Péter Madarasi

Algorithms for Matching Biological Graphs

Mathematics BSc
Applied Mathematics major
Thesis

Supervisor:
Alpár Jüttner
senior research fellow
ELTE Institute of Mathematics,
Department of Operations Research

Eötvös Loránd University
Faculty of Science

Budapest, 2016

Abstract

Subgraph isomorphism is a well-known NP-Complete problem, while its special case, the graph isomorphism problem is one of the few problems in NP neither known to be in P nor NP-Complete. Their appearance in many fields of application such as pattern analysis, computer vision questions and the analysis of chemical and biological systems has fostered the design of various algorithms for handling special graph structures.

The idea of using state space representation and checking some conditions in each state to prune the search tree has made the VF2 algorithm one of the state of the art graph matching algorithms for more than a decade. Recently, biological questions of ever increasing importance have required more efficient, specialized algorithms.

This paper presents VF2++, a new algorithm based on the original VF2, which runs significantly faster on most test cases and performs especially well on special graph classes stemming from biological questions. VF2++ handles graphs of thousands of nodes in practically near linear time including preprocessing. Not only is it an improved version of VF2, but in fact, it is by far the fastest existing algorithm regarding biological graphs.

After defining the problems and briefly summarizing the state of the art algorithms, a detailed description of VF2, VF2 Plus and VF2++ is given including implementation details and tricks. In order to evaluate the effectiveness of VF2++, an extensive comparison to the other contemporary algorithms is shown, using a wide range of inputs, including both real life biological and chemical datasets and standard randomly generated graph series.

This thesis is based on our previous work motivated and sponsored by QuantumBio Inc., and all the developed algorithms are available as the part of the open source LEMON graph and network optimization library (http://lemon.cs.elte.hu).

Contents

1 Introduction 4

2 Problem Statement 6
2.1 Definitions 6
2.2 Common problems 7

3 The VF2 Algorithm 8
3.1 Common notations 8
3.2 Overview of the algorithm 10
3.3 The candidate set P(s) 11
3.4 Consistency 11
3.4.1 Induced subgraph isomorphism 12
3.4.2 Graph isomorphism 12
3.4.3 Subgraph isomorphism 12
3.5 Cutting rules 12
3.5.1 Induced subgraph isomorphism 12
3.5.2 Graph isomorphism 13
3.5.3 Subgraph isomorphism 13

4 The VF2++ Algorithm 15
4.1 Preparations 15
4.2 Idea behind the algorithm 17
4.3 Total ordering 18
4.4 Cutting rules 20
4.4.1 Induced subgraph isomorphism 20
4.4.2 Graph isomorphism 21
4.4.3 Subgraph isomorphism 21
4.5 Implementation details 21
4.5.1 Storing a mapping 21
4.5.2 Avoiding the recurrence 21
4.5.3 Calculating the candidates for a node 22
4.5.4 Determining the node order 23
4.5.5 Cutting rules 23

5 The VF2 Plus Algorithm 25
5.1 Ordering procedure 25

6 Experimental results **26**
6.1 Biological graphs 26
6.1.1 Induced subgraph isomorphism 27
6.1.2 Graph ismorphism 29
6.2 Random graphs 31
6.2.1 Graph isomorphism 31
6.2.2 Induced subgraph isomorphism 34

7 Conclusion **42**

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 studiedness of graphs make them practical for modelling and appear constantly in several seemingly independent fields. Bioinformatics and chemistry are amongst the most relevant and most important fields.

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

For instance, a molecular structure can be considered as a graph, whose nodes correspond to atoms and whose edges to chemical bonds. The secondary structure of a protein can also be represented as a graph, where nodes are associated with aminoacids and the edges with hydrogen bonds. The nodes are often whole molecular components and the edges represent some relationships among them. The similarity and dissimilarity of objects corresponding to nodes are incorporated to the model by *node labels*. Many other chemical and biological structures can easily be modeled in a similar way. Understanding such networks basically requires finding specific subgraphs, which can not avoid the application of graph matching algorithms.

Finally, let some of the other real-world fields related to some variants of graph matching be briefly mentioned: pattern recognition and machine vision [4], symbol recognition [13], face identification [12].

Subgraph and induced subgraph matching 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 algorithms are known for various graph classes, like trees and planar graphs[9], bounded valence graphs[15], interval graphs[14] or permutation graphs[5].

In the following, some algorithms based on other approaches are summarized, which do not need any restrictions on the graphs. However, an overall polynomial behaviour is not expectable from such an alternative, it may often have good performance, even on a graph class for which poly-

nomial algorithm is known. Note that this summary containing only exact matching algorithms is far not complete, neither does it cover all the recent algorithms.

The first practically usable approach was due to **Ullmann**[18] which is a commonly used depth-first search based algorithm 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**[19] presents an improved version of this algorithm based on a bit-vector solution for the binary Constraint Satisfaction Problem.

The **Nauty** algorithm[17] transforms the two graphs to a canonical form before starting to check for the 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[16] uses a depth-first search strategy and formulates the matching as a Constraint Satisfaction Problem to prune the search tree. The constraints are that the mapping has to be injective and edge-preserving, hence it is possible to handle new matching types as well.

The **RI** algorithm[3] 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[20].

The currently most commonly used algorithm is the **VF2**[7], the improved version of VF[11], 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 can't be up to new specialized algorithms, it is still widely used due to its simplicity and space efficiency. VF2 uses a state space representation and checks some conditions in each state to prune the search tree.

Our first graph matching algorithm was the first version of VF2 which recognizes the significance of the node ordering, more opportunities to increase the cutting efficiency and reduce its computational complexity. This project was initiated and sponsored by QuantumBio Inc.[10] and the implementation — along with a source code — has been published as a part of LEMON[8] open source graph library.

This thesis introduces **VF2++**, a new further improved algorithm for

the graph and (induced)subgraph isomorphism problem, which uses efficient cutting rules and determines a node order in which VF2 runs significantly faster on practical inputs.

Meanwhile, another variant called **VF2 Plus**[21] 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 small graph, in which the VF2 works more efficiently.

2 Problem Statement

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

2.1 Definitions

Throughout the paper $G_{small} = (V_{small}, E_{small})$ and $G_{large} = (V_{large}, E_{large})$ denote two undirected graphs.

Definition 2.1.1. *G_{small} and G_{large} are **isomorphic** if $\exists M : V_{small} \longrightarrow V_{large}$ bijection, for which the following is true:*

$$\forall u, v \in V_{small} : (u, v) \in E_{small} \Leftrightarrow (M(u), M(v)) \in E_{large}$$

For the sake of simplicity in this paper subgraphs and induced subgraphs are defined in a more general way than usual:

Definition 2.1.2. *G_{small} is a **subgraph** of G_{large} if $\exists I : V_{small} \longrightarrow V_{large}$ injection, for which the following is true:*

$$\forall u, v \in V_{small} : (u, v) \in E_{small} \Rightarrow (I(u), I(v)) \in E_{large}$$

Definition 2.1.3. *G_{small} is an **induced subgraph** of G_{large} if $\exists I : V_{small} \longrightarrow V_{large}$ injection, for which the following is true:*

$$\forall u, v \in V_{small} : (u, v) \in E_{small} \Leftrightarrow (I(u), I(v)) \in E_{large}$$

Definition 2.1.4. *$lab : (V_{small} \cup V_{large}) \longrightarrow K$ 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 $lab(u) = lab(v)$.*

When node labels are also given, the matched nodes must have the same labels. For example, the node labeled isomorphism is phrased by

Definition 2.1.5. *G_{small} and G_{large} are **isomorphic by the node label function lab** if $\exists M : V_{small} \longrightarrow V_{large}$ bijection, for which the following is true:*

$$(\forall u, v \in V_{small} : (u, v) \in E_{small} \Leftrightarrow (M(u), M(v)) \in E_{large}) \text{ and}$$
$$(\forall u \in V_{small} : lab(u) = lab(M(u)))$$

The other two definitions can be extended in the same way.

Note that edge label function can be defined similarly to node label function, and all the definitions can be extended with additional conditions, but it is out of the scope of this work.

The equivalence of two nodes is usually defined by another relation, $R \subseteq (V_{small} \cup V_{large})^2$. This overlaps with the definition given above if R is an equivalence relation, which does not mean restriction in biological and chemical applications.

2.2 Common problems

The focus of this paper is on two extensively studied topics, the subgraph isomorphism and its variations. However, the following problems also appear in many applications.

The **subgraph matching problem** is the following: is G_{small} isomorphic to any subgraph of G_{large} by a given node label?

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

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

In addition to existence, it may be needed to show such a subgraph, or it may be necessary to list all of them.

It should be noted that some authors misleadingly refer to the term *subgraph isomorphism problem* as an *induced subgraph isomorphism problem*.

The following sections give the descriptions of VF2, VF2++, VF2 Plus and a particular comparison.

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 will be discussed.

3.1 Common notations

Assume G_{small} is searched in G_{large}. The following definitions and notations will be used throughout the whole paper.

Definition 3.1.1. *A set $M \subseteq V_{small} \times V_{large}$ is called **mapping**, if no node of V_{small} or of V_{large} appears in more than one pair in M. That is, M uniquely associates some of the nodes in V_{small} with some nodes of V_{large} and vice versa.*

Definition 3.1.2. *Mapping M **covers** a node v, if there exists a pair in M, which contains v.*

Definition 3.1.3. *A mapping M is* **whole mapping***, if M covers all the nodes in V_{small}.*

Notation 3.1.1. *Let* $\mathbf{M_{small}(s)} := \{u \in V_{small} : \exists v \in V_{large} : (u, v) \in M(s)\}$ *and* $\mathbf{M_{large}(s)} := \{v \in V_{large} : \exists u \in V_{small} : (u, v) \in M(s)\}$.

Notation 3.1.2. *Let* $\mathbf{Pair(M, v)}$ *be the pair of v in M, if such a node exist, otherwise* $\mathbf{Pair(M, v)}$ *is undefined. For a mapping M and $v \in V_{small} \cup V_{large}$.*

Note that if $\mathbf{Pair(M, v)}$ exists, then it is unique

The definitions of the isomorphism types can be rephrased on the existence of a special whole mapping M, since it represents a bijection. For example

$M \subseteq V_{small} \times V_{large}$ represents an induced subgraph isomorphism $\Leftrightarrow$ M is whole mapping and

$$\forall u, v \in V_{small} : (u, v) \in E_{small} \Leftrightarrow (Pair(M, u), Pair(M, v)) \in E_{large}.$$

Definition 3.1.4. *A set of whole mappings is called **problem type**.*

Throughout the paper, **PT** denotes a generic problem type which can be substituted by any problem type.

A whole mapping W **is of type PT**, if $W \in PT$. Using this notations, VF2 searches a whole mapping W of type PT.

For example the problem type of graph isomorphism problem is the following. A whole mapping W is in **ISO**, iff the bijection represented by W satisfies **Definition 2.1.1)**. The subgraph- and induced subgraph matching problems can be formalized in a similar way. Let their problem types be denoted as **SUB** and **IND**.

Definition 3.1.5. *PT is an **expanding problem type** if $\forall\ W \in PT$: $\forall u_1, u_2 \in V_{small}$: $(u_1, u_2) \in E_{small} \Rightarrow (Pair(W, u_1), Pair(W, u_2)) \in E_{large}$, that is each edge of G_{small} has to be mapped to an edge of G_{large} for each mapping in PT.*

Note that ISO, SUB and IND are expanding problem types.

This paper deals with the three problem types mentioned above only, but the following generic definitions make it possible to handle other types as well. Although it may be challenging to find a proper consistency function and an efficient cutting function.

Definition 3.1.6. *Let M be a mapping. A logical function $\mathbf{Cons_{PT}}$ is a **consistency function by** $\mathbf{PT}$, if the following holds. If there exists whole mapping W of type PT for which $M \subseteq W$, then $Cons_{PT}(M)$ is true.*

Definition 3.1.7. *Let M be a mapping. A logical function $\mathbf{Cut_{PT}}$ is a **cutting function by** $\mathbf{PT}$, if the following holds. $\mathbf{Cut_{PT}(M)}$ is false if M can be extended to a whole mapping W of type PT.*

Definition 3.1.8. *M is said to be **consistent mapping by** $\mathbf{PT}$, if $Cons_{PT}(M)$ is true.*

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

Notation 3.1.3. *Let $\mathbf{Cons_{PT}(p, M)} := Cons_{PT}(M \cup \{p\})$ and $\mathbf{Cut_{PT}(p, M)} := Cut_{PT}(M \cup \{p\})$, where $p \in V_{small} \times V_{large}$ and $M \cup \{p\}$ is mapping.*

$Cons_{PT}$ 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.

3.2 Overview of the algorithm

VF2 uses a state space representation of mappings, $Cons_{PT}$ for excluding inconsistency with the problem type and Cut_{PT} for pruning the search tree. Each state s of the matching process can be associated with a mapping $M(s)$.

Algorithm 1) is a high level description of the VF2 matching algorithm.

Algorithm 1 *A high level description of VF2*

1: **procedure** VF2(State s, ProblemType PT)
2: **if** $M(s)$ covers V_{small} **then**
3: Output($M(s)$)
4: **else**
5: Compute the set $P(s)$ of the pairs candidate for inclusion in $M(s)$
6: **for all** $p \in P(s)$ **do**
7: **if** $\text{Cons}_{PT}(p, M(s)) \wedge \neg\text{Cut}_{PT}(p, M(s))$ **then**
8: Compute the nascent state $\tilde{s}$ by adding p to $M(s)$
9: **call** VF2($\tilde{s}$, PT)

The initial state s_0 is associated with $M(s_0) = \emptyset$, i.e. it starts with an empty mapping.

For each state s, the algorithm computes $P(s)$, the set of candidate node pairs for adding to the current state s.

For each pair p in $P(s)$, $Cons_{PT}(p, M(s))$ and $Cut_{PT}(p, M(s))$ are evaluated. If $Cons_{PT}(p, M(s))$ is true and $Cut_{PT}(p, M(s))$ is false, the successor state $\tilde{s} = s \cup \{p\}$ is computed, and the whole process is recursively applied to $\tilde{s}$. Otherwise, $\tilde{s}$ is not consistent by PT or it can be proved that s can not be extended to a whole mapping.

In order to make sure of the correctness see

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

Note that a state may be reached in many different ways, since the order of insertions into M does not influence the nascent mapping. In fact, the number of different ways which lead to the same state can be exponentially large. If G_{small} and G_{large} are circles with n nodes and n different node labels, there exists exactly one graph isomorphism between them, but it will be reached in $n!$ different ways.

However, one may observe

Claim 3.2.2. *Let $\prec$ an arbitrary total ordering relation on V_{small}. If the algorithm ignores each $p = (u, v) \in P(s)$, for which*

$$\exists(\hat{u}, \hat{v}) \in P(s) : \hat{u} \prec u,$$

then no state can be reached more than ones and each state associated with a whole mapping remains reachable.

Note that the cornerstone of the improvements to VF2 is a proper choice of a total ordering.

3.3 The candidate set P(s)

$P(s)$ is the set of the candidate pairs for inclusion in $M(s)$. Suppose that PT is an expanding problem type, see **Definition 3.1.5)**.

Notation 3.3.1. *Let* $\mathbf{T_{small}(s)} := \{u \in V_{small} : u \text{ is not covered by } M(s) \wedge \exists\tilde{u} \in V_{small} : (u, \tilde{u}) \in E_{small} \wedge \tilde{u} \text{ is covered by } M(s)\}$, *and*
$\mathbf{T_{large}(s)} := \{v \in V_{large} : v \text{ is not covered by } M(s) \wedge \exists\tilde{v} \in V_{large} : (v, \tilde{v}) \in E_{large} \wedge \tilde{v} \text{ is covered by } M(s)\}$

The set $P(s)$ includes the pairs of uncovered neighbours of covered nodes and if there is not such a node pair, all the pairs containing two uncovered nodes are added. Formally, let

$$P(s) = \begin{cases} T_{small}(s) \times T_{large}(s) & \text{if } T_{small}(s) \neq \emptyset \wedge T_{large}(s) \neq \emptyset, \\ (V_{small} \setminus M_{small}(s)) \times (V_{large} \setminus M_{large}(s)) & \textit{otherwise.} \end{cases}$$

3.4 Consistency

This section defines the consistency functions for the different problem types mentioned in **Section 2.2)**.

Notation 3.4.1. *Let* $\mathbf{\Gamma_{small}(u)} := \{\tilde{u} \in V_{small} : (u, \tilde{u}) \in E_{small}\}$
Let $\mathbf{\Gamma_{large}(v)} := \{\tilde{v} \in V_{large} : (v, \tilde{v}) \in E_{large}\}$

Suppose $p = (u, v)$, where $u \in V_{small}$ and $v \in V_{large}$, s is a state of the matching procedure, $M(s)$ is consistent mapping by PT and $lab(u) = lab(v)$. $Cons_{PT}(p, M(s))$ checks whether including pair p into $M(s)$ leads to a consistent mapping by PT.

3.4.1 Induced subgraph isomorphism

$M(s) \cup \{(u,v)\}$ is a consistent mapping by $IND \Leftrightarrow (\forall \tilde{u} \in M_{small} : (u,\tilde{u}) \in E_{small} \Leftrightarrow (v, Pair(M(s),\tilde{u})) \in E_{large})$.
The following formulation gives an efficient way of calculating $Cons_{IND}$.

Claim 3.4.1. *$Cons_{IND}((u,v), M(s)) := (\forall \tilde{v} \in \Gamma_{large}(v) \ \cap \ M_{large}(s) : (Pair(M(s),\tilde{v}), u) \in E_{small}) \wedge (\forall \tilde{u} \in \Gamma_{small}(u) \cap M_{small}(s) : (v, Pair(M(s),\tilde{u})) \in E_{large})$ is a consistency function in the case of IND.*

3.4.2 Graph isomorphism

$M(s) \cup \{(u,v)\}$ is a consistent mapping by $ISO \Leftrightarrow M(s) \cup \{(u,v)\}$ is a consistent mapping by IND.

Claim 3.4.2. *$Cons_{ISO}((u,v), M(s))$ is a consistency function by ISO if and only if it is a consistency function by IND.*

3.4.3 Subgraph isomorphism

$M(s) \cup \{(u,v)\}$ is a consistent mapping by $SUB \Leftrightarrow (\forall \tilde{u} \in M_{small} : (u,\tilde{u}) \in E_{small} \Rightarrow (v, Pair(M(s),\tilde{u})) \in E_{large})$.
The following formulation gives an efficient way of calculating $Cons_{SUB}$.

Claim 3.4.3. *$Cons_{SUB}((u,v), M(s)) := (\forall \tilde{u} \in \Gamma_{small}(u) \ \cap \ M_{small}(s) : (v, Pair(M(s),\tilde{u})) \in E_{large})$ is a consistency function by SUB.*

3.5 Cutting rules

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

Notation 3.5.1. *Let $\tilde{\mathbf{T}}_{\mathbf{small}}(s) := (V_{small} \backslash M_{small}(s)) \backslash T_{small}(s)$, and $\tilde{\mathbf{T}}_{\mathbf{large}}(s) := (V_{large} \backslash M_{large}(s)) \backslash T_{large}(s)$.*

3.5.1 Induced subgraph isomorphism

Claim 3.5.1. *$Cut_{IND}((u,v), M(s)) := |\Gamma_{large}(v) \ \cap \ T_{large}(s)| < |\Gamma_{small}(u) \ \cap \ T_{small}(s)| \vee |\Gamma_{large}(v) \cap \tilde{T}_{large}(s)| < |\Gamma_{small}(u) \cap \tilde{T}_{small}(s)|$ is a cutting function by IND.*

Figure 1: Graphs for the proof of **Claim 3.5.4**

3.5.2 Graph isomorphism

Note that the cutting function of induced subgraph isomorphism defined above is a cutting function by ISO, too, however it is less efficient than the following while their computational complexity is the same.

Claim 3.5.2. *$Cut_{ISO}((u,v),M(s)) := |\Gamma_{large}(v) \cap T_{large}(s)| \neq |\Gamma_{small}(u) \cap T_{small}(s)| \vee |\Gamma_{large}(v) \cap \tilde{T}_{large}(s)| \neq |\Gamma_{small}(u) \cap \tilde{T}_{small}(s)|$ is a cutting function by ISO.*

3.5.3 Subgraph isomorphism

Claim 3.5.3. *$Cut_{SUB}((u,v),M(s)) := |\Gamma_{large}(v) \cap T_{large}(s)| < |\Gamma_{small}(u) \cap T_{small}(s)|$ is a cutting function by SUB.*

Note that there is a significant difference between induced and non-induced subgraph isomorphism:

Claim 3.5.4. *$Cut'_{SUB}((u,v),M(s)) := |\Gamma_{large}(v) \cap T_{large}(s)| < |\Gamma_{small}(u) \cap T_{small}(s)| \vee |\Gamma_{large}(v) \cap \tilde{T}_{large}(s)| < |\Gamma_{small}(u) \cap \tilde{T}_{small}(s)|$ is **not** a cutting function by SUB.*

Proof:

Let the two graphs of **Figure 1)** be the input graphs. Suppose the total ordering relation is $u_1 \prec u_2 \prec u_3 \prec u_4$,$M(s) = \{(u_1,v_1)\}$, and VF2 tries to add $(u_2,v_2) \in P(s)$.
$Cons_{SUB}((u_2,v_2),M(s)) = true$, so $M \cup \{(u_2,v_2)\}$ is consistent by SUB. The cutting function $Cut_{SUB}((u_2,v_2),M(s))$ is false, so it does not let cut

the tree.
On the other hand $Cut'_{SUB}((u_2, v_2), M(s))$ is true, since
$0 = |\Gamma_{large}(v_2) \cap \tilde{T}_{large}(s)| < |\Gamma_{small}(u_2) \cap \tilde{T}_{small}(s)| = 1$ is true, but still the tree can not be pruned, because otherwise the $\{(u_1, v_1)(u_2, v_2)(u_3, v_3)(u_4, v_4)\}$ mapping can not be found. ■

4 The VF2++ Algorithm

Although any total ordering relation 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 VF2++' superiority over VF2 is twofold. Firstly, taking into account the structure and the node labeling of the graph, VF2++ determines a state 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, specialized versions for induced subgraph isomorphism and for graph isomorphism have been designed. VF2++ has gained a runtime improvement of one order of magnitude respecting induced subgraph isomorphism and a better asymptotical behaviour in the case of graph isomorphism problem.

Note that a weaker version of the cutting rules and the more efficient candidate set calculating were described in [21], too.

It should be noted that all the methods described in this section are extendable to handle directed graphs and edge labels as well.

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

4.1 Preparations

Claim 4.1.1. *The total ordering relation uniquely determines a node order, in which the nodes of V_{small} will be covered by VF2. From the point of view of the matching procedure, this means, that always the same node of G_{small} will be covered on the d-th level.*

Proof: In order to make the search space a tree, the pairs in $\{(u, v) \in P(s) : \exists \hat{u} : \hat{u} \prec u\}$ are excluded from $P(s)$.
Let $\tilde{P}(s) := P(s) \backslash \{(u, v) \in P(s) : \exists \hat{u} : \hat{u} \prec u\}$
The relation $\prec$ is a total ordering, so $\exists!\ \tilde{u} : \forall\ (u, v) \in \tilde{P}(s) : u = \tilde{u}$. Since a pair form $\tilde{P}(s)$ is chosen for including into $M(s)$, it is obvious, that only $\tilde{u}$ can be covered in V_{small}. Actually, $\tilde{u}$ is the smallest element in $T_{small}(s)$ (or in $V_{small} \backslash M_{small}(s)$, if $T_{small}(s)$ were empty), and $T_{small}(s)$ depends only on the covered nodes of G_{small}.

Simple induction on d shows that the set of covered nodes of G_{small} is unique, if d is given, so $\tilde{u}$ is unique if d is given. ■

Definition 4.1.1. *An order* $(u_{\sigma(1)}, u_{\sigma(2)}, .., u_{\sigma(|V_{small}|)})$ *of* V_{small} *is* ***matching order****, if exists* $\prec$ *total ordering relation, s.t. the VF2 with* $\prec$ *on the d-th level finds pair for* $u_{\sigma(d)}$ *for all* $d \in \{1, .., |V_{small}|\}$*.*

Claim 4.1.2. *A total ordering is matching order, iff the nodes of every component form an interval in the node sequence, and every node connects to a previous node in its component except the first node of the component. The order of the components is arbitrary.*
Formally spoken, an order $(u_{\sigma(1)}, u_{\sigma(2)}, .., u_{\sigma(|V_{small}|)})$ *of* V_{small} *is matching order* $\Leftrightarrow \forall G'_{small} = (V'_{small}, E'_{small})$ *component of* $G_{small} : \forall i : (\exists j : j < i \wedge u_{\sigma(j)}, u_{\sigma(i)} \in V'_{small}) \Rightarrow \exists k : k < i \wedge (\forall l : k \leq l \leq i \Rightarrow u_l \in V'_{small}) \wedge (u_{\sigma(k)}, u_{\sigma(i)}) \in E'_{small}$*, where* $i, j, k, l \in \{1, .., |V_{small}|\}$

Proof: Suppose a matching order is given. It has to be shown that the node sequence has a structure described above.
Let $G'_{small} = (V'_{small}, E'_{small})$ be an arbitrary component and i an arbitrary index.
$(\exists j : j < i \wedge u_{\sigma(j)}, u_{\sigma(i)} \in V'_{small}) \Rightarrow u_{\sigma(i)}$ is not the first covered node in $G_{small} \Rightarrow u_{\sigma(i)}$ is connected to a covered node $u_{\sigma(k)}$ where $k < i$, since $u_{\sigma(i)} \in T_{small}(s)$ for some s and $T_{small}(s) \subseteq V'_{small}$ contains only nodes connected to at least one covered node. It's easy to see, that $\forall l : k \leq l \leq i \Rightarrow u_l \in V'_{small}$, since $T_{small}(s)$ contains only nodes connected to some covered ones, while it is not empty, but if it were empty, then $u_{\sigma(k)}$ and $u_{\sigma(i)}$ were not in the same component.

Now, let us show that if a node sequence has the special structure described above, it has to be matching order.
$(u_{\sigma(1)}, u_{\sigma(2)}, .., u_{\sigma(|V_{small}|)})$ is a total ordering, which determines the matching order $(u_{\sigma(1)}, u_{\sigma(2)}, .., u_{\sigma(|V_{small}|)})$. ■

To summing up, a total ordering always uniquely determines a matching order, and every matching order can be determined by a total ordering, however, more than one different total orderings may determine the same matching order.

4.2 Idea behind the algorithm

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 4.2.1. *Let* $\mathbf{Conn_H(u)} := |\Gamma_{small}(u) \cap H\}|$, *that is the number of neighbours of* u *which are in* H, *where* $u \in V_{small}$ *and* $H \subseteq V_{small}$.

The principal question is the following. Suppose a state s is given. For which node of $T_{small}(s)$ is the hardest to find a consistent pair in G_{large}? The more covered neighbours a node in $T_{small}(s)$ has — i.e. the largest $Conn_{M_{small}(s)}$ it has —, the more rarely satisfiable consistency constraints for its pair are given.

In biology, most of the graphs are sparse, thus several nodes in $T_{small}(s)$ may have the same $Conn_{M_{small}(s)}$, 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 neighbours among which have the same $Conn_{M_{small}(s)}$ to increase $Conn_{M_{small}(s)}$ of uncovered nodes so much, as possible. The tertiary ordering prefers nodes having the rarest uncovered labels.

Note that the secondary ordering is the same as the ordering by deg, 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 easily matchable into G_{large}. To avoid that, a BFS order is used, which provides the shortest possible paths.

In the following, some examples on which the VF2 may be slow are described, although they are easily solvable by using a proper matching order.

Example 4.2.1. *Suppose* G_{small} *can be mapped into* G_{large} *in many ways without node labels. Let* $u \in V_{small}$ *and* $v \in V_{large}$.
$lab(u) := black$
$lab(v) := black$
$lab(\tilde{u}) := red\ \forall \tilde{u} \in (V_{small} \backslash \{u\})$
$lab(\tilde{v}) := red\ \forall \tilde{v} \in (V_{large} \backslash \{v\})$

Now, any mapping by the node label lab must contain (u, v), *since* u *is black and no node in* V_{large} *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 can not correspond.

Example 4.2.2. *Suppose there is no node label given, G_{small} is a small graph and can not be mapped into G_{large} and $u \in V_{small}$.*
Let $G'_{small} := (V_{small} \cup \{u'_1, u'_2, .., u'_k\}, E_{small} \cup \{(u, u'_1), (u'_1, u'_2), .., (u'_{k-1}, u'_k)\})$, that is, G'_{small} is $G_{small} \cup \{$a k long path, which is disjoint from G_{small} and one of its starting points is connected to $u \in V_{small}\}$.
Is there a subgraph of G_{large}, which is isomorph with G'_{small}?
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_{large}, and it would recognize that no path can be extended to G'_{small}.
However, had it started by the matching of G_{small}, it would not have matched any nodes of the path.

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

4.3 Total ordering

Instead of the total ordering relation, the matching order will be searched directly.

Notation 4.3.1. *Let* $\boldsymbol{F_\mathcal{M}(l)} := |\{v \in V_{large} : l = lab(v)\}| - |\{u \in V_{small} \backslash \mathcal{M} : l = lab(u)\}|$ *, where l is a label and* $\mathcal{M} \subseteq V_{small}$.

Definition 4.3.1. *Let* $\mathbf{arg\ max}_f(S) := \{u : u \in S \wedge f(u) = max_{v \in S}\{f(v)\}\}$ *and* $\mathbf{arg\ min}_f(S) := arg\ max_{-f}(S)$, *where S is a finite set and* $f : S \longrightarrow \mathbb{R}$.

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 *lab* and largest *deg*, whose root vertex is the component's minimal node. **Algorithm 3)** and **4)** are two different methods to process a level of the BFS tree.

After sorting the nodes of the current level in descending lexicographic order by $(Conn_\mathcal{M}, deg, -F_\mathcal{M})$, **Algorithm 4)** appends them simultaneously to the matching order $\mathcal{M}$ and refreshes $F_\mathcal{M}$ just once, whereas **Algorithm**

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

```
1: procedure VF2++ORDER
2:     M := ∅                                              ▷ matching order
3:     while V_small \ M ≠ ∅ do
4:         r ∈ arg max_deg (arg min_{F_M ∘ lab}(V_small \ M))
5:         Compute T, a BFS tree with root node r.
6:         for d = 0, 1, ..., depth(T) do
7:             V_d := nodes of the d-th level
8:             Process V_d                     ▷ See Algorithm 3) and 4)
```

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

```
1: procedure VF2++PROCESSLEVEL1(V_d)
2:     while V_d ≠ ∅ do
3:         m ∈ arg min_{F_M ∘ lab}( arg max_deg(arg max_{Conn_M}(V_d)))
4:         V_d := V_d \ m
5:         Append node m to the end of M
6:         Refresh F_M
```

Algorithm 4 *Another method for processing a level of the BFS tree*

```
1: procedure VF2++PROCESSLEVEL2(V_d)
2:     Sort V_d in descending lex. order by (Conn_M, deg, −F_M)
3:     Append the sorted V_d to the end of M
4:     Refresh F_M
```

3) appends the nodes separately to $\mathcal{M}$ and refreshes $F_{\mathcal{M}}$ immediately, so it provides up-to-date label information and may result in a more efficient matching order.

Claim 4.1.2) shows that **Algorithm 2)** provides a matching order.

4.4 Cutting rules

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

Notation 4.4.1. *Let* $\mathbf{\Gamma^l_{small}(u)} := \{\tilde{u} : lab(\tilde{u}) = l \wedge \tilde{u} \in \Gamma_{small}(u)\}$ *and* $\mathbf{\Gamma^l_{large}(v)} := \{\tilde{v} : lab(\tilde{v}) = l \wedge \tilde{v} \in \Gamma_{large}(v)\}$*, where* $u \in V_{small}$*,* $v \in V_{large}$ *and* l *is a label.*

4.4.1 Induced subgraph isomorphism

Claim 4.4.1.

$$LabCut_{IND}((u,v), M(s)) := \bigvee_{l \text{ is label}} |\Gamma^l_{large}(v) \cap T_{large}(s)| < |\Gamma^l_{small}(u) \cap T_{small}(s)| \vee$$

$$\bigvee_{l \text{ is label}} |\Gamma^l_{large}(v) \cap \tilde{T}_{large}(s)| < |\Gamma^l_{small}(u) \cap \tilde{T}_{small}(s)|$$

is a cutting function by IND.

Proof: It has to be shown, that $LabCut_{IND}((u,v), M(s)) = true \Rightarrow$ the mapping can not be extended to a whole mapping.
$LabCut_{IND}((u,v), M(s)) = true$, iff the following holds.
$\exists l : |\Gamma^l_{large}(v) \cap T_{large}(s)| < |\Gamma^l_{small}(u) \cap T_{small}(s)| \vee |\Gamma^l_{large}(v) \cap \tilde{T}_{large}(s)| < |\Gamma^l_{small}(u) \cap \tilde{T}_{small}(s)|$.

Suppose that $|\Gamma^l_{large}(v) \cap T_{large}(s)| < |\Gamma^l_{small}(u) \cap T_{small}(s)|$. Each node of $\Gamma^l_{small}(u) \cap T_{small}(s)$ has to be matched to a node in $\Gamma^l_{large}(v) \cap T_{large}(s)$, so $\Gamma^l_{large}(v) \cap T_{large}(s)$ can not be smaller than $\Gamma^l_{small}(u) \cap T_{small}(s)$. That is why $M(s)$ can not be extended to a whole mapping.

Otherwise $|\Gamma^l_{large}(v) \cap \tilde{T}_{large}(s)| < |\Gamma^l_{small}(u) \cap \tilde{T}_{small}(s)|$ has to be true. Similarly, each node of $\Gamma^l_{large}(v) \cap \tilde{T}_{large}(s)$ has to be matched to a node in $\Gamma^l_{small}(u) \cap \tilde{T}_{small}(s)$, i.e. $\Gamma^l_{large}(v) \cap \tilde{T}_{large}(s)$ can not be smaller than $\Gamma^l_{small}(u) \cap \tilde{T}_{small}(s)$, if $M(s)$ is extendible. ■

The following claims can be proven similarly.

4.4.2 Graph isomorphism

Claim 4.4.2.

$$LabCut_{ISO}((u,v),M(s)) := \bigvee_{l \text{ is label}} |\Gamma^l_{large}(v) \cap T_{large}(s)| \neq |\Gamma^l_{small}(u) \cap T_{small}(s)| \vee$$

$$\bigvee_{l \text{ is label}} |\Gamma^l_{large}(v) \cap \tilde{T}_{large}(s)| \neq |\Gamma^l_{small}(u) \cap \tilde{T}_{small}(s)|$$

is a cutting function by ISO.

4.4.3 Subgraph isomorphism

Claim 4.4.3.

$$LabCut_{IND}((u,v),M(s)) := \bigvee_{l \text{ is label}} |\Gamma^l_{large}(v) \cap T_{large}(s)| < |\Gamma^l_{small}(u) \cap T_{small}(s)|$$

is a cutting function by SUB.

4.5 Implementation details

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

4.5.1 Storing a mapping

After fixing an arbitrary node order $(u_0, u_1, .., u_{|G_{small}|-1})$ of G_{small}, an array M is usable to store the current mapping in the following way.

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

Where $i \in \{0,1,..,|G_{small}|-1\}$, $v \in V_{large}$ and $INVALID$ means "no node".

4.5.2 Avoiding the recurrence

Exploring the state space was described in a recursive fashion using sets (see **Algorithm 1)**), which makes the algorithm easy to understand but does not show directly an efficient way of implementation. The following

approach avoiding recursion and using lookup tables instead of sets is not only fast but has linear space complexity as well.

The recursion of **Algorithm 1)** can be realized as a while loop, which has a loop counter *depth* denoting the all-time depth of the recursion. Fixing a matching order, let M denote the array storing the all-time mapping. The initial state is associated with the empty mapping, which means that $\forall i : M[i] = INVALID$ and $depth = 0$. In case of a recursive call, *depth* has to be incremented, while in case of a return, it has to be decremented. Based on **Claim 4.1.1)**, M is $INVALID$ from index $depth{+}1$ and not $INVALID$ before *depth*, i.e. $\forall i : i < depth \Rightarrow M[i] \neq INVALID$ and $\forall i : i > depth \Rightarrow M[i] = INVALID$. $M[depth]$ 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 easily maintainable or computable by following **Section 3.3)**. A much faster method has been designed for biological- and sparse graphs, see the next section for details.

4.5.3 Calculating the candidates for a node

Being aware of **Claim 4.1.1)**, the task is not to maintain the candidate set, but to generate the candidate nodes in G_{large} for a given node $u \in V_{small}$. In case of an expanding problem type and M mapping, if a node $v \in V_{large}$ is a potential pair of $u \in V_{small}$, then $\forall u' \in V_{small} : (u, u') \in E_{small}$ *and* u' *is covered by* $M \Rightarrow (v, Pair(M, u')) \in E_{large}$. That is, each covered neighbour of u has to be mapped to a covered neighbour of v.

Having said that, an algorithm running in $\Theta(deg)$ time is describable if there exists a covered node in the component containing u. In this case choose a covered neighbour u' of u arbitrarily — such a node exists based on **Claim 4.1.2)**. With all the candidates of u being among the uncovered neighbours of $Pair(M, u')$, there are solely $deg(Pair(M, u'))$ nodes to check.

An easy trick is to choose an u', for which $|\{uncovered\ neighbours\ of\ Pair(M, u')\}|$ is the smallest possible.

Note that if u is the first node of its component, then all the uncovered nodes of G_{large} are candidates, so giving a sublinear method is impossible.

4.5.4 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 $\{0, 1, .., |K|-1\}$, where K is the set of the labels. It enables $F_{\mathcal{M}}$ to be stored in an array, for which $F_{\mathcal{M}}[i] = F_{\mathcal{M}}(i)$ where $i = 0, 1, .., |K| - 1$. At first, $\mathcal{M} = \emptyset$, so $F_{\mathcal{M}}[i]$ is the number of nodes in V_{small} having label i, which is easy to compute in $\Theta(|V_{small}|)$ steps.

$\mathcal{M} \subseteq V_{small}$ can be represented as an array of size $|V_{small}|$.

The BFS tree is computed by using a FIFO data structure which is usually implemented as a linked list, but one can avoid it by using the array $\mathcal{M}$ itself. $\mathcal{M}$ contains all the nodes seen before, a pointer shows where the first node of the FIFO is, and another one shows where the next explored node has to be inserted. So the nodes of each level of the BFS tree can be processed by **Algorithm 3)** and **4)** in place by swapping nodes.

After a node u gets to the next place of the node order, $F_{\mathcal{M}}[lab[u]]$ has to be decreased by one, because there is one less covered node in V_{large} with label $lab(u)$, that is why min selection sort is preferred which gives the elements from left to right in descending order, see **Algorithm 3)**.

Note that using a $\Theta(n^2)$ sort absolutely does not slow down the procedure on biological (and on sparse) graphs, since they have few nodes on a level. If a level had a large number of nodes, **Algorithm 4)** would seem to be a better choice with a $\Theta(nlog(n))$ or Bucket sort, but it may reduce the efficiency of the matching procedure, since $F_{\mathcal{M}}(i)$ can not be immediately refreshed, so it is unable to provide up-to-date label information.

Note that the *while loop* of **Algorithm 2)** takes one iteration per graph component and the graphs in biology are mostly connected.

4.5.5 Cutting rules

In **Section 4.4)**, the cutting rules were described using the sets T_{small}, T_{large}, $\tilde{T}_{small}$ and $\tilde{T}_{large}$, which are dependent on the all-time mapping (i.e. on the all-time state). The aim is to check the labeled cutting rules of VF2++ in $\Theta(deg)$ time.

Firstly, suppose that these four sets are given in such 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 $L[lab(u')]$ for all $u' \in \Gamma_{small}(u) \cap T_{small}(s)$ and decrementing $L[lab(v')]$ for all $v' \in \Gamma_{large}(v) \cap T_{large}(s)$, the first part of the cutting rules is checkable in $\Theta(deg)$ time by considering the proper signs of L. Setting L to zero takes $\Theta(deg)$ 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 $\tilde{T}_{small}$ and $\tilde{T}_{large}$ instead of T_{small} and T_{large}. Thus, the overall complexity is $\Theta(deg)$.

An other integer lookup table storing the number of covered neighbours of each node in G_{large} gives all the information about the sets T_{large} and $\tilde{T}_{large}$, which is maintainable in $\Theta(deg)$ time when a pair is added or substracted by incrementing or decrementing the proper indices. A further improvement is that the values of $L[lab(u')]$ in case of checking u is dependent only on u, i.e. on the size of the mapping, so for each $u \in V_{small}$ an array of pairs (label, number of such labels) can be stored to skip the maintaining operations. Note that these arrays are at most of size deg. Skipping this trick, the number of covered neighbours has to be stored for each node of G_{small} as well to get the sets T_{small} and $\tilde{T}_{small}$.

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

5 The VF2 Plus Algorithm

The VF2 Plus algorithm is a recently improved version of VF2. It was compared with the state of the art algorithms in [21] and has proven itself to be competitive with RI, the best algorithm on biological graphs.
A short summary of VF2 Plus follows, which uses the notation and the conventions of the original paper.

5.1 Ordering procedure

VF2 Plus uses a sorting procedure that prefers nodes in V_{small} with the lowest probability to find a pair in V_{small} and the highest number of connections with the nodes already sorted by the algorithm.

Definition 5.1.1. *(u, v) is a* ***feasible pair****, if $lab(u) = lab(v)$ and $deg(u) \leq deg(v)$, where $u \in V_{small}$ and $v \in V_{large}$.*

$P_{lab}(L) :=$ a priori probability to find a node with label L in V_{large}
$P_{deg}(d) :=$ a priori probability to find a node with degree d in V_{large}
$P(u) := P_{lab}(L) * \bigcup_{d'>d} P_{deg}(d')$
M is the set of already sorted nodes, T is the set of nodes candidate to be selected, and $degreeM$ of a node is the number of its neighbours in M.

Using these notations, **Algorithm 5)** provides the description of the sorting procedure.

Note that $P(u)$ is not the exact probability of finding a consistent pair for u by choosing a node of V_{large} randomly, since P_{lab} and P_{deg} are not independent, though calculating the real probability would take quadratic time, which may be reduced by using fittingly lookup tables.

Algorithm 5

```
 1: procedure VF2 Plus order
 2:     Select the node with the lowest P.
 3:     if more nodes share the same P then
 4:         select the one with maximum degree
 5:     if more nodes share the same P and have the max degree then
 6:         select the first
 7:     Put the selected node in the set M.
 8:     Put all its unsorted neighbours in the set T.
 9:     if M ≠ V_small then
10:         From set T select the node with maximum degreeM.
11:         if more nodes have maximum degreeM then
12:             Select the one with the lowest P
13:         if more nodes have maximum degreeM and P then
14:             Select the first.
15:         goto 7.
```

6 Experimental results

This section compares the performance of VF2++ and VF2 Plus. Both algorithms have run faster with orders of magnitude than VF2, thus its inclusion was not reasonable.

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[20], which has been constructed of Molecule, Protein and Contact Map graphs extracted from the Protein Data Bank[2].

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, the induced subgraph isomorphism and the graph isomorphism will be examined.

6.1.1 Induced subgraph isomorphism

This dataset contains a set of graph pairs, and **all** the induced subgraph ismorphisms have to be found between them. **Figure 2), 3),** and **4)** show the solution time of the problems in the problem set.

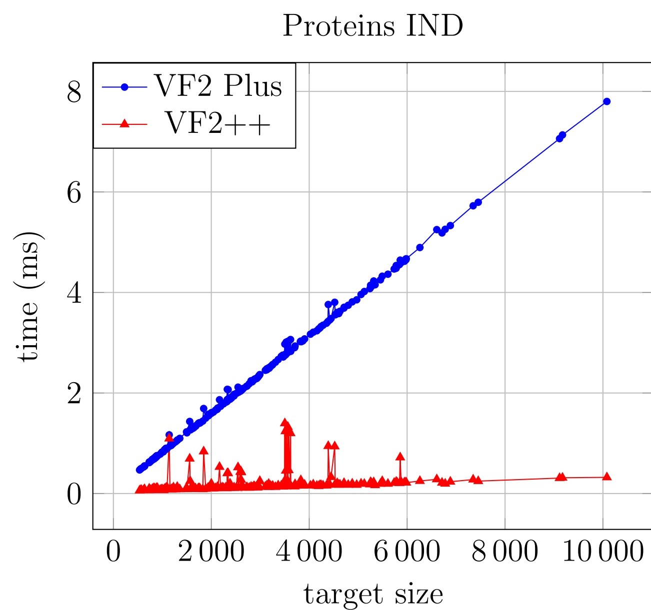

Figure 2: Both the algorithms have linear behaviour on protein graphs. VF2++ is more than 10 times faster than VF2 Plus.

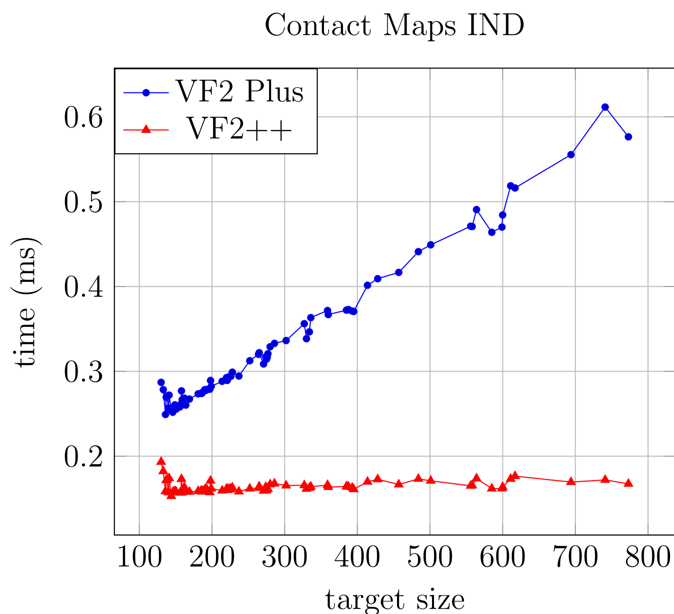

Figure 3: On Contact Maps, VF2++ runs in near constant time, while VF2 Plus has a near linear behaviour.

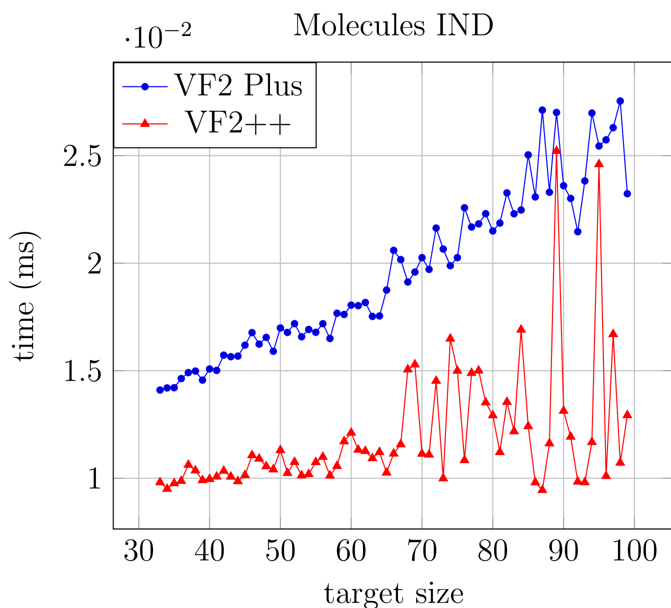

Figure 4: In the case of Molecules, the algorithms seem to have a similar behaviour, but VF2++ is almost two times faster even on such small graphs.

6.1.2 Graph ismorphism

In this experiment, the nodes of each graph in the database have been shuffled and an isomorphism between the shuffled and the original graph has been searched. For runtime results, see **Figure 5), 6),** and **7)**.

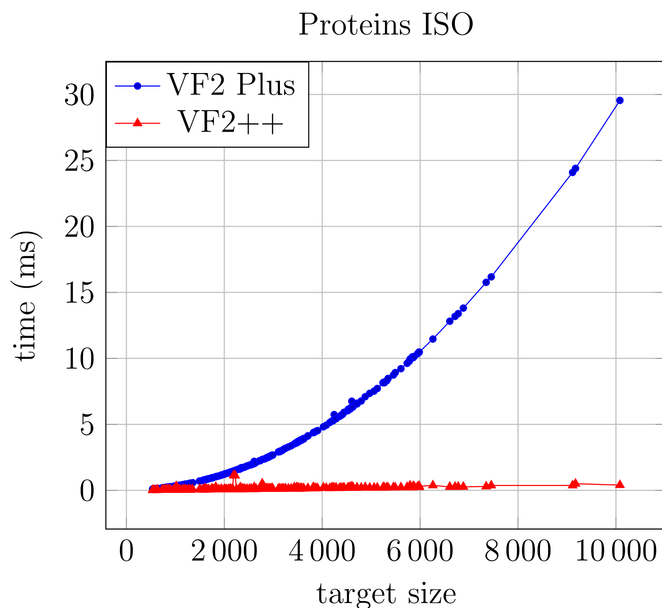

Figure 5: On protein graphs, VF2 Plus has a super linear time complexity, while VF2++ runs in near constant time. The difference is about two order of magnitude on large graphs.

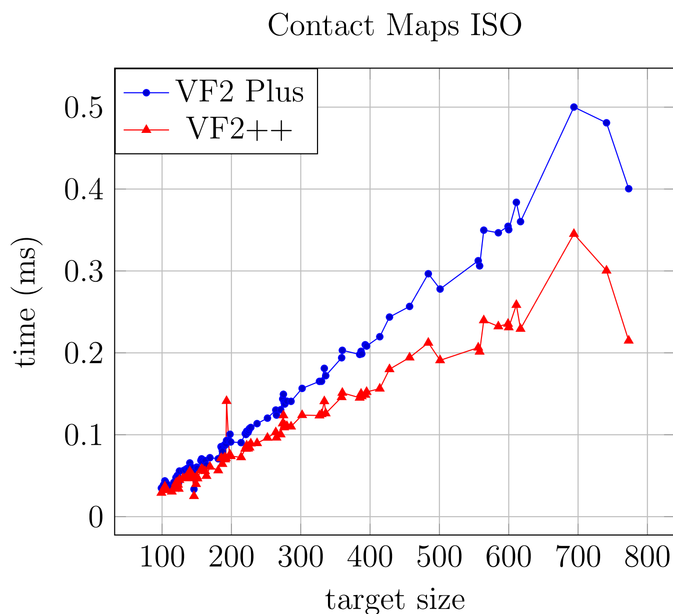

Figure 6: The results are closer to each other on Contact Maps, but VF2++ still performs consistently better.

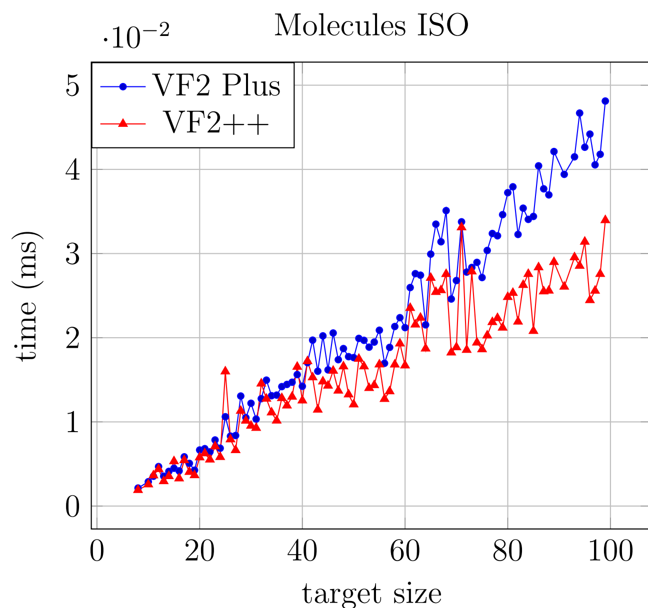

Figure 7: In the case of Molecules, there is not such a significant difference, but VF2++ seems to be faster as the number of nodes increases.

6.2 Random graphs

This section compares VF2++ with VF2 Plus on random graphs of a 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.

6.2.1 Graph isomorphism

To evaluate the efficiency of the algorithms in the case of graph isomorphism, 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. **Figure 8), 9), 10), 11), 12),** and **13)** show the runtime results on graph sets of various density.

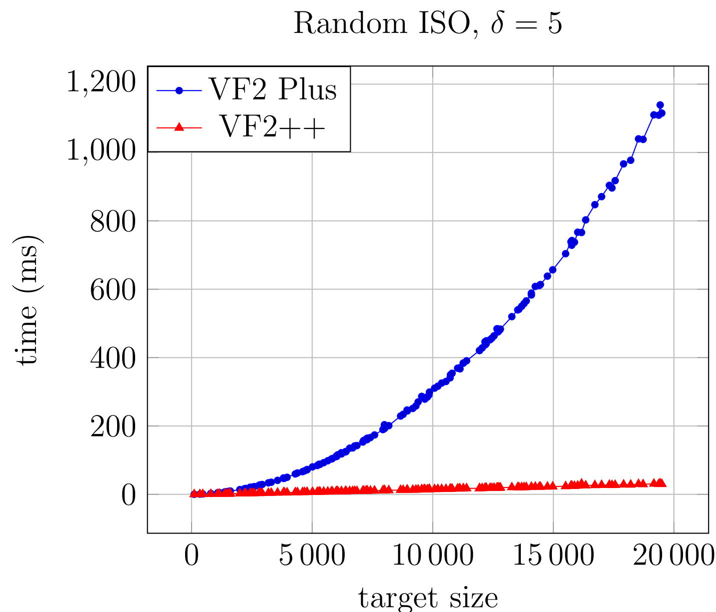

Figure 8

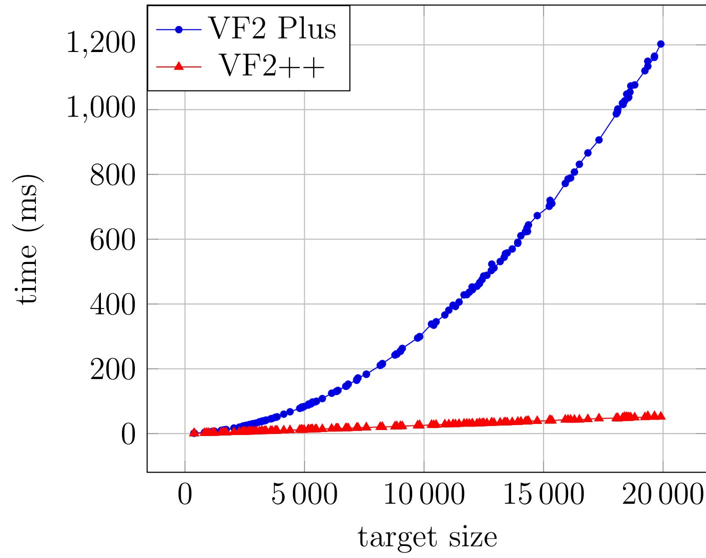

Figure 9

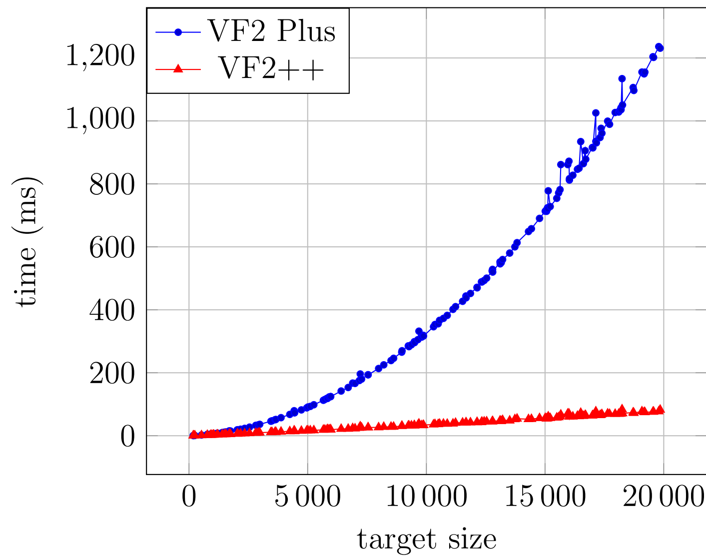

Figure 10

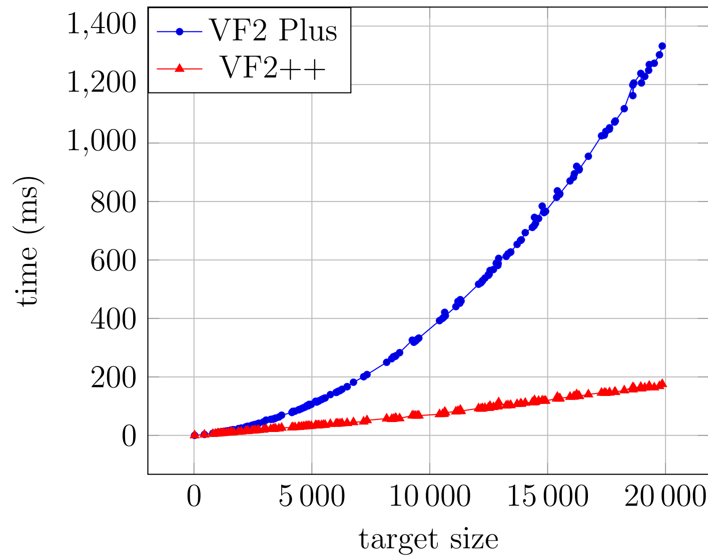

Figure 11

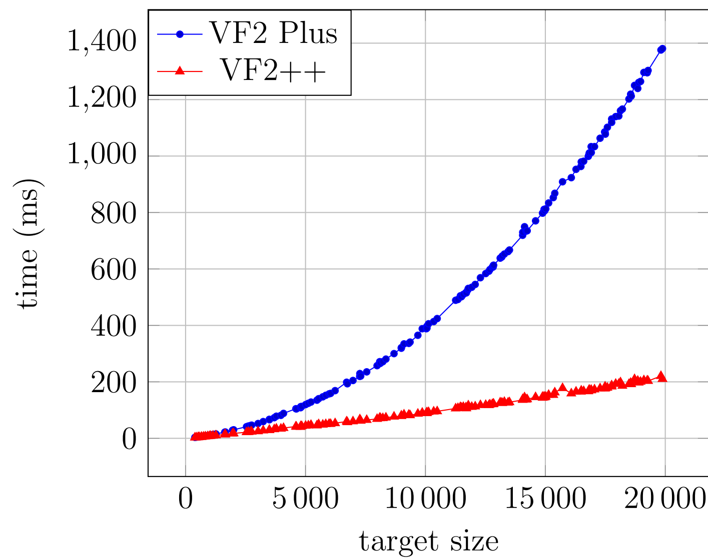

Figure 12

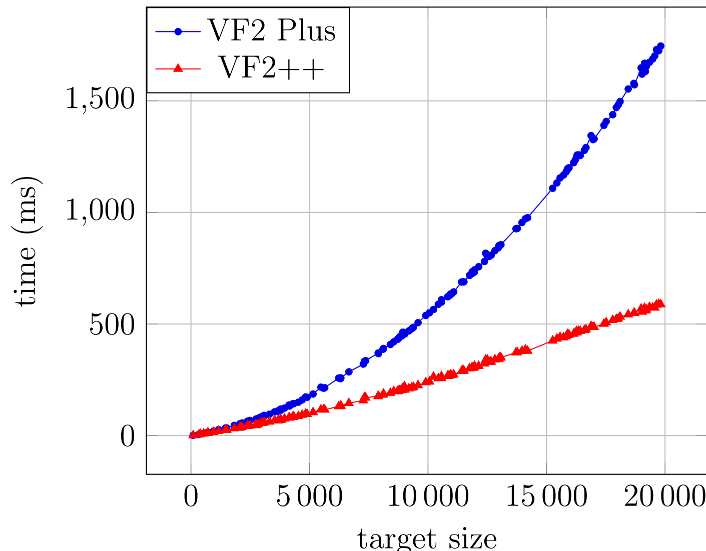

Figure 13

Considering the graph isomorphism problem, VF2++ consistently outperforms its rival especially on sparse graphs. The reason for the slightly super linear behaviour of VF2++ on denser graphs is the larger number of nodes in the BFS tree constructed in **Algorithm 2)**.

6.2.2 Induced subgraph isomorphism

This section provides a comparison of VF2++ and VF2 Plus in the case of induced subgraph isomorphism. In addition to the size of the large graph, that of the small graph dramatically influences the hardness of a given problem too, so the overall picture is provided by examining small graphs 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_{large}, choose 10 of its induced subgraphs having $\rho\,|V_{large}|$ nodes, and for all the 10 subgraphs find a mapping by using both the graph matching algorithms. The $\delta = 5, 10, 35$ and $\rho = 0.05, 0.1, 0.3, 0.6, 0.8, 0.95$ cases have been examined (see **Figure 14), 16)** and **18)**), and for each δ, a cumulative chart is given as well, which excludes $\rho = 0.05$ and 0.1 for the sake of perspicuity (see **Figure 15), 17)** and **19)**).

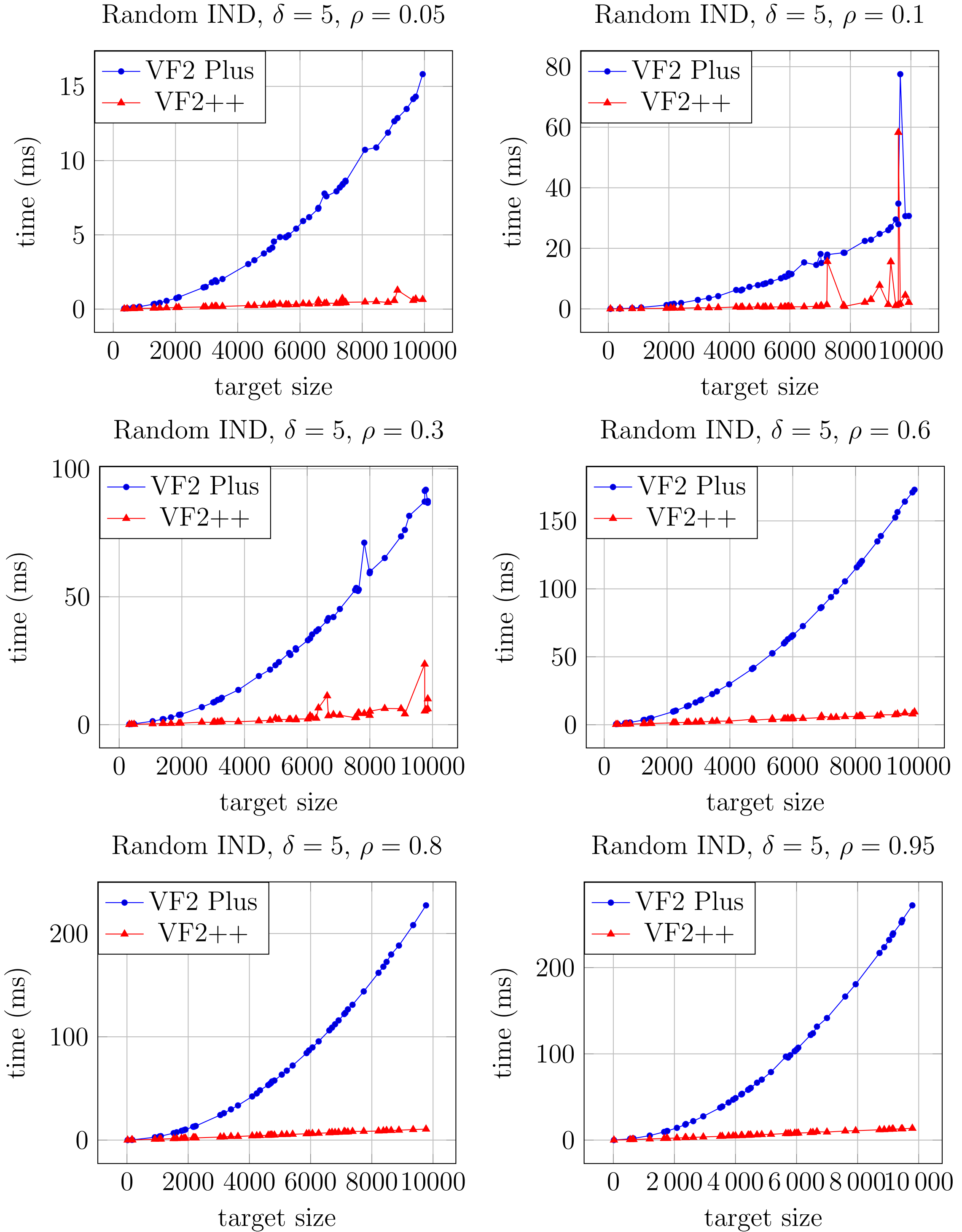

Figure 14: IND on graphs having an average degree of 5.

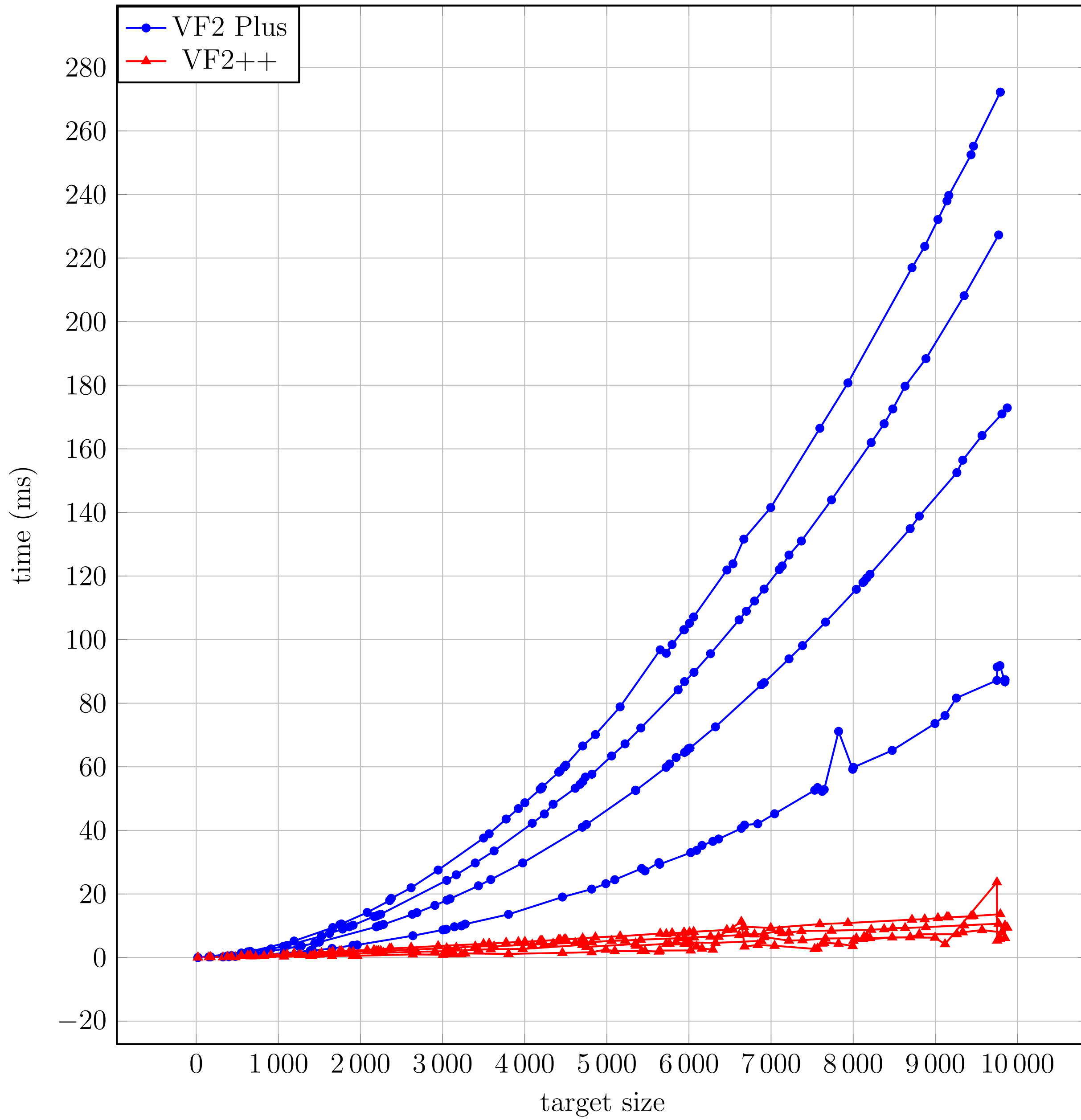

Figure 15: Cummulative chart for $\delta = 5$.

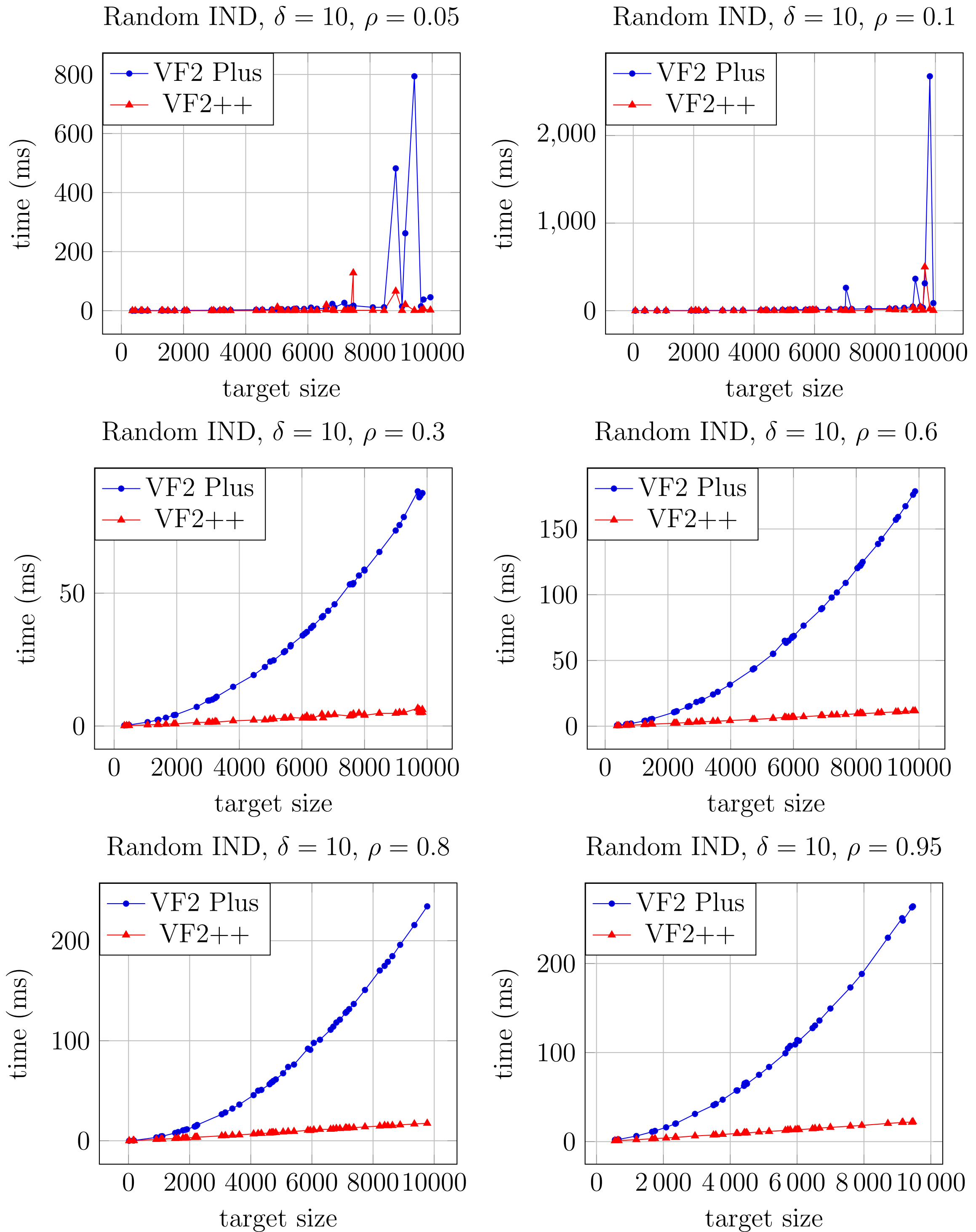

Figure 16: IND on graphs having an average degree of 10.

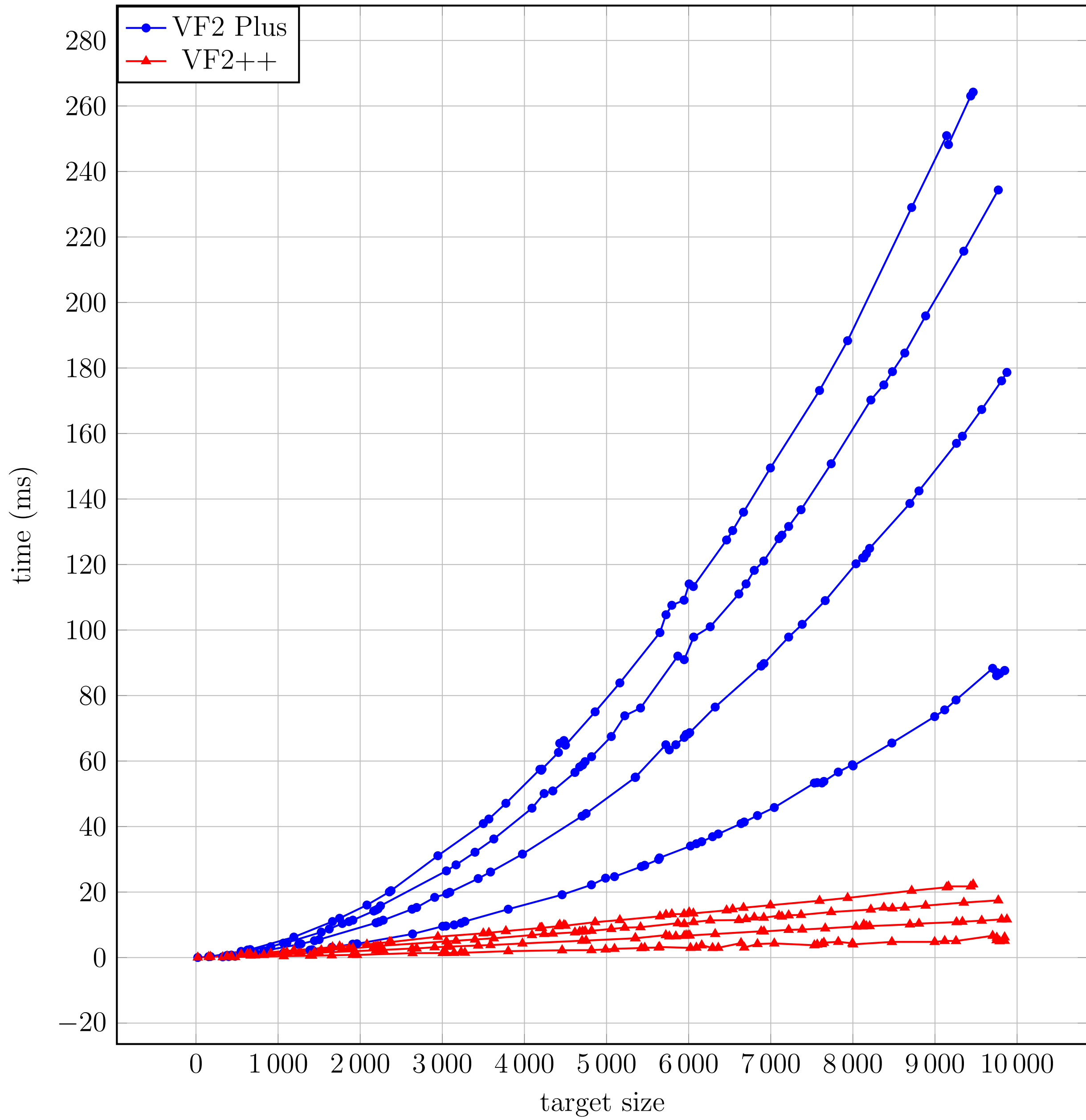

Figure 17: Cummulative chart for $\delta = 10$.

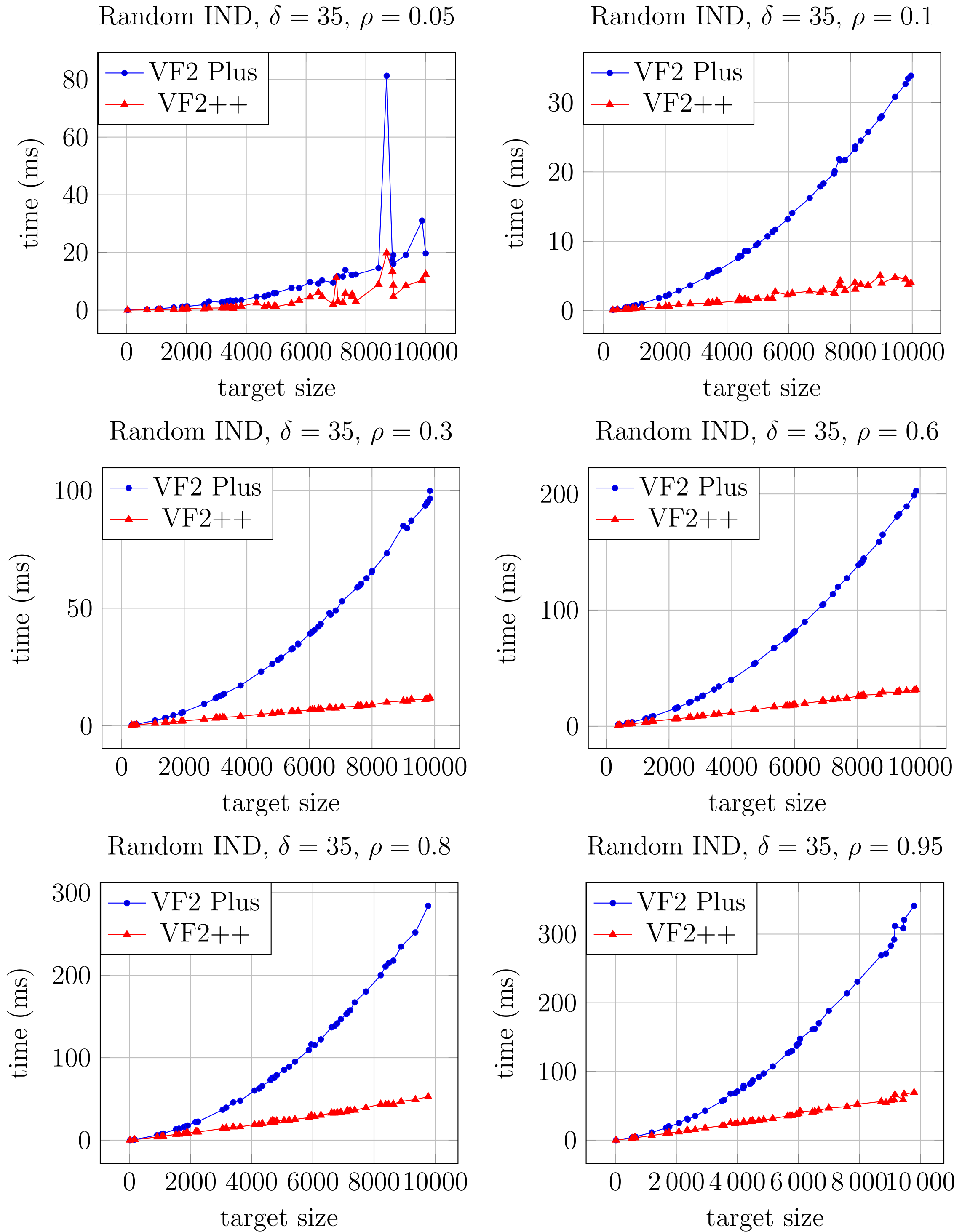

Figure 18: IND on graphs having an average degree of 35.

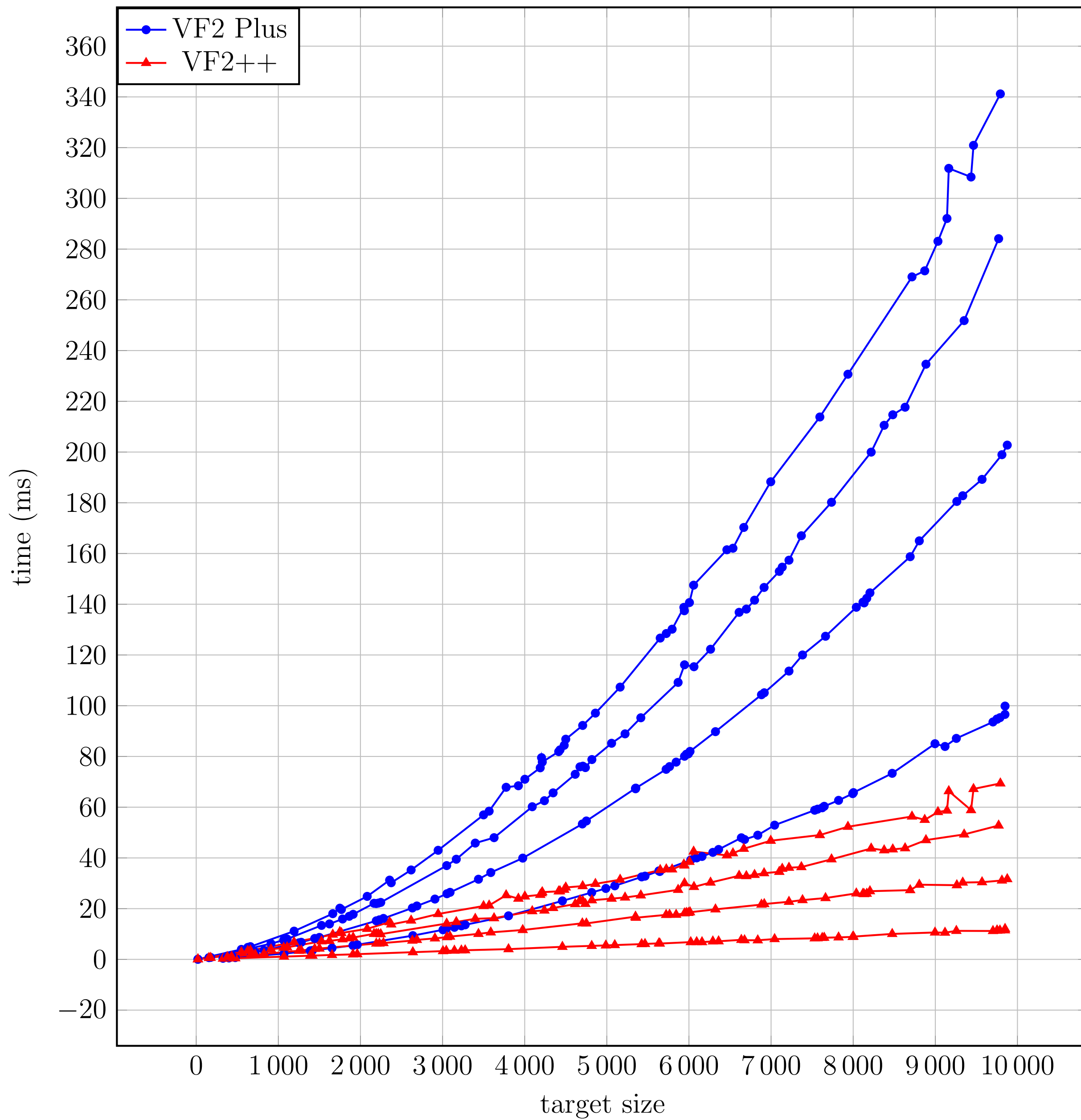

Figure 19: Cummulative chart for $\delta = 35$.

Based on these experiments, VF2++ is faster than VF2 Plus and able

to handle really large graphs in milliseconds. Note that when IND was considered and the small graphs 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 **Figure 16)**). If these examples had been excluded, the charts would have seemed to be similar to the other ones. Unsurprisingly, as denser graphs are considered, both VF2++ and VF2 Plus slow slightly down, but remain practically usable even on graphs having 10 000 nodes.

7 Conclusion

In this thesis, after providing a short summary of the recent algorithms, a new graph matching algorithm based on VF2, called VF2++, has been presented and analyzed from a practical viewpoint.

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 consistency and 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, which is the best concurrent algorithm based on [21].

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, while in the case of graph isomorphism, it has definitely better asymptotic behaviour on protein graphs.

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

References

[1] Alexandru T. Balaban. Applications of graph theory in chemistry. *J. Chem. Inf. Comput. Sci., 1985,25(3),pp 334-343*, March 1985.

[2] Protein Data Bank. http://www.rcsb.org/pdb. June 2015.

[3] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics. 2013; 14(Suppl 7): S13.*, April 2013.

[4] Horst Bunke. Graph matching: Theoretical foundations, algorithms, and applications. *In International Conference on Vision Interface, pp. 82-84*, May 2000.

[5] Charles J. Colbourn. On testing isomorphism of permutation graphs. *Networks, Volume 11, Issue 1, Pages 13-21*, March 1981.

[6] S. A. Cook. The complexity of theorem-proving procedures. *Proc. 3rd ACM Symposium on Theory of Computing, pp. 151-158*, 1971.

[7] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *Journal of the ACM (JACM) JACM Homepage archive Volume 23 Issue 1, Pages 31-42*, 2004.

[8] LEMON: Library for Efficient Modeling and Optimization in Networks. https://lemon.cs.elte.hu. March 2015.

[9] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. *Proceeding STOC '74 Proceedings of the sixth annual ACM symposium on Theory of computing, Pages 172-184*, April 1974.

[10] QuantumBio Inc. http://www.quantumbioinc.com/.

[11] C. Sansone L. P. Cordella, P. Foggia and M. Vento. Performance evaluation of the vf graph matching algorithm. *Proc. of the 10th ICIAP, IEEE Computer Society Press, pp. 1172-1177*, 1999.

[12] Jianzhuang Liu and Yong Tsui Lee. A graph-based method for face identification from a single 2d line drawing. *IEEE Transactions on Pattern Analysis and Machine Intelligence - Graph Algorithms and Computer Vision archive Volume 23 Issue 10*, October 2001.

[13] M. Vento L.P. Cordella. Symbol recognition in documents: a collection of techniques? *International Journal on Document Analysis and Recognition Volume 3, Issue 2, pp 73-88*, December 2000.

[14] George S. Lue and Kellogg S. Booth. A linear time algorithm for deciding interval graph isomorphism. *Journal of the ACM (JACM), Volume 26, Issue 2, Pages 183-195*, April 1979.

[15] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences, Volume 25, Issue 1, Pages 42-65*, August 1982.

[16] Brendan D. McKay. Alldifferent-based filtering for subgraph isomorphism. *Artificial Intelligence 174,850-864*, 2010.

[17] Christine Solnon. Practical graph isomorphism. *Congressus Numerantium, vol.30, pp. 45-87*, 1981.

[18] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM), Volume 23 Issue 1, Pages 31-42*, January 1976.

[19] Julian R. Ullmann. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *Journal of Experimental Algorithmics (JEA),Volume 15, Article No. 1.6, ACM New York, NY, USA*, 2010.

[20] Mario Vento, Xiaoyi Jiang, and Pasquale Foggia. International contest on pattern search in biological databases, http://biograph2014.unisa.it. June 2015.

[21] Carletti Vincenzo, Pasquale Foggia, and Mario Vento. Vf2 plus: An improved version of vf2 for biological graphs. *Conference: Graph-Based Representations in Pattern Recognition, At Beijing*, May 2015.