Exacts and Inexacts methods for graph similarity in Structural Pattern Recognition

Vincenzo Carletti

Normandie Université

THESE

Pour obtenir le diplôme de doctorat

Spécialité Informatique et Applications

Préparée au sein de l'Université de Caen, Normandie

En partenariat international avec l'Université de Salerne, Italie

Exact and Inexact Methods for Graph Similarity in Structural Pattern Recognition

Présentée et soutenue par
Vincenzo CARLETTI

Thèse soutenue publiquement le 21 avril 2016 devant le jury composé de		
Luc BRUN / France	Professeur des Universités / GREYC / ENSICANE	Directeur de thèse
Mario VENTO / Italia	Professeur des Universités / Université de Salerne	Codirecteur de thèse
Antoine TABBONE / France	Professeur des Universités / Université de Nancy	Rapporteur
Francisc SERRATOSA / Espagne	Professeur des Universités/ Universitat Rovira i Virgili	Rapporteur
Carlo SANSONE/ Italia	Professeur des Universités / Université de Naples	Examinateur
Nikolai PETKOV / Pays Bas	Professeur des Universités/ Université de Groningen	Examinateur

Thèse dirigée par Luc BRUN, laboratoire GREYC, et Mario VENTO, Université de Salerne

ED SIMEM

UNICAEN UNIVERSITÉ CAEN NORMANDIE

GREYC

Abstract

Les graphes sont utilisés dans de nombreux domaines applicatifs tels que la biologie, les réseaux sociaux, les bases de données,... Les graphes permettent de décrire un ensemble d'objets ainsi que leurs relations. L'analyse de ces objets réclame souvent de mesurer la similarité entre les graphes. Toutefois, en raison de son aspect combinatoire, ce problème est NP complet et est généralement résolu en utilisant différentes heuristiques.

Dans cette thèse nous avons exploré deux approches pour calculer la similarité entre graphes. La première est basé sur l'appariement exact. Nous avons conçu l'algorithme VF3 qui recherche des motifs dans les graphes. La seconde approche est basée sur un appariement inexact qui calcule une approximation efficace de la distance d'édition entre graphes en la modélisant comme un problème de minimisation quadratique.

Abstract

Graphs are widely employed in many application fields, such as biology, chemistry, social networks, databases and so on. Graphs allow to describe a set of objects together with their relationships. Analysing these data often requires to measure the similarity between two graphs. Unfortunately, due to its combinatorial nature, this is a NP-Complete problem generally addressed using different kind of heuristics.

In this Thesis we have explored two approaches to compute the similarity between graphs. The former is based on the exact graph matching approach. We have designed, VF3, an algorithm aimed to search for pattern structures within graphs. While, the second approach is an inexact graph matching method which aims to compute an efficient approximation of the Graph Edit Distance (GED) as a Quadratic Assignment Problem (QAP).

Rameau index:

1. Graphes, Théorie des
2. Reconnaissance des formes (informatique)
3. Isomorphismes (mathématiques)

GREYC
ENSICAEN
Bâtiment F
6 Boulevard du Maréchal Juin
CS 45053
14050 CAEN cedex 4

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my special appreciation and thanks to my Italian advisor Professor Mario Vento. Every time I felt lost, He has been a beacon of light guiding me through the darkness. His huge experience and passion have always been sources of inspiration and motivation. I consider Him as my scientific father. I would like to thank Him also for the opportunities, the time He has granted me during my PhD.

Another special thanks is addressed to my French advisor, Professor Luc Brun. His experience and knowledge have helped me significantly in developing the research topics I was interested in at the ENSICAEN. I would like to thank Him also for the patience He had in helping me.

A thanks to Prof. Pasquale Foggia, He has been my scientific coach. He gave me the opportunity to glean from His huge experience and culture. I hope to have been a good rookie and make His effort be payed back.

I would like to give a thanks to Prof. Gennaro Percannella and Prof. Pierluigi Ritrovato for all the precious help and the suggestions they gave me during my PhD.

Then I would like to thank all the friends who have made my PhD easier to face. First of all, Alessia Saggese for all the help she gave me in writing papers and this manuscript and for all the weekends we worked together with a mug of hot chocolate. Nicola Strisciuglio, we started the PhD together and we have shared pains and pleasures of this path. He has been an unlimited source of interesting discussions (and distractions). Rosario Di Lascio we worked together on several software projects during these years,

he is a great project manager. Despite he does not like very much scientific research, we become good friends. Benoit Gaüzère who gave me a great help when I was in France and who has been a good research and running partner. I would like to thank him also for his important contribution to this thesis. Antonio Greco and Raffaele Iuliano for their contribution in relieving the days of hard work. All of them are first good friends, then great colleagues.

A special thanks to my family. Words cannot express how grateful I am to all of them for the sacrifices they made on my behalf. They gave me motivation and support to pursue all my dreams.

A thanks to Valentina, she is an amazing partner. I'm very grateful for the love she give me every day and the patience she has to accept all the sacrifices required to achieve my goals.

A thanks to my brothers in-law. Pasquale who shares with me many passions, first of all those for scientific research and wine, and Ivan who is always able to encourage and motivate me. They have supported me by stroking my dreams and adding a pinch of craziness to my life.

Finally, I would like to thank all my dear friends for the presence and the support they gave me during this years.

"Only the children know what they are looking for."
- Antoine de Saint-Exupéry

Chapter 1

Introduction

"Nature uses only the longest threads to weave her patterns, so that each small piece of her fabric reveals the organization of the entire tapestry."
- Richard P. Feynman

1.1 Graphs in Pattern Recognition

In the famous book *Pattern Classification* [1], the authors have defined Pattern Recognition as the "act of taking in raw data and making an action based on the category of the pattern". Such a definition implicitly refers to the ability of a system to create abstractions and recognize the similarity between them and the raw data retrieved from the world.

Let us provide a biological example. Easily, human brain is a pattern recognition system whose main task is recognizing objects belonging to the real world and discriminate among them. To this aim, it retrieves information from the real world and creates abstractions. Objects are, then, recursively decomposed in terms of parts and the relationships between these parts. This decomposition is enriched with other data coming from the senses such as shape, color, smells, taste, sounds and so on. When a new object comes in, the brain searches for the closest model trying to categorize it to something that is already known. If none of the models fits with the object, a new model will be produced. Moreover, human brain is able to recognize objects by using just a little part of these information, e.g. distinguish an apple from an orange by using their smell or taste only. This is due to the capability of leaving out all the useless details and creating general models. Obviously, this example is a simplified view of human brain, but it is useful to highlight a critical point of pattern recognition systems: the representation.

The way we decide to represent objects has a strong impact on the quality of the abstraction the system is able to build. A good abstraction should be complete enough to take into account the fundamental features to characterize objects belonging to a given category, but still sufficiently general to avoid leaving out objects slightly different from the model. Furthermore, the representation affects another important aspect of a pattern recognition system, i.e the similarity measure adopted to compare objects and models. It is trivial to understand that different similarity measures can correspond to a given representation and their complexities depend

on the representation itself.

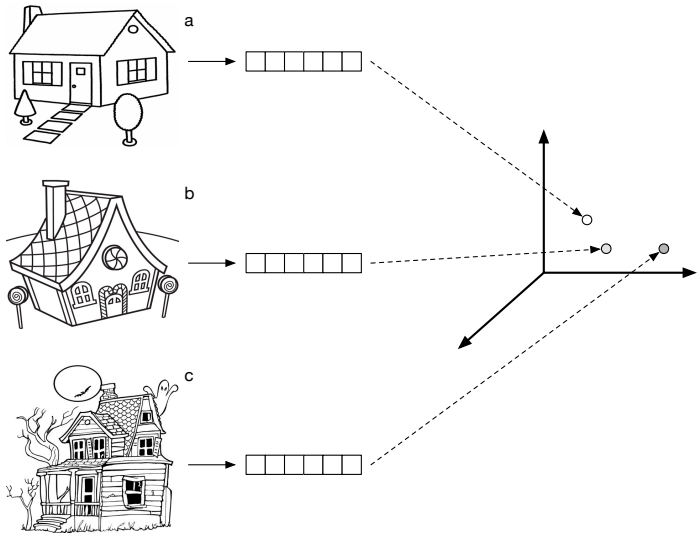


Figure 1.1: Example of vector-based representation. The model (a) and the real world objects (b) and (c) have been represented as vectors of numerical features. In this case, a set of suitable features could be: height, width, depth, number of windows, number of doors, number of rooms.

Among all the possible representations, vectors are the most adopted. Objects are described as vectors by extracting a finite set of numerical features, as shown by the example in Figure 1.1. Vectors provide many benefits, first of all the possibility to define a metric space by using the Euclidean distance and then to take advantage of many instruments provided by the vector space theory. On the other hand, if the problem requires to consider the relationships among the objects, vector-based representation could not provide a satisfying description of the world. In this case, a better representation is based on graphs, i.e. combinatorial structures that are naturally composed of objects and relationships among them.

The use of graph-based representations dates back to the early 70s, when the main application was the classification of visual patterns. Indeed, visual objects were reduced in their sub-parts and

then represented in terms of parts and connections among them, as shown by the example in Figure 1.2. In general, graphs are more expressive than vectors, but the Euclidean metric is not directly applicable on them because of their mathematical properties. In order to deal with this lack different approaches have been proposed in the last decades. We are going to discuss them in the next section.

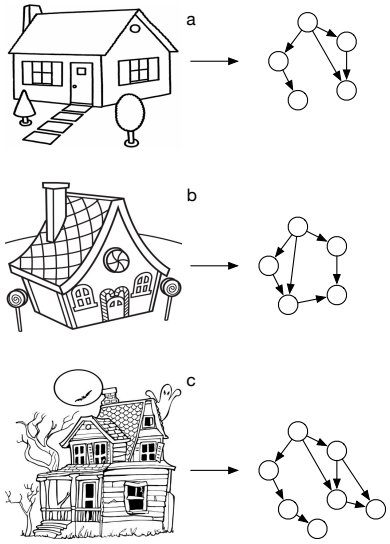


Figure 1.2: Example of graph-based representation. The model (a) and the real world objects (b) and (c) have been represented as graphs. In this case, each node is a sub-part of the house: roof, door, window, wall, three; all the connected parts are linked by an edge in the graph.

1.2 Searching for Similarities

The definition of efficient similarity measures between graphs is a key problem in Structural Pattern Recognition. A recent paper by Vento [2] provides a complete overview of all the approaches proposed to deal with the problem of using graphs-based representation in Pattern Recognition. Some of these aim to avoid

computing the similarity directly into the graph space by moving to other representations. For instance, Graph Kernels extend the concept of inner product to the graph space by defining a suited kernel function able to catch the similarity between graphs. Severe limitations in practical application are imposed by the fact that most of graph kernels require a set of graphs that is known a priori. Other methods like Graph Embedding, extensively described in different papers [3, 4, 5], aim to connect the two representations by transforming graphs in points within a suitable vector space. An important benefit is that, under certains hypotesis, the similarity of graphs is guaranteed to be preserved, in the sense that the more similar two graphs are, the closer the corresponding points in the vector space are. Graph embedding is promising but there is still an open question about the advantages of using both representation instead of using directly graphs or vectors. Indeed, graphs have been introduced when the compact representation provided by vectors is not expressive to model the problem. So can a proper vector representation of graphs be sufficiently expressive when vector representations extracted directly form the data are not?

Although both approaches have been proven to be feasible for pattern recognition problems, often they are not able to model properly the semantic and structural information carried by graphs. So, the most suited solution, to exploit both the semantic and structural information, is to face the problem directly on graphs. Graph matching provides different solutions to compare the structure and the semantic information of two graphs and then to obtain a measure of their similarity. On one hand, exact graph matching is applied when the problem requires to have an exact correspondence between the structures, e.g. when a substructure is searched inside a bigger one (subgraph isomorphism problem). On the other hand, inexact graph matching is more suited when an error tolerant measure is required. Let us consider many real situations where raw data from which the graphs have been extracted are affected by noise; in this cases, structural deformations can happens so an exact approach may not be adopted.

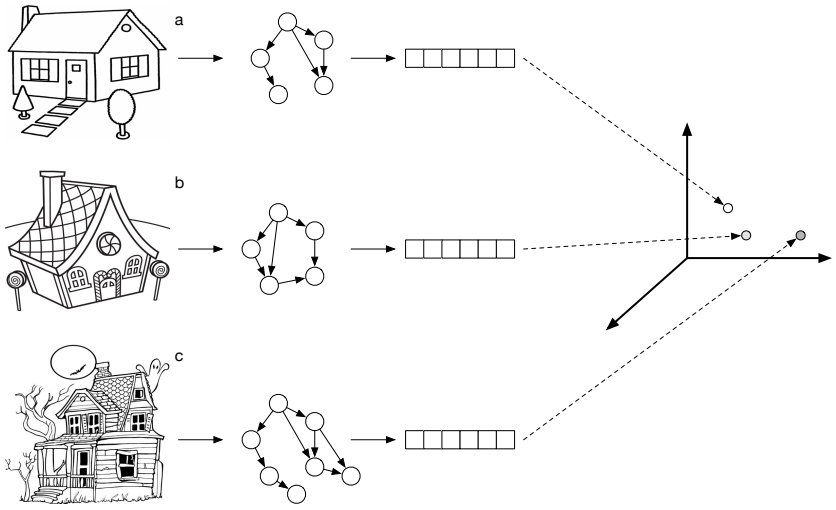


Figure 1.3: Example of graph embedding. The objects are firstly represented as graphs, then a set of features are extracted from the graphs. Different kind of features can be used both structural, such as the centralities, and semantic.

This thesis has been devoted in exploring two different methods to compute the similarity between graphs: the first is based on the subgraph isomorphism an exact graph matching problem where the aim is to search where and how many times a substructure is present inside a bigger one; the second, inexact, is a new method to compute the graph edit distance, the most flexible graph similarity measure that quantifies the distortion required to transform one graph into another one.

1.3 Thesis Overview

The Thesis has been structured as follows:

Chapter 2 provides all the theoretical foundations about graphs, graph matching, graph edit distance and assignment problems, required to understand the contents of the following chapters. The reader familiar with these concepts can skip this section.

Chapter 3 describes a novel exact graph matching algorithm for computing both graph isomorphism and subgraph isomorphism. The attention has been focused on the latter because of its relevance in several practical application. So, the algorithm has been compared with other state of the art algorithms on real biological datasets and on the MIVIA graph dataset for graph and subgraph isomorphism.

Chapter 4 describes an innovative method to compute the graph edit distance based on the quadratic assignment problem. To this aim a novel quadratic cost function has been defined to take into account the whole structure of the graph. The new formulation has been compared with the framework proposed by H.Bunke and K.Riesen [6, 7] on chemoinformatics datasets.

Chapter 5 summarizes all the methods and results obtained by the two approaches and discusses about pros and cons. Moreover, some hypothesis of the future applications and challenges in the next years are provided.

Chapter 2

Preliminaries

"He that breaks a thing to find out what it is has left the path of wisdom." - J.R.R. Tolkien

2.1 Graph Definitions

A graph is a mathematical structure $G = (V, E)$ composed of two finite sets: a *node (or vertices) set* V that represents the objects in the domain and an *edge set* $E \subseteq V \times V$ wich encodes relationships among the objects. Each edge $e \in E$ is a couple of nodes $e = (u, u')$, and the nodes connected are said to be *adjacent*. The set $adj(u) = \{u' \in V | \exists (u, u') \in E\}$ of the nodes adjacent to u is commonly named neighborhood and its size $|adj(u)|$ is the *degree* of u. It is trivial to understand the degree can be equivalently defined as the number of edges of a node.

If the couple (u, u') is ordered the edge e is said to be *directed* in the sense that it is possible to identifiy a direction from u to u', otherwise the edge is *undirected*. A graph is directed if it has at least one directed edge. On the base of the definition provided for the neighborhood, if the edge (u, u') is directed u' will be adjacent to u, but not the inverse. In this case, since a node can have outgoing and incoming edges (incident edges), it is usual to distinguish among incoming degree, outgoing degree (the degree) and total degree of a nodes.

2.1.1 Labeled and Attributed Graphs

As discussed in Chapter 1 graphs can bring semantic information. *Labeled* and *attributed* graphs are characterized by the presence of semantic data on nodes and edges. In particular, a *labeled graph* is a tuple $G = (V, E, \mu, \nu)$ where:

- $\mathcal{L}$ is a finite alphabet of nodes and edges labels.
- $\mu : V \to \mathcal{L}$ is a node labeling function.
- $\nu : E \to \mathcal{L}$ is an edge labeling function.

Similarly, if we do not consider just a finite alphabet but a set of structured information $\mathcal{A}$, an *attributed relational graph* (ARG) is a tuple $G = (V, E, \alpha, \gamma)$, where:

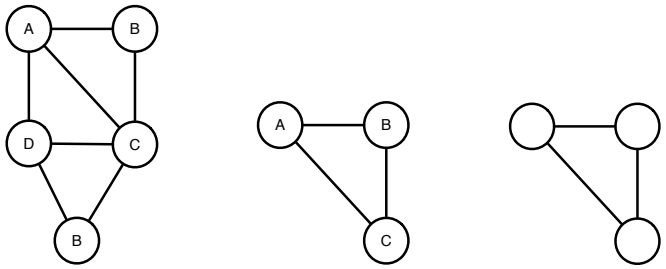
- $\alpha : V \to \mathcal{A}$ is a node attribute function.
- $\gamma : E \to \mathcal{A}$ is an edge attribute function.

2.1.2 Subgraphs and Supergraphs

Intuitively, a subgraph is a substructure contained in a wider graph having a subset of the nodes and edges that belongs to the container. Thus, given a graph $H = (V', E')$, it is a subgraph of the graph $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$. If so, G will be the supergraph of H.

Moreover, it is easy to note that given a graph G we can obtain a subgraph H by extracting a subset V' of V and putting in E' all the edged of E whose endpoint are both in V. In this case, H is called *subgraph induced on the node set* V'.

In the following sections, in particular those concerning graph edit distance, we talk about *structural subgraphs*. They are subgraphs extracted only from the structure of a graph, without considering labels or attributes. Therefore, given a labeled graph G, a structural subgraph of G is a subgraph extracted from the unlabeled graph associated to G. It is trivial to undestand a structural subgraph is always an unlabelled graph.



(a) Graph G. (b) Subgraph of G (c) Structural subgraph of G

Figure 2.1: In Figure an example of a subgraph (b) and a structural subgraph (c) extracted from G.

2.1.3 Bipartite Graphs

A bipartite graph $G = (U, V, E)$ is a graph whose node set can be partitioned in two disjoint subsets U and V such that each edge in E connects a node in U to a node in V (see Figure 2.2). If a non negative weight $c(e)$ is associated to every edge $e \in E$ of a bipartite graph it will be called *weighted bipartite graph.*

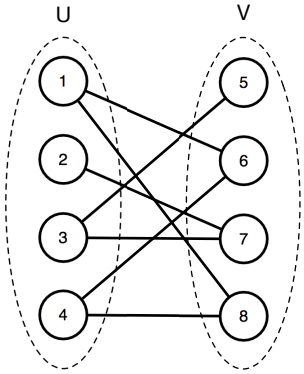


Figure 2.2: Example of bipartire graph.

An important concept that will be widely used in the following sections is those of *Bipartite Matching.* It has not to be confused with the *graph matching* that is explained in Section 2.2.

Definition 1. Bipartite Matching *A* matching *M in a bipartite graph G is a subset of the edges such that every node of G involves at most one edge of the matching. The* cardinality of the matching *is the size $|M|$.*

Definition 2. Maximal Matching *A matching M is called* maximal *if it cannot be enlarged by any edge of the graph.*

Definition 3. Maximum Matching *A matching M is called* maximum *if it has as many edges as the maximum cardinality. A maximal matching may not be maximum, but every maximum matching is maximal.*

Definition 4. Perfect Matching *A matching M is called* perfect *if every node in G is matched, i.e. it corresponds to an edge in M (see Figure 2.3).*

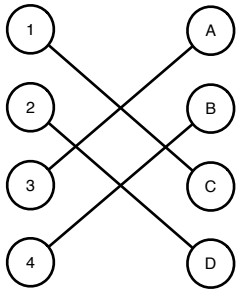


Figure 2.3: Example of perfect matching.

2.2 Graph Matching

In structural pattern recognition an important problem consists in finding a mapping, between the nodes of two graph, that satisfies a given set of structural constraints. Such a problem is commonly known as Graph Matching and comprises a large family of problems whose goal is to find structural correspondences between graphs. A detailed description of all these problems has been provided in two interesting papers of Vento and Foggia [8, 9]. In particular, the papers identify two main categories: exact graph matching where the mapping function has to preserve the structure of the graphs and inexact graph matching where structural deformations are accepted. It is important to point out that most of the graph matching problems are NP-complete and require in the worst case an exponential complexity.

2.2.1 Exact Graph Matching

Exact graph matching requires that the mapping between the nodes of two graph must preserve the adjacency, i.e. if an edge connects two edges of the first graphs they must be mapped to nodes that are linked by an edge. In its strongest form the exact graph matching is known as *graph isomorphism* and requires to preserve all the structure of both the graphs (see Definition 5). If the constraint of non-adjacency preserving is removed the second graph can have extra edges that are not present in the first graph, such kind of matching is called *monomorphism*.

Definition 5. Structure Preserving Mapping *Given two graphs* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$. *A mapping* $\varphi : V_1 \to V_2$ *is said to be:*

- *Adjacency Preserving iff*
  $\forall (i, j) \in E_1 \ \exists (\varphi(i), \varphi(j)) \in E_2$
- *Non-Adjacency Preserving iff*
  $\forall (i, j) \notin E_1 \ (\varphi(i), \varphi(j)) \notin E_2$

- *Structure Preserving iff φ is both Adjacency and Non-Adjacency Preserving*

In structural pattern recognition subgraphs are used to represent patterns of interest to search for inside wider structures, such as the search for a specific amino acid inside a group a proteins. Therefore, the first graph is smaller than the second graph, in this case it is possible to find an isomorphism or a monomorphism by mapping the first graph on a subgraph of the second. Both are *subgraph isomorphism*, but, to be rigorous, the former is usually called *induced subgraph isomorphism*.

In some situations, we are interested in searching for common structures between two graphs. Usually, the size of these structures is fixed to a given value k, but we may even search for the largest common structure. Such a problem is known as *maximum common subgraph* and is one of the hardest exact graph matching problem. In order to be thorough, when we are searching for structural subgraphs (see Section 2.1.2), we will talk of *structural common subgraph* problem.

It is important to note that, in general, exact graph matching problems are NP-complete. It has been proved for the subgraph isomorphism. Therefore, many algorithms aim to reduce the complexity by focusing the attention on a specific kind of graphs. There is not an algorithm which performs better then the others for all the possible families of graphs and in the worst case, i.e. when the graphs are symmetric, all the algorithms have a factorial computational complexity.

2.2.2 Inexact Graph Matching

In some real situations the variability of the patterns, the noise in the acquisition process or other causes, produce deformations in the observed graphs. So, two graphs may have a structure that is almost the same, but with some extra or missing nodes and edges. In this case, the constraints imposed by the exact graph matching are too strong to find out a mapping between the two graphs. The most adopted solution is to make the matching process tolerant

in respect to deformations by introducing the concept of *matching cost* to penalize structural differences. The closer are the structures of the two graphs, the lower is the cost to match them. A well known method to define the matching cost is the graph edit distance that assigns a cost to each operation needed to transform the first graph in the second one then sum all these costs to obtain the overall transformation cost. This method will be described in details in Section 2.4.

Once the cost has been defined, the problem remains how to find out a mapping that minimize such cost. A classical approach to perform the matching is based on transposing the problem in a state space representation where each state is a partial mapping. Then the A* algorithm searches for the optimal mapping by exploiting an heuristic cost function that depends on the kind of graphs is dealing with. Other completely different methods, based on the concept of *continuous optimization*, cast graph matching, that is a discrete optimization problem, into a continuous, nonlinear optimization problem, then an optimization algorithm is used to find the solution.

It is easy to understand that, as for the exact graph matching, the complexity needed to compute the optimal mapping is exponential. For this reason, the attention of the scientific community has been mainly focused on the definition of methods that compute a suboptimal mapping in a polynomial. A recent noteworthy method, proposed by Riesen and Bunke [7]. computes the suboptimal mapping by approaching the inexact matching as an assignment problem.

2.3 Assignment Problems

Every time we are dealing with the problem of assigning n item to other n items we are engaging an assignment problem. Let us think, as example, to the problem of deciding the best way to deliver a set of services to a set of servers in order to obtain the minimum overall response time. In this case, we can represent the problem as a bipartite graph where one set of nodes is composed of services and the other is composed of server. We can attribute a cost to each connection between the two sets that is related to the amount of request expected for a given service if delivered on a given server. The aim is to map each service to each server by respecting the global constraint.

In a more formal way, an assignment is usually represented as a bijective mapping $\varphi : \mathcal{X} \to \mathcal{Y}$ between two finite sets $\mathcal{X} = \{x_i\}_i$ and $\mathcal{Y} = \{y_i\}_i$ of size $|\mathcal{X}| = |\mathcal{Y}| = n$. By assigning the n elements of $\mathcal{X}$ to the n elements of $\mathcal{Y}$ we get an assignment φ corresponding to a *permutation* $(\varphi(1), \ldots, \varphi(n))$, where the first element is assigned to $\varphi(1)$, the second to $\varphi(2)$ and so on. Every permutation φ corresponds to an $n \times n$ *permutation matrix* $X_\varphi = x_{ij}$ where:

$$x_{ij} = \begin{cases} 1 & \text{if } j = \varphi(i) \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

Definition 6. Permutation Matrix *A matrix x_{ij} is said to be a Permutation Matrix iff it is compatible with the assignment constraints:*

$$\sum_{i=1}^{n} x_{ij} = 1 \tag{2.2}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \tag{2.3}$$

$$x_{ij} \in \{0, 1\} \tag{2.4}$$

The assignment constraints ensure that x_{ij} is a binary matrix

and that the mapping function defined by $\varphi(i) = i$ iff $x_{ij} = 1$ represents a one-to-one mapping from $\mathcal{X}$ to $\mathcal{Y}$. It easy to understand that give two sets of items there are several possible assignments, indeed, if S_n is the set of all possible assignments of n items then $|S_n| = n!$. The only way to discriminate among them and define what is an optimal assignment, is to introduce a cost function $C_\varphi : \varphi \to \mathbb{R}$. Thus, the best assignment $\varphi*$ can be defined as the assignment with the minimum cost:

$$\varphi* = \underset{\varphi}{\operatorname{argmin}} C_\varphi \tag{2.5}$$

On the base of the cost function C_φ we can have two different kind of assignment problem: the Linear Sum Assignment Problem (LSAP), whose solution can be computer in cubic time and the Quadratic Assignment Problem (QAP) that is NP-Hard.

2.3.1 Assignment as a Perfect Matching

The concept of perfect matching on a bipartite graph, described in Section 2.1.3, offers another way to represent an assignment. In particular, if we consider the two sets of the assignment problem as node sets of a bipartite graph, it will be quite easy to imagine the strong relationship between an assignment and a perfect matching. A bipartite graph can have more than a perfect matching and finding the number of different perfect matchings is an NP-complete problem, as discussed in [10].

Luckily, the necessary and sufficient condition for the existence of a perfect matching has been provided by the marriage theorem 2.3.2, formulated by Philip Hall in 1935 [10]. The name of this theorem comes from the nice interpretation that was given by Hall of the perfect matching problem. Let us consider a set U of ladies and a set V of men, each edge $(u, v) \in E$ represents the lady u is friend of the man v. So, a perfect matching corresponds to a marriage of all the ladies and men where a couple can only be married if the partners are friends. In particular, Theorem 2.3.2 states that each lady can marry one of her friends and assuming in

addition that $|U| = |V|$, i.e. the ladies and the men are in the same quantity, all ladies and men can marry (the perfect matching).

Theorem 2.3.1. *Let $G = (U, V, E)$ be a bipartite graph. It is possible to match every vertex of U with a vertex of V if and only if for all subsets $U' \subseteq U$:*

$$|U'| \leq |adj(U')| \text{ (Hall's condition)} \tag{2.6}$$

Theorem 2.3.2. *Let $G = (U, V, E)$ be a bipartite graph with $|U| = |V|$. There exists a perfect matching (marriage) in G iff G fulfills Hall's condition.*

The Hall's theorem does not provide an efficient method for finding a perfect matching, but many approaches have been proposed, like those proposed by Hopcroft and Karp [11] that is able to find out the perfect matching with a time complexity of $O(n^{5/3})$.

The problems we will introduce in the following sections are quite different from the problem of deciding if there exists or not an assignment, but the goal is to find the optimal assignment that minimizes a given cost function. This problem is known as *weighted perfect matching problem*, where the aim is to find a perfect matching where the sum of all the weights assigned to the involved edges is a minimum (or a maximum).

2.3.2 Linear Sum Assignment Problem

Given a matrix $\mathbf{C} \in \mathbb{R}_+^{n \times n}$ such as $C_{i,j} = c(x_i \to y_j) = c(y_j \to x_i)$ corresponds to the cost of assigning the element $x_i \in \mathcal{X}$ to the element $y_j \in \mathcal{Y}$. The corresponding Linear Sum Assignment Problem (LSAP) consists in finding an optimal permutation

$$\hat{\varphi} \in \underset{\varphi \in \Phi_n}{\operatorname{argmin}} \sum_{i=1}^{n} C_{i\varphi(i)} \tag{2.7}$$

where Φ_n is the set of all permutations of $\{1, \ldots, n\}$

Generally, the LSAP is equivalently defined in terms of permutation matrices, as follows:

$$\min_x \sum_i \sum_j C_{ij} x_{ij} \tag{2.8}$$

$$\text{s.t.} \sum_{i=1}^{n} x_{ij} = 1 \tag{2.9}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \tag{2.10}$$

$$x_{ij} \in 0, 1 \tag{2.11}$$

where x_{ij} is a mapping matrix under the *assignment constraints*.

This last formulation may be simplified in:

$$\min_{x \in \mathbb{R}^{n^2}} c^t x \tag{2.12}$$

where c and x denote the "vectored" versions of respectively the matrix C encoding mapping costs and the permutation matrix. Note that Equation 2.12 justifies the L(linear) of the LSAP problem.

The resolution of LSAP has been widely studied and several algorithms exist to solve this kind of problems. Among them, we can cite the Hungarian, Kuhn-Munkres or Volgerant-Junker algorithm [12, 13, 14] which finds an optimal solution in $O(n^3)$ time complexity. This algorithm has been generalized in many directions, we refer the reader to [10] for more details.

2.3.3 Quadratic Assignment Problem

The QAP was proposed by Koopmans and Beckmann [15] in order to provide a mathematical formalization for the problem of allocating a set of facilities, i.e. $\mathcal{X}$, to a set of locations, i.e. $\mathcal{Y}$, by minimizing the overall cost.

Differently from the Linear Sum Assignment Problem, in this case, the formulation takes into account even the relationships

among the objects in the same set. For instance, coming back to our example, we have to consider the amount of traffic among the services and the time needed to transfer data among the servers.

The quadratic assignment problem is composed of three matrices $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}_{+}^{n \times n}$, such as:

- $A_{i,k}$ is the flow between the element x_i and the element x_k in $\mathcal{X}$
- $B_{j,l}$ is the distance between the element y_j and the element y_l in $\mathcal{Y}$
- $C_{i,j}$ corresponds to the cost of assigning the element $x_i \in \mathcal{X}$ to the element $y_j \in \mathcal{Y}$

The corresponding Quadratic Assignment Problem (QAP) consists in finding an optimal permutation

$$\hat{\varphi} \in \underset{\varphi \in \Phi_n}{\operatorname{argmin}} \sum_{i=1}^{n} \sum_{k=1}^{n} A_{ik} B_{\varphi(i)\varphi(k)} + \sum_{i=1}^{n} C_{i\varphi(i)} \tag{2.13}$$

where Φ_n is the set of all permutations of $\{1, \ldots, n\}$

As for the LSAP, also the QAP is generally defined as a constrained optimization problem:

$$\min \sum_{i} \sum_{j} \sum_{k} \sum_{l} A_{ik} B_{jl} x_{ij} x_{kl} + \sum_{i} \sum_{j} C_{ij} x_{ij} \tag{2.14}$$

$$\text{s.t.} \sum_{i=1}^{n} x_{ij} = 1 \tag{2.15}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \tag{2.16}$$

$$x_{ij} \in \{0, 1\} \tag{2.17}$$

where x_{ij} is a mapping matrix under the *assignment constraints*.

A most general formulation of the quadratic assignment problem was provided by Lowler [16], such a formulation considers

four-dimensional tensor D_{ijkl} of coefficients instead of the two matrices A and B. Then Equation 2.13 can be rewritten as:

$$\min_{\varphi \in \Phi_n} \sum_i \sum_j D_{i,j,\varphi(i),\varphi(j)} + \sum_i C_{i,\varphi(i)} \tag{2.18}$$

In particular, from the Lowler's formulation it is easy to obtain a quadratic form of the QAP:

$$\min_{x \in \Phi_n} x^T D x + C x^T \tag{2.19}$$

$$\text{s.t. } Mx = 1 \tag{2.20}$$

$$x \in \{0, 1\} \tag{2.21}$$

where $Mx = 1$ and $x \in \{0, 1\}$ is the matrix form to represents the assignment constraints. Such formulation will be used in the successive Sections to describe the solving algorithms and the QAP formulation of the graph edit distance.

Differently from LSAP, for which there exist efficient methods to compute the solution in polynomial time, the QAP has been demonstrated to be an NP-hard problem [10]. It means that it is even impossible to find an approximate solution within some constant factor from the optimum value in polynomial time.

2.4 Graph Edit Distance

Graph edit distance is a flexible graph dissimilarity measure that belongs to the family of inexact graph matching methods. In particular, it measures the deformation between two graphs by considering the cost assigned to the sequence of elementary graph edit operations needed to transform the first graph in the second one. As defined in Definition 7, an elementary graph operation e is the simplest alteration which may be performed on a graph, such add or remove a node. A transformation is composed of a set of edit operations sequentially applied to a graph, namely an *edit path* P (see definition 8).

Definition 7. Elementary graph edit operations *An elementary graph edit operation (or edit operation) is one of the following operation applied on a graph:*

- *Node/Edge removal. Such removals are defined as the removal of the considered element from sets V or E.*
- *Node/Edge insertion. On labeled graphs, a node/edge insertion also associates a label to the inserted element.*
- *Node/Edge substitution if the graph is a labeled one. Such an operation modifies the label of a node or an edge and thus transforms the node or edge labeling functions.*

Definition 8. Edit path

- *An edit path of a graph G is a sequence of elementary operations applied on G, where node removal and edge insertion have to satisfy the following constraints:*
  1. *A node removal implies a first removal of all its incident edges,*
  2. *An edge insertion can be applied only between two existing or already inserted nodes.*

- *An edit path between two graphs G_1 and G_2 is an edit path of G_1 whose last graph is G_2.*

If G_1 and G_2 are unlabeled we assume that no node nor edge substitutions are performed.

It is possible to assign a cost to each simple operation by a set of operation dependant cost functions. More precisely, let x denotes an elementary operation, we distinguish the following cost functions:

- Node ($c_{vd}(x)$) and edge removal ($c_{ed}(x)$)
- Node ($c_{vi}(x)$) and edge ($c_{ei}(x)$) insertion,
- Node ($c_{vs}(x)$) and edge ($c_{es}(x)$) substitution on labeled graphs.

By extension, we will consider that functions c_{vd} and c_{vi} (resp. c_{ed} and c_{ei}) apply on the set of nodes (resp. set of edges) of a graph. Hence, the cost $c_{vd}(v)$ denotes the cost of the elementary operation "removing node v". Moreover, we can assume that a substitution transforming one label into the same label has zero cost:

$$\forall l \in \mathcal{L},\ c_{vs}(l \to l) = c_{es}(l \to l) = 0$$

where $l \to l'$ denotes the substitution of label l into l' on some edge or node.

Given the cost of each elementary edit operation, then the cost of an edit path $\gamma(P)$ is the sum of these elementary costs (equation 2.22).

$$\gamma(P) = \sum_{e \in P} c(e) \tag{2.22}$$

Among all the possible transformations $\mathcal{P}(G_1, G_2)$ that involve two graphs G_1 and G_2, we are interested in the cheapest one whose cost is the *graph edit distance* $d(G_1, G_2)$ between G_1 and G_2 (equation 2.23).

$$d(G_1, G_2) = \min_{P \in \mathcal{P}(G_1, G_2)} \gamma(P) \tag{2.23}$$

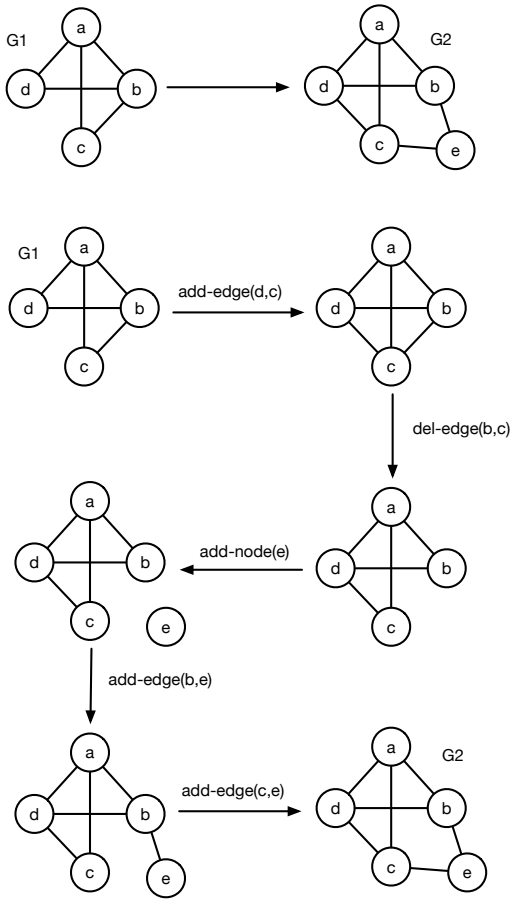


Figure 2.4: A possible edit path to transform the graph G_1 into the graph G_2. If we assume that all the edit operation have an unitary cost, the overall cost of the transformation is equal to 5.

An edit path from G_1 to G_2 with minimal cost is called an *optimal path*.

Chapter 3

Similarity by Subgraph Isomorphism

"It's better to look ahead and prepare, than to look back and regret."
- Jackie Joyner-Kersee

3.1 Introduction

As introduced in Chapter 1, structural representations are widely employed in many application fields, such as biology, chemistry, social networks, databases and knowledge discovery and so on. Because they work on data that are naturally composed of objects connected with each other, let us think about proteins composed of molecules connected to other molecules whose properties are related to the kind of connections they have. The analysis of these data often requires to determine if, how many times and where a substructure of interest is inside. Coming back to the example of proteins in biology, the interest is in searching for a particular molecular structures, inside a protein, providing information about some property the latter could show. But we could even find similar problems in the other fields, for instance, in the case of social networks where people, organization and other social entities are linked by social relationships, such as kinship, friendship, affiliation and so on. Here the aim can be the search for social patterns to discovery new relationships between the entities that are not evident in the initial network.

All these problems share a strong relationship with subgraph isomorphism, an exact graph matching problem that consists in finding all the possible isomorphisms between a pattern graph and a target graph. Unfortunately, subgraph ismorphism has been proven to be NP-complete [8]. The computational complexity in the worst case, i.e. when the two structures are symmetric, is factorial. Despite this limit is intrinsic and can not be defeated, it is often possible to reduce the computational complexity in the average case. The cornerstone to achieve this goal is the knowledge about the domain and the structure of the graphs the algorithm is going to face. Indeed, specific heuristics can provide advantages on given kind of graphs but disadvantages on others. There is no strategy that has been demonstrated to be the best in all the situations. This has contributed, in the last decades to the birth of many algorithms whose aim have been to face effectively subgraph isomorphism in specific applications.

3.1.1 Subgraph Isomorphism Algorithms

A complete description of all the scientific literature concerning subgraph isomorphism in Pattern Recognition has been discussed in the survey of Conte et al. [8] then revised by Foggia et al. in a more recent paper [9]. Among all the algorithms that have been established during the years, we have identified three main paradigms (see Table 3.1): Tree Search based, Constraing Programming based and Graph Indexing based.

The tree search approach comprises many algorithms coming from the field of artificial intelligence that deal with the problem by moving it in a state space representation where each state represents a partial matching. The solution is obtained by searching inside the space usually by adopting a depth-first search with backtracking. The algorithms belonging to this family incrementally build a solution. At each new state, a pair of nodes is added to the current solution, after checking if the addition is consistent with the constraints of the subgraph isomorphism and with the specific heuristic used by the algorithm. If a point where no other pair can be added is reached, they backtrack, removing the previous pair and trying a new one. The most known tree search based algorithms are Ullman [17] and VF/VF2 [18] that have been the state of the art for almost ten years. But, recently, new algorithms have arisen in literature, among them RI [19] is noteworthy because has demonstrated to outperform VF2 in many situations.

The constraing programming approach has been firstly pro-

Approach	**Algorithms**
Three Search	Ullman [17], VF/VF2 [18], RI/RI-DS [19]
Constraing Programming	McGregor [20], Larrosa and Valiente[21], Zampelli [22], Solnon [23], Ullman [24]
Graph Indexing	GraphQL [25], QuickSI [26], GADDI [27], SPath [28], TurboIso [29]

Table 3.1: The three main approaches to face the subgraph isomorphism and their state of the art algorithms

posed by McGregor [20] in the 1979, then improved during the years by Larrosa and Valiente[21], Zampelli [22], Solnon [23] and Ullman [24]. The constraint programming algorithms deals with subgraph isomorphism by using a diametrically opposite method, with respect to those based on tree search. The latter start from an empty mapping and then add new couples until there are no more nodes to map. On the contrary constraint programming algorithms work by filtering, among all the possible couples, those are surely not contained in the solution. In particular, they first compute a *domain* of compatibility for each node of the pattern; then the domains are iteratively reduced by propagating constraints on the structure of the mapping, until only few candidate matchings remain, that can be easily enumerated. The main drawback of this approach is generally the amount of memory required to achieve the matching. Tree search based algorithms can require a memory that grows linearly with respect to the size of the pattern graph; those based on constraint programming have, in the worst case, a quadratic space complexity due to the fact that they need to store the domains of compatibility.

Finally, the graph indexing approach comprises algorithms that extend the pure database approach whose aim is to retrieve from a graph database all those who contain a given pattern. Pure graph indexing algorithms deal with the problem to test if a pattern graph is inside a target graph, without identifying where and how many times. A common way to address the problem is to compute a *graph index*, that is a vector or a tree of features representative of the structural and semantic information of a graph; then it is used to test the presence of the pattern inside each target graphs. So, the online query processing cost is moved to the off-line index construction phase. The more representative is the index the higher is its precision in retrieving the desired target and its computational cost. It worth to point out that the index, generally, does not provide any guarantee about the isomorphism. Two not isomorphic structures may have the same index. Graph indexing based algorithm generalize this approach to the case of subgraph isomorphism. The index is used to search for all the

subgraphs in the target graph that have the same index; then the solution is refined by checking the isomorphism constraints and removing all the inconsistent matchings. Recent algorithms based on Graph Indexing are: GraphQL [25], QuickSI [26], GADDI [27], SPath [28] and TurboIso [29].

3.1.2 Chapter overview

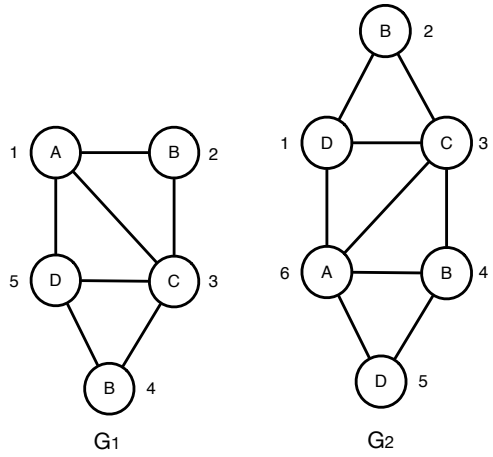
In this Chapter we will describe VF3, a new algorithm that is able to deal with different exact graph matching problem due to its general structure. Despite that, in this Thesis we have focused the attentions on the subgraph isomorphism, so we discuss in details how the algorithm deals with it. In Section 3.2 we will firstly discuss the structure of VF2 that has been inherited by VF3, then in Section 3.3 we will discuss what has been improved by VF3. Finally, in Section 3.4 we will compare VF3 with VF2 and other state of the art algorithm.

3.2 VF2: heritage of a successful approach

VF2 [18] is a very flexible algorithm able to deal with different exact graph matching problems on several kind of graphs. It is based on a state space representation where each state is a partial mapping between the two given graphs. All the states whose mapping is complete, i.e. it can not be further extended by adding new couples of nodes, are possible solutions. But, only those satisfying the constraints imposed by the specific problem are considered as *goal states*. As an example, if the algorithm is searching for a subgraph isomorphism, a state corresponding to a mapping that involves all the nodes of the smallest graph and satisfies the structure preserving constraints (see Section 2.2) is considered to be a goal state.

Therefore, the aim is to start from an *empty state*, representing a *void mapping*, then explore the state space by increasing iteratively the number of couples involved by the mapping until a goal state is reached, if it exists.

The state space is naturally represented as a graph, so it can be explored in different ways, VF2 adopts a depth-first strategy with backtracking, as other tree search based algorithms, because of it efficiency in terms of space and time. But the real innovation of VF2 has been the introduction of feasibility rules to prune, in advance, unfruitful search paths combined with a memory efficient representation of the state space. These two elements have allowed it to have a space complexity that is linear with respect to the size of the smaller graph and a time complexity that is quadratic in the average case. It worth to point out that a recent paper of N. Dahm and H. Bunke [30] has reconfirmed the effectiveness of feasibility rules in terms of search space reduction and complexity. Nevertheless, the exponential nature of the subgraph isomorphism problem and the wide variety of application fields raise the need for an algorithm that is generic with respect to the specific contexts, but easy to specialize, by using simple heuristics if possible, in order to reduce the explored search space and consequently the



(a) Pattern graph. (b) Target graph.

Figure 3.1: Graphs used in the following examples about VF2 and VF3. On the left the pattern graphs the algorithm is searching for inside the target graph, on the right.

time to find the solutions.

3.2.1 Representation of the problem

As introduced above, the process of finding an exact matching between two graphs can be solved by means of a State Space Representation (SSR). More formally, given the graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, each state s, in the *state space* S, represents a *partial mapping* $M(s)$ between V_1 and V_2, that involves only a subset of all the node couples in the goal mapping M. So that, each partial mapping $M(s) \equiv M_1(s) \times M_2(s)$ is composed of a set of ordered node couples (u, v), where $u \in M_1(s) \subseteq V_1$ and $v \in M_2(s) \subseteq V_2$. Recalling the fact that a subgraph can be extracted from a graph by considering a subset of its nodes (see Section 2.1.2), it is trivial to imagine that the subsets $M_1(s)$ and $M_2(s)$ induce two subgraphs $G_1(s) = (M_1(s), B_1(s))$ and $G_2(s) = (M_2(s), B_2(s))$, respectively of G_1 and G_2. Where, in-

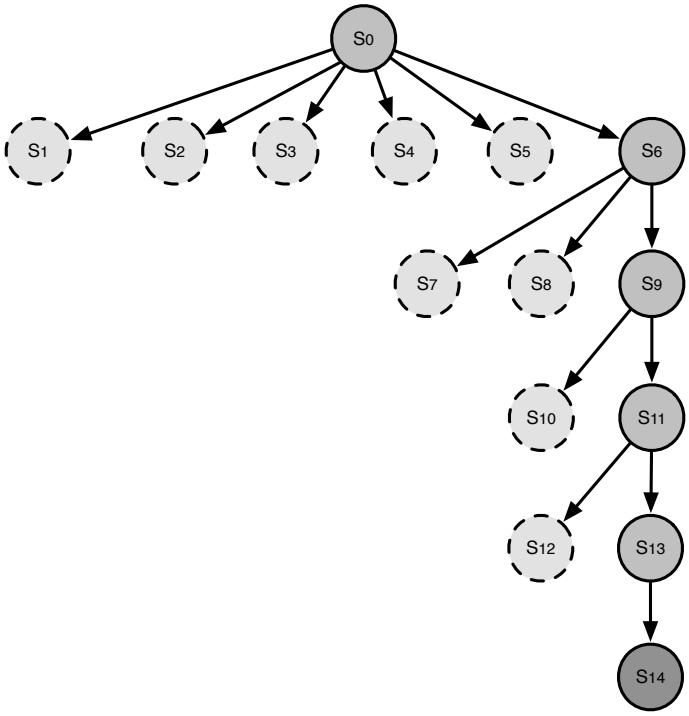
tuitively, the sets $B_1(s)$ and $B_2(s)$ contain only the edges of $G_1(s)$ and $G_2(s)$ connecting the nodes in $M_1(s)$ and $M_2(s)$. Therefore, if these subgraphs satisfy the constraints of the wanted matching then the state s will be *consistent*. For instance, if the algorithm is searching for a graph isomorphism then s will be consistent iff $G_1(s)$ and $G_2(s)$ are isomorphic. An example of the sets described above is shown in Figure 3.2 and Figure 3.3.

3.2.1.1 Exploring the space

According to the representation described in previous Section, the matching process is a search inside the SSR where the algorithm starts form a state s_0, that represents a void mapping $M(s_0) = \emptyset$ and searches for one or more goal states s_g, where $M(s_g) \equiv M$. Each new state s' is generated from a *parent state* s by adding a new ordered couple (u, v), where $u, v \notin M(s)$. The state transition from s to s' corresponds to the addition of the node u to $G_1(s)$ and the node v to $G_2(s)$.

It is easy to undestand that if we generate all the possible state, through an exhaustive exploration, we will find all the goal states, if at least one exists. But, due to the combinatorial nature of the problem, the computational cost of exhaustive exploration is factorial with respect to the size of the graphs. Nevertheless, just the consistent states will lead the algorithm toward a solution and the number of such states is substantially smaller than the whole size of the state space. Indeed, a non consistent state s will not generate any consistent states. This is due to the fact that, if $G_1(s)$ is not isomorphic with $G_2(s)$ then $G_1(s')$ and $G_2(s')$ obtained by adding u to $G_1(s)$ and v to $G_2(s)$ will not be isomorphic. Therefore, it is possible to focus the search process on consistent states only.

As shown in Algorithm 1, VF2 explores the search space according to a depth-first search strategy with backtracking. At each iteration, the algorithm searches for a new candidate couple (u, v) to generate a new state $s' = s \cup (u, v)$. But it is not explored as soon as the new couple is added. Before that, VF2 checks, first,



State	Mapping
$\boldsymbol{S_0}$	$\boldsymbol{M(S_0) = \emptyset}$
S_1	$M(S_1) = \{(1,1)\}$
S_2	$M(S_2) = \{(1,2)\}$
S_3	$M(S_3) = \{(1,3)\}$
S_4	$M(S_4) = \{(1,4)\}$
S_5	$M(S_5) = \{(1,6)\}$
$\boldsymbol{S_6}$	$\boldsymbol{M(S_6) = \{(1,6)\}}$
S_7	$M(S_7) = \{(1,6),(2,1)\}$
S_8	$M(S_8) = \{(1,6),(2,3)\}$
$\boldsymbol{S_9}$	$\boldsymbol{M(S_9) = \{(1,6),(2,4)\}}$
S_{10}	$M(S_{10}) = \{(1,6),(2,4),(3,1)\}$
$\boldsymbol{S_{11}}$	$\boldsymbol{M(S_{11}) = \{(1,6),(2,4),(3,3)\}}$
S_{12}	$M(S_{12}) = \{(1,6),(2,4),(3,3),(4,1)\}$
$\boldsymbol{S_{13}}$	$\boldsymbol{M(S_{13}) = \{(1,6),(2,4),(3,3),(4,2)\}}$
$\boldsymbol{S_{14}}$	$\boldsymbol{M(S_{14}) = \{(1,6),(2,4),(3,3),(4,2),(5,1)\}}$

Figure 3.2: In the figure are considered the two graphs in Figure 3.1, it shows all the states found by VF2 during the exploration. The states represented by dashed lines are those generated but not explored because they are unfeasible. On the other hand, the states represented by solid lines are those consistent. S_{14} is a goal state because is both complete and consistent. Note that the numbering depends on the order the states are generated by the algorithm. Finally, in the table is shown Core set for each state, the consistent mappings are highlighted by using a bold text.

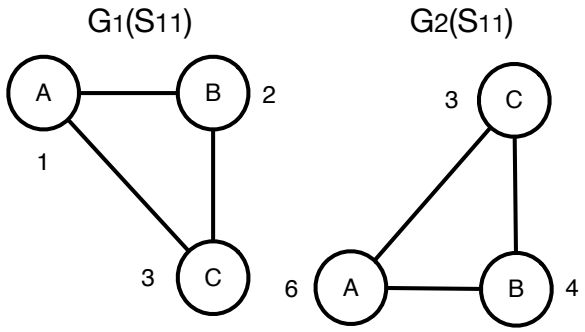


Figure 3.3: The figures is referred to the example in Figure 3.2. It shows the two isomorphic subgraphs induced by the nodes that are inside the mapping at the state S_{11}. It is cleat that S_{11} is consistent because it satisfies the constraints imposed by the graph isomorphism problem.

if the the state s' is consistent, then, if it has at least a consistent descendant Indeed, it the last condition is not verified, it is sure that we will not found any goal state by exploring it, so s' can be cut off. To this aim, VF2 uses a set of look-ahead rules through which the algorithm is able to determine if a new couple is feasible to generate a consistent state before it is explored. It worth to point out that the conditions checked by these rules are necessary, but not sufficient to satisfy the matching constraints. But such a strategy allows the algorithm to reduce significantly the number of states that are explored.

All the main aspects that have been introduced in this section will be explained in details in the followings.

3.2.2 Feasibility Rules

The concept of feasibility is a key point of VF2 and the whole structure of this algorithm is based on it. Indeed, the idea of feasible node couple is directly related to that of state consistency. Since the algorithm has to explore consistent states only, the transaction function $s' = s \cup (u, v)$, used to generate a new state s' from a consistent state s, must ensure that the addition

Algorithm 1 Structure of the matching procedure used by VF2. The inputs provided to the procedure are the start state s and the two graphs G_1, G_2. The procedure returns true if the solution exists, false otherwise.

```
1: function MATCH(s, G1, G2)
2:
3:     if IsGoal(s) then
4:         return True
5:     end if
6:
7:     if IsDead(s) then
8:         return False
9:     end if
10:
11:    Set u = ε ∧ v = ε
12:    (u', v') = GetNextCandidate(s, (u, v), G1, G2)
13:    while u' ≠ ε ∧ v' ≠ ε do
14:        if IsFeasible((u', v'), G1, G2) then
15:            s' = s ∪ (u', v')
16:            if Match(s', N_G1, G1G1, G2) is True then
17:                return True
18:            end if
19:        end if
20:        (u', v') = GetNextCandidate(s, (u', v'), G1, G2)
21:    end while
22:    return False
23: end function
```

of the pair (u, v) will lead the algorithm toward a new consistent state. So that, before generating a new state VF2 analyses the feasibility of a candidate pair (u, v) by using a *feasibility function* $F(s, u, v)$ that takes into account both structural and semantic information of each node:

$$F(s, u, v) = F_{sem}(s, u, v) \wedge F_{str}(s, u, v) \tag{3.1}$$

The semantic term $F_{sem}(s, u, v)$ of the function depends only on the attributes of the two nodes and is used to evaluate if the se-

mantic information of the two nodes is equivalent. The structural term $F_{str}(s,u,v)$ is more complex because it analyses the neighbourhood of each node by considering the consistency of three different subsets: the neighbours that are already in the mapping set $M(s)$ of the current state s, the neighbours that are not into $M(s)$ but are connected with nodes in $M(s)$ and those are not into $M(s)$ and are not connected nodes in $M(s)$. To this aim, VF2 for each states uses two distinct sets, one for each graph: the *core sets*, $M_1(s) \subseteq V_1(s)$ and $M_2(s) \subseteq V_1(s)$, to store the nodes already in $M(s)$ and the *terminal sets*, $T_1(s) \subseteq V_1(s)$ and $T_2(s) \subseteq V_1(s)$, containing the nodes connected to those are inside the core set. Then, the structural term of the feasibility function can be evaluated by considering three different rules, one for each subset defined above:

$$F_{str}(s,u,v) = R_{core}(s,u,v) \wedge R_{term}(s,u,v) \wedge R_{new}(s,u,v) \quad (3.2)$$

$R_{core}(s,u,v)$ is the main rule used to evaluate if the algorithm is going towards a new consistent state or not by adding the new couple (u,v). The rule checks if all the constraints imposed by the matching problem the algorithm is facing are respected. E.g. in case of isomorphism $R_{core}(s,u,v)$ ensures that adding the node u to the subgraph $G_1(s)$ and the node v to the subgraph $G_2(s)$ than $G_1(s')$ and $G_2(s')$ are still isomorphic.

$$\begin{aligned} R_{core}(s,u,v) \iff & \\ \forall u' \in adj(G_1,u) \cap M_1(s) & \\ \exists! v' \in adj(G_2,v) \cap M_2(s) : (u',v') \in M(s) & \\ \wedge \forall v' \in adj(G_2,v) \cap M_2(s) & \\ \exists! u' \in adj(G_1,u) \cap M_1(s) : (u',v') \in M(s) & \end{aligned} \quad (3.3)$$

Differently from R_{core}, the other two rules define conditions that are necessary but not sufficient to determine if a state is consistent. Nevertheless, they are useful to reduce the number of explored state by looking ahead. Indeed, $R_{term}(s,u,v)$ and $R_{new}(s,u,v)$ are used to look for the consistency respective one and two steps after the current state with the aim to determine if among all the descendants of s there will be or not at least a goal state. So, if they

are false we are sure that the algorithm will not found any goal state by exploring s. An important consideration about the look ahead rules is related to the computational complexity, indeed, the more the algorithm looks far the more the computational cost of the rule increases. Therefore, in order to limit such a cost the two look ahead rules just consider the cardinality of the sets used to analyse the consistency.

$$\begin{aligned} R_{term}(s,u,v) &\iff \\ |adj(G_1,u) \cap T_1(s)| &\leq |adj(G_2,v) \cap T_2(s)| \end{aligned} \tag{3.4}$$

$$\begin{aligned} R_{new}(s,u,v) &\iff \\ |adj(G_1,u) \cap \widetilde{V_1}(s)| &\leq |adj(G_2,v) \cap \widetilde{V_2}(s)| \end{aligned} \tag{3.5}$$

In Equation 3.5 two new sets have been introduced, $\widetilde{V_1}(s) \subseteq V_1(s)$ and $\widetilde{V_2}(s) \subseteq V_2(s)$, to represents the nodes that are neither in $M(s)$ nor into the terminal sets.

3.2.2.1 Extension to directed graphs

The feasibility rules can be easily extended for directed graphs by considering different sets for the edge directions, respectively $Prec(G_1,u)$, $Prec(G_2,v)$ for incoming edges and $Succ(G_1,u)$, $Succ(G_2,v)$ for outgoing edges. The terminal sets $T_1(s)$ and $T_2(s)$ will be divided each one in two subsets: $T_1^{in}(s)$, $T_1^{out}(s)$, $T_2^{in}(s)$ and $T_2^{out}(s)$. In this case, the rule $R_{core}(s,u,v)$ is composed of two subrules $R_{succ}(s,u,v)$ and $R_{pred}(s,u,v)$ used to check the consistencies on both the directions. Similarly, the look-ahead rules should be adapted to consider separately the two kind of edges.

$$R_{core}(s,u,v) \iff R_{succ}(s,u,v) \wedge R_{pred}(s,u,v) \tag{3.6}$$

$$\begin{aligned} & R_{succ}(s,u,v) \iff \\ & \forall u' \in Succ(G_1,u) \cap M_1(s) \\ \exists! v' \in Succ(G_2,v) \cap M_2(s) : (u',v') \in M(s) & \\ & \wedge \forall v' \in Succ(G_1,v) \cap M_1(s) \\ \exists! u' \in Succ(G_2,u) \cap M_2(s) : (u',v') \in M(s) & \end{aligned} \tag{3.7}$$

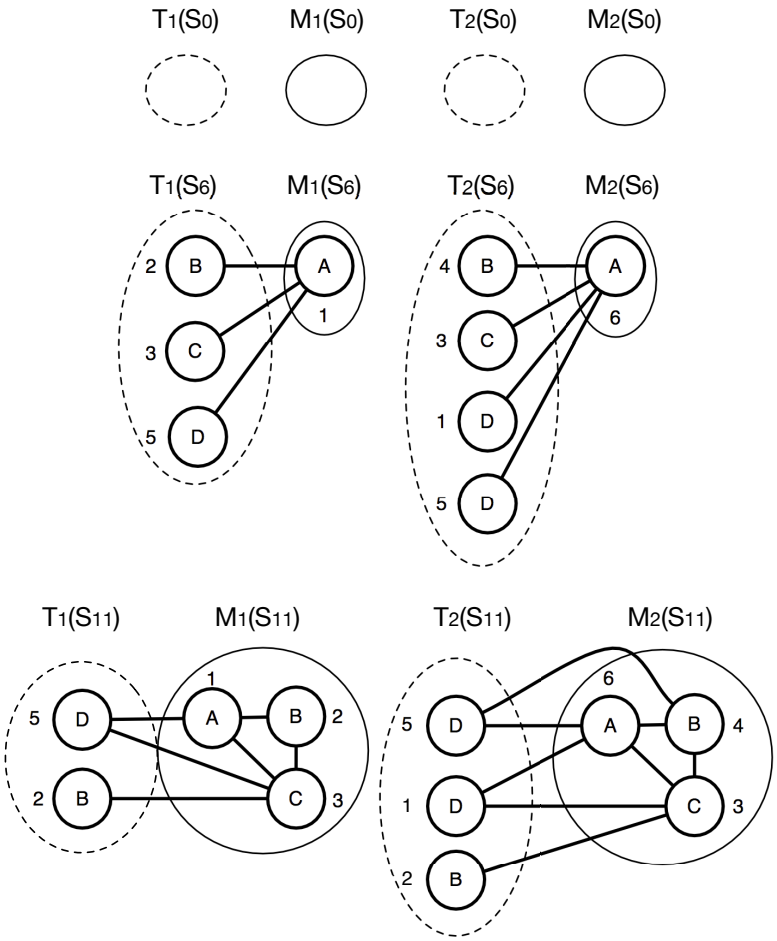


Figure 3.4: The Core sets (solid lines) and the Terminal set (dashed lines) related to the state S_0, S_6 and S_{11} in VF2 , (see Figure 3.2).

$$\begin{aligned} R_{pred}(s,u,v) \iff \\ \forall u' \in Pred(G_1,u) \cap M_1(s) \\ \exists! v' \in Pred(G_2,v) \cap M_2(s) : (u',v') \in M(s) \\ \wedge \forall v' \in Pred(G_1,v) \cap M_1(s) \\ \exists! u' \in Pred(G_2,u) \cap M_2(s) : (u',v') \in M(s) \end{aligned} \tag{3.8}$$

$$\begin{aligned} R_{term}(s,u,v) \iff \\ |Pred(G_1,u) \cap T_1(s)| \leq |Pred(G_2,v) \cap T_2(s)| \\ \wedge |Succ(G_1,u) \cap T_1(s)| \leq |Succ(G_2,v) \cap T_2(s)| \end{aligned} \tag{3.9}$$

$$\begin{aligned} R_{new}(s,u,v) \iff \\ |Pred(G_1,u) \cap \widetilde{V_1}(s)| \leq |Pred(G_2,v) \cap \widetilde{V_2}(s)| \\ \wedge |Succ(G_1,u) \cap \widetilde{V_1}(s)| \leq |Succ(G_2,v) \cap \widetilde{V_2}(s)| \end{aligned} \tag{3.10}$$

For simplicity of notations we will refer to undirected graphs only on the following sections.

3.2.3 Generating new states

Exploring a state requires to generate all its direct descendants. To this aim VF2 defines two sets $P_1(s)$ and $P_2(s)$, shown in Equation 3.11 and Equation 3.12, where to search for a new couple (u, v). If the terminal sets are not empty, the algorithm will search for the next candidate inside these, otherwise it will explore the set of nodes that are outside the terminal sets and the core set, represented by the sets $\widetilde{N_1}(s)$ and $\widetilde{N_2}(s)$. After a new couple (u, v), candidate to be added in the current mapping $M(s)$, has been found then VF2 check for its feasibility by using the above mentioned rules.

$$P_1(s) = \begin{cases} T_1(s), & \text{if } T_1(s) \neq 0 \wedge T_2(s) \neq 0 \\ \widetilde{N_1}(s), & \text{otherwise} \end{cases} \tag{3.11}$$

$$P_2(s) = \begin{cases} T_2(s), & \text{if } T_1(s) \neq 0 \wedge T_2(s) \neq 0 \\ \widetilde{N_2}(s), & \text{otherwise} \end{cases} \tag{3.12}$$

To be thorough, in Equation 3.13 and Equation 3.14 is shown the extension of the sets $P_1(s)$ and $P_2(s)$ in the case of directed graphs.

$$P_1(s) = \begin{cases} T_1^{out}(s), & \text{if } T_1^{out}(s) \neq 0 \wedge T_2^{out}(s) \neq 0 \\ T_1^{in}(s), & \text{if } T_1^{in}(s) \neq 0 \wedge T_2^{in}(s) \neq 0 \\ \widetilde{N_1}(s), & \text{otherwise} \end{cases} \tag{3.13}$$

$$P_2(s) = \begin{cases} T_2^{out}(s), & \text{if } T_1^{out}(s) \neq 0 \wedge T_2^{out}(s) \neq 0 \\ T_2^{in}(s), & \text{if } T_1^{in}(s) \neq 0 \wedge T_2^{in}(s) \neq 0 \\ \widetilde{N_2}(s), & \text{otherwise} \end{cases} \tag{3.14}$$

3.2.3.1 Avoiding cycles

The depth-first search, as all the algorithms aiming to explore graphs, has to deal with the cycles. It means, in the case of the state space, to avoid the generation of states that have been already discovered. In particular, a state s whose partial mapping $M(s)$ contains k couples is reachable from $k!$ different paths. A set that contains all the explored states is usually employed to face the problem, but this solution is unsuitable for large graphs.

Since the problem concerns graphs only another possibility is to reshape the state space as a tree. This can be obtained by defining an arbitrary total order relationship (denoted by $\prec$) over the second graph that states the order the nodes have to be explored. The strategy is feasible because the order of the couples belonging to a goal state is not relevant to the correctness of the solution. Indeed, the mapping just defines and association between two nodes of two different graphs, but not an order relationship among them.

Therefore, during the search of a new candidate couple, VF2 has to take into account the existing order relationship and then it ignores any pair $v_i \in P_2(s)$ if this set already contains a node $v_j \prec v_i$. This simple strategy allows the algorithm to generate each state only once.

3.3 VF3: a novel subgraph isomorphism algorithm

After more than ten years, the problems the subgraph isomorphism is applied to changed and the structural and semantic characteristics of the graphs involved too. In several new applications VF2 is no more competitive with the state of the art, as it has been shown in some recent papers [23, 19, 31, 32, 33]. Therefore, new strategies are needed to be able to compete with the challenges that recently arose.

VF3 is new algorithm that has been designed with the intent to improve VF2 by preserving its structure based on a SSR, a DFS with backtracking and a set of feasibility rules. In this respect, VF3 has introduced a pattern preprocessing step based on a new total order relationships, a new procedure to select the candidate couple and, finally, a set of class-based look-ahead rules that improve the pruning power of those original.

3.3.1 Pattern preprocessing

3.3.1.1 A new total order relationship

As described in the previous section, VF2 defines an arbitrary order relationship on the second graph to deal with the problem of cycles in the state space. But VF3 has the additional aim to prepare a node exploration sequence for the patter graph, that is successively used to preprocess the pattern graph and prepare in advance all the structure needed to explore the graph.

Differently from RI, the new order relationship used by VF3 does not takes into account only the structural features of the pattern graph, but tries to combine them with the structural and semantic ones of the target graph. In fact, the basic idea is giving priority to the nodes with the lowest probability to find a feasible candidate and the highest number of connections with the nodes already inserted in the sequence. In this way, the algorithm will explore first the nodes with highest number of constraints. So that,

let us enter into details and define the basic concepts. The procedure *GenerateNodeSequence* has the aim of exploring the graph G_1 and generating a *node exploration sequence* N_{G_1} by applying the order relationships. It needs to know, for each node $u \in G_1$, the probability $P_{feasible}(u)$ to find a feasible candidate $v \in G_2$. This probability is obtained by combining $P_{lab}(L)$ and $P_{deg}(d)$ that are respectively the probability to find a node with label L and the probability to find a node with degree d in G_2. In the case of the subgraph isomorphism $P_{feasible}(u)$ can be calculated as defined in Equation 3.15.

$$P_{feasible_{subiso}}(u) = P_{lab}(lab(u)) \cdot \sum_{d' \geq deg(u)} P_{deg}(d') \tag{3.15}$$

It is interesting to note that in the case of the graph isomorphism, the structural constraint given by the degree is stronger and $v \in G_2$ will be compatible to $u \in G_1$ if $deg(u) = deg(v)$. This constraint affects the definition of $P_{feasible}(u)$, as shown in Equation 3.16.

$$P_{feasible_{iso}}(u) = P_{lab}(lab(u)) \cdot P_{deg}(deg(u)) \tag{3.16}$$

Once the probability has been computed, there is another information used by the procedure to take into account the structural constraints that came from the nodes already in N_{G_1}. To this aim, we have introduced the concept of *node mapping degree*, namely the *degreeM*. Given a node $u' \in G_1$, it is defined as the number of incoming and outgoing edges between u' and all the nodes that are inside N_{G_1}. So, at each step, when a new node is inserted in N_{G_1}, the procedure has to update the constraint of all the node $u' \notin N_{G_1}$.

Now, we are ready to describe how *GenerateNodeSequence* works. In Figure 2 is shown each step of this procedure. The idea is to use the *degreeM* (deg_M in Figure 2) first as ordering criterion, then if two or more nodes share the same *degreeM* the choice will be to take that has the lowest probability to be mapped, so if more nodes have even the same probability the procedure takes the one with highest degree. Obviously, more nodes could share the same

degreeM, degree and probability, in this case the procedure just selects randomly. At the beginning N_{G_1} is empty, so all the nodes have *degreeM* equal to zero, then node with the lowest probability to be mapped is selected to be inserted in N_{G_1}.

Algorithm 2 Procedure to generate an exploration sequence. The inputs provided to the procedure are the graph G_1 whose nodes have to be ordered and the probabilities to find a feasible pair $P_{feasible}$ for each node of G_1. The output provided is N_{G_1}, a node exploration sequence over G_1.

1: **function** GENERATENODESEQUENCE(G_1,$P_{feasible}$)
2: $\forall n \in G_1$ set $deg_M(n) = 0$
3: Extract $n_0 = \min_{n \in G_1} P_{feasible}(n) \wedge \max_{n \in G_1} deg(n)$
4: Add n_0 in N_{G_1}
5: Update the set deg_M
6: **for all** $n \in G_1 \wedge n \notin N_{G_1}$ **do**
7: $n_i = \max_{n \in G_1} deg_M(n) \wedge \min_{n \in G_1} P_{feasible}(n) \wedge \max_{n \in G_1} deg(n)$
8: Add n_i in S_{G_1}
9: $\forall n \in G_1 \wedge n \notin N_{G_1}$ update $deg_M(n)$
10: **end for**
11: **return** N_{G_1}
12: **end function**

It worth to point out that the exploration sequence generated by the procedure represents a fixed traversing path used by VF3 to visit the fist graph. Each position of the sequence corresponds to a specific level of the depth-first search. Therefore, all the states at the level i will be generated by adding to those at the level $i-1$ couples that shares the same node of the pattern graph. For instance, if we consider an arbitrary node sequence $N_G = \{1, 2, 4, 3\}$, each state s, at the second level, is generated by adding a couple that have 2 as first node. In order to make more clear how the whole process works, a complete example is provided in Section 3.3.1.3.

3.3.1.2 State structures precalculation

Having a fixed exploration sequence for the first graph provides a side benefit to the algorithm. Indeed, by using such sequence, VF3 is able to pre-process the graph before starting the matching and compute, at every level of DFS, the state of each set (terminal sets and core set). Additionally, during the pre-processing, the algorithm is capable to generate a coverage tree used successively to improve significantly the selection of a new node couple.

Algorithm 3 shows the procedure used to pre-process the first graph. It explores iteratively the neighbourhood of each node u in the sequence S_{G_1}. If a neighbour u' is not yet in the terminal set T_1 of G_1, it will be inserted in T_1 and u will become the parent of u'. Since, one of the aims of such procedure is to compute the state of T_1 at each level of DFS, when a node is inserted in the set the procedure will take note of the level the node has been inserted. This information can be easily obtained by the procedure recalling the relationship between the level of DFS and the position of a node in the sequence S_{G_1}. Finally, the coverage tree is effortless obtained by using the parent set returned by the procedure. Obviously, the first node in the sequence has a dummy parent, generally identified as a null node.

3.3.1.3 Example

In this Section we are going to provide a simple, but effective example to clarify how sorting and preprocessing are performed by VF3. As discussed above, the first step the generation of an exploration sequence N_{G_1}; to this aim, it computes label and degree frequencies on the target graph G_2 and then it calculates the probability to find a feasible pair for all the nodes in G_1. In Table 3.2 and Table 3.3 are shown the frequencies and probabilities concerning the graphs in Figure 3.1. At the beginning N_{G_1} is empty, so the first node to insert is selected only on the base of the probability. Referring to Table 3.3 the node 3 has the lowest probability, then the algorithm will choose it first. Thus, $N_{G_1} = \{3\}$, all the other nodes are connected to 3, so they have the same $degM$.

Algorithm 3 Procedure to preprocess the first graph G_1 given an exploration node sequece S_{G_1}. It returns the set of parents for each node in the sequence.

```
1: function PreprocessFirstGraph(G1,SG1,T1)
2:     i = 0
3:     for all u ∈ SG1 do
4:         for all u' ∈ adj(u) do
5:             if u' ∉ T1 then
6:                 Put u' in T1 at level i
7:                 Parent(u') = u
8:             end if
9:             i = i + 1
10:        end for
11:    end for
12:    return Parent
13: end function
```

Label	Frequency
A	$0,16$
B	$0,34$
C	$0,16$
D	$0,34$

Degree	Frequency
1	0
2	$0,34$
3	$0,34$
4	$0,34$

Table 3.2: Label and degree frequencies extracted from G_2.

Node	Degree	Label	Total
1	$0,66$	$0,16$	$0,11$
2	1	$0,34$	$0,34$
3	$0,34$	$0,16$	$0,05$
4	1	$0,34$	$0,34$
5	$0,66$	$0,34$	$0,22$

Table 3.3: Probabilities for the nodes in G_1.

Among them 1 and 5 have the highest degree, but 1 has the lowest probability, so it is selected. Now, $N_{G_1} = \{3, 1\}$, 2 and 5 have the highest $degM$, but 5 is chosen because of its higher degree. Finally, the 2 and 4 are inserted in the sequence. The resulting

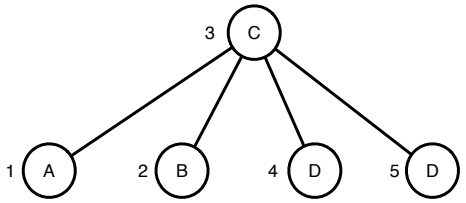


Figure 3.5: Coverage tree produced by VF3 using the exploration sequence $N_{G_1} = \{3, 1, 5, 2, 4\}$.

exploration sequence is $N_{G_1} = \{3, 1, 5, 2, 4\}$.

Once the sequence is ready, VF3 can explore the graph G_1 in order to prepare the terminal sets (see Table 3.4) and generate the coverage tree shown in Figure 3.5. Note that the maximal depth is equal to the size of G_1. At the end of this process, VF3 is

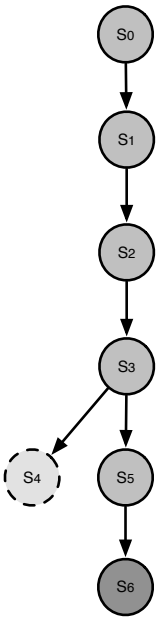
Level	Core Set	Terminal Set
0	$\emptyset$	$\emptyset$
1	$\{3_3\}$	$\{1_4, 2_2, 4_2, 5_1\}$
2	$\{3_3, 1_4\}$	$\{2_2, 4_2, 5_1\}$
3	$\{3_3, 1_4, 5_1\}$	$\{2_2, 4_2\}$
4	$\{3_3, 1_4, 5_1, 2_2\}$	$\{4_2\}$
5	$\{3_3, 1_4, 5_1, 2_2, 4_2\}$	$\emptyset$

Table 3.4: Core and Terminal sets of G_1 for each level of the depth search. For each node in the terminal set has been indicated the class it belongs to.

finally ready to start the matching.

3.3.2 Nodes Classification

In Section 3.2.2 we have described a set of rules, namely the feasibility rules, used by VF2 to be more effective in exploring the state space by looking ahead for the consistency of a new state. Such rules can be further strengthened by partitioning the sets used by the algorithms in several subsets, each of them containing all the nodes belonging to a given class. A node class contains all the



State	Mapping
$\boldsymbol{S_0}$	$\boldsymbol{M(S_0) = \emptyset}$
$\boldsymbol{S_1}$	$\boldsymbol{M(S_1) = \{(3,3)\}}$
$\boldsymbol{S_2}$	$\boldsymbol{M(S_2) = \{(3,3),(1,6)\}}$
$\boldsymbol{S_3}$	$\boldsymbol{M(S_3) = \{(3,3),(1,6),(5,1)\}}$
S_4	$M(S_4) = \{(3,3),(1,6),(5,1),(2,2)\}$
$\boldsymbol{S_5}$	$\boldsymbol{M(S_5) = \{(3,3),(1,6),(5,1),(2,4)\}}$
$\boldsymbol{S_6}$	$\boldsymbol{M(S_6) = \{(3,3),(1,6),(5,1),(2,4),(4,2)\}}$

Figure 3.6: In the figure are considered the two graphs in Figure 3.1, it shows all the states tryed by VF3 during the exploration. Similarly to Figure 3.2, the states represented by dashed lines are those generated but not explored because they are unfeasible. While, the states represented by solid lines are those consistent. S_6 is a goal state because is both complete and consistent. The numbering depends on the order the states are generated by the algorithm. The Core set for each state is shown in the table, the consistent mappings are highlighted by using a bold text. It worths to note that thanks to the node classification and the new candidate selection procedure, VF3 is able to generate far less unfeasible couples.

nodes with particular semantic and structural features such that if two nodes of G_1 and G_2 can generate a feasible couple they will be assigned to the same class. In particular, in the case of graph isomorphism, it is possible to use both the degree and the label of a node to define a class. So, if we consider the graph G_1 having the following set of distinct node degrees $deg(G_1) = \{1, 4, 5\}$, then the set associated to it by the algorithm, when a generic state s is explored, will be: $M_1(s) = M_1^1(s) \cup M_1^4(s) \cup M_1^5(s)$, $T_1(s) = T_1^1(s) \cup T_1^4(s) \cup T_1^5(s)$, $\widetilde{V}_1(s) = \widetilde{V_1^1}(s) \cup \widetilde{V_1^4}(s) \cup \widetilde{V_1^5}(s)$. Using such sets the rules described in the previous subsection will become more selective due to the fact that they can check separately the consistency on each subset.

In general, it is possible to identify a *Node Classification Function* that assigns each node to a class $c_i \in C = \{c_1, c_2, \ldots, c_n\}$, we call this function $\psi : V \in G \rightarrow C$, In this way the algorithm will be able to divide the set used to explore the space state in distinct subsets, one for each class $c_i \in C$. The function ψ must have the following properties:

1. $\bigcap_{i=1}^{n} C_i = \emptyset$, where $C_i = \{u \in V : \psi(u) = c_i\}$.
2. Given $u \in G_1$ and $v \in G_2$, $F(s, u, v) \Rightarrow \psi(u) = \psi(v)\ \forall s \in S$.

The first property ensures that the set C is composed of non overlapped classes in order to obtain distinct and non overlapped state subsets. Moreover, since the algorithm will analyse each class subset separately, the second property guarantees that we are not going to exclude feasible pairs, and so consistent states, by performing such partition.

It worths pointing out that, in the case of subgraph isomorphism, the first property of ψ disallows us to use the degree as structural information to classify the nodes. Indeed, it will happen that a node is assigned to more classes. In order to clarify this drawback, let us consider a generic node $u \in G_1$ it may be properly mapped to all the nodes in G_2 whose degree is greater or equal then u. Then, all of these nodes must share the same class c_i of u, but they are even compatible to all the nodes in G_1

Label	Class
D	1
B	2
C	3
A	4

Table 3.5: Function ψ used to classify the nodes in the example. A class for each label in G_2 has been considered.

having a degree that is less or equal then u. So, even these nodes must belong to the class c_i. Now let us consider the node $u' \in G_1$ whose degree is greater then u and belongs to the class c_j. All the nodes in G_2 that may be mapped to u' must be assigned to c_j, but these nodes may be even mapped to u. Thus, if a node cannot be assigned to two distinct classes, then u and u' will belong to the same class $c_i = c_j$. Extending this line of reasoning to all the nodes of G_1 and G_2, we will get that only a class can exist.

Given the set of classes C, the sets used by VF3 for the graph G_1 can be defined as follows:

$$\begin{aligned} M_1(s) &= M_1^{c_1}(s) \cup M_1^{c_2}(s) \cup \cdots \cup M_1^{c_n}(s) \\ T_1(s) &= T_1^{c_1}(s) \cup T_1^{c_2}(s) \cup \cdots \cup T_1^{c_n}(s) \\ \widetilde{V_1}(s) &= \widetilde{V_1^{c_1}}(s) \cup \cdots \cup \widetilde{V_1^{c_2}}(s) \cup \widetilde{V_1^{c_n}}(s) \end{aligned} \tag{3.17}$$

Similarly, it is possible to divide the sets used for the graph G_1. Once these sets have been obtained the feasibility rules can be defined as:

$$\begin{aligned} & R_{term}(s,u,v) \iff \\ & |adj(G_1,u) \cap T_1^{c_1}(s)| \leq |adj(G_2,v) \cap T_2^{c_1}(s)| \\ \wedge & |adj(G_1,u) \cap T_1^{c_2}(s)| \leq |adj(G_2,v) \cap T_2^{c_2}(s)| \\ \wedge \cdots \wedge & |adj(G_1,u) \cap T_1^{c_n}(s)| \leq |adj(G_2,v) \cap T_2^{c_n}(s)| \end{aligned} \tag{3.18}$$

$$
\begin{aligned}
& R_{new}(s,u,v) \iff \\
& |adj(G_1,u) \cap \widetilde{V_1}^{c_1}(s)| \leq |adj(G_2,v) \cap \widetilde{V_2}^{c_1}(s)| \\
& \wedge |adj(G_1,u) \cap \widetilde{V_1}^{c_2}(s)| \leq |adj(G_2,v) \cap \widetilde{V_2}^{c_2}(s)| \\
& \wedge \cdots \wedge |adj(G_1,u) \cap \widetilde{V_1}^{c_n}(s)| \leq |adj(G_2,v) \cap \widetilde{V_2}^{c_n}(s)|
\end{aligned}
\tag{3.19}
$$

In the Equation 3.5 two new sets have been introduced, $\widetilde{V_1}(s) \subseteq V_1(s)$ and $\widetilde{V_2}(s) \subseteq V_2(s)$, to represents the nodes that are neither in $M(s)$ nor into the terminal sets.

3.3.3 A new candidate selection

Recalling the procedure used by VF2; it searches for the next candidate couple inside the sets $P_1(s)$ and $P_2(s)$. These two sets contain all the nodes that are the terminal sets. In many situations, especially if the target graph is considerably larger then the pattern, just a little part of these sets contains nodes suitable to compose a feasible couple. To this reason, the algorithm could check for the feasibility or, even worse, backtrack several times, more than is necessary.

The intent of the new candidate selection procedure is to reduce the set where the algorithm explores. Indeed, when it has to search for a candidate to couple a node u, the algorithm can analyze only the neighborhood of those nodes in the core set $M(s)$ that are connected to it. This is, generally, a little subset of all the nodes connected to all the nodes in $M(s)$, as it is done in VF2. An example of the difference between VF2 and VF3 is shown in Figure 3.8.

By using the coverage tree computed during the preprocessing of the pattern graph, VF3 is able to reduce significantly the size of the set explored to search for a candidate node. The procedure *GetNextCandidate*, shown in Figure 4, starts form the current pair (u, v) and searches for a new pair (u', v'). Firstly, the procedure gets the next node u' of the sequence S_{G_1}, if it exists, and retrieves the parent $p_{u'}$ from the set *Parent* generated by the procedure

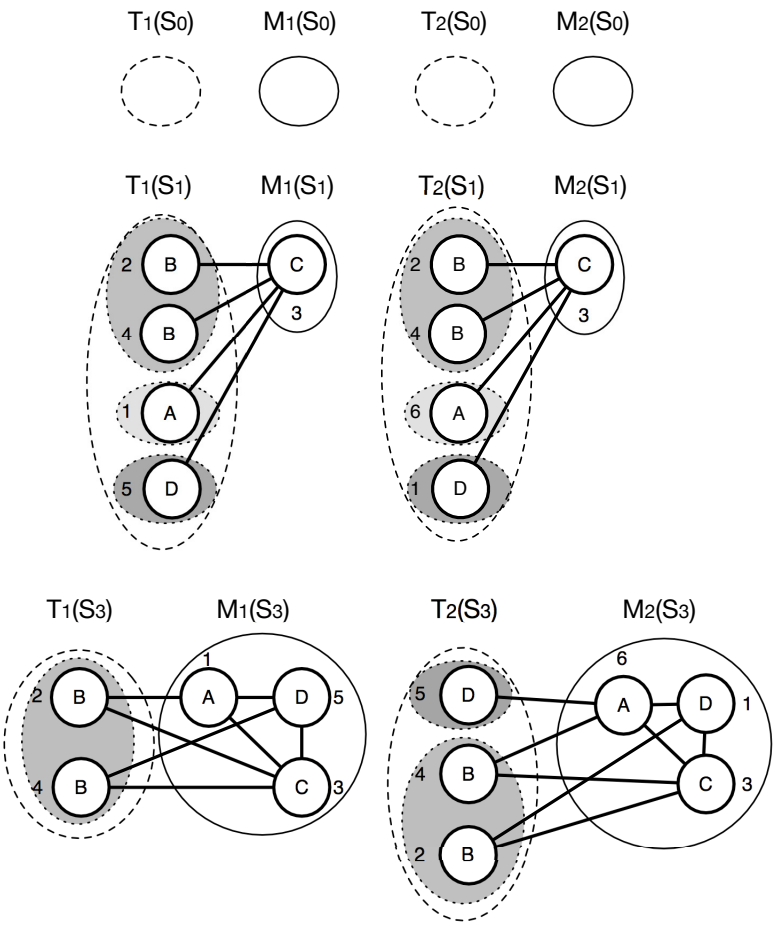


Figure 3.7: Core sets (solid lines) and Terminal sets (dashed lines) related to the state S_0, S_1 and S_3 in VF3 (see Figure 3.6). The classes considered to divide the Terminal sets are defined in Table 3.5.

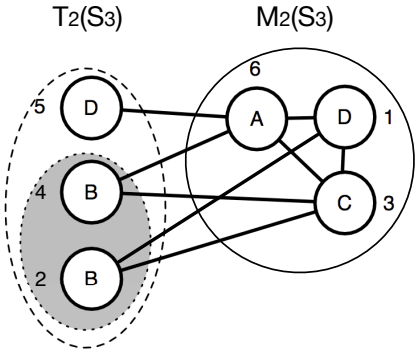


Figure 3.8: Recalling the terminal sets in Figure 3.7, the figure shows the subset of the Terminal set $T_2(S_3)$ used to search for the candidate nodes in G_2 to couple with 2, the next node in the sequence N_{G_1}. It is possible to note, as described in details in Section 3.3.3, that VF3 does not take into account the whole Terminal set $T_2(S_3)$, but just the nodes connected to 3, the parent node of 2 in the coverage tree (see Figure 3.5)

PreprocessFirstGraph. If the parent is a null node ϵ then two situations would have happened: the node u' is the first in the sequence or u' belongs to another connect component of the graph. In both these cases, the procedure has to pick the pair node for u' outside the terminal sets of G_2, i.e. in the set of $\widetilde{N_2}(s)$. If the parent of u' is not a real node the procedure will search for the pair of u' among the unmapped neighbours of the node $\widetilde{v}$, the pair of $p_{u'}$ in $M(s)$. Once an available neighbourhood v' has been found the procedure will return the couple (u', v') as next candidate to check for the feasibility. If there are no more available neighbours of $\widetilde{v}$ to be coupled with u' then the procedure will return an empty couple (ϵ, ϵ). When it happens VF3 has the only option to backtrack and remove the couple (u, v) from the current state s.

Algorithm 4 Procedure to generate the next candidate couple. The inputs provided are the current state s, the current node u of G_1, the exploration sequence S_{G_1} and the graphs G_1 and G_2. The procedure will return a candidate couple (u', v') to be checked for the feasibility or a null couple (ϵ, ϵ) if there are no more couple to explore.

```
1: function GetNextCandidate(s, u, S_G1, G_1, G_2)
2:     u' = GetNextInSequence(S_G1, u)
3:     if u' = ε then
4:         return (ε, ε)
5:     end if
6:     p_u' = Parent(u')
7:     if p_u' ≠ ε then
8:         ṽ = ψ(p_u')                          ▷ ṽ is the pair of p_u' in M(s)
9:         while v' ∈ M(s) ∨ ψ(v') ≠ ψ(u') do
10:            Put v' in adj_exp(ṽ)
11:            Pick v' from adj(ṽ, G_2) − adj_exp(ṽ)
12:        end while
13:    else
14:        while v' ∈ M(s) ∨ ψ(v') ≠ ψ(u') do
15:            Put v' in Ñ_2exp(s)
16:            Pick v' from Ñ_2(s) − Ñ_2exp(s)
17:        end while
18:    end if
19:    if v' ≠ ε then
20:        return (u', v')                      ▷ Next pair has been found
21:    else
22:        return (ε, ε)                        ▷ An empty pair is returned
23:    end if
24: end function
```

3.4 Experiments and Results

A complete characterization of a graph matching algorithm requires to analyze its behavior, in terms of time and memory usage, on a large variety of graphs. Our main intent has been to show that VF3 represents a completely new algorithm able to outperform VF2 and to be competitive with the state of the art in several situations. So that, we have compared VF3 with VF2, RI [19] and LAD [23] by using the new version of MIVIA [34, 35, 36] subgraph isomorphism dataset, a dataset of biological graphs composed of proteins and contact maps and a synthetic dataset of large and dense random graphs. VF3 has been tested over more than 100000 graphs organized in different families and having different size, density and label distributions. The experiments have required about 7000 running hours.

In this Section, we are going to describe the experimental setup used to perform the comparison: platforms, datasets and algorithms considered and then we will comment the results obtained.

3.4.1 Experimental Setup

3.4.1.1 Environment

Experiments on graphs are very time consuming. On one hand, because the search for patterns on large graphs can require much time. On the other hand, performing a time benchmark that provides significant results requires to run a single algorithm per machine. To deal with this problem, we have conducted the time benchmarks on batch of homogeneous virtual machines hosted by VMWare ESXi 5. Each of them is equipped with Ubuntu Server running on two AMD Opteron Processors 6376 2300Mhz, 2Mb of cache and 8Gb or RAM. It is important to point out that the way clock signals are kept on virtual machines is a critical point. Phenomena like *time drifting* or *lost clocks* can introduce bias in the measures. In our case, it has been faced by following the suggestions provided by VMWare Team to use correctly the virtual clock signal generated by the hypervisor. In order to be more robust

with respect to different biases introduced by the operating system each single measure has been taken several time. Given that dealing with large or regular graphs can require a large amount of time we have set a timeout of 30 minutes. Only for the additional synthetic random graphs dataset we let RI and VF3 work without timeout in order to understand better the behaviour on graphs very hard to face.

Luckily, memory benchmarks have not the same limitations and more algorithms can run concurrently on the same machine. This task has been performed on Ubuntu Server by using Valgrind, a well known software library to profile the memory usage of a process. The aim of this benchmark has been to measure the maximum amount of data memory, stack and heap, used by the algorithm during the matching. Valgring, in particular its tool Massif, has allowed us to measure the data memory only with an accuracy of 99% in getting the memory peak. Since some algorithm is memory hungry when it faces large graphs, we have limited to 1 Gb the maximum amount of memory available to each competitor.

3.4.1.2 Algorithms

We have introduced in Section 3.1.1 two state of the art algorithms we have taken into account in our experiments: RI [19] and LAD [23]. Both are are specialized in solving the subgraph isomorphism but using two different approaches. Recalling the classification we have provided in the state of the art, RI is based on a Tree Search approach, while LAD is a Constraint Programming algorithm. We have selected these two algorithms because of the following reasons. First, they are both very recent algorithms belonging to different approaches. Then, they have demonstrated to be more than one order of magnitude faster than VF2 in solving the subgraph isomorphism. Finally, their source code is easy to obtain. Therefore, we were able to add the code needed to properly perform our benchmarks.

3.4.1.2.1 RI RI is based on a static pattern preordering procedure, namely Greatest Constraint First, that aims to produce a exploration sequence on the first graph and a coverage tree used by the searching algorithm. Such procedure orders the nodes of the pattern graph by considering only the structural constrains provided by the nodes already sorted. Once the exploration sequence is ready, the matching algorithm generates all the possible paths in the search space by using a DFS with backtracking. All the path that are not consistent with the constraints of subgraph isomorphism are pruned. The pruning is based only on the check of basic semantic and structural constraints without any look-ahead. Similarly to VF2, RI has a complexity linear in space and quadratic in time in the average case.

Before discussing the results, a first difference that can be realized between RI and VF3 is in the ability of pruning the state space. On one hand, RI is less effective in pruning then VF2 and VF3 because it does not use any look-ahead or backjump rule. On the other hand, this leak is balanced both by the fact that time required to elaborate each state is considerably reduced and by the sorting procedure. Indeed, the latter is able, in the average case, to provide a good exploration sequence that reduces the number of states explored by RI.

3.4.1.2.2 LAD Similarly to other Constraint Programming algorithms, LAD works basically by filtering the starting search tree composed of all the possible couples. The filter is applied iteratively by the algorithm since no other couples can be removed. This stop condition may be verified in two different situations: all the nodes belonging to the pattern have been mapped with at least one node in the target graph (there is at least one solution), or at least one pattern node has not any correspondence with a target node (no solutions exist).

The filter used in LAD is based on a set of global constraints derived from those of subgraph isomorphism. At each step, the algorithm selects a node couple and checks that the constraints are satisfied for all the neighbors of the two nodes. This check is

performed by using a bipartite mapping whose aim is to verify that all the neighbors of the first node can have at least one pair in the target. If such condition is not satisfied the couple will discarded.

The space complexity of LAD is quadratic in the average case, while the time complexity is between N^2 and N^4, where N is the size of the smaller graph.

LAD works in a completely different way with respect to RI, VF2 and VF3 and its state space size and pruning power can not be easily compare to the others. In many situations it is faster than VF2 in finding all the solutions, even though the space required is generally higher.

3.4.1.3 MIVIA Dataset

For the most part the experiments have been conducted on the MIVIA dataset due to its completeness. Indeed, it is a synthetic dataset composed both of unlabelled and labelled graphs belonging to different families, each of them related to specific mathematical model. Moreover, for each target graph there are three patterns having a size that is respectively the 20%, 40% and 60% of the target one.

It worths pointing out that, due to the time needed to perform the experiments, we have not used the entire datasets, but only the hardest part of it. In particular, we have used only patterns large the 20% of their respective targets.

3.4.1.3.1 Bounded Valence Graph These are graphs whose total degree (ongoing and outgoing) is lower than a given threshold, namely the valence. If the number of edges is equal for all the nodes, they are commonly called *fixed valence graph*. The dataset includes the latter kind of graphs, that have been generated by inserting random edges, by using an uniform distribution, with the constraint that the valence of a node cannot exceed a selected value. Three different values of valence v have been considered (3, 6 and 9) with graph from 20 to 1000 nodes, but the dataset can be further extend by using the generators provided by the authors

together with the datasets.

In addition to fixed valence graph, the dataset contains even *irregular bounded valence graphs*, where the average valence of the nodes is bounded, so the single nodes may have a valence which is quite different from the average. It is clear that, in this case, the degree is not bounded by a constant value. The irregularities are obtained by moving the edges of fixed valence graphs according to a random distribution with uniform probability.

The dataset contains 6000 matching couples for the subgraph isomorphism problem, for each kind of labelling.

3.4.1.3.2 Regular Meshes Graphs The dataset includes connected 2D, 3D and 4D meshes as regular graphs, in order to provide a kind of graph that represent a worst case for general graph matching algorithms.

A mesh is a graph in which each node, except those in the border, is connected respectively with its k neighborhood nodes. k depends on the type of grid considered: 4 for 2D, 6 for 3D and 8 for 4D meshes.

Moreover, similarly to the bounded valence graphs, the dataset contains a set of *irregular mesh-connected graph*, obtained by adding a certain number of edges to the regular meshes. The nodes connected by each new edge have been randomly selected according to a uniform distribution.

The composition of the dataset is shown in details in Table 3.6.

mesh	k	**size range**	**# of couples per labeling**
2D	4	$[16-1024]$	4000
3D	6	$[27-1000]$	3200
4D	8	$[16-1296]$	2000

Table 3.6: Details about the mesh graph dataset structure

3.4.1.3.3 Random Graphs These are graphs where the edges connect nodes without any structural regularity. The model

used to generate such kind of random graphs has been formulated by Erodős and Rényi in [37]. In the graphs belonging to the dataset it is assumed that the probability that exists an edge, connecting two nodes, is independent on the nodes themselves. The probability distribution is assumed to be uniform and is strongly related to the desired density by means of the parameter $\eta \in [0, 1]$, where 0 represents a graph without edges and 1 is the case of a complete graph.

In the dataset are considered three different values of the edge density η (0.01, 0.05, 0.1) and 3000 matching couples for each labelling distribution. The size of the graphs is between 20 and 1000 nodes.

It is important to note that the higher is the density the harder is for a graph matching algorithm to process the graphs.

3.4.1.4 Additional Datasets

In addition to the MIVIA dataset we have considered other two datasets.

The first is a biological dataset composed of graphs extracted from real molecular structures of proteins. Such a dataset has been proposed during the International Contest on Graph Matching Algorithms for Pattern Search in Biological Databases hosted by the 22nd International Conference of Pattern Recognition [33]. The intent in this choice has been to understand how VF3 would have performed with respect to the winner of contest. In particular, the dataset we have used contains proteins from 500 to 10000 nodes and contact maps from 99 to 700 nodes.

The second is a synthetic dataset composed only of random graphs (unlabelled and labelled), generated by using the Erodős and Rényi model. Differently from the MIVIA dataset, the graphs contained in this one are considerably more large and dense. In particular, the size is from 300 to 8000 node and the values of η considered are 0.2, 0.3 and 0.4. We have taken into account this dataset to evaluate better the behaviour of VF3 on very hard graphs.

It is important to point out that, in this case we have not set a timeout so, due to the long time required to perform the experiments, on these datasets we have only measured the time usage, but not the memory. Moreover, not all the algorithms have been able to find a the solution in an acceptable time. Indeed, we have no measures of VF2 and Lad on the unlabelled graphs.

Dataset	**Graph Size**	**Labelling**	**Matching Couples**
Bounded Valence (Regular and Irregular)	$[20 - 1000]$	Unlabelled, 8 labels	30000
Mash Grids 2D/3D/4D (Regular and Irregular)	$[16 - 1296]$	Unlabelled, 8 labels	46000
Random (Regular and Irregular)	$[20 - 1000]$	Unlabelled, 8 labels	15000
Proteins	$[500 - 1000]$	5 labels	1800
Contact Maps	$[99 - 800]$	21 labels	1800
Extended Regular Random	$[300 - 8000]$	Unlabelled, 8 labels	2500

Table 3.7: Summary of the datasets used to perform the experiments .

3.4.2 Results

Finally we reach the showdown, now we need to understand if VF3 is really able to substitute VF2 and, above all, if is able to compete with the state of the art. As discussed in the previous Section, we have performed a wide analysis in order to show both strong points and weakness of VF3.

Before discussing the results, it worths to point out that the limits we have set in memory and time have made some algorithm unable to solve all the instances. In these situations we have not been able to get the correct measure then we have discarded the latter from all the algorithms. At the same time, we have keept track of all the instances that each algorithm has not been able to solve. Indeed, they provide useful information about the graphs have been hard to face for each competitor. Therefore, with the

aim to complete the data provided, each figure has been annotated with additional labels representing the number of instances, for a given target size, that the algorithm has not been able to solve. Moreover, in order to avoid the figures appear confused we have not labelled the sizes where the algorithm has solved all the instances.

Another preliminary remark to make is about VF3. We let the algorithm to generate a class for each different label and, as described in Section 3.3.2, we are not able to use structural information in the classification function. Thus, the latter does not affect the results on unlabeled graphs and, in this case, VF3 is basically rewarded by the sorting procedure, the preprocessing and the new procedure to select candidate couples. On labeled graphs the function can be both an advantage and a disadvantage. Indeed, even though the new look-ahead class-based rules provide a stronger pruning of unfeasible states, when the number of different labels increases (e.g. for the case of scale free and small world graphs with 256 different labels) the classification requires an additional amount of memory.

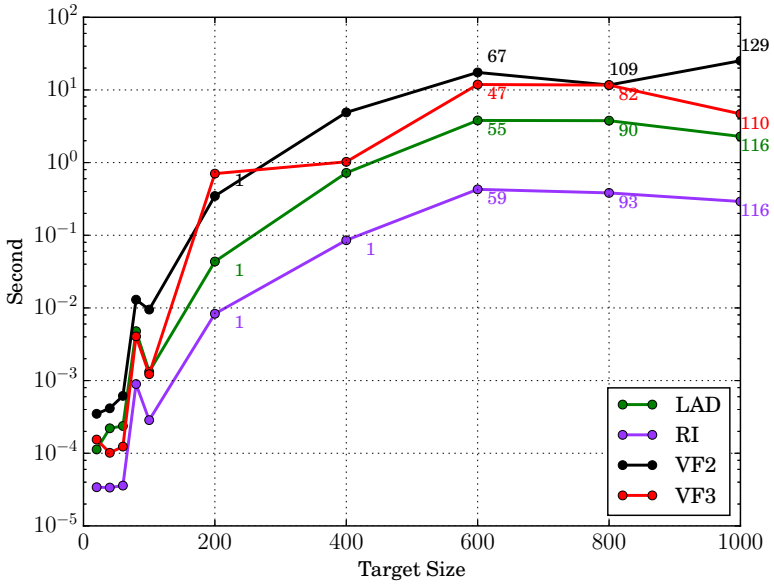
Analyzing the time usage on the MIVIA dataset, RI appears to outperform the other algorithms. But, entering into details and considering the number of unsolved instances, it is clearly a partial conclusion. Indeed, despite VF3 is slower then RI, it is able to solve more instances. The low time usage shown by RI is due to its simpler state consistency check, that differently from VF3 does not use any kind of look-ahead. It is based only on checking the semantic and structural feasibility of a new state. On the one hand, without using the look-ahed, RI is able to save time in processing a single state, but, on the other hand, it performs worse than VF3 when the the graphs become more dense and large. This behaviour is more evident by analyzing the results on the additional random graphs dataset where VF3 outperforms RI on very large and dense graphs.

As previously highlighted, on unlabeled graphs, VF3 is penalized by the fact that the classification function does not provide any benefit, because it takes into account only semantic informa-

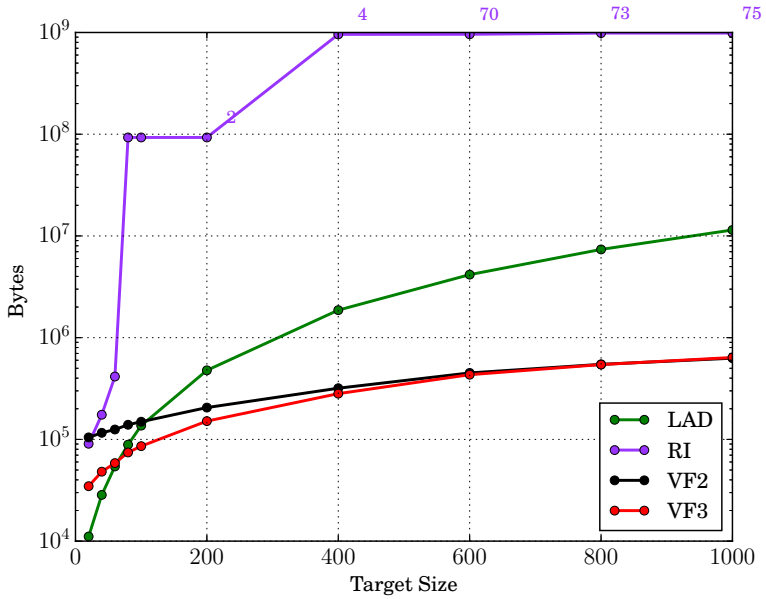
tion. In addition, the sorting procedure can use only the structure of the graphs. Nevertheless, VF3 outperforms its predecessors; the difference in time usage is generally more than one order of magnitude and it always solves more instances. Compared with the other two competitors, VF3 is not the faster but it is very close to RI and is generally able to solves matchings that the others are not. On labeled graphs, VF3 can rely on very strong look-ahead rules and a more selective procedure to generate new states. It results in a wider gap with respect to VF2. The distance between RI and VF3 is reduced and the latter is even now able to solve more instances than the former, especially with a non uniform labelling.

A noteworthy strong point of VF3 is the memory usage. Similarly to VF2, its space complexity is linear and its behavior is stable irrespective of kind of graphs it is facing. Their memory usage depends only on the size. The main motivation lies into the depth-first approach together with an optimized structure that allow both VF2 and VF3 to share many of the information used to explore the search space. But, as already discussed, differently to VF2, VF3 needs an additional space to manage classes and strengthen the feasibility rules. Even so, VF3 and VF2 have almost the memory usage and generally on small graphs VF3 requires less memory. We can justify the advantage of VF3 with respect to the other competitors, especially to VF2, as due to the reduced number of states explored.

Observing the results achieved VF3, has demonstrated to be suitable in different situations, especially on large and dense graphs, and able to compete with the algorithms in the state of the art. And it is clear that, in most of the cases, VF3 outperforms VF2 in time and memory. Therefore, we can claim that it is ready to inherit the position of VF2 in the state of the art.

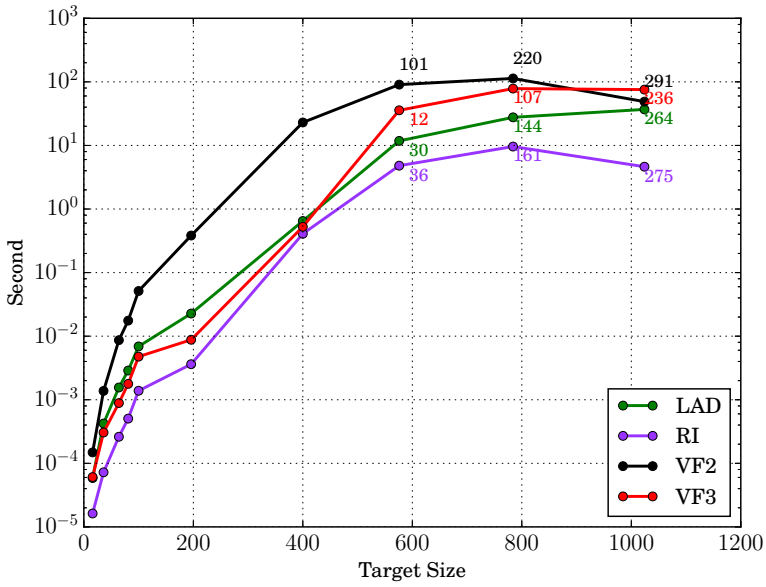


(a) Time usage.

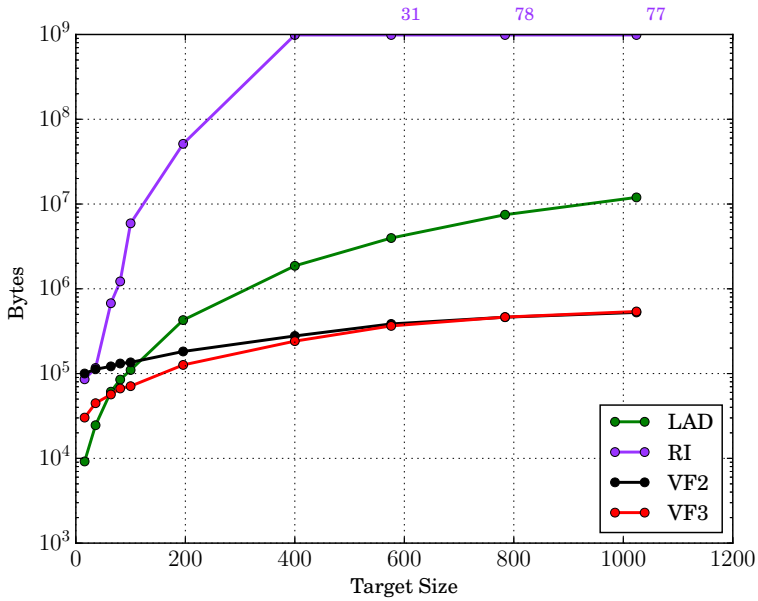


(b) Memory usage.

Figure 3.9: Time and memory usage on unlabelled bounded valence graphs. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

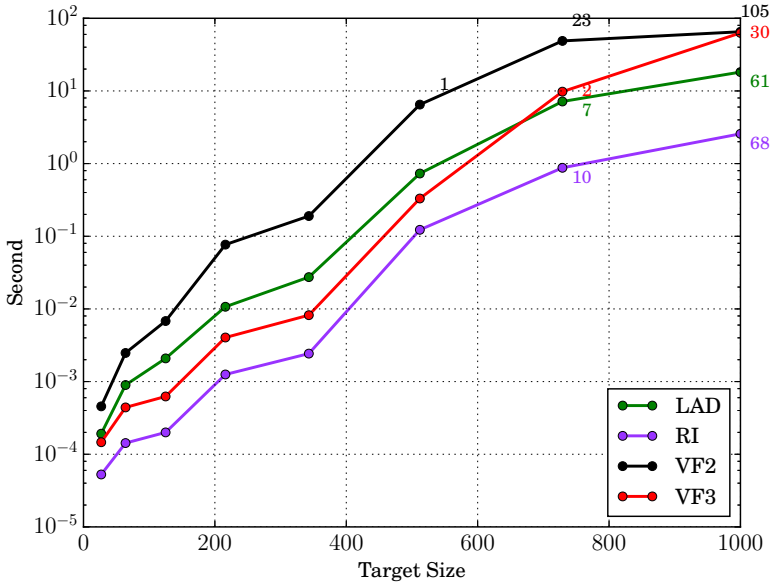


(a) Time usage.

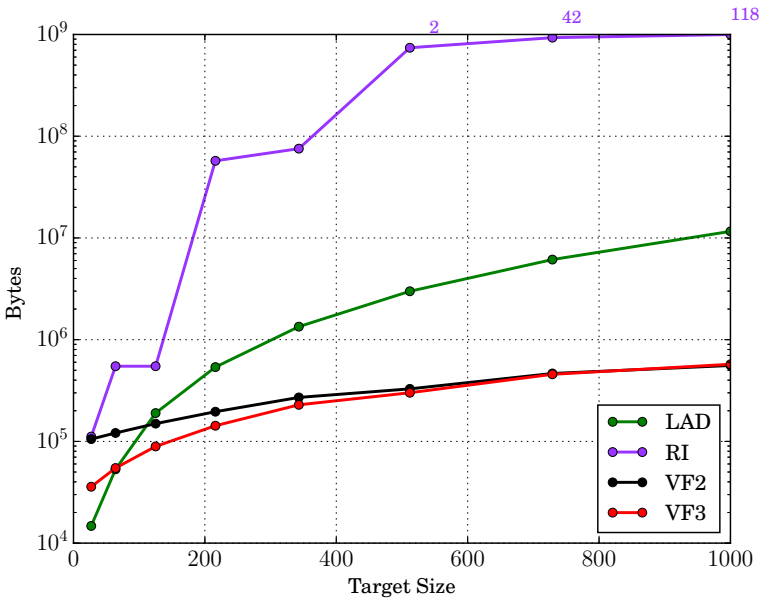


(b) Memory usage.

Figure 3.10: Time and memory usage on unlabelled 2D open meshes. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

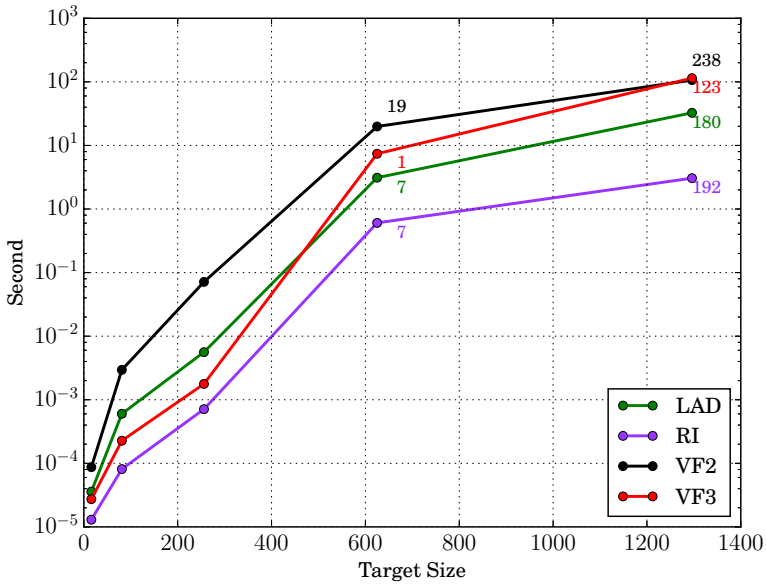


(a) Time usage.

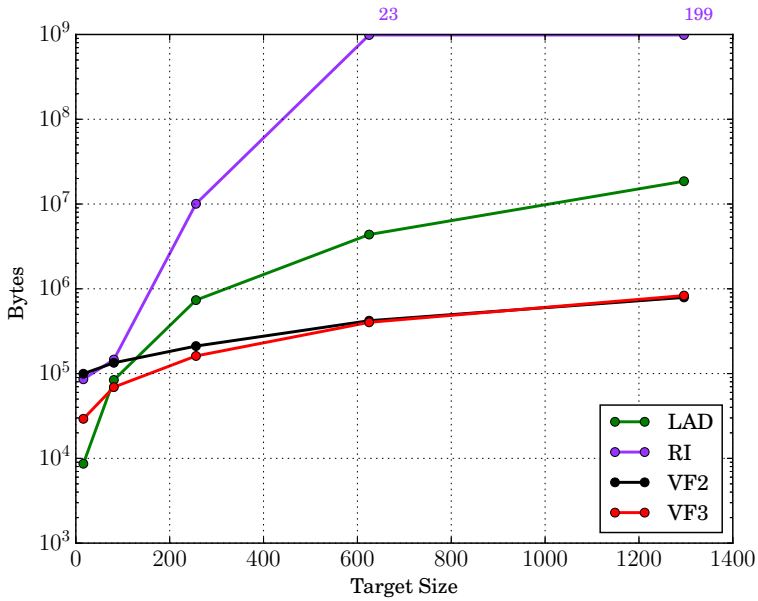


(b) Memory usage.

Figure 3.11: Time and memory usage on unlabelled 3D open meshes. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

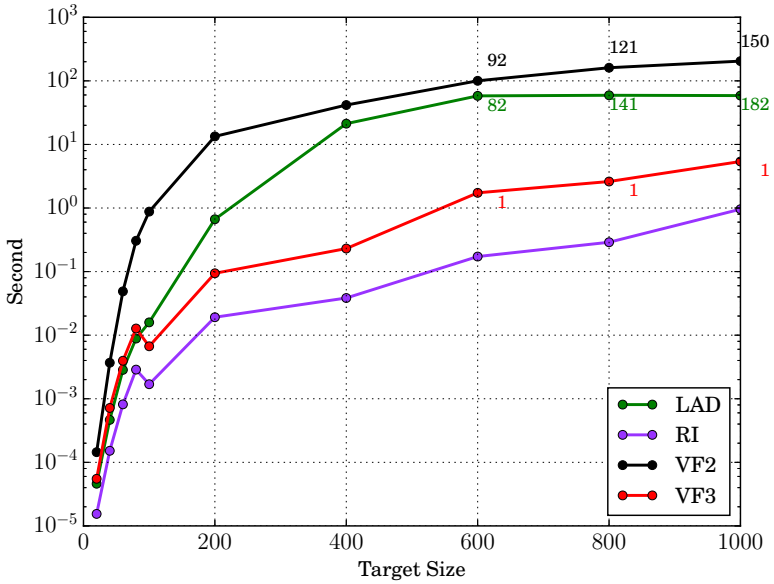


(a) Time usage.

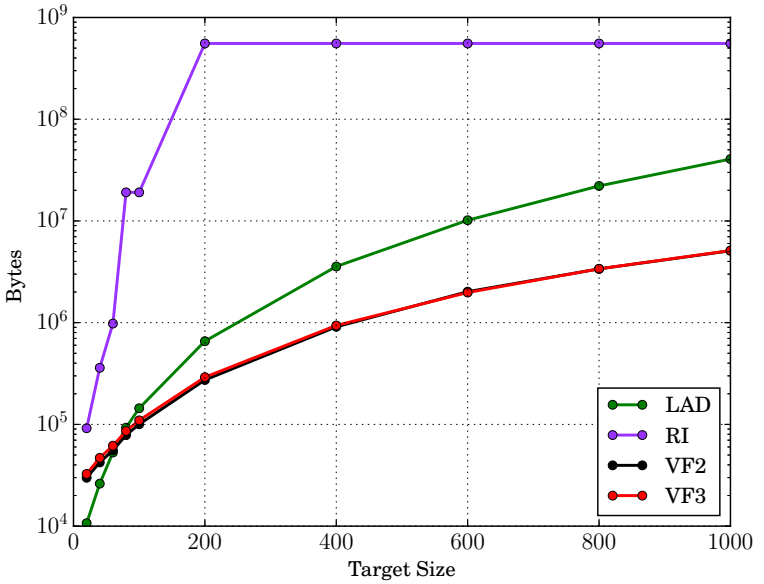


(b) Memory usage.

Figure 3.12: Time and memory usage on unlabelled 4D open meshes. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

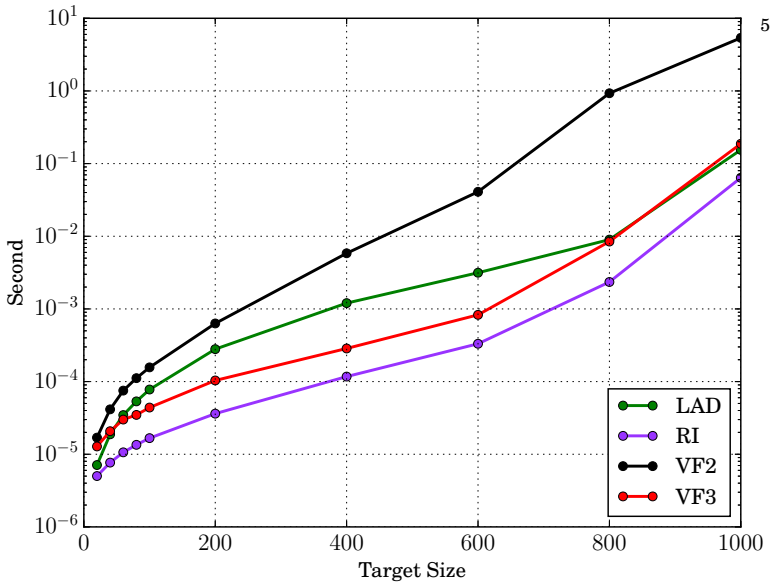


(a) Time usage.

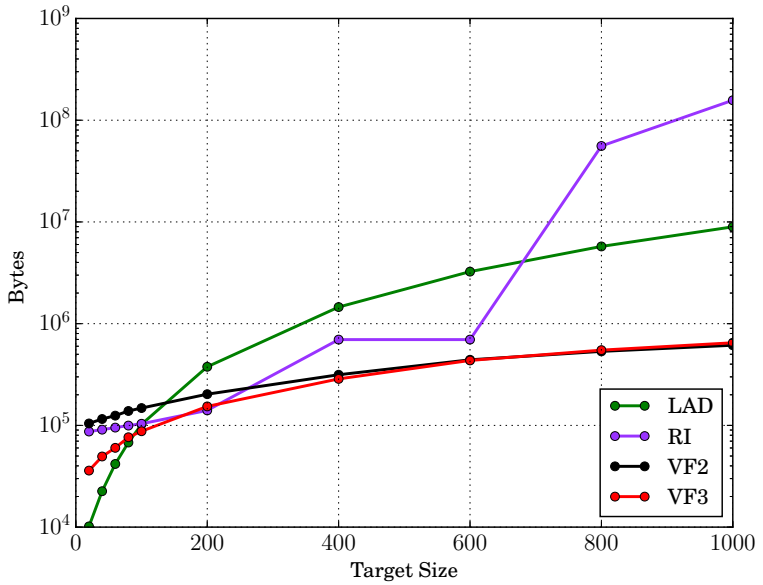


(b) Memory usage.

Figure 3.13: Time and memory usage on unlabelled random graphs. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

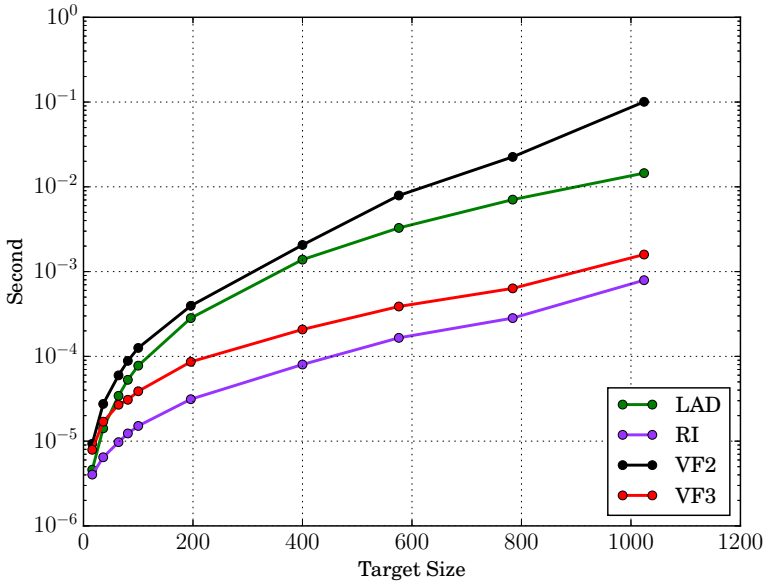


(a) Time usage.

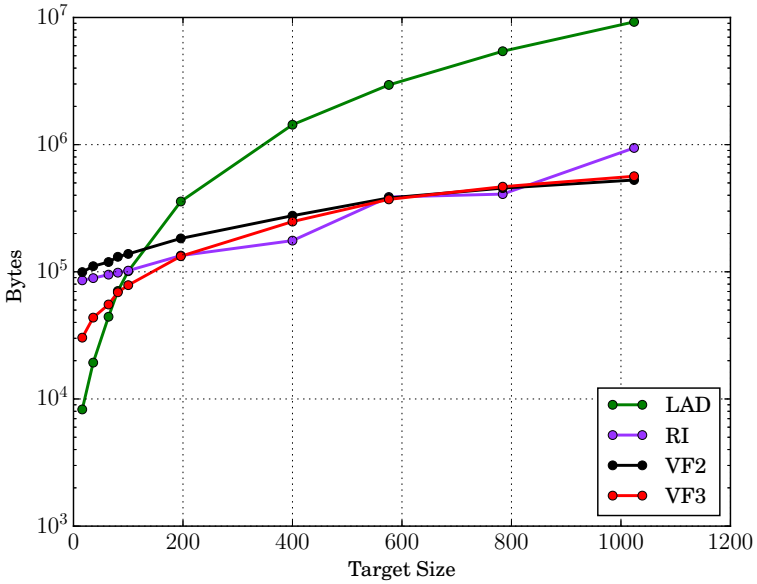


(b) Memory usage.

Figure 3.14: Time and memory usage on bounded valence graphs with 8 labels uniformly assigned. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

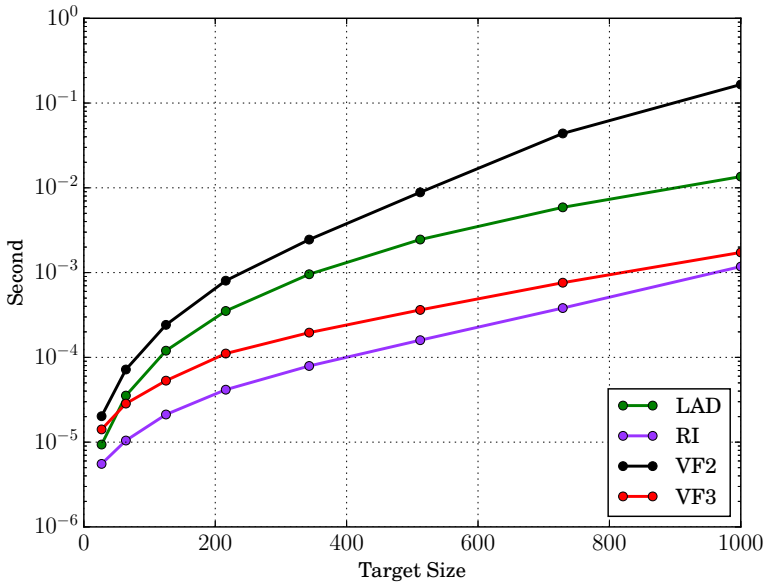


(a) Time usage.

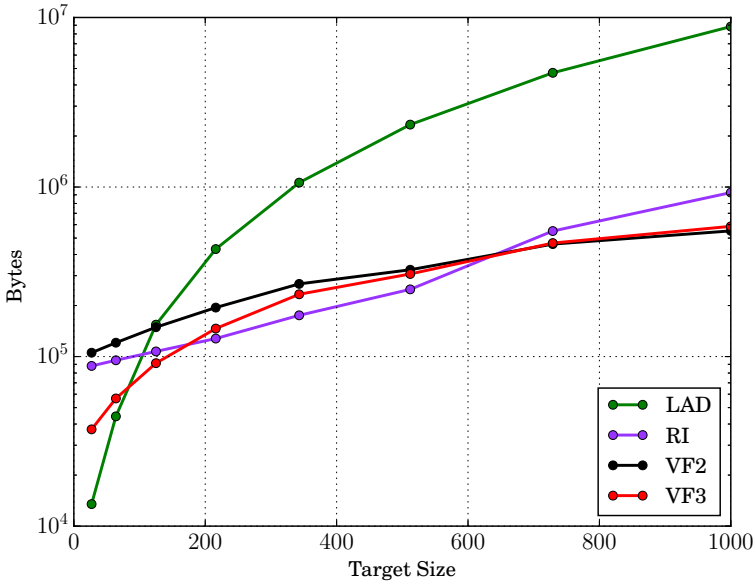


(b) Memory usage.

Figure 3.15: Time and memory usage on 2D open meshes with 8 labels uniformly assigned. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

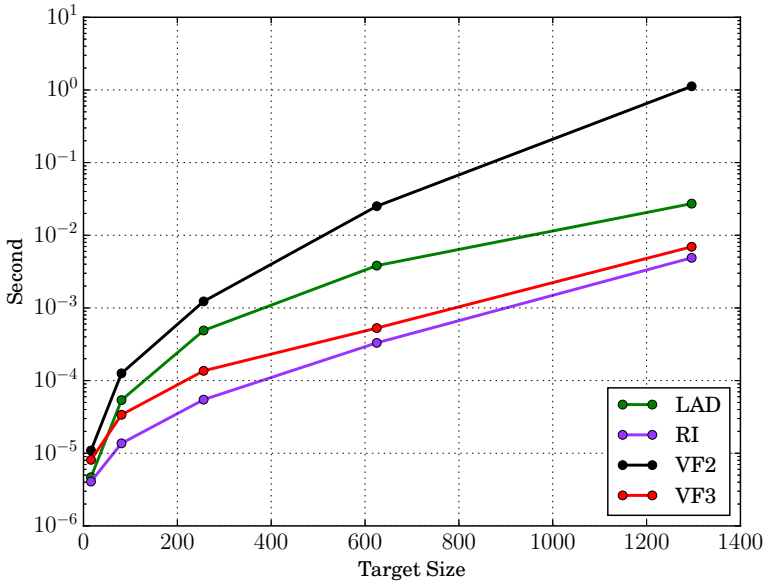


(a) Time usage.

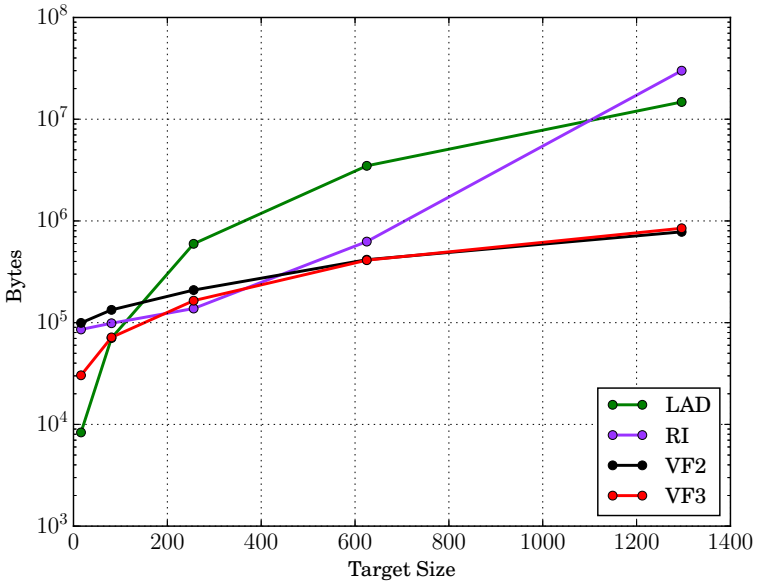


(b) Memory usage.

Figure 3.16: Time and memory usage on 3D open meshes with 8 labels uniformly assigned. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

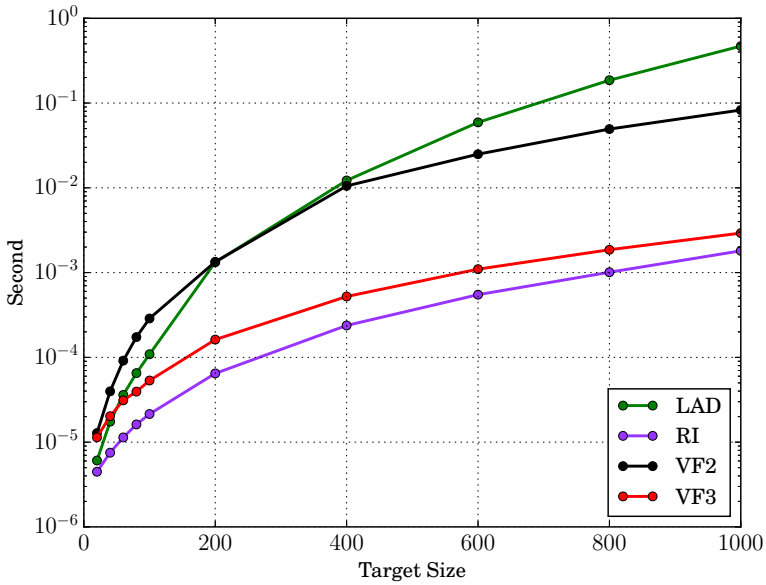


(a) Time usage.

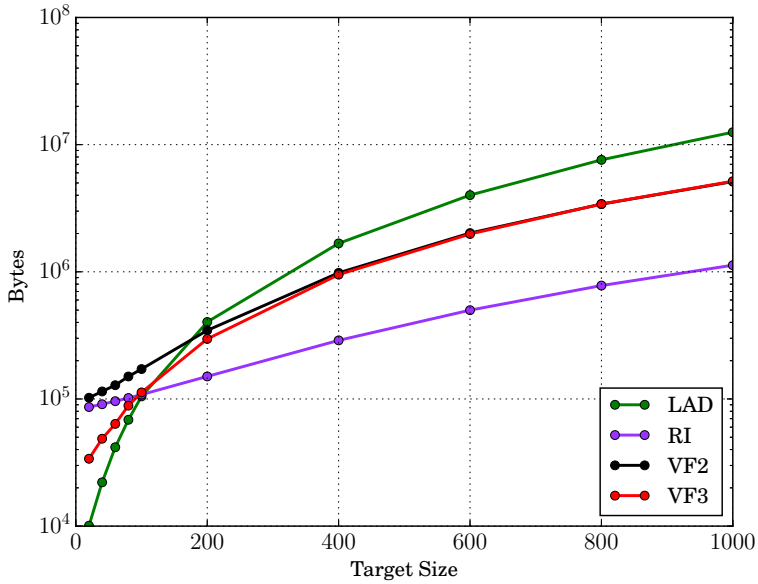


(b) Memory usage.

Figure 3.17: Time and memory usage on 4D open meshes with 8 labels uniformly assigned. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

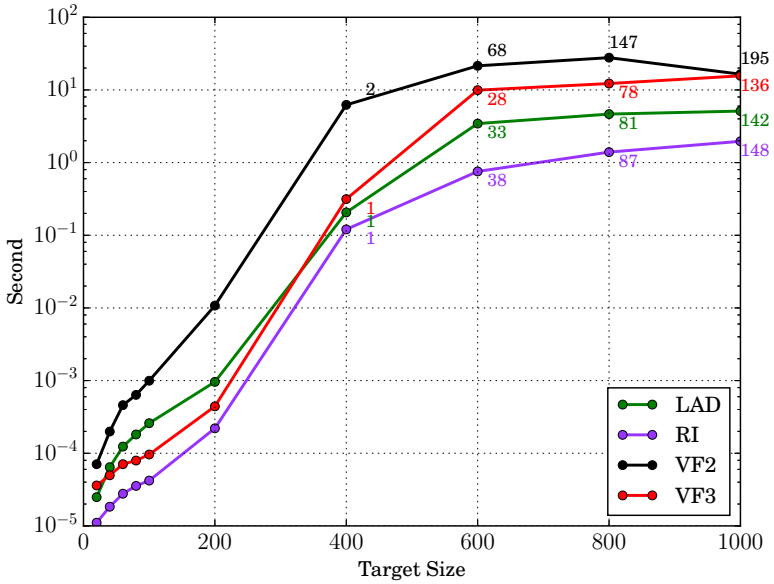


(a) Time usage.

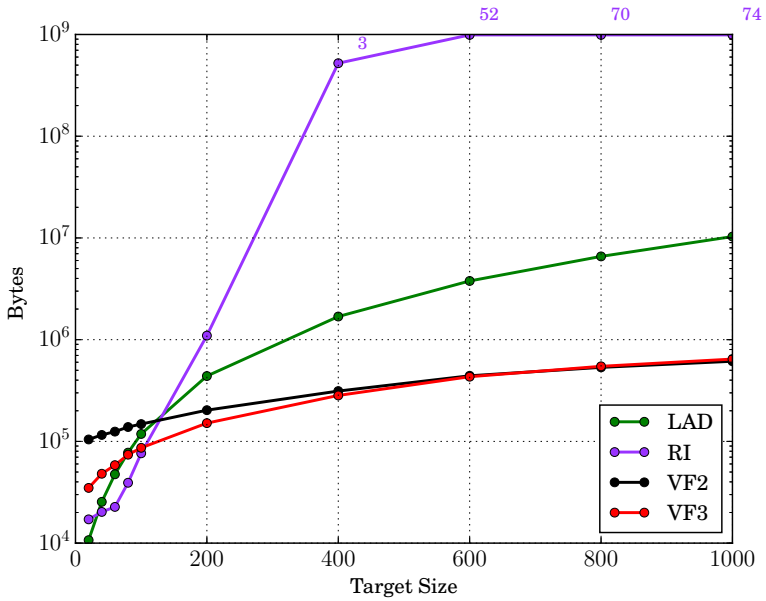


(b) Memory usage.

Figure 3.18: Time and memory usage on random graphs with 8 labels uniformly assigned. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

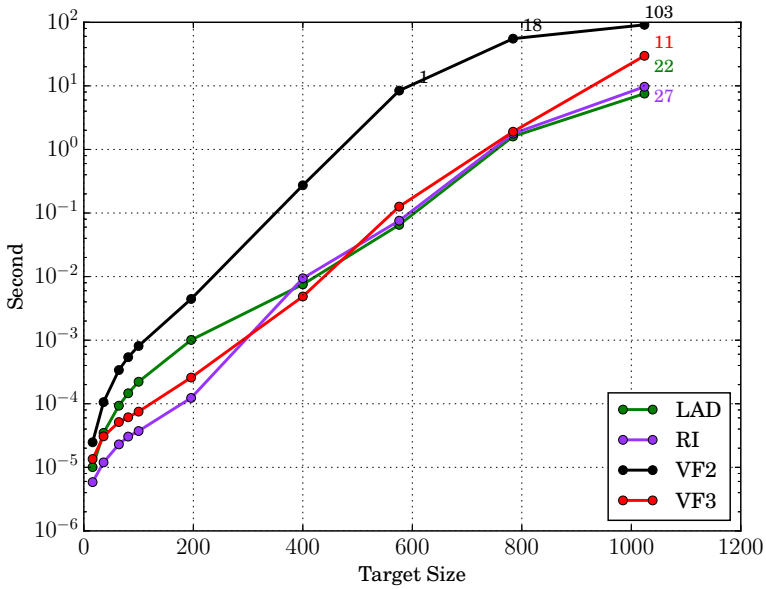


(a) Time usage.

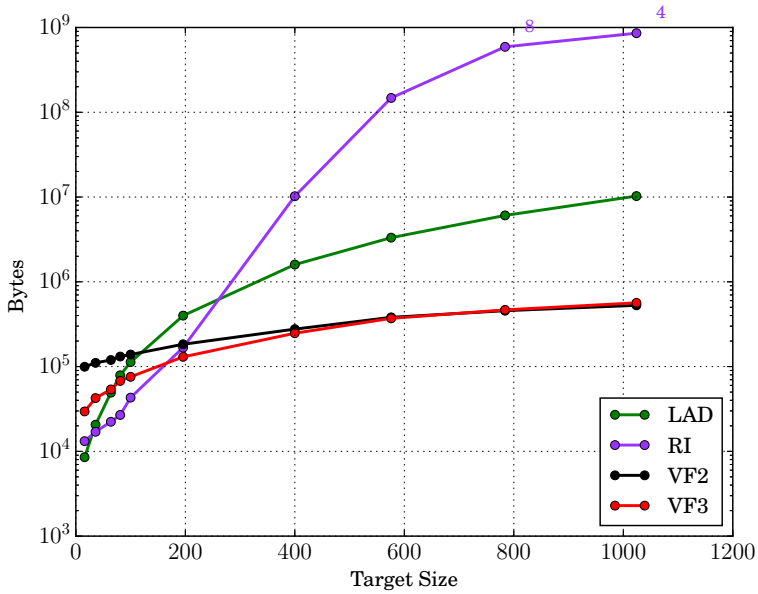


(b) Memory usage.

Figure 3.19: Time and memory usage on bounded valence graphs with 8 labels non uniformly assigned. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

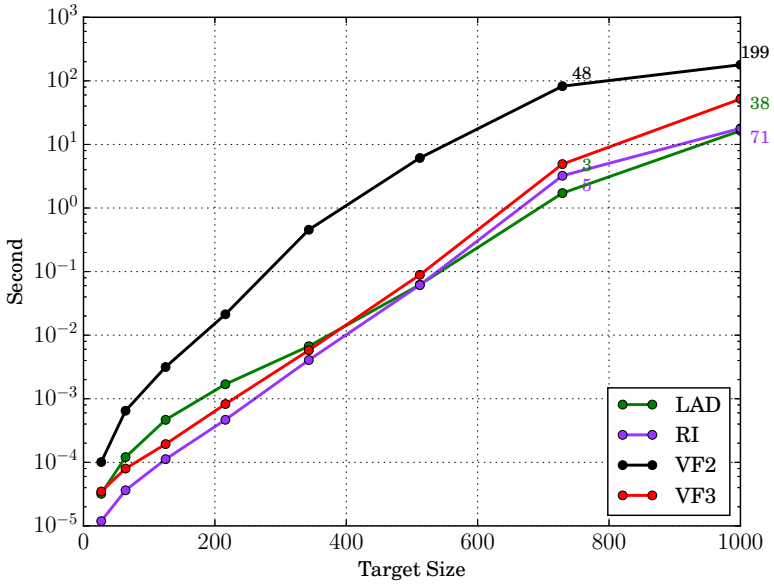


(a) Time usage.

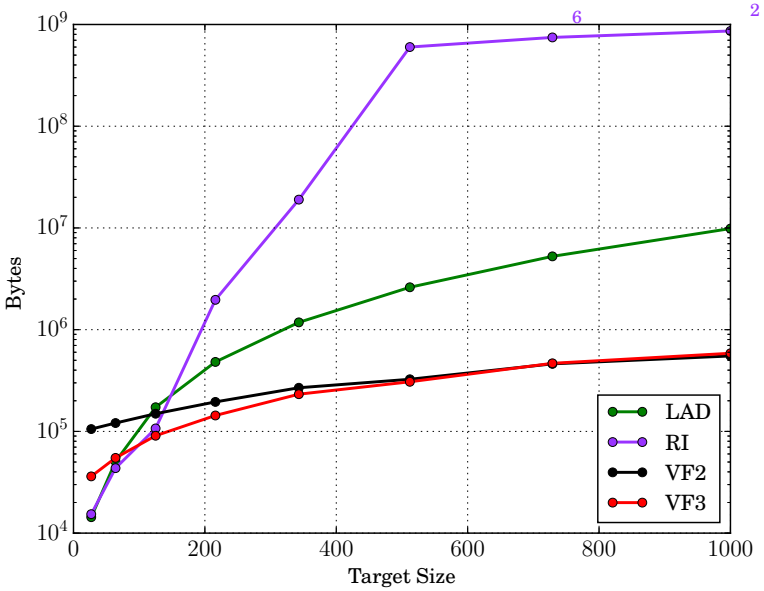


(b) Memory usage.

Figure 3.20: Time and memory usage on 2D open meshes with 8 labels non uniformly assigned. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

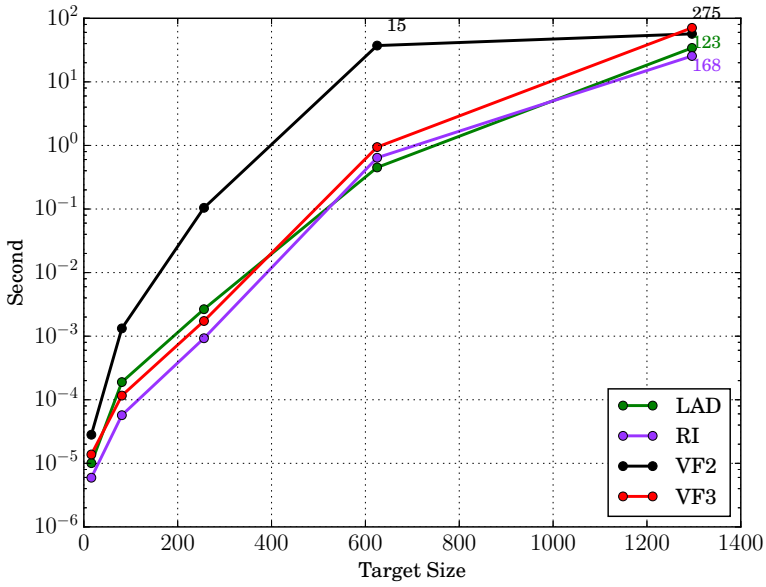


(a) Time usage.

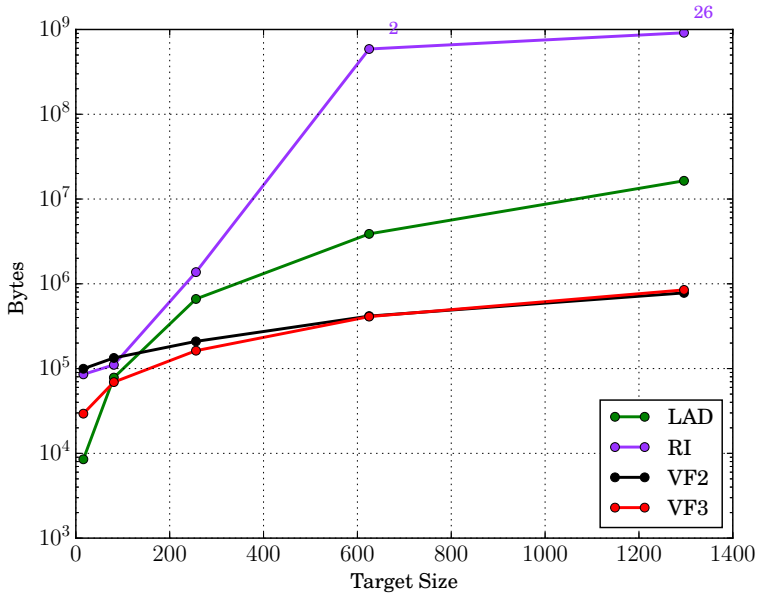


(b) Memory usage.

Figure 3.21: Time and memory usage on 3D open meshes with 8 labels non uniformly assigned. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

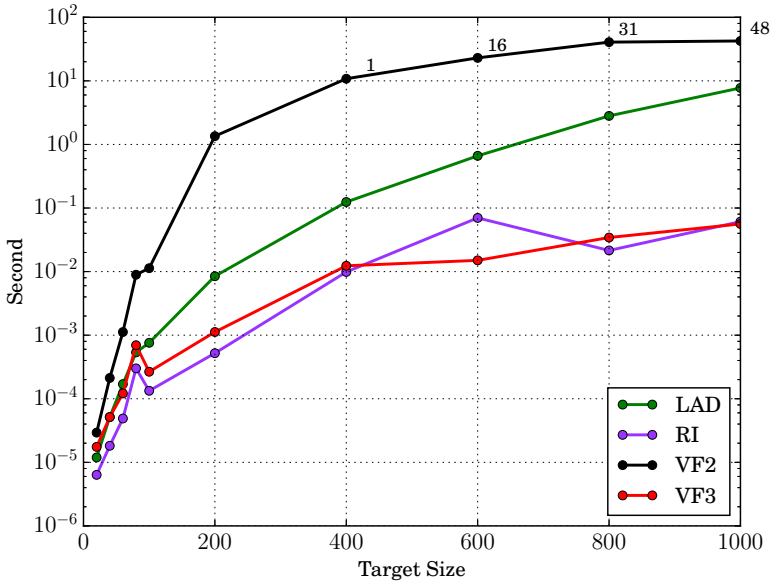


(a) Time usage.

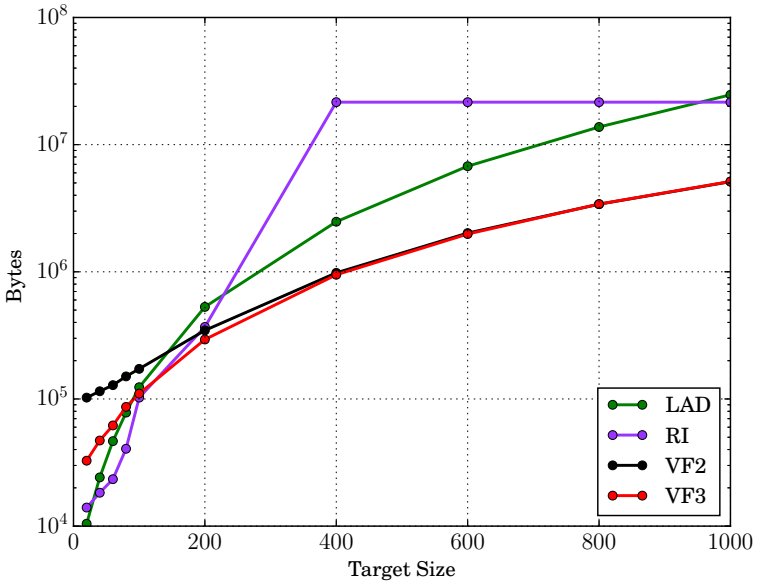


(b) Memory usage.

Figure 3.22: Time and memory usage on 4D open meshes with 8 labels non uniformly assigned. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

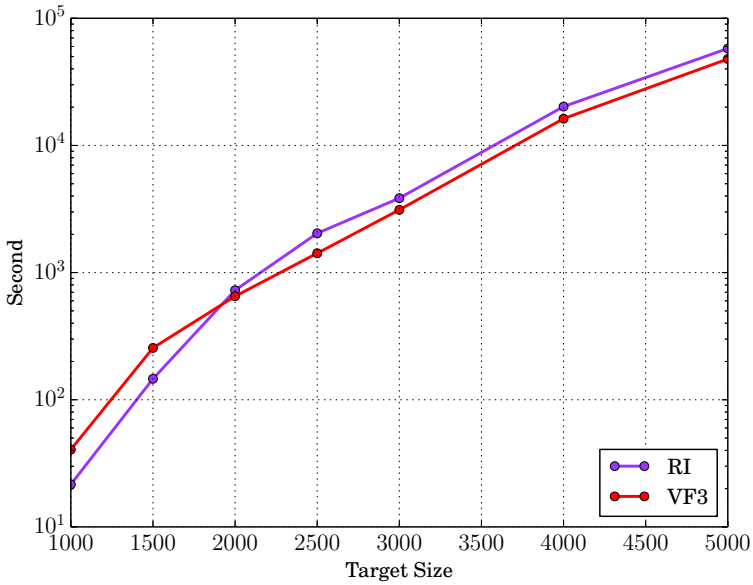


(a) Time usage.

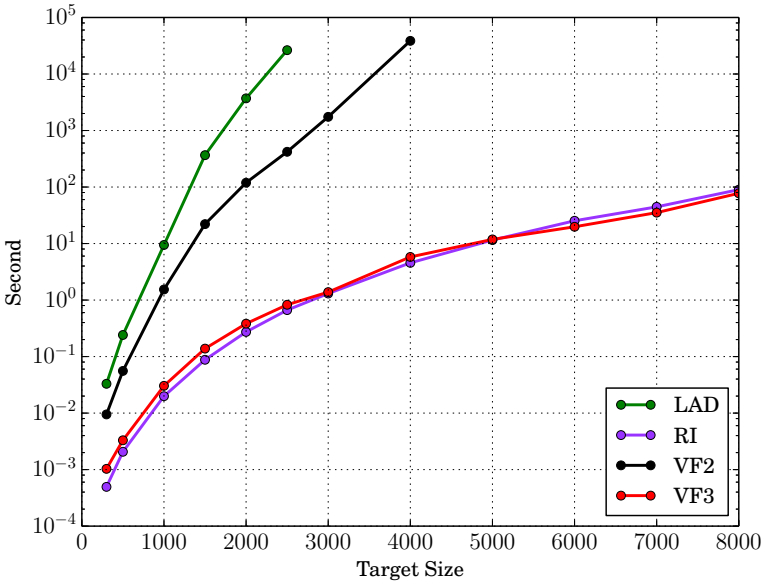


(b) Memory usage.

Figure 3.23: Time and memory usage on random graphs with 8 labels non uniformly assigned. Note that the figures are annotated with additional labels. They represents the number of instances, for a given target size, that the algorithm has not been able to solve due to the limits we have set in memory and time. If a size is not labelled then all the instances have been solved.

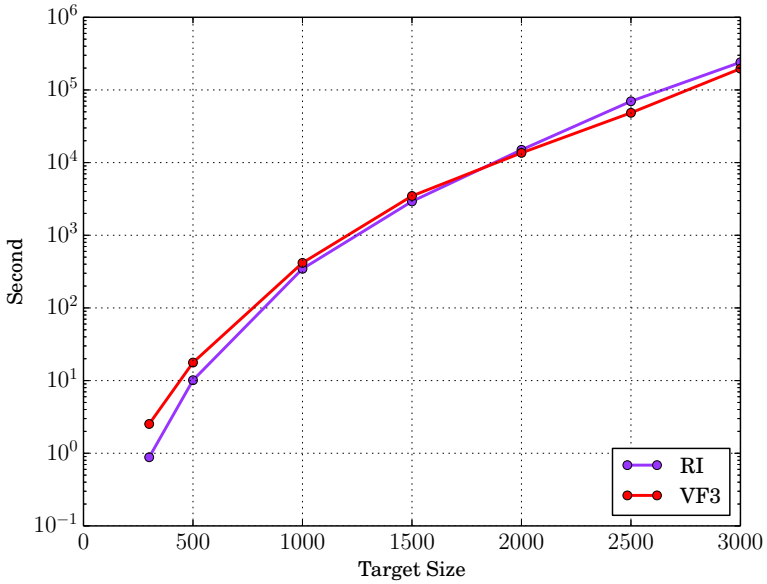


(a) Unlabelled Random Graphs with $\eta = 0.2$.

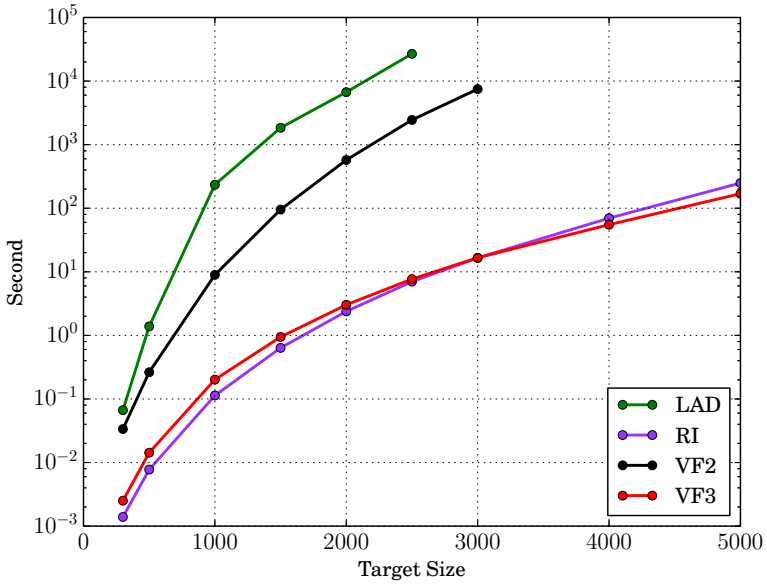


(b) Labelled Random Graphs with $\eta = 0.2$.

Figure 3.24: Time usage on random graphs. Note that on unlabelled graphs VF2 and Lad are not plotted becouse we were not able to get measures.

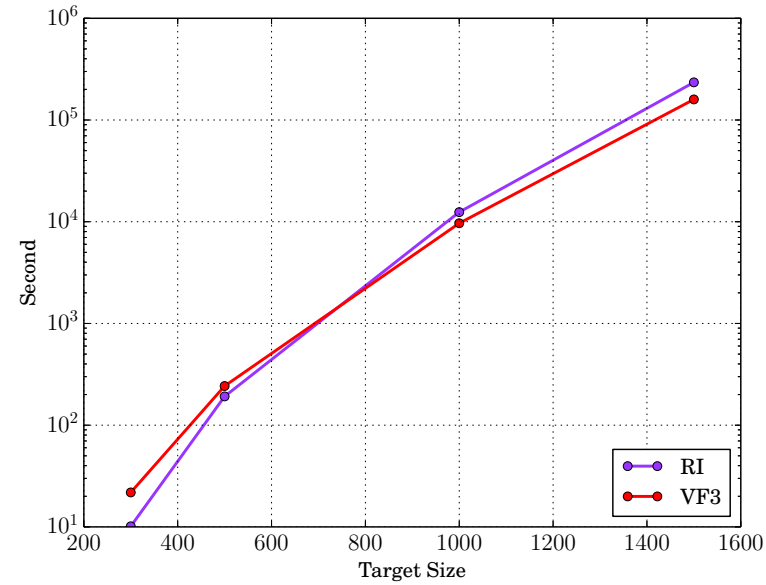


(a) Unlabelled Random Graphs with $\eta = 0.3$.

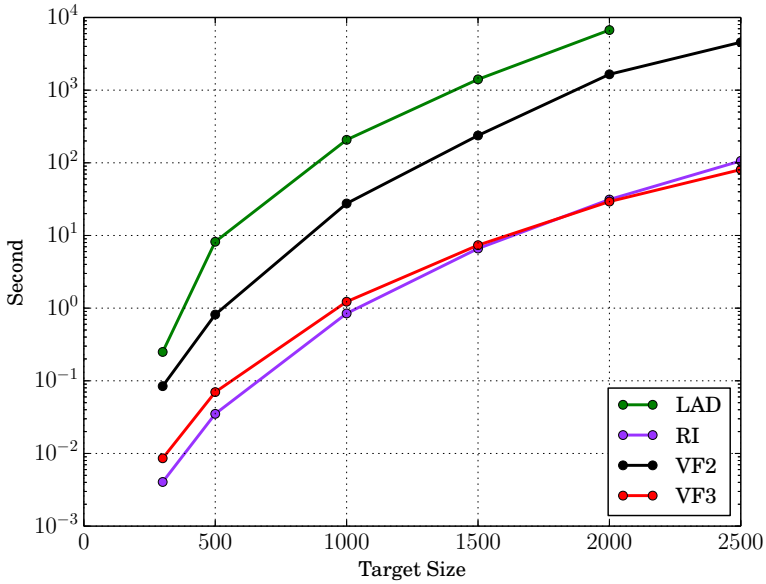


(b) Labelled Random Graphs with $\eta = 0.3$.

Figure 3.25: Time usage on random graphs. Note that on unlabelled graphs VF2 and Lad are not plotted becouse we were not able to get measures.

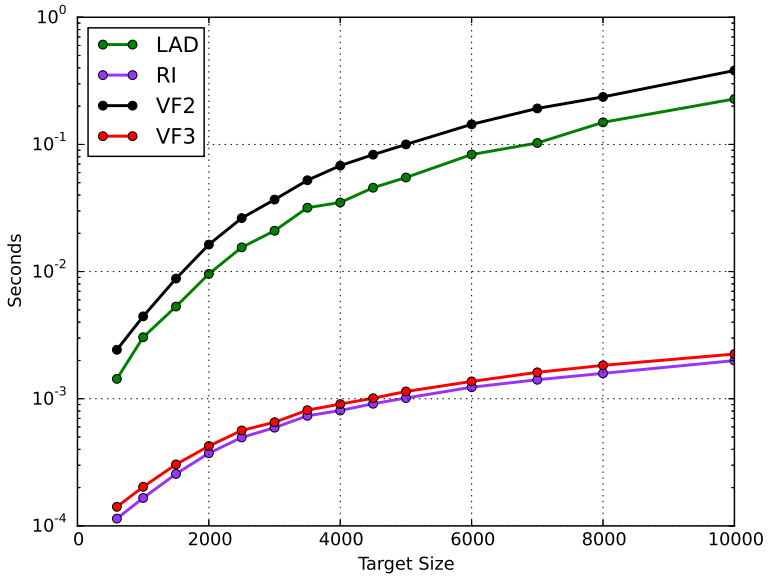


(a) Labelled Random Graphs with $\eta = 0.4$.

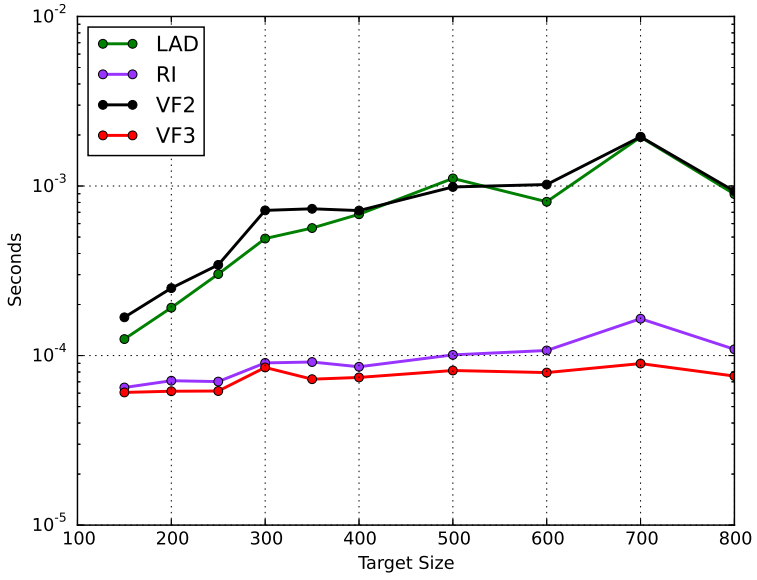


(b) Unlabelled Random Graphs with $\eta = 0.4$.

Figure 3.26: Time usage on unlabelled random graphs. Note that on unlabelled graphs VF2 and Lad are not plotted becouse we were not able to get measures.



(a) Proteins.



(b) Contact Maps.

Figure 3.27: Time usage on biological graphs representing respectively proteins structures (a) and proteins contact maps (b).

Chapter 4

Similarity by Graph Edit Distance

"My big thesis is that although the world looks messy and chaotic, if you translate it into the world of numbers and shapes, patterns emerge and you start to understand why things are the way they are."
- Marcus du Sautoy

4.1 Introduction

Graph edit distance (GED) is the most employed error tolerant method to compute the similarity between two graphs; it is frequently used when the aim is to apply statistical pattern recognition methods, such as LVQ, K-NN and so on, directly on graphs. But, computing the exact value of GED is a NP-hard problem that is commonly addressed by using A^*, a search algorithm whose complexity is exponential and then unsuitable for large graphs.

Even though, the complexity required to compute the GED is unsuitable for large graphs, in most of real world problems an approximation is often sufficient and, under certain conditions, it can be computed in a polynomial time. For this reason, the interest of the scientific community has been focused in defining efficient methods that provide efficient approximations of GED.

In particular, a recent successful method proposed by K.Riesen and H.Bunke [6, 7] approximates the GED by solving a *weighted bipartite matching problem*. Moreover, several improvements and applications of this methods have been proposed in the last years [38, 39, 40, 41, 42, 43, 43, 44].

In the following sections we will discuss this approach in details because it represents the state of the art and is the starting point of our method.

4.1.1 Linear Sum Assignment Methods

The graph edit distance between two graphs can be approximated by solving a weighted bipartite matching problem. The reason of this lies into the fact that an edit path may be deduced from a sequence of edit operations applied on nodes only. As we will see in the following sections, this is possible because of the relationship between assignments and some particular kinds of edit paths, i.e. the *restricted edit paths*. Indeed, any mapping between the nodes of two graphs induces an edit path which substitutes all mapped nodes together with all their incident edges and inserts or removes the non-mapped nodes/edges. Conversely, given an

edit path between two graphs, such that each node and each edge is substituted only once, one can define a mapping between the substituted nodes and edges of both graphs.

Therefore, with the intent to provide an introduction to the framework that will be detailed below, let us discuss how the GED is computed as a weighted bipartite matching (see Figure 4.3). First, the nodes of two graphs G_1 and G_2 are arranged in two different node sets of a bipartite graph, then the aim is to find the optimal assignment between the two sets. Each mapping between two nodes represents an edit operation of node addition, deletion or substitution on G_1 and is penalized by a non negative cost. The cost of the assignment is, then, defined as the sum of the costs of its corresponding mappings. Under mild assumptions, starting from the optimal node assignment it is possibile to deduce, in a non ambiguous way, an edit path. The cost of this path is an approximate GED. In order to understand where is the approximation, it is important to point out that the operations on the edges are not directly considered. But, they are generally included in the cost of mapping two nodes. Because of this, the cost of the optimal assignment is an approximation of the real one, hence the edit path deducted is not minimal but short and provides an overestimated GED.

The method proposed by K.Riesen and H.Bunke [6, 7] faces the weighted bipartite matching problem as a Linear Sum Assignment Problem (LSAP) by using a polynomial time solver, such as the Hungarian Algorithm [45, 10]. In order to define the cost to map two nodes, a bag of patterns is attached to each node; it takes into account both the label and the structure around it. Each possible substitution is then penalized by a cost that measures the affinity between the bags. Similarly, node removals and insertions are penalized by a cost measuring the importance of the bag.

So that, the definition of the bags of patterns is a key point, as well as the associated measure of affinity. Incident edges have been initialy proposed in [6, 7], and the cost between two nodes (or bags) is itself defined as the cost of the linear sum assignment of the patterns within the bags, following the same framework as

the one defined for the nodes. The cost of substituting, removing and inserting the patterns depends on the original cost function used to penalize the edit operations.

Although the bipartite GED provides a good approximation it is still an overestimation. As shown by several works, the overestimation can only be marginally reduced, for instance by considering more global information than the one supported by incident edges [40, 44] (see Figure 4.1), or by modifying the resulting edit path by means of genetic algorithms [41], see [43] for more details. These methods provide an interesting compromise between time complexity and approximation quality, but they are inherently limited to compute linear approximations of the GED.

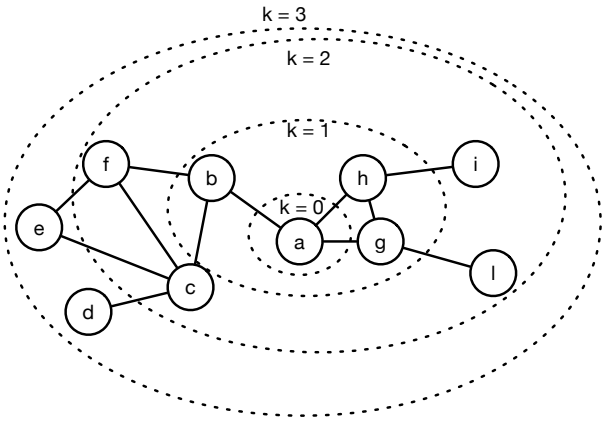


Figure 4.1: Example of local structure of a node used in [44] as bag of patterns. The cost to map two nodes is the exact edit distance between their local structures of size k. The parameter k represents the distance from the considered node in terms of path length. Note that, in the current example, when $k = 3$, the local structure of the node is the whole graph. This is a extreme situation where the entire structure of the graph has been considered to compute the distance.

4.1.2 A Quadratic Assignment Approach

A complete representation of the GED as a bipartite matching can be obtained only by considering both node and edge assignments

simultaneously. Indeed, operations on edges can only be deduced from operations performed on their two incident nodes. For instance, an edge can be substituted to another one only if its incident nodes are substituted. This pairwise constraint on nodes is closely related to the one involved in graph matching. It is known that graph matching problems, and more generally problems that incorporate pairwise constraints, can be cast as a quadratic assignement problem (QAP) [46, 16, 47, 48, 10]. QAPs are NP-hard and so different relaxation algorithms have been proposed to find an approximate solution, such as Soft-Assign [49, 50, 51], Integer Projected Fixed Point (IPFP) [52], or Graduated NonConvexity and Concavity Procedure [53]. Even if computing the GED is generally not equivalent to solve a graph matching problem, it may also be formalized as a QAP. So far, this aspect has only been considered through the definition of fuzzy paths by [54]. Thus, the strong relationships between the GED and the QAP have not yet been fully analyzed.

4.1.3 Chapter overview

In this chapter, we will describe in details how we have extended the LSAP with insertions and removals [6, 7] to a quadratic problem. To this aim, in Chapter 2 we have provided some established results about edit paths. These results are used to formalize the relation between the LSAP (Section 4.3) or the QAP (Section 4.4), and the edit paths. In particular, in the next sections, we will show that, under the assumption of working with restricted edit paths, the GED may be computed by solving a QAP. Then, in Section 4.5, we will describe two algorithms commonly used to solve the QAP, Soft-Assign and IPFP, and our improved version of IPFP algorithm adapted to the minimization of quadratic functionals to approximate the GED. Finally, in Section 4.6, we will validate experimentally our approach and we will show that it generally provides a more accurate approximation than the state of the art.

4.2 Edit paths and assignments

As introduced before, the approach proposed by Riesen and Bunke [7] mainly consists in finding an optimal assignment between the sets of nodes of the graphs according to a given set of edit operation costs. This problem can be faced as a linear sum assignment problem (LSAP) where the sets to be mapped correspond to the two node sets respectively of G_1 and G_2.

The original definition of LSAP consists in mapping two sets having the same number of elements. Trivially transposing LSAP to the assignment of graphs will restrict the application of such approach to graphs having the same size. However, in order to compare graph in a more general way, the formulation of LSAP must include the removal and the insertion of elements between the two sets. Such operations on elements then correspond to insertion and deletion of nodes within the edit path.

Hence, let $\mathcal{X}$ and $\mathcal{Y}$ be two finite sets, with $|\mathcal{X}| = n$ and $|\mathcal{Y}| = m$. Without loss of generality, we assume that $\mathcal{X} = \{1, \ldots, n\}$ and $\mathcal{Y} = \{1, \ldots, m\}$. Each element of $\mathcal{X}$ can be assigned to an element of $\mathcal{Y}$. Such a mapping represents a possible substitution. Also each element of $\mathcal{X}$ can be removed, and each element of $\mathcal{Y}$ can be inserted into $\mathcal{X}$. In order to represent insertions, $\mathcal{X}$ is augmented by m dummy elements $\mathcal{E}_\mathcal{X} = \{\epsilon_1, \ldots, \epsilon_m\}$, such that j can only be inserted into $\mathcal{Y}$ by assiging ϵ_j to j. Similarly, the set $\mathcal{Y}$ is augmented by n dummy elements $\mathcal{E}_\mathcal{Y} = \{\epsilon_1, \ldots, \epsilon_n\}$, such that $i \in \mathcal{X}$ is removed by assigning it to ϵ_i. In other terms, it is not possible to assign an element $i \in \mathcal{X}$ to an element $\epsilon_k \in \mathcal{E}_\mathcal{Y}$ with $k \neq i$, and similarly any assignment from $\epsilon_j \in \mathcal{E}_\mathcal{X}$ to $k \in \mathcal{Y}$ with $k \neq j$ is forbidden.

Let $\mathcal{X}^\epsilon = \mathcal{X} \cup \mathcal{E}_\mathcal{X}$ and $\mathcal{Y}^\epsilon = \mathcal{Y} \cup \mathcal{E}_\mathcal{Y}$ be the two augmented sets, which thus have the same size $n + m$. We assume without loss of generality that symbols ϵ_i and ϵ_j represent integers, *i.e.* $\mathcal{E}_\mathcal{X} = \{n + 1, \ldots, n + m\}$ and $\mathcal{E}_\mathcal{Y} = \{m + 1, \ldots, m + n\}$. It is now possible to define assignments that take into account removal, substitution, and insertion of elements. In the following sections we better formalize the relationship between edit paths and assignments in order to compute the GED as an assignment problem.

It worths to point out that in order to make the Section easier to read, propositions and corollaries have been provided without proofs. However, they may be found in [55].

4.2.1 Independent edit path

An independent edit path between two labeled graphs G_1 and G_2 is an edit path such that:

1. No node nor edge is both substituted and removed,
2. No node nor edge is simultaneously substituted and inserted,
3. Any inserted element is never removed,
4. Any node or edge is substituted at most once,

An important thing to note is that an independent edit path is not minimal in the number of operations. Indeed, it is still allowed to replace one substitution by one removal followed by one insertion (but such an operation can be performed only once for each node or edge thanks to condition 3). However, we forbid useless operations such as the substitution of one node followed by its removal (condition 1) or the insertion of a node with a wrong label followed by its substitution (condition 2). In the following we will only consider independent edit paths that we simply call edit paths.

Proposition 1. *The elementary operations of an independent edit path between two graphs G_1 and G_2 may be ordered into a sequence of removals, followed by a sequence of substitutions and terminated by a sequence of insertions.*

Proposition 2. *Let P be an edit path between two graphs G_1 and G_2. Let us further denote by R, S and I the sequence of node and edge Removals, Substitutions and Insertions performed by P, the order in each sequence being deduced from the one of P (see Figure 4.2). Then:*

- *the graph $\hat{G}_1$ obtained from G_1 by applying removal operations R is a sub graph of G_1,*
- *the graph $\hat{G}_2$ obtained from G_1 by applying the sequence of operations (R, S) is a sub graph of G_2,*
- *Both $\hat{G}_1$ and $\hat{G}_2$ correspond to a same common structural sub graph of G_1 and G_2.*

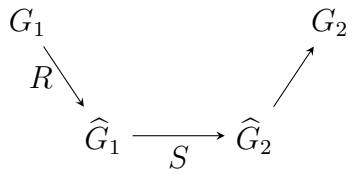


Figure 4.2: Edit path of Proposition 2.

One should note that it may exist several structural isomorphisms between $\hat{G}_1$ and $\hat{G}_2$. The set of substitutions S fixes one of them, say f such that the image of any element of $\hat{G}_1$ by f have the same label than the one defined by the substitution. More precisely, let us suppose that we enlarge the set of substitutions S by 0 cost substitutions so that all the nodes and edges of $\hat{G}_1 = (\hat{V}_1, \hat{E}_1, \mu_1, \nu_1)$ are substituted. In this case, we have:

$$\begin{cases} \forall v \in \hat{V}_1, & \mu_2(f(v)) = l_v \\ \forall e \in \hat{E}_1, & \nu_2(f(e)) = l_e \end{cases}$$

Where l_v and l_e correspond to the labels defined by the substitutions of v and e and μ_2 and ν_2 define respectively the node and edge labeling functions of G_2.

Corollary 1. *Using the same notations than in Proposition 2, the cost $\gamma(P)$ of an edit path is defined by:*

$$\gamma(P) = \sum_{v \in V_1 \setminus \hat{V}_1} c_{vd}(v) + \sum_{e \in E_1 \setminus \hat{E}_1} c_{ed}(e) + \sum_{v \in \hat{V}_1} c_{vs}(v) + \sum_{e \in \hat{E}_1} c_{es}(e) + \sum_{v \in V_2 \setminus \hat{V}_2} c_{vi}(v) +$$

$\sum_{e \in E_2 \setminus \hat{E}_2} c_{ei}(e)$

Remark 1. *Using the same notations than Proposition 2 if both G_1 and G_2 are undirected we have:*

$$\gamma(P) = \gamma_v(P) + \gamma_e(P)$$

with

$$\begin{aligned}
\gamma_v(P) &= \sum_{i \in V_1 \setminus \hat{V}_1} c_{vd}(i) + \sum_{i \in \hat{V}_1} c_{vs}(i) + \sum_{k \in V_2 \setminus \hat{V}_2} c_{vi}(k) \\
\gamma_e(P) &= \frac{1}{2}\left(\sum_{(i,j) \in \hat{E}_1} c_{es}((i,j)) + \sum_{(i,j) \in E_1 \setminus \hat{E}_1} c_{ed}((i,j)) + \sum_{(k,l) \in E_2 \setminus \hat{E}_2} c_{ei}((k,l)) \right)
\end{aligned}$$

Indeed, if both graphs G_1 and G_2 are undirected both (i,j) and (j,i) belong to E_1 while encoding a single edge e. The removal or the substitution of the edge e is thus counted twice in $\gamma_e(P)$. In the same way (k,l) and (l,k) represent the same edge e of $E_2 \setminus \hat{E}_2$ which is thus inserted twice in $\gamma_e(P)$. The factor $\frac{1}{2}$ of $\gamma_e(P)$ removes this double couting of edge operations.

Corollary 2. *If all costs do not depend on the node/edge involved the cost of an edit path P is equal to:*

$$\begin{aligned}
\gamma(P) &= (|V_1| - |\hat{V}_1|)c_{vd} + (|E_1| - |\hat{E}_1|)c_{ed} + V_f c_{vs} + E_f c_{es} \\
&\quad + (|V_2| - |\hat{V}_2|)c_{vi} + (|E_2| - |\hat{E}_2|)c_{ei}
\end{aligned}$$

where V_f (resp. E_f) denotes the number of nodes (resp. edges) substituted with a non zero cost and c_{vd}, c_{ed}, c_{vs}, c_{es}, c_{vi}, and c_{ei} denote the constant costs of the associated functions.

Moreover, in this case minimizing the cost of the edit path is equivalent to maximizing the following formula:

$$M(P) \stackrel{not.}{=} |\hat{V}_1|(c_{vd} + c_{vi}) + |\hat{E}_1|(c_{ed} + c_{ei}) - V_f c_{vs} - E_f c_{es}$$

Note that similar results have been discussed by Bunke in [56].

4.2.2 Restricted edit path

A restricted edit path is an independent edit path in which any node or any edge cannot be removed and then inserted. The term

restricted should be understood as edit path with the minimal number of operations. It is worth to point out that the cost of a restricted edit path may not be minimal (among all edit paths) if the cost of a removal operation followed by an insertion is lower than the cost of the associated substitution. However, such a drawback may be easily solved by setting a new substitution cost equal to the minimum between the old substitution cost and the sum of the costs of a removal and an insertion. In this case all non-zero cost substitutions, for nodes and edges, may be equivalently replaced by a removal followed by an insertion.

Proposition 3. *If G_1 and G_2 are simple graphs, there is a one-to-one mapping between the set of restricted edit paths between G_1 and G_2 and the set of injective functions from a subset of V_1 to V_2. We denote by φ_0, the special function from the empty set onto the empty set.*

Proposition 4. *Let P be a restricted edit path not associated with φ_0 (hence with some substitutions). Let us denote by $\varphi : \hat{V}_1 \to V_2$ the injective function associated to P and let us denote $\varphi(\hat{V}_1)$ by $\hat{V}_2$. We further introduce the following two sets:*

$$\begin{cases} R_{12} &= \{(i,j) \in E_1 \cap (\hat{V}_1 \times \hat{V}_1) \mid (\varphi(i), \varphi(j)) \notin E_2\} \\ I_{21} &= \{(k,l) \in E_2 \cap (\hat{V}_2 \times \hat{V}_2) \mid (\varphi^{-1}(k), \varphi^{-1}(k)) \notin E_1\} \end{cases}$$

- *The set of substituted/removed/inserted nodes by P are respectively equal to: $\hat{V}_1$, $V_1 \setminus \hat{V}_1$ and $V_2 \setminus \hat{V}_2$.*
- *The set of edges substituted/removed/inserted by P are respectively equal to:*
  - *Removed:* $E_1 \setminus \hat{E}_1 = \Big(E_1 \cap \Big(((V_1 \setminus \hat{V}_1) \times V_1) \cup (V_1 \times (V_1 \setminus \hat{V}_1))\Big)\Big) \cup R_{12}$
  - *Inserted:* $E_2 \setminus \hat{E}_2 = \Big(E_2 \cap \Big(((V_2 \setminus \hat{V}_2) \times V_2) \cup (V_2 \times (V_2 \setminus \hat{V}_2))\Big)\Big) \cup I_{21}$
  - *Substituted:* $\hat{E}_1 = \Big(E_1 \cap (\hat{V}_1 \times \hat{V}_1)\Big) \setminus R_{12}$ *with* $\hat{E}_2 = \varphi(\hat{E}_1) = \Big(E_2 \cap (\hat{V}_2 \times \hat{V}_2)\Big) \setminus I_{21}$

4.2.3 ϵ-assignment

An importart concept required to discuss the relationships between graph edit distance and bipartite graph assignment problems is the ϵ-assignment. It represents a bijective mapping $\psi : \mathcal{X}^\epsilon \to \mathcal{Y}^\epsilon$ (see Section 4.2), here a permutation, such that for each element of $\mathcal{X}^\epsilon$ one of the four following cases occurs:

1. Substitutions: $\psi(i) = j$ with $(i, j) \in \mathcal{X} \times \mathcal{Y}$.
2. Removals: $\psi(i) = \epsilon_i$ with $i \in \mathcal{X}$.
3. Insertions: $\psi(\epsilon_j) = j$ with $j \in \mathcal{Y}$.
4. Finally $\psi(\epsilon_j) = \epsilon_i$ allow to complete the bijective property of ψ, and then should be ignored. This occurs when $i \in \mathcal{X}$ and $j \in \mathcal{Y}$ are both substituted.

Let $\Psi_\epsilon(\mathcal{X}, \mathcal{Y})$ be the set of all ϵ-assignments from $\mathcal{X}$ to $\mathcal{Y}$. In other terms, an ϵ-assignment is a bijection (or permutation) with additional constraints. The corresponding $(n+m) \times (m+n)$ permutation matrix can be decomposed as follows:

$$\mathbf{X} = \begin{array}{c} \begin{array}{cc} 1 \cdots m & \epsilon_1 \cdots \epsilon_n \end{array} \\ \left(\begin{array}{c|c} \mathbf{X}^{\text{sub}} & \mathbf{X}^{\text{rem}} \\ \hline \mathbf{X}^{\text{ins}} & \mathbf{X}^{\epsilon} \end{array} \right) \end{array} \begin{array}{c} 1 \\ \vdots \\ n \\ \hline \epsilon_1 \\ \vdots \\ \epsilon_m \end{array} \tag{4.1}$$

where matrix $\mathbf{X}^{\text{sub}} \in \{0,1\}^{n \times m}$ encodes node substitutions, $\mathbf{X}^{\text{rem}} \in \{0,1\}^{n \times n}$ encodes node removals, and $\mathbf{X}^{\text{ins}} \in \{0,1\}^{m \times m}$ encodes node insertions. Matrix $\mathbf{X}^\epsilon \in \{0,1\}^{m \times n}$ is an auxiliary matrix (see case 4), it ensures that $\mathbf{X}$ is a permutation matrix. Due to the constraints on dummy nodes (cases 2 and 3 above) matrices $\mathbf{X}^{\text{rem}}$ and $\mathbf{X}^{\text{ins}}$ always satisfy:

$$\begin{aligned} &\forall (i,j) \in \{1,\ldots,n\}^2, i \neq j, \quad x^{\text{rem}}_{i,j} = 0 \\ &\forall (i,j) \in \{1,\ldots,m\}^2, i \neq j, \quad x^{\text{ins}}_{i,j} = 0. \end{aligned} \tag{4.2}$$

Definition 9 (ϵ-assignment matrix)**.** *A $(n+m)\times(m+n)$ matrix satisfying equations 4.1 and 4.2 is called an ϵ-assignment matrix. The set of all $(n+m)\times(m+n)$ ϵ-assignment matrices is denoted by $\mathcal{A}_{n,m}$.*

The auxiliary matrix $\mathbf{X}^\epsilon$ in Eq. 4.1 suggests the definition of an equivalence relation between ϵ-assignment matrices.

Definition 10. *Two ϵ-assignment matrices $\mathbf{X}_1$ and $\mathbf{X}_2$, defined by the two sequences of block matrices $(\mathbf{X}_1^{sub}, \mathbf{X}_1^{rem}, \mathbf{X}_1^{ins}, \mathbf{X}_1^{\epsilon})$ and $(\mathbf{X}_2^{sub}, \mathbf{X}_2^{rem}, \mathbf{X}_2^{ins}, \mathbf{X}_2^{\epsilon})$, are equivalent iff*

$$(\mathbf{X}_1^{sub} = \mathbf{X}_2^{sub}) \wedge (\mathbf{X}_1^{rem} = \mathbf{X}_2^{rem}) \wedge (\mathbf{X}_1^{ins} = \mathbf{X}_2^{ins}).$$

The set of ϵ-assignment matrices up to this equivalence relation is denoted by $\mathcal{A}_{n,m}^{\sim}$.

Proposition 5. *There is a one-to-one relation between $\mathcal{A}_{n,m}^{\sim}$ and the set of injective functions from a subset of $\mathcal{X}$ to $\mathcal{Y}$.*

So, ϵ-assignment matrices on $\mathcal{X}^\epsilon$ and $\mathcal{Y}^\epsilon$, and injective functions defined on a subset of $\mathcal{X}$ onto $\mathcal{Y}$, are in one-to-one correspondence.

It is now possible to link ϵ-assignments to edit paths. Consider two simple graphs G_1 and G_2, with node sets V_1 and V_2 respectively. An ϵ-assignment from V_1 to V_2 can be defined by constructing the sets V_1^ϵ and V_2^ϵ. According to the above proposition, there is a one-to-one correspondence between ϵ-assignment matrices on V_1^ϵ and V_2^ϵ, and injective functions defined on a subset of V_1 onto V_2. By using Proposition 3, we can connect such a mapping to a restricted edit path between G_1 and G_2. Recalling the equivalence relation (Definition 10), we can state that there is a one-to-one correspondence between ϵ-assignment matrices and restricted edit paths; this shows that restricted edit paths can be deduced from ϵ-assignments.

4.3 Graph Edit distance as a bipartite graph assignment

4.3.1 LSAP for ϵ-assignments

Let $\mathcal{X} = \{1, \ldots, n\}$ and $\mathcal{Y} = \{1, \ldots, m\}$ be two sets. These two sets are augmented by dummy elements as described in the previous section, *i.e.* $\mathcal{X}^\epsilon = \mathcal{X} \cup \mathcal{E}_\mathcal{X}$ and $\mathcal{Y}^\epsilon = \mathcal{Y} \cup \mathcal{E}_\mathcal{Y}$. An ϵ-assignment from $\mathcal{X}$ to $\mathcal{Y}$, *i.e.* a bijective mapping from $\mathcal{X}^\epsilon$ onto $\mathcal{Y}^\epsilon$, represents a set of edit operations.

The selection of a relevant ϵ-assignment is realized through the design of a pairwise cost function adapted to edit operations. To this, each possible mapping of an element $i \in \mathcal{X}^\epsilon$ to an element $j \in \mathcal{Y}^\epsilon$ is penalized by a non-negative cost $c_{i,j}$. All costs can be encoded by a $(n+m) \times (m+n)$ matrix (having the same structure as ϵ-assignment matrices) [6, 7]

$$\mathbf{C} = \begin{array}{c} \begin{array}{cc} 1 \cdots m & \epsilon_1 \cdots \epsilon_n \end{array} \\ \left(\begin{array}{c|c} \mathbf{C}^{\text{sub}} & \mathbf{C}^{\text{rem}} \\ \hline \mathbf{C}^{\text{ins}} & \mathbf{0}_{m,n} \end{array}\right) \end{array} \begin{array}{c} 1 \\ \vdots \\ n \\ \epsilon_1 \\ \vdots \\ \epsilon_m \end{array} \tag{4.3}$$

where the matrix $\mathbf{C}^{\text{sub}} \in [0, +\infty)^{n \times m}$ encodes substitution costs, $\mathbf{C}^{\text{rem}} \in [0, +\infty)^{n \times n}$ encodes removal costs, and $\mathbf{C}^{\text{ins}} \in [0, +\infty)^{m \times m}$ encodes insertion costs. According to cases 2 and 3 considered in the definition of an ϵ-assignment (see Section 4.2.3, off-diagonal values of $\mathbf{C}^{\text{rem}}$ and $\mathbf{C}^{\text{ins}}$ are typically set to a large value ω, such that $\max\{c_{i,\psi(i)} \,|\, \forall i \in \mathcal{X}^\epsilon, \forall \psi \in \Psi_\epsilon(\mathcal{X}, \mathcal{Y})\} \ll \omega < +\infty$, in order to avoid forbidden mappings. Moreover, according to case 4, the mapping of any ϵ_i to an ϵ_j should not induce any cost, so the last block of $\mathbf{C}$ is set to the null matrix $\mathbf{0}_{m,n}$. As it is possible to deduce from the decomposition in Section 4.2.3 the cost of an

ϵ-assignment ψ can then be measured by the following sum:

$$\sum_{i=1}^{n+m} c_{i,\psi(i)} = \underbrace{\sum_{i\in\hat{\mathcal{X}}} c_{i,\psi(i)}}_{\text{substitutions}} + \underbrace{\sum_{i\in\mathcal{X}\setminus\hat{\mathcal{X}}} c_{i,\epsilon_i}}_{\text{removals}} + \underbrace{\sum_{\epsilon_j\in\mathcal{E}_\mathcal{X}} c_{\epsilon_j,\psi(\epsilon_j)}}_{\text{insertions}}.$$

where $\hat{\mathcal{X}} = \{i \in \mathcal{X} \mid \exists j \in \mathcal{Y},\ \psi(i) = j\}$.

An optimal ϵ-assignment is then defined as one having a minimal cost (several optimal ϵ-assignment may exist) among all ϵ-assignments:

$$\hat{\psi} \in \operatorname{argmin}\left\{\sum_{i=1}^{n+m} c_{i,\psi(i)} \mid \psi \in \Psi_\epsilon(\mathcal{X},\mathcal{Y})\right\} \tag{4.4}$$

which is a LSAP (Section 2.3). It can thus be rewritten as a binary programming problem (Eq. 2.12)

$$\hat{\mathbf{x}} \in \operatorname{argmin}\left\{\mathbf{c}^T\mathbf{x} \mid \mathbf{x} \in \operatorname{vec}[\mathcal{A}^{\sim}_{n,m}]\right\}, \tag{4.5}$$

where $\mathbf{x} = \operatorname{vec}(\mathbf{X}) \in \{0,1\}^{(n+m)^2}$ is the vectorization of the ϵ-assignment matrix $\mathbf{X}$ associated with ψ (Eq. 4.1), $\mathbf{c} = \operatorname{vec}(\mathbf{C}) \in [0,+\infty)^{(n+m)^2}$ is the vectorization of the cost matrix $\mathbf{C}$, and $\operatorname{vec}[\mathcal{A}^{\sim}_{n,m}] \subset \{0,1\}^{(n+m)^2}$ is the set of all vectorized ϵ-assignment matrices.

The optimal solution of the LSAP defined by Eq. 4.5 can be computed by any algorithm that solves LSAPs, such as Hungarian-type algorithms. Note that mappings in Ψ_ϵ, or matrices in $\mathcal{A}_{n,m}$, are much more constrained than bijective mappings or permutation matrices. These constraints, *i.e.* forbidden assignments, are satisfied in [6, 7] through the large ω values in the cost matrix. This is a classical trick used in LSAPs to avoid some specific assignments of elements [10]. While these assignments are avoided, the corresponding large ω values are still treated by the algorithms. A better way to take into account the additional constraints would be to modify the algorithms such that forbidden assignments are not treated at all. This is the choice we made in our experimentations. This improves the time complexity.

4.3.2 Bipartite GED

It is now possible to define a framework to approximate the GED, based on ϵ-assignments and the corresponding LSAP [6, 7, 40, 43]. Within this framework, a resulting approximate GED is called a bipartite GED.

Let G_1 and G_2 be two graphs, with node sets V_1 and V_2 respectively. The computation of a bipartite GED from G_1 to G_2 is performed in four main steps detailed below.

Step 1 - construction of the bags of patterns. For each node of each graph a bag of patterns is constructed. It represents a part of the graph connected to a specific node by some structured subgraphs, such as incident edges [6, 7] or walks [40]. A set of bags of patterns is then obtained for each graph. Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be the ones associated to G_1 and to G_2 respectively. We have thus $|\mathcal{B}_1| = |V_1|$ and $|\mathcal{B}_2| = |V_2|$. The idea is then to find an optimal ϵ-assignment from $\mathcal{B}_1$ to $\mathcal{B}_2$, according to a given pairwise cost matrix.

Step 2 - construction of the cost matrix. Each possible mapping of a bag $b_i \in \mathcal{B}_1$ to a bag $b_j \in \mathcal{B}_2$ is penalized by a cost measuring the affinity between the two bags. This cost is initially defined as the cost of editing b_i such that it is transformed into b_j, *i.e.* the cost of an optimal ϵ-assignment of the elements of the two bags [6, 7, 40, 43]. Moreover, the bags of $\mathcal{B}_1$ can be removed, and the bags of $\mathcal{B}_2$ can be inserted into $\mathcal{B}_1$, which is penalized by a cost measuring the relevance of the bag. In order to approximate the GED, all theses costs depends on the original edit cost functions defined in Section 2.4. They are encoded by a cost matrix $\mathbf{C}$ (Eq. 4.3). Note that in this step $|V_1| \times |V_2|$ LSAP are solved to compute the costs of assigning bags of $\mathcal{B}_1$ to bags of $\mathcal{B}_2$.

Step 3 - construction of an ϵ-assignment between the nodes. Given the cost matrix $\mathbf{C}$ computed in the previous step,

an optimal ϵ-assignment from $\mathcal{B}_1$ to $\mathcal{B}_2$ is then computed by solving again a LSAP. The computed optimal assignment hence provides an optimal mapping $\psi \in \Psi_\epsilon(V_1, V_2)$.

Step 4 - construction of a restricted edit path. The ϵ-assignment ψ can be interpreted as a partial edit path between the graphs G_1 and G_2. Indeed, it is only composed of edit operations involving nodes. Therefore this partial edit path has to be completed with edit operations applied on edges. This set of edit operations is directly induced by the set of edit operations operating on nodes, defined by the mapping computed in the previous step. The substitution, removal or insertion of any edge depends thus on the edit operations performed on its incident nodes. The cost of the complete edit path is finally defined by the sum of edit operations on nodes and edges. This cost only corresponds to an approximation of the GED between G_1 and G_2 since the mapping computed during Step 3 may not be optimal. Therefore, this cost corresponds to an overestimation of the exact GED, known as bipartite GED.

The definition of the cost matrix $\mathbf{C}$ in Step 2 is a keypoint of the framework. The initial approach proposed in [6, 7] defines bag of patterns as the corresponding node itself and its direct neighborhood, i.e. the set of incident edges and adjacent nodes. The cost of assigning a bag $b_i \in \mathcal{B}_1$ to a bag $b_j \in \mathcal{B}_2$ is then defined as the substitution cost of the associated node $i \in V_1$ and $j \in V_2$, plus the cost associated to an optimal ϵ-assignment between the two sets composed of their incident edges and their adjacent nodes. Using such bags of patterns can be discriminant enough, in which case the bipartite GED provides a good approximation of the GED. But this approach lacks of accuracy in some cases, in particular when the direct neighbourhood of the nodes is homogeneous in the graph. When considering such graphs, the cost associated to each pair of bags does not differ sufficiently, and the optimal ϵ-assignment depends more on the traversal of the nodes by the LSAP solver than on the graph's structure.

In order to improve the accuracy of the bipartite GED, the information attached to each node needs to be more global, for instance by considering bags of walks up to a length k [40], instead of the direct neighbourhood. This approach follows the same scheme as the one used in [6, 7], except that patterns associated to a node are defined as walks of length k starting at this node. Considering bags of such patterns allow to extend the amount of information associated to the nodes, which leads to a better approximation of the GED. However, the use of bags of walks induces some drawbacks. First, the set of computed walks suffers from the tottering phenomenon which leads to consider irrelevant patterns, especially when considering high values of k. These irrelevant patterns affect the cost of the ϵ-assignment, and thus the quality of the approximation of the GED. In addition, the mapping cost between two bags of walks can only be approximated, which induces another loss of accuracy.

These drawbacks can be avoided by using bags of subgraphs rather than bags of walks, such as all subgraphs centered on a given node and up to a radius k [44]. The cost associated to the mapping of two bags of such patterns is defined as the edit distance between the two k-subgraphs centered on the respective nodes. Despite the fact that we can control the size of these subgraphs thanks to the parameter k, this approach requires significantly more computational time than the previous ones. However, the use of accurate sub-structures allows to obtain a better approximation of the GED.

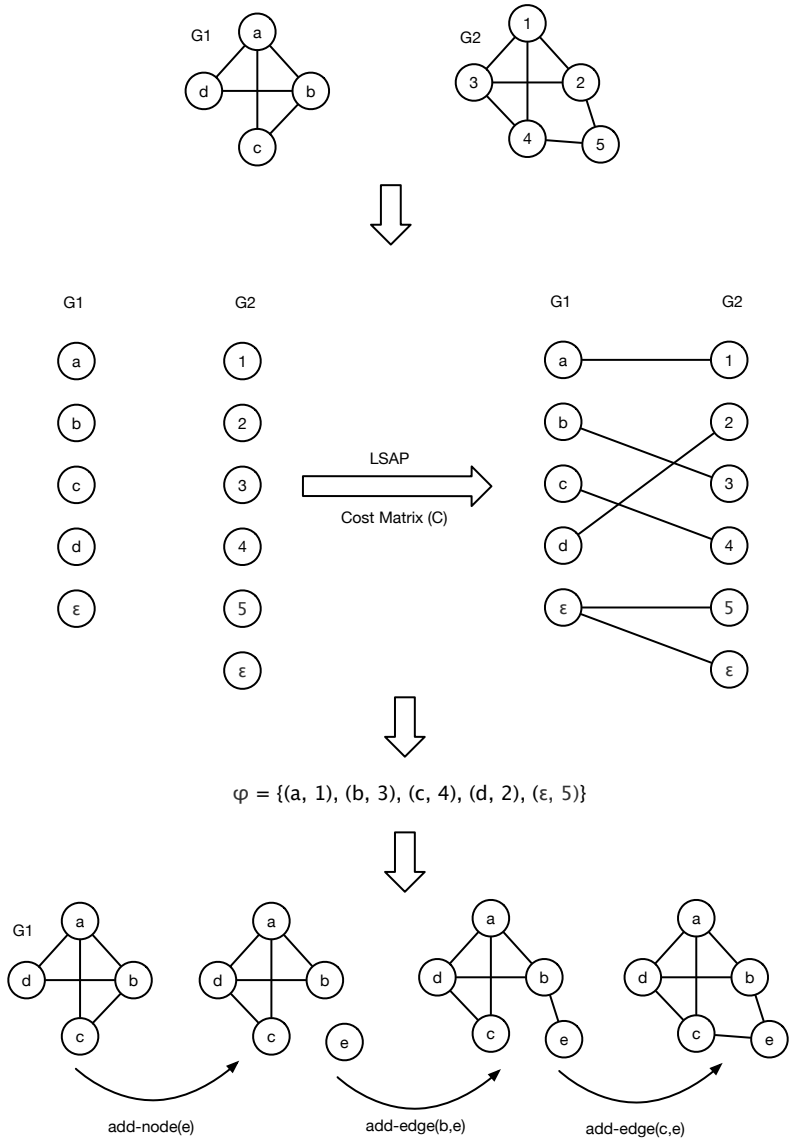


Figure 4.3: General framework to compute the GED as a LSAP.

4.4 Graph Edit Distance as a Quadratic Assignment Problem

The bipartite GED is a good candidate approximation of the GED, but it is based on the construction of a restricted edit path which generally does not have a minimal cost. Costs on edges can only be deduced from operations performed on their two incident nodes. This last point cannot be taken into account by the approach based on the LSAP, which considers information about edges separately in each node. In order to obtain a complete description of the GED, the model must consider simultaneously node and edge assignments. To this aim, a suitable formalization it is provided by the quadratic assignment problem [6, 7].

In this section, we propose to extend the linear model to a quadratic one based on ϵ-assignments, and we show that this model corresponds to the cost of a restricted edit path.

4.4.1 Simultaneous assignments and quadratic cost

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs, and let $\psi \in \Psi(V_1^\epsilon, V_2^\epsilon)$ be an ϵ-assignment (Section 4.2.3). When a pair $(i, j) \in V_1^\epsilon \times V_1^\epsilon$ is assigned by ψ to a pair $(\psi(i), \psi(j)) \in V_2^\epsilon \times V_2^\epsilon$, one of the following cases occurs:

1. Edge substitution: $(\psi(i), \psi(j)) \in E_2$ with $(i, j) \in E_1$.
2. Edge removal: $(\psi(i), \psi(j)) \notin E_2$ with $(i, j) \in E_1$.
3. Edge insertion: $(\psi(i), \psi(j)) \in E_2$ with $(i, j) \notin E_1$.
4. Finally $(\psi(i), \psi(j)) \notin E_2$ with $(i, j) \notin E_1$ allows to complete the bijection property.

Each possible simultaneous mapping of nodes $i, j \in V_1^\epsilon$ onto respectively nodes k and l in V_2^ϵ, is penalized by a non-negative cost $d_{ik,jl}$ which depends on the underlying edit operation described by one

of the above mentioned cases. The overall cost of edges associated to a simultaneous node assignment is then measured by:

$$d(\psi) = \sum_{i=1}^{n+m} \sum_{j=1}^{n+m} d_{i\psi(i),j\psi(j)}, \tag{4.6}$$

where cost values are defined as follows.

Let us recall that all mappings from a node of V_1^ϵ to a node of V_2^ϵ are not allowed. Indeed, as described in Section 4.2.3), $i \to \epsilon_j$ with $i \in V_1$ and $j \neq i$, and reciprocally $\epsilon_k \to l$ with $l \in V_2$ and $k \neq l$ are forbidden. So, a simultaneous node mapping involving at least one of these two cases is also forbidden. We denote a forbidden mapping by $\not\to$. The cost is set to a (large) value ω in this case, as discussed in Section 4.3.1,

For any other simultaneous node mapping $(i \to k, j \to l)$, with $i, j \in V_1^\epsilon$ and $k, l \in V_2^\epsilon$, its cost depends on the presence or the absence of edges $(i, j) \in E_1$ and $(k, l) \in E_2$:

- If $(i, j) \in E_1$ and $(k, l) \in E_2$ then $d_{ik,jl}$ is the cost of the edge assignment $(i, j) \to (k, l)$, *i.e.* edge substitution.
- If $(i, j) \in E_1$ and $(k, l) \notin E_2$ then $d_{ik,jl}$ is the cost of removing the edge (i, j).
- If $(i, j) \notin E_1$ and $(k, l) \in E_2$ then $d_{ik,jl}$ is the cost of inserting the edge (k, l).
- Else, the simultaneous mapping must not influence the overall cost and so its cost is always set to 0.

By using the edit cost functions defined in Section 2.4, the cost of an allowed simultaneous node mapping is then defined by

$$\begin{aligned} c_e(i \to k, j \to l) = {} & c_{es}\left((i, j) \to (k, l)\right) \delta_{(i,j)\in E_1} \delta_{(k,l)\in E_2} \\ & + c_{ed}\left(i, j\right) \delta_{(i,j)\in E_1} (1 - \delta_{(k,l)\in E_2}) \\ & + c_{ei}\left(k, l\right) (1 - \delta_{(i,j)\in E_1}) \delta_{(k,l)\in E_2} \end{aligned} \tag{4.7}$$

where $\delta_{e \in E} = 1$ if $e \in E$ and 0 else. Since graphs do not have self-loops, we have $d_{ik,ik} = 0$ for all $i \in V_1^\epsilon$ and $k \in V_2^\epsilon$. Remark also

that the symmetry of $c_e(i \to k, j \to l)$ depends both on the edit operations and on the directed edges, when the two graphs are directed.

Finally, the cost of a simultaneous node mapping is given by

$$d_{ik,jl} = \begin{cases} \omega & \text{if } (i \not\to k) \vee (j \not\to l) \\ c_e(i \to k, j \to l) & \text{else} \end{cases} \tag{4.8}$$

Let $\mathbf{x} \in \text{vec}[\mathcal{A}^{\sim}_{n,m}] \subset \{0,1\}^{(n+m)^2}$ be the vectorization of the ϵ-assignment matrix associated to ψ. All costs can be represented by a $(n+m)^2 \times (n+m)^2$ matrix $\mathbf{D} = (d_{ik,jl})_{i,k,j,l}$. Where $d_{ik,jl}x_{ik}x_{jl}$ will be equivalent to $d_{i\psi(i),j\psi(j)}$ if $x_{ik} = x_{jl} = 1$, and 0 else. In this way, each row and each column of $\mathbf{D}$, and $\mathbf{x}$, have the same organization of pairwise indices, and then the total cost of the simultaneous node assignment can be written in quadratic form as:

$$d(\psi) = \sum_{i=1}^{n+m} \sum_{k=1}^{m+n} \sum_{j=1}^{n+m} \sum_{l=1}^{m+n} d_{ik,jl}x_{ik}x_{jl} = \mathbf{x}^T\mathbf{D}\mathbf{x},$$

The cost matrix $\mathbf{D}$ can be decomposed as follows into blocks:

$$\mathbf{D} = \left(\begin{array}{ccc|ccc} \mathbf{D}^{1,1} & \cdots & \mathbf{D}^{1,n} & \mathbf{D}^{1,\epsilon_1} & \cdots & \mathbf{D}^{1,\epsilon_m} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{D}^{n,1} & \cdots & \mathbf{D}^{n,n} & \mathbf{D}^{n,\epsilon_1} & \cdots & \mathbf{D}^{n,\epsilon_m} \\ \hline \mathbf{D}^{\epsilon_1,1} & \cdots & \mathbf{D}^{\epsilon_1,n} & \mathbf{D}^{\epsilon_1,\epsilon_1} & \cdots & \mathbf{D}^{\epsilon_1,\epsilon_m} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{D}^{\epsilon_m,1} & \cdots & \mathbf{D}^{\epsilon_m,n} & \mathbf{D}^{\epsilon_m,\epsilon_1} & \cdots & \mathbf{D}^{\epsilon_m,\epsilon_m} \end{array}\right) \tag{4.9}$$

where each block $\mathbf{D}^{i,j} \in [0, +\infty)^{(m+n)\times(m+n)}$ defines the cost of assigning i and j of V_1^{ϵ} to respectively k and l for all $k, l \in V_2^{\epsilon}$, *i.e.* $[\mathbf{D}^{i,j}]_{k,l} = d_{ik,jl}$. Remarking that blocks $\mathbf{D}^{i,j}$ are organized in four main blocks corresponding to the nature of nodes i and j (dummy nodes or not). Each block $\mathbf{D}^{i,j}$ is itself decomposed into four blocks

as follows:

$$\mathbf{D}^{i,j} = \begin{array}{c} \begin{array}{cc} j1 \cdots jm & j\epsilon_1 \cdots j\epsilon_n \end{array} \\ \left(\begin{array}{c|c} \mathbf{D}^{i,j}_{1,1} & \mathbf{D}^{i,j}_{1,2} \\ \hline \mathbf{D}^{i,j}_{2,1} & \mathbf{D}^{i,j}_{2,2} \end{array}\right) \end{array} \begin{array}{c} i1 \\ \vdots \\ im \\ \hline i\epsilon_1 \\ \vdots \\ i\epsilon_n \end{array} \tag{4.10}$$

where $\mathbf{D}^{i,j}_{1,1} \in [0,+\infty)^{m\times m}$, $\mathbf{D}^{i,j}_{1,2} \in [0,+\infty)^{m\times n}$, $\mathbf{D}^{i,j}_{2,1} \in [0,+\infty)^{n\times m}$ and $\mathbf{D}^{i,j}_{2,2} \in [0,+\infty)^{n\times n}$. The different values taken by the elements of $\mathbf{D}$, depending on the values of the indices, are reported in Table 4.1.

Proposition 6. *If both G_1 and G_2 are undirected, then:*

$$\forall (i,k,j,l) \in V_1^\epsilon \times V_2^\epsilon \times V_1^\epsilon \times V_2^\epsilon, \quad d_{ik,jl} = d_{jl,ik}.$$

Proof. If both G_1 and G_2 are undirected, then $\delta_{(i,j)\in E_1} = \delta_{(j,i)\in E_1}$, $\delta_{(k,l)\in E_2} = \delta_{(l,k)\in E_2}$ and :

$$\left\{\begin{array}{lcl} c_{es}((i,j)\to(k,l)) & = & c_{es}((j,i)\to(l,k)) \\ c_{ed}(i,j) & = & c_{ed}(j,i) \\ c_{ei}(k,l) & = & c_{ei}(l,k) \end{array}\right.$$

Hence, if none of i, j, k or l are equal to ϵ, the first line of Table 4.1 will remain unchanged. Moreover, if $\epsilon \in \{i,j,k,l\}$, then permuting indices (i,k) and (j,l) will lead to the following permutations of the lines of Table 4.1 (after the appropriate renaming of variables):

(2,3)(4)(5,9)(6,11)(7,10)(8,12)(13)(14,15)(16)

One can check that in each case $d_{ik,jl} = d_{jl,ik}$. □

Remark 2. *If the rows of matrix $\mathbf{D}$ correspond to (i,k) and the columns to (j,l), then under the hypothesis of Proposition 6, $\mathbf{D}$ is symmetric and we get $\mathbf{D}^T = \mathbf{D}$.*

In order to provide a complete representation of edit operations we need also to consider those performed on nodes. This can be measured by the linear sum $\mathbf{c}^T\mathbf{x}$ defined in Section 4.3.1, where $\mathbf{c} = \text{vec}(\mathbf{C}) \in [0, +\infty)^{(n+m)^2}$ represents the cost of edit operations on nodes (Eq. 4.3):

$$\begin{aligned} c_{i,k}^{\text{sub}} &= c_{vs}(i \to k) \\ c_{i,k}^{\text{rem}} &= \begin{cases} c_{vd}(i) & \text{if } k = i \\ \omega & \text{else} \end{cases} \\ c_{i,k}^{\text{ins}} &= \begin{cases} c_{vi}(k) & \text{if } i = k \\ \omega & \text{else} \end{cases} \end{aligned} \tag{4.11}$$

4.4.2 QAP for ϵ-assignments, restricted edit paths and GED

According to the following result, summing the quadratic and the linear costs defined above leads to the cost of a restricted edit path.

Proposition 7. *Let $\Delta = \mathbf{D}$ if both G_1 and G_2 are undirected and $\Delta = \mathbf{D} + \mathbf{D}^T$ else. Note that using Proposition 6, Δ is symmetric. Any non-infinite value of $\frac{1}{2}\mathbf{x}^T\Delta\mathbf{x} + \mathbf{c}^T\mathbf{x}$ corresponds to the cost of a minimal edit path. Conversely the cost of any minimal edit path may be written as $\frac{1}{2}\mathbf{x}^T\Delta\mathbf{x} + \mathbf{c}^T\mathbf{x}$ with the appropriate $\mathbf{x}$.*

In worths pointing out that the prof of Proposition 7 is discussed in details in [55].

So that, the determination of a restricted edit path with a minimal cost is equivalent to searching for an optimal ϵ-assignment

$$\hat{\mathbf{x}} \in \text{argmin} \left\{ \frac{1}{2}\mathbf{x}^T\Delta\mathbf{x} + \mathbf{c}^T\mathbf{x} \mid \mathbf{x} \in \text{vec}[\mathcal{A}_{n,m}^{\sim}] \right\} \tag{4.12}$$

In other terms, for the class of graphs under consideration, *i.e.* simple graphs, we have

$$\text{GED}(G_1, G_2) = \min \left\{ \frac{1}{2}\mathbf{x}^T\Delta\mathbf{x} + \mathbf{c}^T\mathbf{x} \mid \mathbf{x} \in \text{vec}[\mathcal{A}_{n,m}^{\sim}] \right\} \tag{4.13}$$

This is a QAP, see [16, 10] for more details on QAPs. In particular, QAPs are NP-hard and exact algorithms can solve QAPs of small size only. So, many heuristics able to find suboptimal solutions in short computing time have been explored.

Remark 3. *Note that the functional involved in the QAP defined by Eq. 4.12 can be rewritten as a general quadratic term:*

$$\frac{1}{2}\mathbf{x}^T\Delta\mathbf{x} + \mathbf{c}^T\mathbf{x} = \frac{1}{2}\mathbf{x}^T\Delta\mathbf{x} + \mathbf{x}^T \mathit{diag}(\mathbf{c})\mathbf{x} = \mathbf{x}^T\left(\frac{1}{2}\Delta + \mathit{diag}(\mathbf{c})\right)\mathbf{x} \tag{4.14}$$

where $\mathit{diag}(\mathbf{c})$ *is the diagonal matrix with* $\mathbf{c}$ *as diagonal. So the GED can be equivalently defined by*

$$\mathit{GED}(G_1, G_2) = \min\left\{\, \mathbf{x}^T\overline{\Delta}\mathbf{x} \mid \mathbf{x} \in \mathit{vec}[\mathcal{A}^{\sim}_{n,m}] \,\right\} \tag{4.15}$$

where $\overline{\Delta} = \frac{1}{2}\Delta + \mathit{diag}(\mathbf{c})$ *represents the cost of both node and edge edit operations. As graphs are simple, they have no self-loops and then the diagonal elements of* Δ *are all equal to* 0*. So the diagonal of* $\overline{\Delta}$ *is always equal to* $\mathbf{c}$*.*

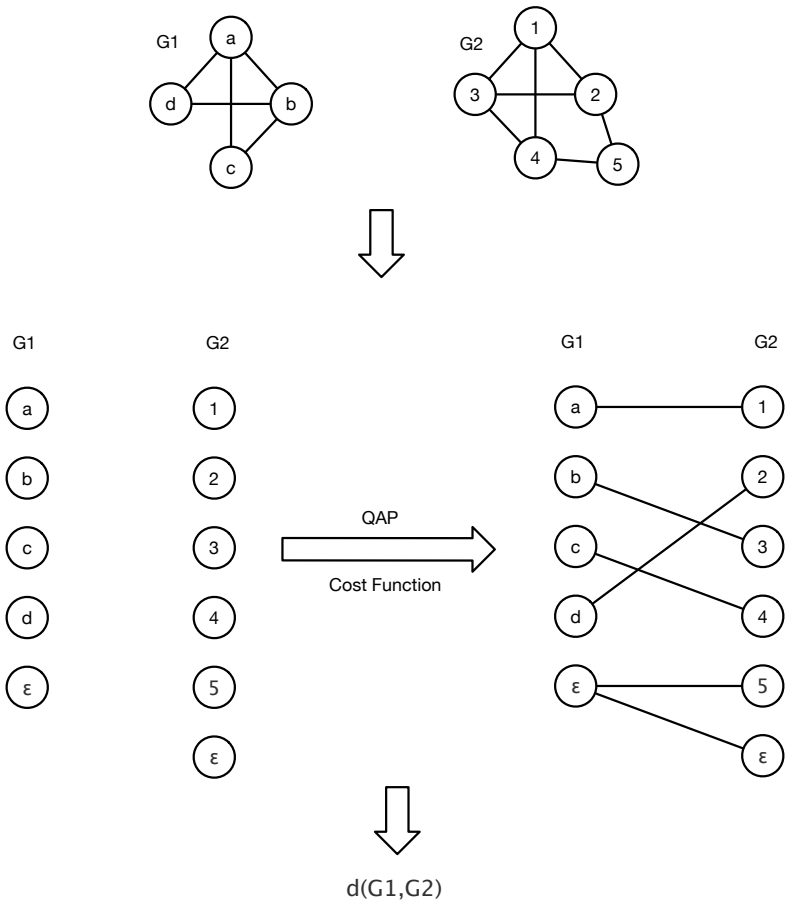


Figure 4.4: General framework to compute the GED as a GED.

case	block	nodes in V_2^ϵ	$d_{(i,j),(k,l)}$
1	$\mathbf{D}_{1,1}^{i,j}$	k, l	$c_{es}((i,j) \to (k,l))\delta_{(i,j)\in E_1}\delta_{(k,l)\in E_2} +$ $c_{ed}(i,j)\delta_{(i,j)\in E_1}(1-\delta_{(k,l)\in E_2}) +$ $c_{ei}(k,l)(1-\delta_{(i,j)\in E_1})\delta_{(k,l)\in E_2}$
2	$\mathbf{D}_{1,2}^{i,j}$	k, ϵ_j	$c_{ed}(i,j)\delta_{(i,j)\in E_1}$
		else	ω
3	$\mathbf{D}_{2,1}^{i,j}$	ϵ_i, l	$c_{ed}(i,j)\delta_{(i,j)\in E_1}$
		else	ω
4	$\mathbf{D}_{2,2}^{i,j}$	ϵ_i, ϵ_j	$c_{ed}(i,j)\delta_{(i,j)\in E_1}$
		else	ω
5	$\mathbf{D}_{1,1}^{i,\epsilon_l}$	k, l	$c_{ei}(k,l)\delta_{(k,l)\in E_2}$
		else	ω
6	$\mathbf{D}_{1,2}^{i,\epsilon_l}$	k, ϵ	0
7	$\mathbf{D}_{2,1}^{i,\epsilon_l}$	ϵ_i, l	0
		else	ω
8	$\mathbf{D}_{2,2}^{i,\epsilon_l}$	ϵ_i, ϵ	0
		else	ω
9	$\mathbf{D}_{1,1}^{\epsilon_k,j}$	k, l	$c_{ei}\,(k,l)\,\delta_{(k,l)\in E_2}$
		else	ω
10	$\mathbf{D}_{1,2}^{\epsilon_k,j}$	k, ϵ_j	0
		else	ω
11	$\mathbf{D}_{2,1}^{\epsilon_k,j}$	ϵ, l	0
12	$\mathbf{D}_{2,2}^{\epsilon_k,j}$	ϵ, ϵ_j	0
		else	ω
13	$\mathbf{D}_{1,1}^{\epsilon_k,\epsilon_l}$	k, l	$c_{ei}\,(k,l)\,\delta_{(k,l)\in E_2}$
		else	ω
14	$\mathbf{D}_{1,2}^{\epsilon_k,\epsilon_l}$	k, ϵ	0
		else	ω
15	$\mathbf{D}_{2,1}^{\epsilon_k,\epsilon_l}$	ϵ, l	0
		else	ω
16	$\mathbf{D}_{2,2}^{\epsilon_k,\epsilon_l}$	ϵ, ϵ	0

Table 4.1: Elements of matrix D according to the configuration of its indices. We consider that $i, j \in V_1$, $k, l \in V_2$, $\epsilon_k, \epsilon_l \in \mathcal{E}_1$, and $\epsilon_i, \epsilon_j \in \mathcal{E}_2$. Epsilon values without indices mean any ϵ-value.

4.5 Solving the Quadratic Assignment Problem

Now it is possible to discuss how to approximate the GED by solving the QAP defined by Eq. 4.12. The latter can be rewritten as the following binary quadratic programming problem:

$$\operatorname{argmin} \left\{ S(\mathbf{x}) \stackrel{\text{def.}}{=} \frac{1}{2}\mathbf{x}^T \Delta \mathbf{x} + \mathbf{c}^T \mathbf{x} \mid \mathbf{A}\mathbf{x} = \mathbf{1}_n,\ \mathbf{x} \in \{0,1\}^n \right\} \quad (4.16)$$

where $\mathbf{A}\mathbf{x} = \mathbf{1}_n$, with $\mathbf{x} \in \{0,1\}^n$ and $\mathbf{A} \in \{0,1\}^{n \times n}$, is the matrix version of the bijectivity constraints given by Def. 6, see [10, 45] for more details. Also, we suppose that $\mathbf{c} \in [0, +\infty)^n$, and $\Delta \in [0, +\infty)^{n \times n}$ is assumed to be symmetric. Note that Eq. 4.12 is equivalent to Eq. 4.16, with additional constraints on $\mathbf{x}$ (Eq. 4.2) imposed by ω values in the expression of Δ (Eq. 4.8 and Proposition 7) and $\mathbf{c}$ (Eq. 4.11).

QAPs are generally NP-hard, which depends on the structure of the cost matrix $\overline{\Delta} = \frac{1}{2}\boldsymbol{\Delta} + \operatorname{diag}(\mathbf{c})$ (see previous section), and so most algorithms find approximate local or global optimal solutions by relaxing the bijectivity constraints on the solution, which leads to find a continuous solution instead of a discrete one:

$$\operatorname{argmin} \{ S(\mathbf{x}) \mid \mathbf{A}\mathbf{x} = 1,\ \mathbf{x} \in [0, +\infty)^n \}. \quad (4.17)$$

While this relaxed problem is also NP-hard, several polynomial-time algorithms have been designed to converge close to a local or global solution in a short computing time. The ones based on linearization of the cost function S are known to be particularly efficient. They transform the relaxed problem into a sequence of convex problems, such that a given initial solution is improved iteratively by decreasing the cost function up to a fixed point. Then, the final continuous solution is binarized and used as a solution of the QAP. But as shown experimentally in [52] in the context of graph matching, the continuous optimum is not necessarily close to the global discrete optimum. Indeed the relaxed problem is equivalent to the original QAP when $\overline{\Delta}$ is positive definite, which

is generally not the case in most situations. In order to try to overcome this problem, it seems to be more efficient to find a discrete solution as close as possible to a continuous one, at each iteration and then derive the discrete solution from it. This is done by Sof-Assign [49, 50, 51] or Integer Projected Fixed Point (IPFP) [52].

4.5.1 Adapting IPFP to solve the GED

IPFP is an iterative optimization algorithm for constrained optimization problems proposed by Leordeanu et al. [52]. In the original version, the algorithm is designed to solve a generic QAP by maximizing the cost function. Thus, in order to compute efficiently the GED, we have firstly adapted the algorithm to search for the minimum and then we have reduced its computational complexity by improving different steps.

Given an initial continuous (or discrete) candidate solution $\mathbf{x}_0$, the idea of [52] is to iteratively improve (here reduce) the corresponding quadratic cost in two steps at each iteration:

1. Compute a discrete linear approximation $\mathbf{b}_{k+1}$ of the quadratic cost S around the current solution $\mathbf{x}_k$ by solving a LSAP.

2. Compute the next candidate solution $\mathbf{x}_{k+1}$ by solving the relaxed problem, reduced to compute the extremum of S between $\mathbf{x}_k$ and $\mathbf{b}_{k+1}$ included.

The iteration of these steps converges to an optimum of the relaxed problem, which is either continuous or discrete but generally not the global one. This last point depends on the initialization. The whole process is detailed in Algorithm 5.

At each iteration, in the first step, the cost S is linearly approximated. The differential of S in $\mathbf{x}_k$ in the direction $\mathbf{h}$ is given by:

$$DS(\mathbf{x}_k) \cdot \mathbf{h} = \mathbf{x}_k^T \Delta \mathbf{h} + \mathbf{c}^T \mathbf{h} \qquad \text{(since } \Delta \text{ is symmetric)}.$$

Algorithm 5 Optimized IPFP.

function IPFPMIN($\mathbf{x}_0$,$\mathbf{c}$,Δ,$k_{\max}$)
  $k \leftarrow 0,\ \ L \leftarrow \mathbf{c}^T\mathbf{x}_0,\ \ S_k \leftarrow \frac{1}{2}\mathbf{x}_0^T\Delta\mathbf{x}_0 + L$
  repeat
    $\mathbf{b}_{k+1} \leftarrow \operatorname{argmin}\{(\mathbf{x}_k^T\Delta + \mathbf{c}^T)\mathbf{b} \mid \mathbf{b} \in \mathcal{A}_{n,m}^{\sim}\}$
    $L' \leftarrow \mathbf{c}^T\mathbf{b}_{k+1}$
    $S_{\mathbf{b}_{k+1}} \leftarrow \frac{1}{2}\mathbf{b}_{k+1}^T\Delta\mathbf{b}_{k+1} + L'$
    $\alpha \leftarrow R(\mathbf{b}_{k+1}) - 2S_k + L$
    $\beta \leftarrow S_{\mathbf{b}_{k+1}} + S_k - R(\mathbf{b}_{k+1}) - L$
    $t_0 \leftarrow -\alpha/(2\beta)$
    if $(\beta \leq 0) \vee (t_0 \geq 1)$ **then**
      $\mathbf{x}_{k+1} \leftarrow \mathbf{b}_{k+1},\ \ S_{k+1} \leftarrow S_{\mathbf{b}_{k+1}},\ \ L \leftarrow L'$
    else
      $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + t_0(\mathbf{b}_{k+1} - \mathbf{x}_k)$
      $S_{k+1} \leftarrow S_k - \alpha^2/(4\beta)$
      $L \leftarrow \mathbf{c}^T\mathbf{x}_{k+1}$
    end if
    $k \leftarrow k + 1$
  until $(\mathbf{x}_{k+1} = \mathbf{x}_k) \vee (k \geq k_{\max})$
  return $(\mathbf{x}_{k+1}, S_{k+1})$
end function

Hence the first-order Taylor expansion of S around the current solution $\mathbf{x}_k$ is given by:

$$\begin{aligned} S(\mathbf{b}) &\approx S(\mathbf{x}_k) + \left(\mathbf{x}_k^T\Delta + \mathbf{c}^T\right)(\mathbf{b} - \mathbf{x}_k) \\ &\approx S(\mathbf{x}_k) + R(\mathbf{b}) - R(\mathbf{x}_k) \end{aligned} \tag{4.18}$$

where $R(\mathbf{x}) = (\mathbf{x}_k^T\Delta + \mathbf{c}^T)\mathbf{x}$ and $\mathbf{b} \geq \mathbf{0}$. Keeping $\mathbf{x}_k$ fixed, $S(\mathbf{x}_k)$ and $R(\mathbf{x}_k)$ are constant, and so the minimization of $S(\mathbf{b})$ is approximatively equivalent to the minimization of $R(\mathbf{b})$:

$$\mathbf{b}_{k+1} \in \operatorname{argmin}\left\{\left(\mathbf{x}_k^T\Delta + \mathbf{c}^T\right)\mathbf{b} \mid \mathbf{A}\mathbf{b} = \mathbf{1},\ \mathbf{b} \geq \mathbf{0}_n\right\}. \tag{4.19}$$

This is a linear programming problem with totally unimodular constraint matrix $\mathbf{A}$ and the right-hand side vector of the linear system $\mathbf{A}\mathbf{b} = \mathbf{1}$ is integer valued. So, by standard tools in linear

programming, there is an integer optimal solution, here binary and equal to the solution of the LSAP [45, 10]:

$$\mathbf{b}_{k+1} \in \operatorname{argmin} \left\{ \left(\mathbf{x}_k^T \Delta + \mathbf{c}^T \right) \mathbf{b} \mid \mathbf{A}\mathbf{b} = \mathbf{1},\ \mathbf{b} \in \{0,1\}^{n \times n} \right\} \quad (4.20)$$

In our experiments, this problem is solved by the $O(n^3)$ version of the Hungarian algorithm [47, 10], modified such that forbidden assignments represented by ω values in $\mathbf{\Delta}$ and $\mathbf{c}$ are not treated. The resulting assignment $\mathbf{b}_{k+1}$ determines a direction of largest possible decrease of S in the first-order approximation. Let us additionally note that the first order approximation of $S(\mathbf{b})$ is lower than $S(\mathbf{x}_k)$ since $R(\mathbf{b}_{k+1}) \leq R(\mathbf{x}_k)$. However we cannot yet conclude since this is only an approximation.

The second step of each iteration of Algorithm 5 consists in minimizing the quadratic function S in the continuous domain along the direction given by $\mathbf{b}_{k+1}$. This can be done analytically. Let $\mathbf{x}_t = \mathbf{x}_k + t(\mathbf{b}_{k+1} - \mathbf{x}_k)$, with $t \in [0,1]$, be a parameterization of the segment between $\mathbf{x}_k$ and $\mathbf{b}_{k+1}$. The evolution of S on this segment is provided by:

$$\begin{aligned}
S(\mathbf{x}_t) &= S(\mathbf{x}_k + t(\mathbf{b}_{k+1} - \mathbf{x}_k)) \\
&= \tfrac{1}{2}[\mathbf{x}_k + t(\mathbf{b}_{k+1} - \mathbf{x}_k)]^T \Delta [\mathbf{x}_k + t(\mathbf{b}_{k+1} - \mathbf{x}_k)] + \mathbf{c}^T [\mathbf{x}_k + t(\mathbf{b}_{k+1} - \mathbf{x}_k)] \\
&= \tfrac{1}{2}\mathbf{x}_k^T \Delta \mathbf{x}_k + \mathbf{c}^T \mathbf{x}_k + t\mathbf{x}_k^T \Delta (\mathbf{b}_{k+1} - \mathbf{x}_k) + \tfrac{1}{2} t^2 (\mathbf{b}_{k+1} - \mathbf{x}_k)^T \Delta (\mathbf{b}_{k+1} - \mathbf{x}_k) \\
&\quad + t\mathbf{c}^T (\mathbf{b}_{k+1} - \mathbf{x}_k) \\
&= S(\mathbf{x}_k) + t[\mathbf{x}_k^T \Delta (\mathbf{b}_{k+1} - \mathbf{x}_k) + \mathbf{c}^T (\mathbf{b}_{k+1} - \mathbf{x}_k)] + \tfrac{1}{2} t^2 (\mathbf{b}_{k+1} - \mathbf{x}_k)^T \Delta (\mathbf{b}_{k+1} - \mathbf{x}_k) \\
&= S(\mathbf{x}_k) + tR(\mathbf{b}_{k+1} - \mathbf{x}_k) + \tfrac{1}{2} t^2 (\mathbf{b}_{k+1} - \mathbf{x}_k)^T \Delta (\mathbf{b}_{k+1} - \mathbf{x}_k) \\
&= S(\mathbf{x}_k) + \alpha t + \beta t^2
\end{aligned}$$

where

$$\begin{aligned}
\alpha &= R(\mathbf{b}_{k+1} - \mathbf{x}_k) = R(\mathbf{b}_{k+1}) - R(\mathbf{x}_k) \leq 0 \\
&= R(\mathbf{b}_{k+1}) - \mathbf{x}_k^T \Delta \mathbf{x}_k - \mathbf{c}^T \mathbf{x}_k = R(\mathbf{b}_{k+1}) - 2(\tfrac{1}{2}\mathbf{x}_k^T \Delta \mathbf{x}_k + \mathbf{c}^T \mathbf{x}_k) + \mathbf{c}^T \mathbf{x}_k \\
&= R(\mathbf{b}_{k+1}) - 2S(\mathbf{x}_k) + \mathbf{c}^T \mathbf{x}_k \\
\beta &= \tfrac{1}{2}(\mathbf{b}_{k+1} - \mathbf{x}_k)^T \Delta (\mathbf{b}_{k+1} - \mathbf{x}_k) \\
&= \tfrac{1}{2}\mathbf{b}_{k+1}^T \Delta \mathbf{b}_{k+1} - \mathbf{x}_k^T \Delta \mathbf{b}_{k+1} + \tfrac{1}{2}\mathbf{x}_k^T \Delta \mathbf{x}_k \\
&= \tfrac{1}{2}\mathbf{b}_{k+1}^T \Delta \mathbf{b}_{k+1} + \mathbf{c}^T \mathbf{b}_{k+1} - R(\mathbf{b}_{k+1}) + \tfrac{1}{2}\mathbf{x}_k^T \Delta \mathbf{x}_k \\
&= S(\mathbf{b}_{k+1}) + S(\mathbf{x}_k) - R(\mathbf{b}_{k+1}) - \mathbf{c}^T \mathbf{x}_k
\end{aligned}$$

The main advantage of the above expression for the calculation of α, and not used in the original algorithm [52], is that $R(\mathbf{b}_{k+1})$

is already computed by the LSAP algorithm that computes $\mathbf{b}_{k+1}$. Moreover $S(\mathbf{x}_k)$ and $\mathbf{c}^T\mathbf{x}_k$ may be stored from the previous iteration of the algorithm. Note that since $\alpha = R(\mathbf{b}_{k+1}) - R(\mathbf{x}_k)$ we have $\alpha \leq 0$. For β, the improvement is a bit more tedious. We have indeed to compute $S(\mathbf{b}_{k+1})$. Hence the gain is not obvious compared to the direct computation of $(\mathbf{b}_{k+1} - \mathbf{x}_k)^T\Delta(\mathbf{b}_{k+1} - \mathbf{x}_k)$ as done in the original algorithm [52]. Note, however, that $S(\mathbf{b}_{k+1})$ will also be used in a following step, so computations are factorized. The problem is thus transformed into finding the optimal value

$$t_0 = \operatorname{argmin}\left\{S(\mathbf{x}_t) = S(\mathbf{x}_k) + \alpha t + \beta t^2 \mid t \in [0,1]\right\}. \qquad (4.21)$$

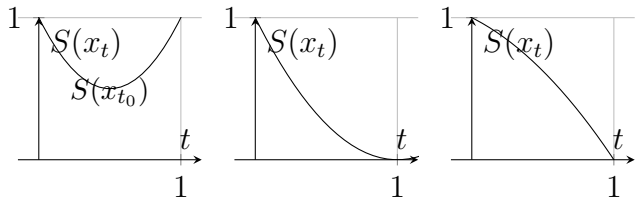
The derivative of $S(\mathbf{x}_t)$ cancels at $t_0 = -\alpha/(2\beta)$. Then as shown in Fig. 4.5, we have:

- If $\beta > 0$
  - If $t_0 \leq 1$, $S(\mathbf{x}_{t_0})$ is the minimum of $S(\mathbf{x}_t)$, in particular it is lower than $S(\mathbf{x}_k)$ and $S(\mathbf{b}_{k+1})$. Moreover:

$$S(\mathbf{x}_{t_0}) = S(\mathbf{x}_k) - \frac{\alpha^2}{2\beta} + \frac{\alpha^2}{4\beta} = S(\mathbf{x}_k) - \frac{\alpha^2}{4\beta}$$

  - If $t_0 \geq 1$, then $S'(\mathbf{x}_t) < 0\ \forall t \in [0,1]$, and the minimal value of $S(\mathbf{x}_t)$ is $S(\mathbf{x}_1) = S(\mathbf{b}_{k+1})$.
- If $\beta \leq 0$, since $\alpha \leq 0$, $S(\mathbf{x}_t)$ decreases between $t = 0$ and $t = 1$. Its minimal value is thus $S(\mathbf{x}_1) = S(\mathbf{b}_{k+1})$.

So, if either $\beta < 0$ or $\beta > 0$, but $t_0 \geq 1$, the minima of $S(\mathbf{x}_t)$ within the range $t \in [0,1]$ is obtained for $t_0 = 1$, *i.e.* $\mathbf{x}_{t_0} = \mathbf{b}_{k+1}$ (lines 11-12). Note that in this case the new solution is discrete. In the remaining case (lines 13-16), $S(\mathbf{x}_t)$ passes by a minimal value lower than $S(\mathbf{x}_k)$ and $S(\mathbf{b}_{k+1})$. In both cases $\mathbf{x}_{t_0}$ is taken as the solution $\mathbf{x}_{k+1}$ for the next iteration. Hence, as in the original algorithm [52], $S(\mathbf{x}_k)$ decreases at each iteration, and since Δ and $\mathbf{c}$ are positive, S is bounded bellow 0 and the algorithm converges.



(a) $\beta \geq 0, t_0 \leq 1$ (b) $\beta \geq 0, t_0 \geq 1$ (c) $\beta < 0$

Figure 4.5: Illustration of the 3 cases relating β and t_0.

The whole process is iterated until a fixed point is reached, in which case $\mathbf{x}_{k+1}$ is ensured to be a minimum of the relaxed problem defined by Eq. 4.17. When a minimum of the original problem defined by Eq. 4.16 is requested as for the GED, the discrete vector closest to the minimum of the relaxed problem is selected. The minimum of the relaxed problem is not guaranteed to be global. This depends on the initial vector x_0, which influences both the value of the resulting cost and the number of iterations required to reach the convergence. For the approximation of the GED, we have tested several initializations based on the LSAP, as described in the following section. We have observed that the exact GED is often obtained, meaning that the optimal solution of the original problem can be reached by the algorithm.

4.6 Experiments and Results

The experiments have been mainly devoted to demonstrate the effectiveness of the QAP approach with respect to the ones based on LSAPs. In particular, our intent has been to prove that the accuracy of the bipartite GED can be considerably improved by a formulation which considers nodes and edges simultaneously. To this aim, we compare our approach with three LSAP methods of the state of the art [7, 40, 44] on four molecular datasets obtained form real chemistry problems. As discussed in Section 4.3.2, each of them differs on the definition of the cost between elements.

In addition to these methods we have included in our experiment the A^* algorithm, with the intent to have a reference about the exact GED. However, A^* is restricted to very simple graphs and it wasn't possible to compute exact GEDs for two over the four used datasets.

In order to provide comparable computational times, all the experiments have been performed by using Matlab on the same macchine equipped with an Quadcore Intel Xeon E5310 1,60GHz, 4Mb of cache and 32 Gb of RAM.

4.6.1 Datasets

The molecular datasets used in our experiments are provided by *GREYC* and *LCMT* laboratories. Both are UMR research units placed under the joint responsibility of University of Caen Basse-Normandie and ENSICAEN. Each dataset is composed of graphs extracted from a different kind of molecules: *Alkane*, *Acyclic*, *MAO* and *PAH*. These datasets are available on the following web page: http://iapr-tc15.greyc.fr/links.html. We have finally generated a synthetic dataset by extending MAO, in order to have larger graphs, its composition is described below.

4.6.1.0.1 Alkane Dataset The Alkane database is a purely structural database, only carbons, with a problem of regression. Also this dataset contains acyclic but not labeled graphs. The

number of graph contained is 150, and size of graphs is between 1 and 10 vertices.

4.6.1.0.2 Acyclic Dataset The Acyclic database is concerned with the determination of the boiling point using regression and is composed by acyclic molecules with hetero atoms. All graphs are labeled and acyclic. Acyclic dataset contains 183 graphs, with a size between 3 and 11 vertices.

4.6.1.0.3 MAO Dataset The MAO database is composed by 68 molecules belonging to two classes: molecules which inhibit the monoamine oxidase and those which do not. The average size of graphs is about 18 vertices.

4.6.1.0.4 PAH Dataset The PAH database is composed by cyclic unlabeled graphs. There are only carbons molecules, all bounds are aromatics. Few acyclic bounds connect some atoms to cycles. The idea is to find cancerous or not cancerous molecules.

4.6.1.0.5 Extended MAO Dataset In the molecular dataset we do not have graphs larger than 30 nodes, but it is important to understand how our IPFP is able to scale with respect to the size. To this aim, we have generated a synthetic dataset with the same node and edge labels distribution and same ratio between the number of edges and the number of nodes as MAO dataset, but that is generalized to different graph sizes. For each node size, from 10 to 100 nodes, we have generated 100 couples of pattern and target graphs. Each target graph has been generated by removing one node and substituting another one from the associated source graph. The overall edit distance between source and target graphs is then defined by the cost associated to this two node operations together with the induced edit operations on edges. The graph edit distance between each pair of graphs is around 10.

Dataset	*Number of graphs*	*Mean size*	*Mean degree*	*Min size*	*Max size*
Alkane	150	8.9	1.8	1	10
Acyclic	183	8.2	1.8	3	11
MAO	68	18.4	2.1	11	27
PAH	94	20.7	2.4	10	28

Table 4.2: GREYC's Chemistry Dataset

4.6.2 Results

IPFP approach allows to drastically improve the accuracy of the approximation with respect to LSAP approaches while keeping a reasonable computational time. In order to confirm this, we have considered three relevant information in the comparison: the average edit distance (d), the average approximation error (e) with respect to the exact graph edit distance and the average computational time (t) required to get the graph edit distance for a pair of graphs. For these three measures, lower values correspond to better results. Indeed, since approximation approaches overestimate graph edit distance, a lower average edit distance can be considered as better than an higher one. Moreover, as regard the average error, due to the computational complexity required by A* algorithm, the exact graph edit distance has not been computed on PAH and MAO datasets that are composed of larger graphs. Then, it has not been possible to compute the error, but a useful information about the accuracy can be obtained by the average edit distance. Moreover, It is worth to point out that, in order to avoid some bias, the results have been computed using random permutations of the adjacency matrices before computing graph edit distances.

Results shown in Table 4.3 also highlight the importance of the initialization step. Since the functional of QAP formulation is not convex, different initializations can lead to different local minima. Obviously, initializations close to the global minimum have an

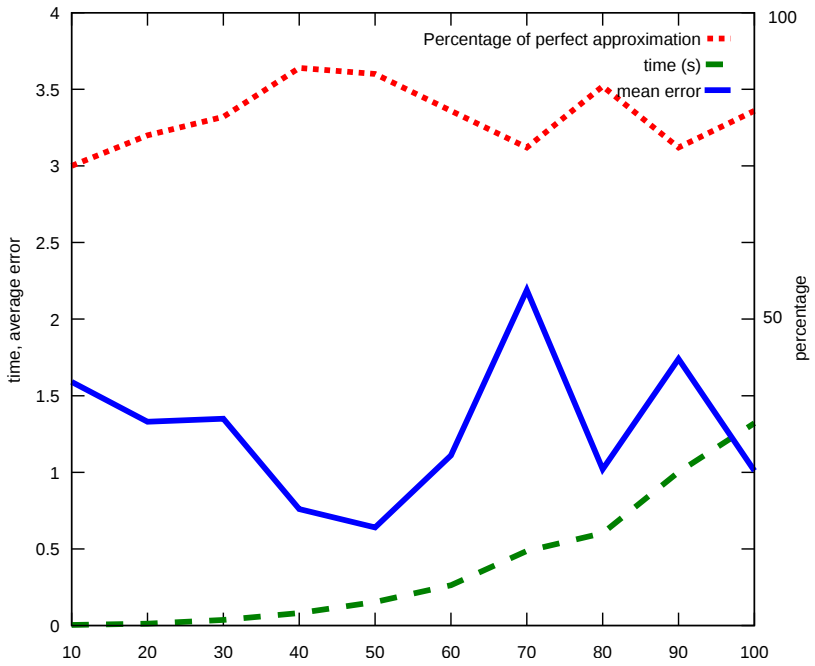


Figure 4.6: Analysis of complexity on the extended MAO dataset.

higher probability to reach it than initializations far from it. This behavior is observed in the results where better approximations are obtained by using the approach giving the best approximation considering LSAP framework. Moreover, less iterations are required to reach convergence since the algorithm is initialized close to the minima. This phenomenon explains the low differences of computational time between the different approaches. Note that we didn't test the method presented in [44] due to its high computational time. In conclusion, these results show that the QAP approach is a relevant approach to approximate graph edit distance and it outperforms methods based on LSAP formulation while keeping an interesting computational time with respect to the one required to compute an exact graph edit distance. This confirms our initial observation.

In Figure 4.6 shows how our IPFP approach scales with respect to the size of the graphs. The dashed green line corresponds to the

computational time required to compute an approximation of the graph edit distance, the plain blue line to the average approximation error and the dotted red line corresponds to the percentage of pairs for which the exact graph edit distance is computed using IPFP. The x axis corresponds to the size of the graphs. The y axis on the left of Figure 4.6 represents simultaneously the mean execution times and the average error using a same scaling. So, for instance, 0.5 should be read as 0.5 seconds on the dashed green line and as an average error of 0.5 on the plain blue curve. The y axis on the right corresponds to the percentage of exact graph edit distances computed by our algorithm and should be used for the analysis of the dotted red curve. As we can see, the accuracy of the approximation using IPFP is stable for all tested sizes. The average error for each dataset remains about 5 to 10 % of the exact graph edit distance which corresponds to a good approximation. Moreover, the percentage of perfect approximation shows that we are able to compute the exact graph edit distance for 75% to 91% pairs of graphs. From a computational point of view, the dashed green curve seems to describe a polynomial function with respect to the size of graphs. Considering a bounded number of iterations, this observation is conform with the cubic complexity associated to the algorithm used to resolve LSAP problems, which is used in each iteration of the IPFP algorithm.

Table 4.3: Accuracy and complexity scores. d is the average edit distance, e the average error and t the average computational time.

Algorithm	Alkane			Acyclic			MAO		PAH	
	d	e	t	d	e	t	d	t	d	t
A^*	15		1.29	17		6.02				
[7]	35	18	$\simeq 10^{-3}$	35	18	$\simeq 10^{-3}$	105	$\simeq 10^{-3}$	138	$\simeq 10^{-3}$
[40]	33	18	$\simeq 10^{-3}$	31	14	$\simeq 10^{-2}$	49	$\simeq 10^{-2}$	120	$\simeq 10^{-2}$
[44]	26	11	2.27	28	9	0.73	44	6.16	129	2.01
$IPFP_{\text{Random init}}$	22.6	7.1	0.007	23.4	6.1	0.006	65.2	0.031	63	0.04
$IPFP_{\text{init [7]}}$	22.4	7.0	0.007	22.6	5.3	0.006	59	0.031	62.2	0.04
$IPFP_{\text{Init [40]}}$	20.5	5	0.006	20.7	3.4	0.005	33.6	0.016	52.5	0.037

Chapter 5

Conclusions

"I've seen people spend days, if not months, researching and gathering data, but only at the end did they finally figure out what they were really looking for; then they have to redo a lot of stuff. If after a day or so you force yourself to put together your tentative conclusions, then you'll have guidance for the rest of your research."

- Robert Pozen

The definition of efficient methods to compute the similarity between graphs is a relevant open problem in structural pattern recognition, where the well known metrics applied on vectors are not suitable. To this aim, the work done in this Thesis has been devoted on improving the state of the art problems by proposing two new algorithms. In particular, the first one, described in Chapter 3, is based on exact graph matching and deals with the problem of stating if a pattern graph is contained inside a target one, i.e. the subgraph isomorphism problem. The second, described in Chapter 4, is based on the inexact graph matching and computes the similarity between two graphs by using a new formulation of the graph edit distance.

The first algorithm, VF3, is a new version of the well known VF2, that has been the state of the art for more then ten years. It has been proven to be competitive with respect to the state of the art thanks to its efficiency both in time and in memory. VF3 has been presented as an algorithm to solve the subgraph isomorphism, but similarly to VF2 it can be applied to all the exact graph matching problems (e.g. Graph isomorphism, monomorphism, maximum common subgraph and so on). Moreover, VF3 has a more flexible structure and may be easily adapted to specific applications by defining different sorting procedures and classification functions. Nevertheless, VF3 is not an arrival point, many research directions are open. Among them the most interesting one is the design of an algorithm able to exploit the advantages provided by parallel architectures in order to deal with larger graphs.

Concerning the inexact graph matching approach, we have proposed a new formulation to compute an approximated value of the graph edit distance. Such a formulation is based on the quadratic assignment and uses the strong relationship between the graph edit distance and ϵ-assignments. A new algorithm to approximate the graph edit distance has been realized by specializing the IPFP, a generic solver for quadratic assignment problems. Then, it has been compared with the state of the art. The results are promising, indeed, a more accurate approximation has been achieved without impacting significantly the execution time. Moreover, it is still

possible to improve the accuracy and reduce the time required to approximate the graph edit distance as a quadratic assignment by considering other solvers and further improvement of the IPFP algorithm.

Bibliography

[1] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification (2Nd Edition)*. Wiley-Interscience, 2000.

[2] M. Vento, "A long trip in the charming world of graphs for Pattern Recognition," *Pattern Recognition*, pp. 1–11, Jan. 2014. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0031320314000053

[3] K. Riesen, M. Neuhaus, and H. Bunke, "Graph embedding in vector spaces by means of prototype selection," in *Graph-Based Representations in Pattern Recognition*. Springer Berlin Heidelberg, 2007, vol. 4538, pp. 383–393.

[4] A. Torsello and E. R. Hancock, "Graph embedding using tree edit-union," *Pattern Recognition*, vol. 40, no. 5, pp. 1393 – 1405, 2007.

[5] D. Emms, R. Wilson, and E. Hancock, "Graph embedding using quantum commute times," in *Graph-Based Representations in Pattern Recognition*. Springer Berlin Heidelberg, 2007, pp. 371–382.

[6] K. Riesen, M. Neuhaus, and H. Bunke, "Bipartite graph matching for computing the edit distance of graphs," *Graph-Based Representations in ...*, pp. 1–12, 2007. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-72903-7{_}1

[7] K. Riesen and H. Bunke, "Approximate graph edit distance computation by means of bipartite graph matching," *Image and Vision Computing*, vol. 27, pp. 950–959, 2009.

[8] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in Pattern Recognition," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 18, no. 3, pp. 265–298, 2004.

[9] P. Foggia, G. Percannella, and M. Vento, "Graph Matching and Learning in Pattern Recognition on the last ten years," *Journal of Pattern Recognition and Artificial Intelligence*, vol. 28, no. 1, 2014.

[10] R. Burkard, M. Dell'Amico, and S. Martello, *Assignment Problems*. SIAM, 2009.

[11] J. E. Hopcroft and R. M. Karp, "A n5/2 algorithm for maximum matchings in bipartite," in *Switching and Automata Theory, 1971., 12th Annual Symposium on*, Oct 1971, pp. 122–125.

[12] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955. [Online]. Available: http://dx.doi.org/10.1002/nav.3800020109

[13] J. Munkres, "Algorithms gor the assignment and trasportation problems." *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, pp. 32–38, 1957.

[14] R. Jonker and A. Volgenant, "A shortest augmenting path algorithm for dense and sparse linear assignment problems," *Computing*, vol. 38, no. 4, pp. 325–340, 1987. [Online]. Available: http://dx.doi.org/10.1007/BF02278710

[15] T. Koopmans and M. J. Beckmann, "Assignment problems and the location of economic activities," Cowles Foundation for Research in Economics, Yale University, Cowles Foundation Discussion Papers 4, 1955. [Online]. Available: http://EconPapers.repec.org/RePEc:cwl:cwldpp:4

[16] E. L. Lawler, "The quadratic assignment problem," *Management Science*, vol. 9, no. 4, pp. pp. 586–599, 1963.

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

[18] L. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, pp. 1367–1372, 2004.

[19] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro, "A subgraph isomorphism algorithm and its application to biochemical data," *BMC Bioinformatics*, vol. 14, 2013.

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

[21] J. LARROSA and G. VALIENTE, "Constraint satisfaction algorithms for graph pattern matching," *Mathematical Structures in Computer Science*, vol. 12, pp. 403–422, 8 2002.

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

[23] C. Solnon, "Alldifferent-based filtering for subgraph isomorphism," *Artificial Intelligence*, vol. 174, no. 12–13, pp. 850 – 864, 2010.

[24] J. Ullmann, "Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism," *Journal of Experimental Algorithmics (JEA)*, vol. 15, no. 1, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1921702

[25] H. He and A. Singh, "Graphs-At-A-Time: Query Language And Access Methods For Graph Databases," *Proceedings of the 2008 ACM SIGMOD international ...*, pp. 405–417, 2008.

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

[27] S. Zhang, S. Li, and J. Yang, "GADDI: Distance Index Based Subgraph Matching In Biological Networks," *... of the 12th International Conference on ...*, 2009.

[28] P. Zhao and J. Han, "On Graph Query Optimization In Large Networks," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 340–351, Sep. 2010.

[29] W. Han, J.-h. Lee, and J. Lee, "Turbo Iso: Towards Ultra-fast And Robust Subgraph Isomorphism Search In Large Graph Databases," *... of the 2013 international conference on ...*, pp. 337–348, 2013.

[30] N. Dahm, H. Bunke, T. Caelli, and Y. Gao, "Efficient subgraph matching using topological node feature constraints," *Pattern Recognition*, Jun. 2014.

[31] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," *Proc. VLDB Endow.*, vol. 6, no. 2, pp. 133–144, Dec. 2012.

[32] V. Carletti, P. Foggia, and M. Vento, "Performance Comparison of Five Exact Graph Matching Algorithms on Biological Databases," *New Trends in Image Analysis and Processing - ICIAP*, 2013.

[33] V. Carletti, P. Foggia, M. Vento, and X. Jiang, "Report on the first contest on graph matching algorithms for pattern search in biological databases," in *Proc. of the 10th IAPR-TC15 Intl. Workshop on Graph-based Representations in Pattern Recognition (GbR2015)*, ser. LNCS, no. 9069, 2015, pp. 178–187.

[34] P. Foggia, C. Sansone, and M. Vento, "A Database of Graphs for Isomorphism and Subgraph Isomorphism Benchmarking," 2001, pp. 176–187.

[35] M. De Santo, P. Foggia, C. Sansone, and M. Vento, "A Large Database of Graphs and Its Use for Benchmarking Graph Isomorphism Algorithms," *Pattern Recogn. Lett.*, vol. 24, no. 8, pp. 1067–1079, May 2003.

[36] V. Carletti, P. Foggia, and M. Vento. (2015) A large database of graphs for benchmarking graph isomorphism algorithms. [Online]. Available: http://mivia.unisa.it/datasets/graph-database/arg-database/

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

[38] S. Fankhauser, K. Riesen, and H. Bunke, "Speeding up graph edit distance computation through fast bipartite matching," *Graph-based representations in ...*, pp. 102–111, 2011. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-20844-7{_}11

[39] F. Serratosa, "Fast computation of Bipartite graph matching," *Pattern Recognition Letters*, vol. 45, pp. 244–250, aug 2014. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0167865514001330

[40] B. Gaüzère, S. Bougleux, K. Riesen, and L. Brun, "Approximate Graph Edit Distance Guided by Bipartite Matching of Bags of Walks," in *Structural, Syntactic and Statistical Pattern Recognition*, ser. Lecture Notes in Computer Science. Springer, 2014, vol. 8621, pp. 73–82.

[41] K. Riesen, A. Fischer, and H. Bunke, "Improving Approximate Graph Edit Distance Using Genetic Algorithms," in *Structural, Syntactic and Statistical Pattern Recognition*, ser. Lecture Notes in Computer Science. Springer, 2014, vol. 8621, pp. 63–72.

[42] ——, "Combining Bipartite Graph Matching and Beam Search for Graph Edit Distance Approximation," *Artificial Neural Networks in Pattern ...*, pp. 117–128, 2014. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-11656-3{_}11

[43] M. Ferrer, F. Serratosa, and K. Riesen, "A First Step Towards Exact Graph Edit Distance Using Bipartite Graph Matching," in *Graph Based Representations in Pattern Recognition*. Springer, 2015, vol. 9069, pp. 77–86.

[44] V. Carletti, B. Gauzere, B. L., and M. Vento, "Approximate graph edit distance computation combining bipartite matching and exact neighborhood substructure distance." in *Graph Based Representations in Pattern Recognition*. Springer, 2015.

[45] G. Sierksma, *Linear and Integer Programming: Theory and Practice*, 2nd ed., ser. Advances in Applied Mathematics. CRC Press, 2001.

[46] T. Koopmans and M. Beckmann, "Assignment Problems and the Location of Economic Activities," *Econometrica*, vol. 25, no. 1, pp. 53–76, 1957.

[47] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*. New York: Holt, Rinehart and Winston, 1976.

[48] E. M. Loiola, N. M. M. de Abreu, P. O. Boaventura-Netto, P. Hahn, and T. Querido, "A survey for the quadratic assignment problem," *European Journal of Operational Research*, vol. 176, pp. 657–690, 2007.

[49] S. Gold, E. Mjolsness, and A. Rangarajan, "Clustering with a domain-specific distance measure." *Advances in Neural Information Processing Systems*, vol. 6, pp. 96–103, 1994.

[50] S. Gold and A. Rangarajan, "A graduated assignment algorithm for graph matching." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18(4), pp. 377–388, 1996.

[51] ——, "Softmax to softassign: Neural network algorithms for combinatorial optimization." *Journal of Artificial Neural Networks*, vol. 2(4), pp. 381–399, 1996.

[52] M. Leordeanu, M. Hebert, and R. Sukthankar, "An Integer Projected Fixed Point Method for Graph Matching and MAP Inference." *Neural Information Processing Systems*, 2009.

[53] Z. Y. Liu and H. Qiao, "GNCCP - Graduated Nonconvexity and concavity procedure," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 6, pp. 1258–1267, 2014.

[54] M. Neuhaus and H. Bunke, "A quadratic programming approach to the graph edit distance problem," in *Graph-Based Representations in Pattern Recognition*, ser. LNCS, 2007, vol. 4538, pp. 92–102.

[55] S. Bougleux, L. Brun, V. Carletti, P. Foggia, B. Gaüzère, and M. Vento, "A quadratic assignment formulation of the graph edit distance," NormaSTIC, FR 3638 CNRS, Normandie, France, Tech. Rep., 2015. [Online]. Available: https://bougleux.users.greyc.fr/TR/TR.pdf

[56] H. Bunke, "On a relation between graph edit distance and maximum common subgraph," *Pattern Recognition Letters*, vol. 18, pp. 689–694, 1997.