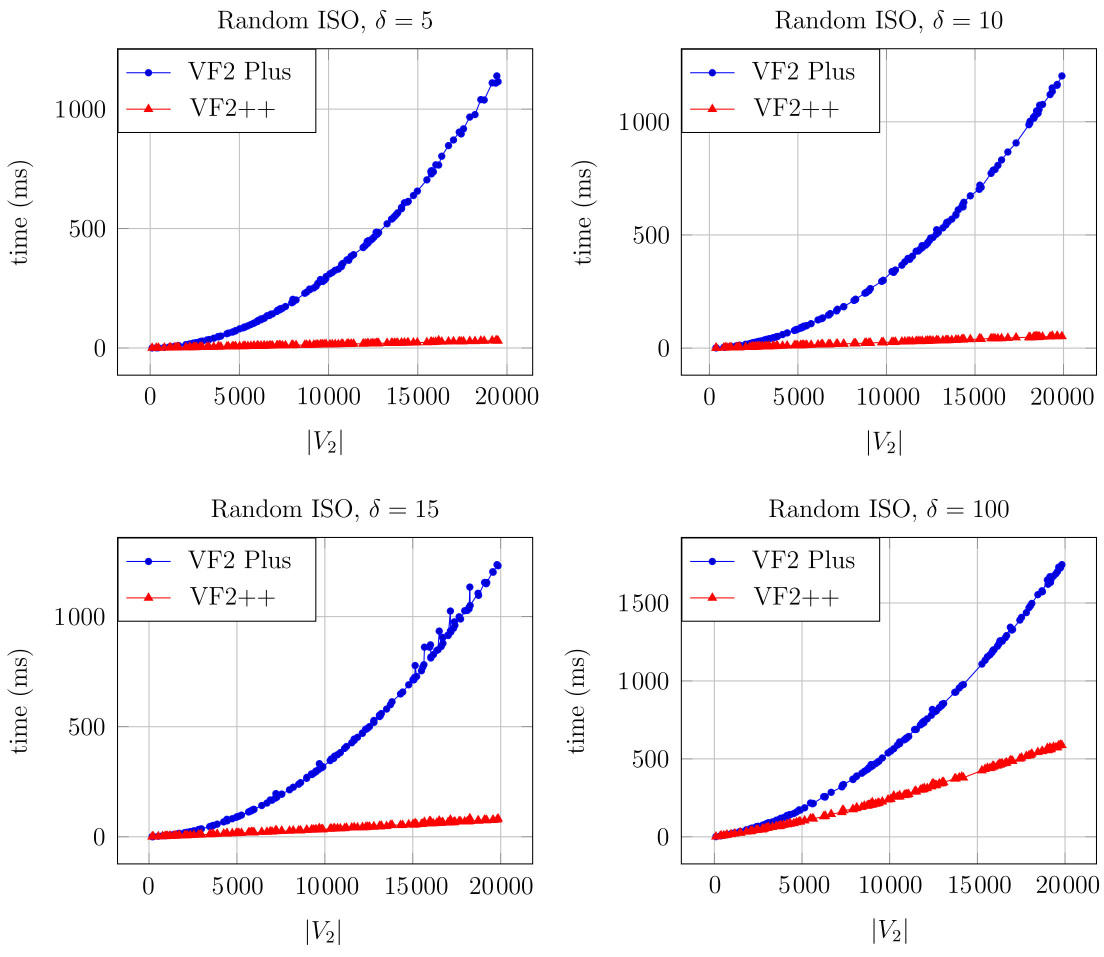


Fig. 3. Graph isomorphism problem on random graphs.

index *depth* + 1 and not *INVALID* before *depth*. $M[depth]$ changes while the state is being processed, but the property is held before both stepping back to a predecessor state and exploring a successor state.

The necessary part of the candidate set is easy to maintain or compute by following the steps described in Section 3.3. A much faster method has been designed for biological and sparse graphs, see the next section for details.

5.3. Calculating the candidates for a node

The task is not to maintain the candidate set, but to generate the candidate nodes in G_2 for a given node $u \in V_1$. In case of any of the three problem types and a mapping $\mathfrak{m}$, if a node $v \in V_2$ is a potential pair of $u \in V_1$, then $\forall u' \in \mathfrak{D}(\mathfrak{m}) : (u, u') \in E_1 \Rightarrow (v, \mathfrak{m}(u')) \in E_2$. That is, each covered neighbor of u has to be mapped to a covered neighbor of v, i.e. selecting arbitrarily a covered neighbor u' of u, all of the admissible candidates for u are among the neighbors of $\mathfrak{m}(u')$.

Having said that, an algorithm running in $\Theta(\Delta_2)$ time is describable if there exists a covered node in the component containing u, and a linear one otherwise.

5.4. Determining the node order

This section describes how the node order preprocessing method of VF2++ can efficiently be implemented.

For using lookup tables, the node labels are associated with the numbers $\{0, 1, \ldots, |K| - 1\}$, where K is the set of the labels. It enables $F_{\mathcal{M}}$ to be stored in an array. At first, the node order $\mathcal{M} = \emptyset$, so $F_{\mathcal{M}}[i]$ is the number of nodes in V_2 having label i, which is easy to compute in $\Theta(|V_2|)$ steps.

Representing $\mathcal{M} \subseteq V_1$ as an array of size $|V_1|$, both the computation of the BFS tree, and processing its levels by Algorithm 3 can be done in-place by swapping nodes.

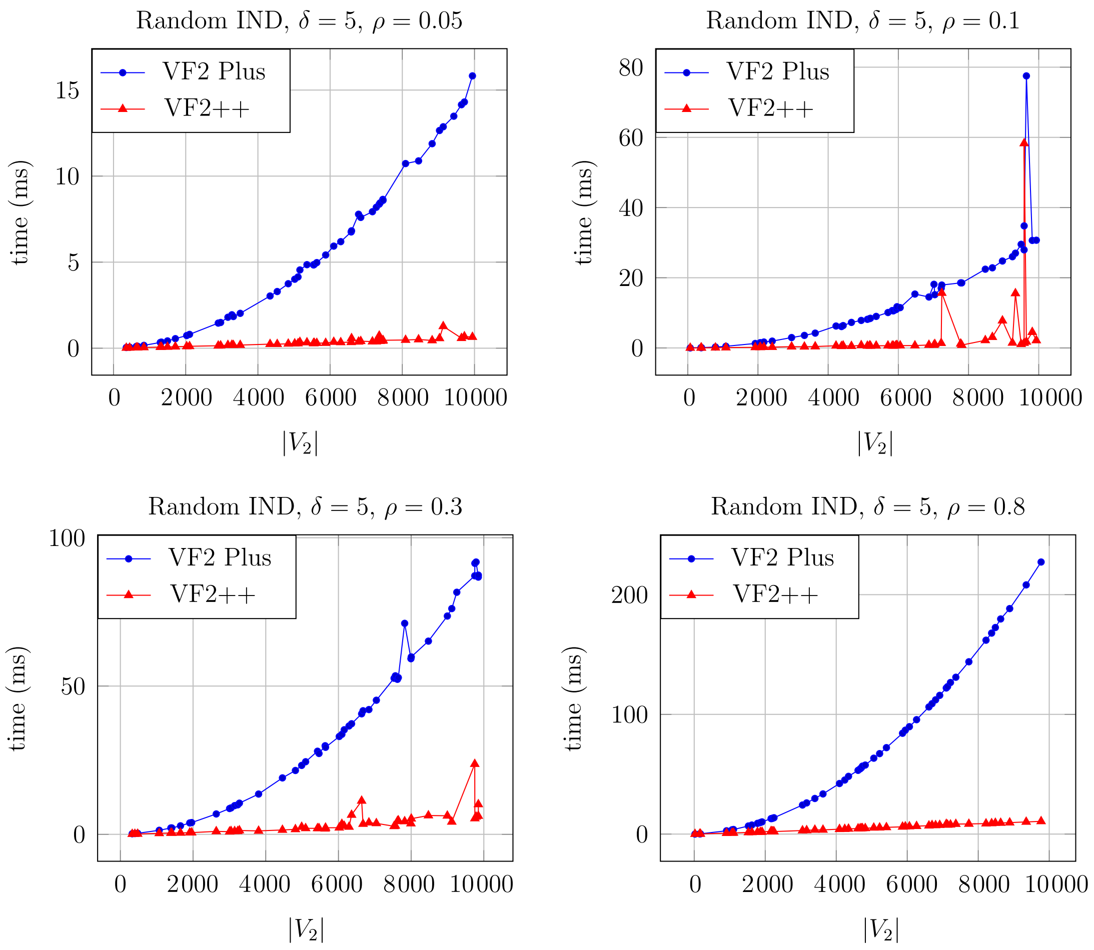


Fig. 4. Induced subgraph isomorphism problem on random graphs having an average degree of 5.

5.5. *Cutting rules*

Section 4.2 described the cutting rules using the sets T_1, T_2, $\tilde{T}_1$ and $\tilde{T}_2$, which are dependent on the current mapping. The aim is to check the labeled cutting rules of VF2++ in $\Theta(\Delta)$ time.

Firstly, suppose that these four sets are given a way, that checking whether a node is in a certain set takes constant time, e.g. they are given by their 0–1 characteristic vectors. Let L be an initially zero integer lookup table of size $|K|$. After incrementing $L[\mathcal{L}(u')]$ for all $u' \in \Gamma_1(u) \cap T_1(\mathfrak{m})$ and decrementing $L[\mathcal{L}(v')]$ for all $v' \in \Gamma_2(v) \cap T_2(\mathfrak{m})$, the first part of the cutting rules can be checked in $\Theta(\Delta)$ time by considering the proper signs of L. Setting L to zero takes $\Theta(\Delta)$ time again, which makes it possible to use the same table through the whole algorithm. The second part of the cutting rules can be verified using the same method with $\tilde{T}_1$ and $\tilde{T}_2$ instead of T_1 and T_2. Thus, the overall time complexity is $\Theta(\Delta)$.

To maintain the sets T_1, T_2, $\tilde{T}_1$ and $\tilde{T}_2$, two other integer lookup tables storing the number of covered neighbors of the nodes of the two graphs can be used. This representation allows constant-time membership checking, furthermore it is maintainable in $\Theta(\Delta)$ time whenever a node pair is added or subtracted by incrementing or decrementing the proper indices. A further improvement is that the values of $L[\mathcal{L}(u')]$ in case of checking u are dependent only on u, i.e. on the current depth of the recursion, so for each $u \in V_1$, an array of pairs *(label, number of such labels)* can store L. Note that these arrays are at most of size Δ_1 if pairs with non-appearing node labels are discarded.

Using similar techniques, the consistency function can be evaluated in $\Theta(\Delta)$ steps, as well.

6. Experimental results

This section compares the performance of VF2++ and VF2 Plus. According to our experience, both algorithms run faster than VF2 with orders of magnitude, thus its inclusion was not reasonable.

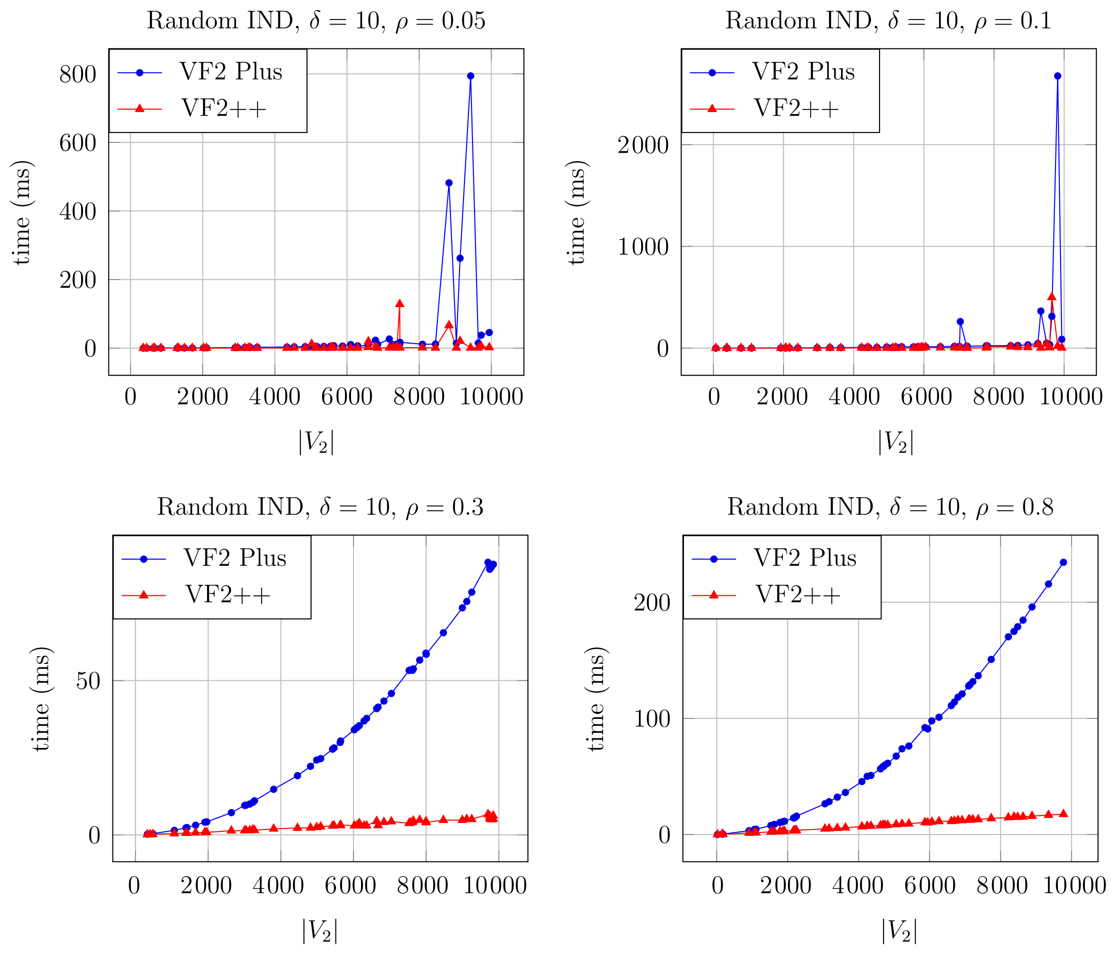


Fig. 5. Induced subgraph isomorphism problem on random graphs having an average degree of 10.

The algorithms were implemented in C++ using the open-source LEMON graph and network optimization library [10,12]. The tests were carried out on a Linux-based system with an Intel i7 X980 3.33 GHz CPU and 6 GB of RAM.

6.1. Biological graphs

The tests have been executed on a recent biological dataset created for the International Contest on Pattern Search in Biological Databases [22], which has been constructed of molecule, protein and contact map graphs extracted from the Protein Data Bank [17].

The molecule dataset contains small graphs with less than 100 nodes and an average degree of less than 3. The protein dataset contains graphs having 500–10 000 nodes and an average degree of 4, while the contact map dataset contains graphs with 150–800 nodes and an average degree of 20.

In the following, both the induced subgraph and the graph isomorphism problems will be examined. This dataset provides graph pairs, between which all the induced subgraph isomorphisms have to be found. For the running times, please see Fig. 1.

In an other experiment, the nodes of each graph in the database had been shuffled, and an isomorphism between the shuffled and the original graph was searched. The running times are shown in Fig. 2.

6.2. Random graphs

This section compares VF2++ with VF2 Plus on random graphs of large size. The node labels are uniformly distributed. Let δ denote the average degree. For the parameters of problems solved in the experiments, please see the top of each chart.

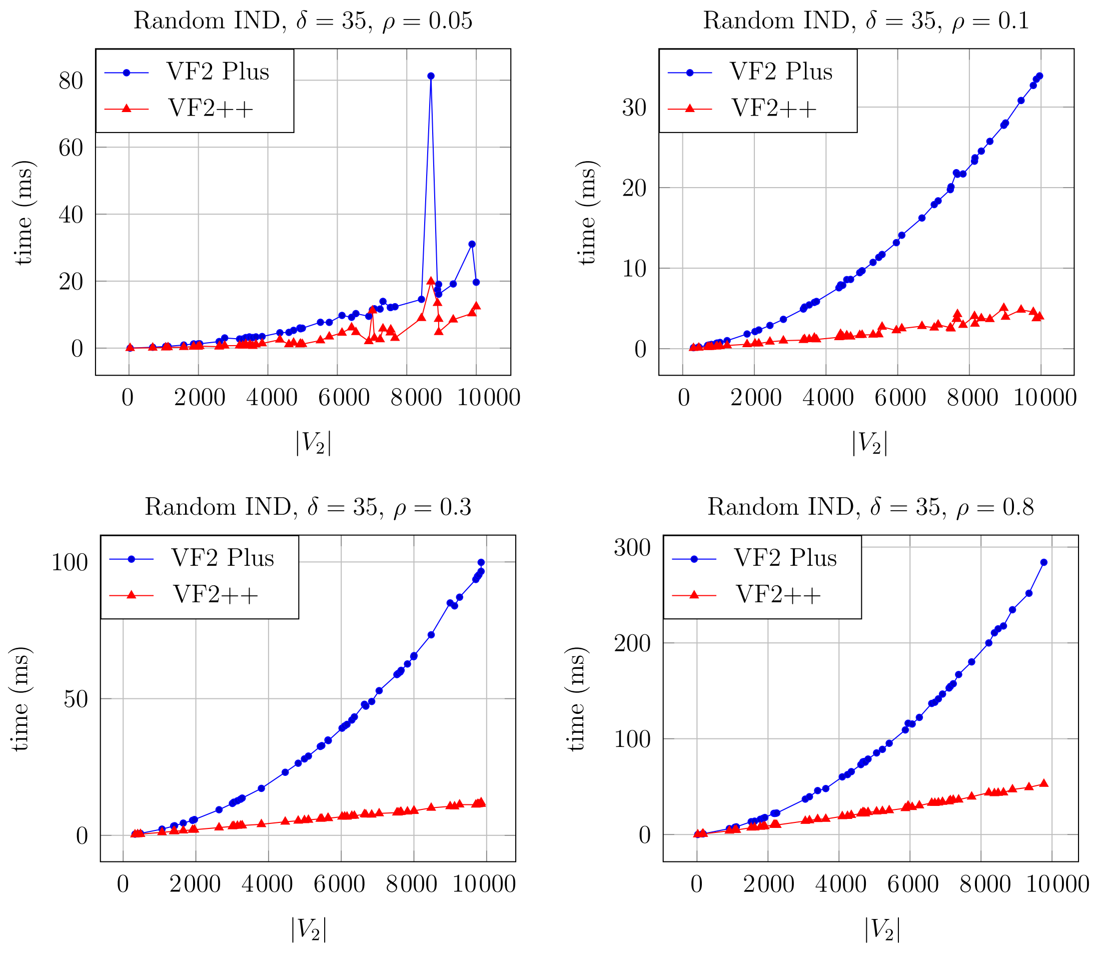


Fig. 6. Induced subgraph isomorphism problem on random graphs having an average degree of 35.

6.2.1. *Graph isomorphism problem*

To evaluate the efficiency of the algorithms in the case of graph isomorphism problem, random connected graphs of less than 20 000 nodes have been considered. Generating a random graph and shuffling its nodes, an isomorphism had to be found. Fig. 3 shows the running times on graph sets of various density.

6.2.2. *Induced subgraph isomorphism problem*

This section presents a comparison of VF2++ and VF2 Plus in the case of induced subgraph isomorphism problem. In addition to the size of graph G_2, that of G_1 dramatically influences the hardness of a given problem too, so the overall picture is provided by examining graphs to be embedded of various size.

For each chart, a number $0 < \rho < 1$ has been fixed, and the following has been executed 150 times. Generating a large graph G_2 of an average degree of δ, choose 10 of its induced subgraphs having $\rho|V_2|$ nodes, and for all the 10 subgraphs find a mapping by using both graph isomorphism algorithms. The $\delta = 5, 10, 35$ and $\rho = 0.05, 0.1, 0.3, 0.8$ cases have been examined, see Figs. 4–6.

As the experiments above demonstrates, VF2++ is faster than VF2 Plus and able to handle really large graphs in milliseconds. Note that when *IND* was considered and the graph to be embedded had proportionally few nodes ($\rho = 0.05$, or $\rho = 0.1$), then VF2 Plus produced some inefficient node orders (e.g. see the $\delta = 10$ case on Fig. 5). If these instances had been excluded, the charts would have looked similarly to the other ones. Unsurprisingly, as denser graphs are considered, both VF2++ and VF2 Plus slow down slightly, but remain practically usable even on graphs having 10 000 nodes.

7. Conclusion

This paper presented VF2++, a new (sub)graph isomorphism algorithm based on VF2, and analyzed it from a practical point of view.

Recognizing the importance of the node order and determining an efficient one, VF2++ is able to match graphs of thousands of nodes in near practically linear time including preprocessing. In addition to the proper order, VF2++ uses more efficient cutting rules, which are easy to compute and make the algorithm able to prune most of the unfruitful branches without going astray.

In order to show the efficiency of the new method, it has been compared to VF2 Plus [4], which is the best contemporary algorithm.

The experiments show that VF2++ consistently outperforms VF2 Plus on biological graphs. It seems to be asymptotically faster on protein and on contact map graphs in the case of induced subgraph isomorphism problem, while in the case of graph isomorphism problem, it has definitely better asymptotic behavior on protein graphs.

Regarding random sparse graphs, not only has VF2++ proved itself to be faster than VF2 Plus, but it also has a practically linear behavior both in the case of induced subgraph and graph isomorphism problems.

Acknowledgments

This research project was initiated and sponsored by QuantumBio Inc. [18].

The authors were supported by the Hungarian Scientific Research Fund - OTKA, K109240 and by the János Bolyai Research Fellowship program of the Hungarian Academy of Sciences.

References

[1] V. Arvind, B. Das, J. Köbler, S. Toda, Colored hypergraph isomorphism is fixed parameter tractable, Algorithmica 71 (2015) 120–138.
[2] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, A. Ferro, A subgraph isomorphism algorithm and its application to biochemical data, BMC Bioinformatics 14 (Suppl. 7) (2013) S13.
[3] H. Bunke, Graph matching: Theoretical foundations, algorithms, and applications, in: International Conference on Vision Interface, 2000, pp. 82–84.
[4] V. Carletti, P. Foggia, M. Vento, VF2 Plus: An improved version of VF2 for biological graphs, in: Conference: Graph-Based Representations in Pattern Recognition, At Beijing, 2015.
[5] C.J. Colbourn, On testing isomorphism of permutation graphs, Networks 11 (1) (1981) 13–21.
[6] S.A. Cook, The complexity of theorem-proving procedures, in: Proc. 3rd ACM Symposium on Theory of Computing, 1971, pp. 151–158.
[7] L.P. Cordella, P. Foggia, C. Sansone, M. Vento, Performance evaluation of the VF graph matching algorithm, in: Proc. of the 10th ICIAP, IEEE Computer Society Press, 1999, pp. 1172–1177.
[8] L.P. Cordella, P. Foggia, C. Sansone, M. Vento, A (sub)graph isomorphism algorithm for matching large graphs, IEEE Trans. Pattern Anal. Mach. Intell. 26 (10) (2004) 1367–1372.
[9] L.P. Cordella, M. Vento, Symbol recognition in documents: a collection of techniques? Int. J. Doc. Anal. Recognit. 3 (2) (2000) 73–88.
[10] B. Dezső, A. Jüttner, P. Kovács, LEMON - an open source C++ graph template library, Electron. Notes Theor. Comput. Sci. 264 (5) (2011) 23–45 Proceedings of the Second Workshop on Generative Technologies (WGT) 2010.
[11] J.E. Hopcroft, J.K. Wong, Linear time algorithm for isomorphism of planar graphs, in: Proceeding STOC '74 Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, 1974, pp. 172–184.
[12] LEMON: Library for Efficient Modeling and Optimization in Networks. URL http://lemon.cs.elte.hu.
[13] J. Liu, Y.T. Lee, A graph-based method for face identification from a single 2D line drawing, IEEE Trans. Pattern Anal. Mach. Intell. 23 (10) (2001) 1106–1119.
[14] G.S. Lue, K.S. Booth, A linear time algorithm for deciding interval graph isomorphism, J. ACM 26 (2) (1979) 183–195.
[15] E.M. Luks, Isomorphism of graphs of bounded valence can be tested in polynomial time, J. Comput. Syst. Sci. 25 (1) (1982) 42–65.
[16] B.D. McKay, Practical graph isomorphism, Congr. Numer. 30 (1981) 45–87.
[17] Protein Data Bank. URL http://www.rcsb.org/pdb.
[18] QuantumBio Inc. URL http://www.quantumbioinc.com.
[19] C. Solnon, AllDifferent-based filtering for subgraph isomorphism, Artificial Intelligence 174 (2010) 850–864.
[20] J.R. Ullmann, An algorithm for subgraph isomorphism, J. ACM 23 (1) (1976) 31–42.
[21] J.R. Ullmann, Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism, J. Exp. Algorithmics 15 (2010) Article No. 1.6.
[22] M. Vento, X. Jiang, P. Foggia, International contest on pattern search in biological databases, 2015. http://biograph2014.unisa.it.