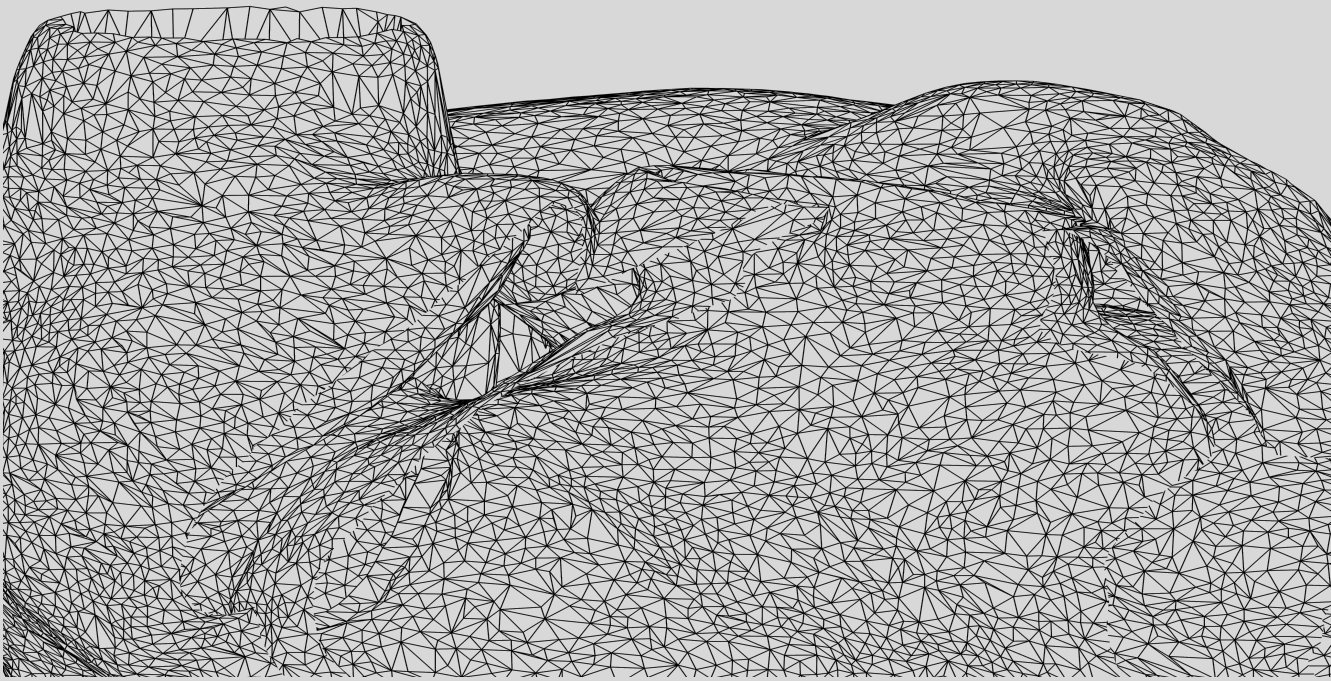
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS

Jose Luis Santacruz Muñoz

UNIVERSITAT
ROVIRA i VIRGILI

Error-Tolerant Graph Matching on Huge Graphs and Learning Strategies on the Edit Costs

Pep Santacruz Muñoz



DOCTORAL THESIS
2019

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

Error-Tolerant Graph Matching on Huge Graphs and Learning Strategies on the Edit Costs

Pep Santacruz Muñoz

Doctoral Thesis

Supervised by Dr. Francesc Serratosa

Departament of Computer Engineering and Mathematics
Univerty Rovira i Virgili

UNIVERSITAT
ROVIRA i VIRGILI

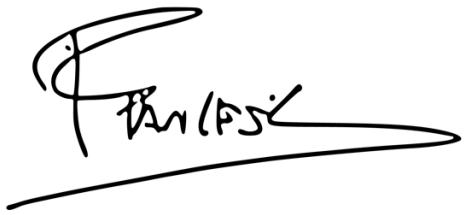
Tarragona, April 2019

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

UNIVERSITAT
ROVIRA i VIRGILI

FAIG CONSTAR que aquest treball, titulat “Error-tolerant Graph Matching on Huge Graphs and Learning Strategies on the Edit Costs”, que presenta Jose Luis Santacruz Muñoz per a l’obtenció del títol de Doctor, ha estat realitzat sota la meva direcció al Departament d’Enginyeria i Matemàtiques d’aquesta universitat.

Tarragona, 10/04/2019



Les director de la tesi doctoral

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

Dedico aquesta tesi als meus pares Maria José i Vicent.
Gràcies al vostre esforç jo he pogut tenir una educació. Us trobo a faltar.
"Les coses ben fetes només es fan una vegada". Eduard del Rey.

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

Acknowledgements

Vull donar les gràcies pel suport rebut en l'elaboració d'aquesta tesi a dues persones. La primera és el Dr. Francesc Serratosa, que m'ha acompanyat com a director durant aquest viatge. Hem discutit sobre idees, algorismes, articles, correccions, etc., i sempre ha tingut un forat per atendre'm tot i trucar-lo en cap de setmana. Sense el teu ajut aquesta tesi no hauria estat possible. La segona és la Dra. Annabel Folch que no només m'ha assessorat durant tot el procés, sinó que ha estat un pilar de recolzament indispensable en la meva vida.

Moltes gràcies.

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

Abstract

Graphs are abstract data structures used to model real problems with two basic entities: nodes and edges. Each node or vertex represents a relevant point of interest of a problem, and each edge represents the relationship between these points. Nodes and edges could be attributed to increase the accuracy of the modeled problem, which means that these attributes could vary from feature vectors to description labels. Due to this versatility, many applications have been found in fields such as computer vision, bio-medics, network analysis, etc. Graph Edit Distance has become an important tool in structural pattern recognition since it allows to measure the dissimilarity of attributed graphs. One of its main constraints is that it requires an adequate definition of the substitution, deletion and insertion of nodes and edges, which eventually determines which graphs are considered similar.

The first part of this thesis presents a method to generate a pair of graphs together with an upper and lower bound distance and a correspondence in a linear computational cost. Through this method, the behaviour of the known -or the new- sub-optimal Error-Tolerant graph matching algorithm can be tested against a lower and an upper bound Graph Edit Distance on large graphs, even though we do not have the true distance.

Next, the present thesis is focused on how to measure the dissimilarity between two huge graphs (more than 10.000 nodes), using a new Error-Tolerant graph matching algorithm called Belief Propagation Algorithm, Belief Algorithm for short. It has a $O(d^{3.5} \cdot n)$ computational cost.

For the first time, we have presented an error-tolerant graph-matching algorithm with linear computational and linear space costs with respect to the nodes. This algorithm is useful for computing the correspondence between huge graphs or social networks. The first tests were performed by the graph generation method as has been commented above.

This thesis also presents a general framework to learn the edit costs involved in the Graph Edit Distance calculations automatically. This is because we did not want to impose the costs but deduce them through an optimisation process. Then, we concretise this framework in two different models based on neural networks and probability density functions. An exhaustive practical validation on 14 public databases has been performed. This validation shows that the accuracy is higher with the learned edit costs, than with some manually imposed costs or other costs automatically learned by previous methods.

Finally we propose an application of the Belief algorithm applied to muscle mechanics. This application closes this thesis giving sense to the need of a linear graph matching method, a learning method and a method to generate synthetic graphs.

We propose a new discrete model for the simulation of muscle mechanics where the mesh grid is recomputed in each iteration. Our model solves this problem by using a graph matching algorithm that deduces a sub-optimal correspondence in linear cost our method presents higher accuracy at the expense of a linear and low increase of runtime.

Table of contents

List of figures **xiii**

List of tables **xv**

Nomenclature **xvii**

1 Introduction **1**

1.1 Graphs 1
1.1.1 Attributed Graphs 2
1.1.2 Node, Degree and other local structures 3
1.1.3 Star of a node 3
1.2 Graph Matching 4
1.2.1 Graph Edit Distance 4
1.2.2 Star Edit Distance 6
1.2.3 Suboptimal GED 7
1.3 Graph Matching Algorithms 7
1.3.1 Optimal GED computation 8
1.3.2 Sub-optimal GED computation 8
1.3.2.1 Bipartite algorithm 9
1.3.2.2 Greedy algorithm 11
1.3.2.3 Experiments evaluation 12
1.4 Contribution of this thesis 12

2 Graph Edit Distance Testing through Synthetic Graphs Generation 15
2.1 Introduction . . . 15
2.2 The method . . . 17
2.2.1 The algorithm . . . 19
2.3 Experimental Results . . . 21

3 Error tolerant graph matching in linear computational cost 29
3.1 Introduction . . . 29
3.2 The method . . . 31
3.2.1 The Belief Propagation Algorithm . . . 31
3.2.2 Computational cost . . . 35
3.2.3 A toy example . . . 38
3.3 Experimental validation . . . 49
3.3.1 Validation using synthetic graphs . . . 50
3.3.2 Real applications . . . 53

4 A general framework to learn the edit cost based on an embedded model 59
4.1 Introduction . . . 59
4.2 The method . . . 60
4.2.1 Learning Error Tolerant Graph Matching . . . 60
4.2.1.1 General Problem setting . . . 60
4.2.1.2 Definition of the stars' sets . . . 62
4.2.1.3 Embedding stars into vectors . . . 64
4.2.2 Neural Network . . . 67
4.2.3 Probabilistic Density Distribution . . . 69
4.3 Experimental Results . . . 69

5 Graph Edit distance applied to define muscle mechanics 81
5.1 Introduction . . . 81

5.2 The method 82
5.2.1 Simulation of muscle mechanics 82
5.2.2 Differential model with mesh recomputing 86
5.3 Experimental validation 88
5.3.1 Comparing graph matching algorithms 90
5.3.1.1 Comparing the sub-optimal graph matching algorithms 90
5.3.1.2 Simulating the dynamics of a human heart 93

6 Conclusions **97**
6.1 Graph Edit Distance Testing through Synthetic Graphs Generation 97
6.2 Error-tolerant graph matching in linear computational cost . 98
6.3 A general framework to learn the graph edit costs 99
6.4 Incorporating a graph matching algorithm into a muscle mechanics model 100

7 List of Publications **101**
7.1 Journals 101
7.2 Congresses 101

References **103**

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

List of figures

1.1 A graph example. 2
1.2 Most common local structures. 3
1.3 General contribution of this thesis. 13
1.4 Random Graph pair generation. 13
1.5 Linear Graph Matching in huge graphs. 13
1.6 Learning the *EditCost* function. 14
1.7 Application in muscle mechanics. 14

2.1 Graphical representation of both graphs 18
2.2 GED Distance in graph generation 23
2.3 Runtime spent by the three graph-matching algorithms. . . . 24
2.4 GED of Belief algorithm with large graphs 25
2.5 Runtime spent by the Belief algorithm. 25
2.6 Runtime spent by the graphs generation 26

3.2 Number of times *Match_Star* is executed 36
3.3 Correspondences generated by *Mach_Star* 37
3.4 k with regard to the graph order 38
3.5 Graph examples 39
3.6 Correspondence $f_{sub}(S_1, S'_1)$ between S_1 and S'_1 40
3.7 Correspondence $f_{sub}(S_2, S'_2)$. 41
3.8 Correspondence $f_{sub}(S_3, \varnothing)$. 42
3.9 Correspondence $f_{sub}(S_4, S'_6)$. 42

3.10 Final correspondence between G and G'. 50
3.11 *GED* obtained by the Bipartite, Greedy and Belief algorithms 52
3.12 Runtime of Bipartite, Greedy and Belief algorithms 53
3.13 Runtime of Belief algorithm given several graph orders . . . 54
3.14 Counting different neighbours. 55
3.15 Normalised variation of the Tarragona-Facebook social network friendships. 57

4.1 General machine learning scheme composed of two steps. . 61
4.2 General graph matching scheme. 62
4.3 Ground-truth correspondence $\hat{f}^p$ from G^p to G'^p. 63
4.4 Classes and mappings given the example of Figure 4.3 . . . 65
4.5 The E_a embedding of *Star* S_a. 66
4.6 The E_a embedding of *Star* S_a. 68

5.1 Classical iterative model for the muscle mechanics that uses a finite-element model. 83
5.2 Representation of a mesh based on hexahedrons 84
5.3 Iterative model 85
5.4 An example of the evolution of a mesh, represented as a cloud with three nodes in the new model. 86
5.5 General scheme of the Mesh recomputing model. 87
5.6 General scheme of the physical property imposition through three options: *Seeds*, substitutions and insertions. 89
5.7 Evolution of the cylinder given an initial perturbation 91
5.8 Mesh representation of a real human heart. 94
5.9 Zoomed mesh of a real human heart. 95

List of tables

3.1 Sets used by Algorithm 4 32
3.2 Debug of Algorithm 4 40
3.3 Debug of Algorithm 4 in 1^{st} iteration 41
3.4 Debug of Algorithm 4 in 2^{nd} iteration 43
3.5 Debug of Algorithm 4 in 3^{rd} iteration 44
3.6 Debug of Algorithm 4 in 4^{th} iteration 45
3.7 Debug of Algorithm 4 in 5^{th} iteration 46
3.8 Debug of Algorithm 4 in 6^{th} iteration 47
3.9 Debug of Algorithm 4 in 7^{th} iteration 48
3.10 Debug of Algorithm 4 in 8^{th} iteration 49

4.1 Main features of the fourteen databases. First experiment . . 71
4.2 Main features of the fourteen databases. Second experiment 72
4.3 Main features of the fourteen databases. Third experiment . 73
4.4 Configuration of the Neural networks and Probability density functions 74
4.5 Accuracies of the differents methods given fourteen databases 78

5.1 Normalised Graph edit distance 92
5.2 Runtime of the graph matching algorithms 93

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

Nomenclature

Roman Symbols

C	Function that represents the cost
e	Edge
f	Bijective matching between nodes
$\hat{f}$	Bijective ground truth correspondence
G	Graph
h	Histogram
s	Computational cost of computing the distance between local structures
S	The *Star* of a node
T	Set of bijection functions
v	Node or vertex
D	Number of deleted nodes or edges

Greek Symbols

Δ	Attributes values
γ	Function to assign a set of values

ϕ	Embedding function
ψ	Set of node correspondences in cost matrix C
Σ	Set of nodes
$\hat{\Sigma}$	Set of null-nodes
φ	Column of cost matrix C
K	Cost of deleting or inserting a node or edge

Subscripts

d	Deletion
e	Edge
i	Insertion
s	Substitution
v	Node or vertex

Acronyms / Abbreviations

BP	Bipartite algorithm
GED	Graph Edit Distance
$Q1$	Matrix of node substitution costs
$Q2$	Matrix of node deletion costs
$Q3$	Matrix of node insertion costs
$Q4$	Matrix filled with 0 values
Seed	One or several node-to-node mappings
Star	Local graph structure

Chapter 1

Introduction

1.1 Graphs

A graph is an abstract data structure compounded of nodes (or individual components) with a morphological relationship between them called edges (Figure 1.1). Graphs are used to model real problems, where each node represents a relevant entity (point of interest) and each edge is the relationship between nodes [34]. This definition allows to understand graphs as a composition of subparts (or subgraphs) with their own relationships inside and outside the subpart. In most cases graphs could be attributed, so the nodes and the edges could have attributes associated to better fit the modeled problem and they represent local information or characterization (in section 1.1.1 this model is described mathematically). Attributes could have different natures, from nominal labels to feature vectors, and they could be assigned to the nodes, the edges or both. The versatility and flexibility of the graphs allows them to be suitable in many applications such as biology, computer vision, network analysis or machine learning, among others. A wider scope of the state of the art of graphs representation models and applications can be found in [17, 7, 15].

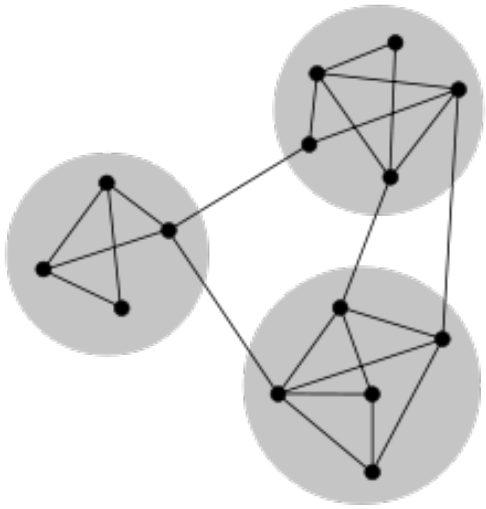
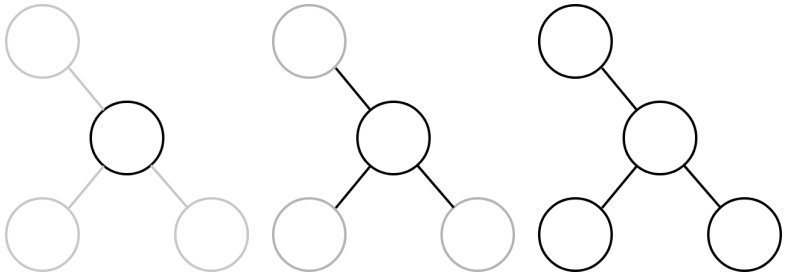


Fig. 1.1 A graph example.

1.1.1 Attributed Graphs

Let $G = (\Sigma_v, \Sigma_e, \gamma_v, \gamma_e)$ and $G' = (\Sigma'_v, \Sigma'_e, \gamma'_v, \gamma'_e)$ be two Attributed graphs. $\Sigma_v = v_a | a = 1, \ldots, n$ is the set of vertices and $\Sigma_e = e_{a,b} | a, b \in 1, \ldots, n$ is the set of edges. Functions $\gamma_v : \Sigma_v \rightarrow \Delta_v$ and $\gamma_e : \Sigma_e \rightarrow \Delta_e$ assign attribute values in any domain to vertices and edges. $\gamma_v(v_a) = v_a$ and $\gamma_e(e_{a,b}) = e_{a,b}$. Coherent definitions hold for $G' = (\Sigma'_v, \Sigma'_e, \gamma'_v, \gamma'_e)$. The symbol n stands for order of the graph.

A local structure of a node is the set of edges and nodes of the graph adjacent to it. The influence on selecting different local structures was analysed in [10, 50]. The most common local structures are the *Node* (Figure 1.2a), the *Degree* (Figure 1.2b) and the *Star* (Figure 1.2c, also called *Clique* in some papers). In the *Node*, the local structure is composed of only one node and any edges or other nodes are not considered. In the *Degree*, the local structure is composed of a node and its connecting edges. Finally, in the *Star*, the local structure is composed of a node, its connecting edges and also the nodes that



(a) Node. (b) Degree. (c) Star.

Fig. 1.2 Most common local structures.

these edges connect. These structures are defined as Attributed graphs with their specific node and edge structure.

1.1.2 Node, Degree and other local structures

The *Star* local structure is used in the present thesis instead of *Node* and *Dregree*. It has been demonstrated in [50] that this structure holds the best accuracy-runtime ratio.

The section 3.2 presents a new algorithm based in the *Star* local structure, so this structure is explained in depth in section 1.1.3. Other larger local structures have been considered but in [10] and [50], it is empirically demonstrated that using those larger structures, with Bipartite (*BP*) algorithm (see section 1.3.2.1), increases the computational cost with a negligible increment of the accuracy.

1.1.3 Star of a node

Formally, the *Star* of a node v_a, named S_a, on an Attributed graph G, is another graph $S_a = (\Sigma_v^{S_a}, \Sigma_e^{S_a}, \gamma_v^{S_a}, \gamma_e^{S_a})$ composed of the node v_a and the nodes connected to v_a by an edge and these edges. Formally, $\Sigma_v^{S_a} = v_a \cup v_b | e_{ab} \in \Sigma_e$ and $\Sigma_e^{S_a} = e_{ab} | e_{ab} \in \Sigma_e$. Moreover, $\gamma_v^{S_a}(v_b) = \gamma_v(v_b), \forall v_b \in \Sigma_v^{S_a}$ and $\gamma_e^{S_a}(e_{ab}) = \gamma_e(e_{ab}), \forall e_{ab} \in \Sigma_e^{S_a}$. Defining the *Stars* as graphs, the graph matching algo-

rithms described in the next section can also be applied to deduce the distance between *Stars*.

1.2 Graph Matching

Once the problem is modeled with graphs, the main challenge is to classify or compare them. Since early seventies until now, several graph matching algorithms have been released and many surveys have been published [7, 16, 15]. This thesis is centered in the Error-Tolerant Graph Matching algorithms to find the correspondence between graphs and the Graph Edit Distance (or *GED*, see section 1.2.1) to calculate the distance between them. This sort of algorithms have been chosen because they have the lower runtime cost.

1.2.1 Graph Edit Distance

One of the most widely used methods to deduce a distance between graphs and to extract a "logical" correspondence between them is the Graph Edit Distance [34]. In the Graph Edit Distance, distance is defined as the minimum amount of required distortion to transform one graph into the other. To this end, a number of distortion or edit operations, consisting of the insertion, deletion or substitution of nodes and edges are defined.

Edit cost functions are introduced to quantitatively evaluate the edit operations. The basic idea is to assign a penalty cost to each edit operation according to the amount of distortion introduced in the transformation. To allow maximum flexibility in the matching process, both graphs are theoretically extended with null nodes and edges to have the same order n. The *null* nodes and edges are assigned to the set $\hat{\Sigma}_v$ and $\hat{\Sigma}_e$ for graph G and $\hat{\Sigma'}_v$ and $\hat{\Sigma'}_e$ for graph G'. Thus, the deletion and insertion operations are transformed to assignations of a $non-null$ node of the first or second graph to a *null* node of the second or first graph. Substitutions simply indicate node-to-node assignations.

Using this transformation, given two graphs G and G' and a bijective matching between their nodes f, the graph edit cost $EditCost(G,G',f)$ (Equation 1.1) is computed. It is based on the following constants and functions: C_{vs} is a function that represents the cost of substituting node v_i of G with node $f(v_i)$ of G'. C_{es} is a function that represents the cost of substituting edge $e_{i,k}$ of G with edge $f(e_{i,k})$ of G'. C_{vd} and C_{vi} are the costs of deleting node v_i of G (mapping it to a *null* node) or inserting node v'_j of G' (or being mapped from a *null* node). Likewise, C_{ed} and C_{ei} are the costs of assigning edge $e_{i,k}$ of G to a *null* edge of G' or assigning edge $e'_{j,p}$ of G' to a *null* edge of G. Note that the cases in which two *null* nodes or *null* edges are mapped are not considered; this is because this cost is zero by definition. The expression $EditCost$ is formally described as follows:

$$\begin{aligned} EditCost(G,G',f) = & \sum_{\substack{v_a\in\Sigma_v-\hat{\Sigma}_v \\ v'_i\in\Sigma'_v-\hat{\Sigma}'_v}} C_{vs}(v_a,v'_i) + \sum_{\substack{e_{ab}\in\Sigma_e-\hat{\Sigma}_e \\ e'_{ij}\in\Sigma'_e-\hat{\Sigma}'_e}} C_{es}(e_{ab},e'_{ij}) + \\ & \sum_{\substack{v_a\in\hat{\Sigma}_v \\ v'_i\in\Sigma'_v-\hat{\Sigma}'_v}} C_{vd}(v_a,v'_i) + \sum_{\substack{e_{ab}\in\hat{\Sigma}_e \\ e'_{ij}\in\Sigma'_e-\hat{\Sigma}'_e}} C_{ed}(e_{ab},e'_{ij}) + \\ & \sum_{\substack{v_a\in\Sigma_v-\hat{\Sigma}_v \\ v'_i\in\hat{\Sigma}'_v}} C_{vi}(v_a,v'_i) + \sum_{\substack{e_{ab}\in\Sigma_e-\hat{\Sigma}_e \\ e'_{i,j}\in\hat{\Sigma}'_e}} C_{ei}(e_{ab},e'_{ij}) \end{aligned} \tag{1.1}$$

Where $f(v_a)=v'_i$ and $f(v_b)=v'_j$. The Graph edit distance GED is defined as the minimum cost under any bijection in T:

$$GED(G,G') = min_{f\in T}\left\{EditCost(G,G',f)\right\} \tag{1.2}$$

1.2.2 Star Edit Distance

In this thesis the Star Edit Distance is a key concept. $StarEditDistance(S,S')$ is defined as the cost of transforming the *Star* structure S into another S' (Figure 1.2c) from a graph, and is represented as follows:

- $C^S(S_a,S'_i)$: Cost of substituting a $Star\ if S,S' \neq NULL$.
- $C^D(S_a)$: Cost of deleting a $Star\ if S' = NULL$.
- $C^I(S'_i)$: Cost of inserting a $Star\ if S = NULL$.

The Equation 1.1 can be reformulated using these terms as:

$$GED(G,G',f) = \sum_{\substack{v_a \in \Sigma_v - \hat{\Sigma}_v \\ v'_i \in \Sigma'_v - \hat{\Sigma}'_v}} C^S(v_a,v_i) + \sum_{\substack{v_a \in \Sigma_v - \hat{\Sigma}_v \\ v'_i \in \hat{\Sigma}'_v}} C^I(v_i) + \sum_{\substack{v_a \in \hat{\Sigma}_v \\ v'_i \in \Sigma'_v - \hat{\Sigma}'_v}} C^D(v_a) \tag{1.3}$$

$C^S(v_a,v_i)$ denotes the cost of substituting the local structure S_a centred at node v_a by the local structure S'_i centred at node v'_i where the central nodes are forced to be mapped. $C^D(v_a)$ denotes the cost of deleting the local structure S_a and $C^I(v_i)$ denotes the cost of inserting the local structure S'_i. These local structure costs depends on the structure itself and also on the costs of nodes and edges $C^S_n(v_a,v_i)$, $C^D_n(v_a)$, $C^I_n(v_i)$, $C^S_e(v_a,v_i,v_b,v_j)$, $C^D_e(v_a,v_b)$ and $C^I_e(v_i,v_j)$.

As commented in section 1.1.3, a Star is a graph with the structure composed by a node and its adjacent edges and nodes. The *GED* between two stars comprise the costs of substituting, deleting and inserting Stars, defined as Equation 1.1 and 1.2. In the substitution case, both Stars have a non-empty structure. In the other two cases (insertion and deletion), one of the stars is an empty structure:

$$\begin{aligned} Substitution &: C^S(v_a, v'_i) = C_v(v_a, v_i) + T_(v_a, v'_i) \cdot (K_v + K_e) + \breve{C}(v_a, v'_i) \\ Deletion &: C^D(v_a) = K_v + n_(N_a) \cdot (K_v + K_e) \\ Insertion &: C^I(v'_i) = K_v + n_(N_i) \cdot (K_v + K_e) \end{aligned} \tag{1.4}$$

The Star substitution cost is composed of three terms: substitution of the central nodes:$C_v(v_a, v_i)$, deleting or inserting the external nodes: $T_(v_a, v'_i) \cdot (K_v + K_e)$ and substitution of the external edges and nodes: $\breve{C}(v_a, v'_i)$.

$T_(v_a, v'_i) = |n_{N_{v_a}} - n_{N_{v'_i}}|$ is the difference between the order of stars. Function $\breve{C}(v_a, v'_i)$ is computed through a linear solver [19] that obtains the best mapping g between external nodes of the Star. Accordingly:

$$\breve{C}(v_a, v'_i) = \sum_{a'=1}^{M_{a,i}} \left[C_v(v_{\{a'\}}, v_{g(\{a'\})}) + C_e(e_{a\{a'\}}, e_{i\, g(\{a'\})}) \right] \tag{1.5}$$

Where $M_{a,i}$ is the number of substituted external nodes, $M_{a,i} = min\{n_{N_a}, n_{N_i}\}$. Node $v_{\{a'\}}$ represents the a'^{th} external node of the Star.

The Belief Propagation graf mathching algorithm, Belief algorithm for short, is a new suboptimal algorithm presented in section 3.1 and the general learning framework presented in section 4.1 are based on the distance between *Stars* to calculate the *GED*.

1.2.3 Suboptimal GED

Some current algorithms use a suboptimal *GED* called GED^{sub} (see section 1.3.2) due to the computational cost associated with the *GED* calculation.

1.3 Graph Matching Algorithms

Error-tolerant graph matching has been demonstrated to be an NP-problem [18], therefore, some authors have presented several algorithms that apply

certain heuristics in order to reduce the computational cost [7, 15, 51] or [1]. However, sub-optimal algorithms that deduce a distance and a matching between nodes in polynomial time have also been presented. For instance, the Graduated assignment [20], the Bipartite graph matching [36, 44, 45] and [46] or the Greedy edit distance algorithm [37, 10]. All of these algorithms define a bi-dimensional matrix in which the number of rows or columns is related to the graph order.

1.3.1 Optimal GED computation

The optimal computation of the *GED* (Equation 1.2) is usually carried out by means of the A* algorithm [5, 26, 13] and [23]. In this algorithm, a tree of solutions is explored to determine the final distance between graphs, where each node represents a partial edition, and the leaves represent the whole transformation from one to another. Different A*-based algorithms have been published in the literature [12, 14] differing in accuracy and time.

Unfortunately, the computational complexity of these methods is exponential in the number of nodes of the involved graphs (see Algorithm 1 in paper [36]). For this reason, several sub-optimal methods to compute the *GED* have been presented in previous research [36, 44–46, 1, 23, 33] and [14]. The main idea is to optimize local criteria instead of global criteria, so a sub-optimal *GED* can be computed in polynomial time.

1.3.2 Sub-optimal GED computation

Computing the Graph edit distance and the optimal matching is a known NP-problem [18]. Because of this, several optimal algorithms have been defined to compute it and deduce the matching that obtains the minimum cost applying different search strategies [13]. Nevertheless, due to runtime reasons, applications usually apply suboptimal algorithms that search for a

suboptimal distance and a matching in polynomial time. In this thesis this suboptimal distance is called GED^{sub} (Equation 1.6).

One of the classical ones is the Graduated assignment [20] that has a $O(n^6)$ computational cost. Nowadays, one of the most used algorithms is the Bipartite graph matching [36, 44] that has a $O(s \cdot n^2 + n^3)$ computational cost, where s is the computational cost of computing the distance between local structures. Finally, it is worth mentioning the Greedy edit distance algorithm [37, 10] that returns a distance in $O(s \cdot n^2 + n^2)$ computational cost. All of these suboptimal algorithms try to minimize the cost of the Equation 1.3.

$$GED^{sub}(G,G') = min_{f \in T} \left\{ EditCost^{sub}(G,G',f^{sub}) \right\} \tag{1.6}$$

Where the superscript *sub* refers to the suboptimal version of the variable.

In the next subsection, the main algorithms used in these thesis to compute the sub-optimal *GED* are explained.

1.3.2.1 Bipartite algorithm

The Bipartite algorithm [36] uses a cost matrix C to approximate the *GED* to the linear assignment problem. This approximation reduces the computational cost to a cubic order ($O(n^3)$). The Equation 1.7 defines the cost matrix.

$$C = \left\{ \begin{array}{cc} \overbrace{\begin{bmatrix} C^S_{11} & C^S_{12} & \cdots & C^S_{1M} \\ C^S_{21} & C^S_{22} & \cdots & C^S_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ C^S_{N1} & C^S_{N2} & \cdots & C^S_{NM} \end{bmatrix}}^{Q1} & \overbrace{\begin{bmatrix} C^D_{11} & \infty & \cdots & \infty \\ \infty & C^D_{22} & \cdots & \infty \\ \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \cdots & C^D_{NN} \end{bmatrix}}^{Q2} \\ \underbrace{\begin{bmatrix} C^I_{11} & \infty & \cdots & \infty \\ \infty & C^I_{22} & \cdots & \infty \\ \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \cdots & C^I_{MM} \end{bmatrix}}_{Q3} & \underbrace{\begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}}_{Q4} \end{array} \right\} \quad (1.7)$$

Given two attributed graphs $G = (\Sigma_v, \Sigma_e, \gamma_v, \gamma_e)$ and $G' = (\Sigma'_v, \Sigma'_e, \gamma'_v, \gamma'_e)$ with $|\Sigma_v| = N$ and $|\Sigma'_v| = M$, the C matrix is calculated from matrices $Q1$, $Q2$, $Q3$ and $Q4$. The $Q1$ matrix represents the cost of the substitution of node i by node j C_{ij}. This cost can be only the cost of substitute the node (the distance between their attributes), or can include information of the edges. In the present thesis the *Star* set explained in section 1.1.3 is used to calculate the substitution cost. The $Q2$ matrix represents the cost of deleting the *node* i, where each *node* in G has its own delete node C^D_{ij} and it is not interchangeable. In the same way, the $Q3$ matrix represents the cost of inserting a new *node* l instead of node i in graph G', where each *node* in G' has its own insertion node l and it is not interchangeable. The cost of insertion (C^I) and deletion (C^D) is constant in this method. Finally, the $Q4$ matrix is filled with 0's and it will never be taken into account to calculate the correspondence between G an G'. Once this cost matrix C is built, the Hungarian [24] or Munkres [30] algorithms could be used to compute the correspondences that minimize the cost of the assignment problem.

The algorithm 1 is divided in two parts. In the first part, it fills the cost matrix described in Equation 1.7. In the second part, it finds the correspon-

Algorithm 1 Bipartite (G,G')

1: Build cost matrix C form G and G' (Equation 1.7)
2: $f =$ Compute the correspondences between nodes(Hungarian [24] or Munkres [30] method)
3: **return** f

dence between the nodes of both graphs solving the assignment problem with the Hungarian [24] or Munkres [30] method. This algorithm returns the final correspondence f found.

1.3.2.2 Greedy algorithm

The Greedy algorithm [37] is closely related to the Bipartite algorithm (section 1.3.2.1) because it also calculates a cost matrix C to compute the GED, but $O(n^3)$ complexity is avoided by using suboptimal $O(n^2)$ solutions for the assignment problem.

This method iterates through the cost matrix C from top to bottom, and for each row assigns every component to the minimum unused component of the current row. Algorithm 2 describes this process.

Algorithm 2 Bipartite-Greedy (G,G')

1: Build cost matrix C form G and G' (Equation 1.7)
2: $\psi = \{\}$
3: **for** $i = 1, \cdots, N+M$ **do**
4: $\quad \varphi_i = argmin_j(C_{ij})$
5: $\quad$ Remove column φ_i from C
6: $\quad \psi = \psi \cup \{(u_i \rightarrow v_{\varphi_i})\}$
7: **end for**
8: **return** ψ

This algorithm covers every row of the cost matrix C. For each iteration, it finds the minimum cost entry $\varphi_i = argmin_j(C_{ij})$ and adds to ψ the node edit operation. Finally, it removes the column φ_i to consider every column of

the cost matrix only once. This algorithm has a $O(m+n)$ computational cost, where $|\Sigma_v| = N$ and $|\Sigma'_v| = M$.

1.3.2.3 Experiments evaluation

All the experiments in this thesis have been compared at least with the Bipartite and Greedy algorithms. As mentioned in section 1.3.2, these algorithms are the most commonly used in graph matching due to runtime and accuracy reasons [34, 45, 37, 10].

1.4 Contribution of this thesis

The contributions of the present thesis are (Figure 1.3):

1. A method to generate a synthetic pair of graphs (Figure 1.4) with a $\hat{f}^{pq}$ correspondence, where the *GED* is between minimum and maximum bounds. These minimum and maximum *GED* bounds are outputs of the algorithm.

2. An error-tolerant graph matching algorithm in linear computational cost using an initial partial correspondence (Figure 1.5). The aim of this method is to handle huge graphs (composed by more than 10.000 nodes) speeding up the matching process. This method is called Belief algorithm.

3. A general framework to learn the *StartEditCost* of nodes and edges (C^S, C^I and C^D) automatically focused on the *Star* as the local structure (Figure 1.6).

4. Muscle mechanics simulation based on attributed graphs. The aim of this method is to increase the accuracy of the simulation, at the expense of a certain increase of runtime (Figure 1.7).

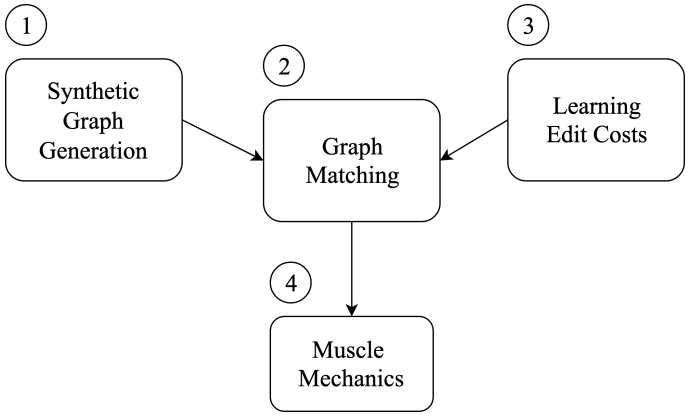


Fig. 1.3 General contribution of this thesis.

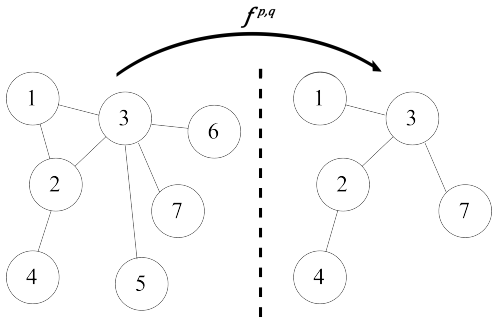


Fig. 1.4 Random Graph pair generation.

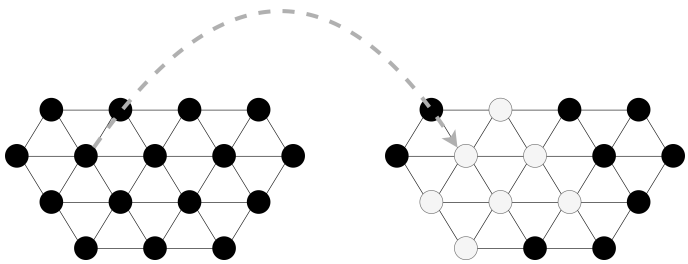


Fig. 1.5 Linear Graph Matching in huge graphs.

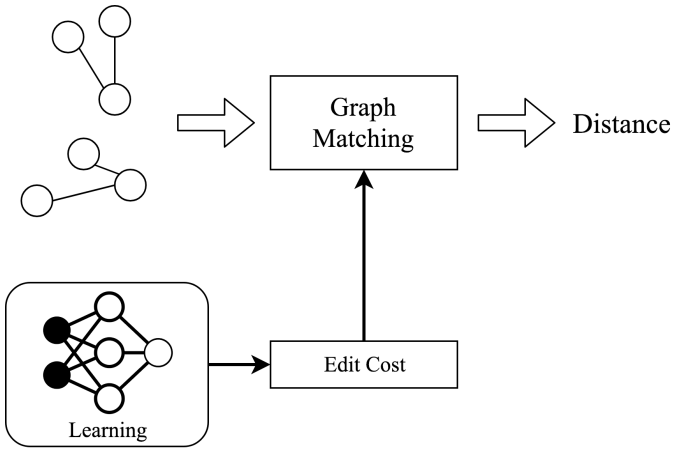


Fig. 1.6 Learning the *EditCost* function.

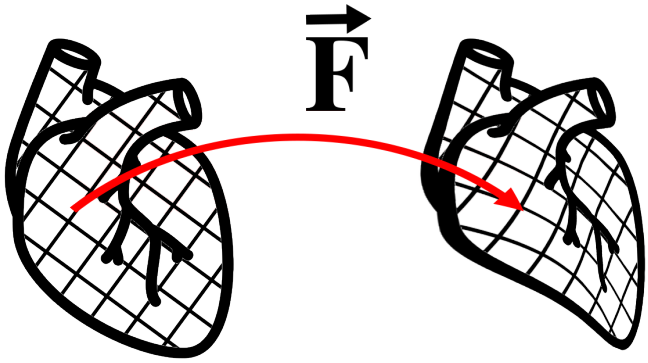


Fig. 1.7 Application in muscle mechanics.

Chapter 2

Graph Edit Distance Testing through Synthetic Graphs Generation

2.1 Introduction

Attributed graphs have found widespread applications in pattern recognition and machine learning due to their ability to represent structured objects through unary and binary local entities. Since the first use of attributed graphs in pattern recognition in early 80s, several graph-matching algorithms have been presented that compute the distance between graphs in an optimal or sub-optimal way, summarized in the following surveys [7, 15] and [52].

These authors presented an experimental validation section in all surveys [7, 15] and [52] as well as other authors did in the papers commented in this section. For the purpose of reproducibility, most of them use publically-available graph databases or synthetically-generated graphs with known parameters. Some of these databases are discussed below. The IAM graph database repository [35] maybe is the most used database in this field. It was published in 2008 and it is composed of several databases with diverse

attributed graphs, such as proteins, fingerprints, or hand written characters, among others. Another public database was presented in [6] in 2009. It is composed of attributed graphs extracted from image sequences. Graphs nodes represent salient points of these images and graph edges have been generated through Delaunay triangulation [11] or represent shape edges. Another database with unattributed graphs was presented in 2003 [43]. The aim of this database is to perform exact isomorphism benchmarking and it cannot be used for testing error-tolerant graph matching since nodes and edges are unattributed. Finally, the database presented in [29] has the particularity that the elements in the database are not classified graphs, but a triplet composed of a pair of graphs and a ground-truth correspondence between their nodes. It has been designed to test algorithms that aim to learn the graph-matching parameters.

In these databases, the average order of the graphs is between 10 and 50. This is because if the order increases, the runtime of any exact algorithm makes impossible to obtain the true graph edit distance within a reasonable time. Thus, below this graph order, is possible to deduce the gap between the exact distance and the obtained sub-optimal distance. However, with larger orders it is possible to obtain classification results or a comparison of several sub-optimal algorithms, but an exact $A*$ algorithm cannot be executed. Therefore, the current assumption results or behaviour of sub-optimal algorithms can be extrapolated from matching small graphs to large graphs. In this case, it is not clear how the error induced by the use of sub-optimal algorithms grows with problem size, since the $A*$ algorithms have never been executed on large graphs.

In this chapter, we show how to address this methodological problem by manufacturing pairs of attributed graphs of a given lower and upper graph edit distance. That turns out to be computationally easier than generating two graphs (randomly or extracted from other objects such as images, proteins, and so on) and then calculating the graph edit distance between them given

an error-tolerant graph matching algorithm. See [48] for more information. In fact, the computation of these bounds is done simultaneously while the random graph is synthesised, therefore, little effort is required.

2.2 The method

In the present section, we describe a method and an algorithm to generate graphs. The main idea is to use this algorithm to generate synthetic databases of huge graphs. Each element of these databases has four components, which are the two graphs, an upper bound distance and a lower bound distance between them. As commented in section 2.1, these bounds are going to be used to estimate how far is a computed sub-optimal distance to the true distance, without the need to compute this optimal distance. It should be noted that the computation of the upper and lower bounds is implicit in the generation of both graphs, so the computational order does not increase.

The graph generation is based on an assumed suboptimal correspondence close to the optimal one. To do so, one graph is randomly generated and the other one is generated as a sub-graph of it. Any number of attributes or type of attributes can be defined in this step. Then some edges are deleted on both graphs and finally, the attributes on some nodes and edges are altered. Figure 2.1 shows both initial graphs and its correspondence. G^p is the input graph and G^q is the output graph, with orders $n^p \geq n^q$.

The graph G^q is defined as the sub-graph of the nodes with the lowest indices in the graph G^p, plus some *null* nodes. That implies that the first nodes in G^p belong to $\Sigma_v^q - \hat{\Sigma}_v^q$ ($non-null$ nodes) and the rest of the nodes belong to $\hat{\Sigma}_v^q$ (*null* nodes). Then, we define the best correspondence $\breve{f}^{p,q}$ as the identity function:

$$\breve{f}^{p,q}(v_i^p) = v_i^q \in \begin{Bmatrix} \Sigma_v^q - \hat{\Sigma}_v^q & if 1 \leq i \leq n^q \\ \hat{\Sigma}_v^q & n^q \leq i \leq n^p \end{Bmatrix} \tag{2.1}$$

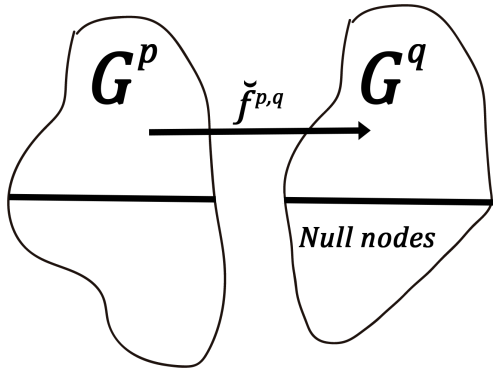


Fig. 2.1 Graphical representation of both graphs and the identity correspondence. G^q is a copy of a sub-graph of G^p with some alterations. Some extra *null* nodes have been added too.

The lower bound only has structural information and attributes are not considered.

$$DminDB = |n^p - n^q| \cdot K_v + |\hat{n}^p - \hat{n}^q| \cdot K_e \tag{2.2}$$

Where K_v is the cost of inserting a node and K_e is the cost of deleting a node.

Clearly, it is impossible to have a smaller distance than the *DminDB* value, since it only appears if G^q is an exact sub-graph of G^p .In this case, "exact" means having the same structure and also the same attribute values in the nodes and edges. The upper bound is defined as the edit cost of $\breve{f}^{p,q}$ correspondence,

$$DmaxDB = EditCost_{K_v,K_e}(G^p, G^q, \breve{f}^{p,q}) \tag{2.3}$$

The computational cost of $EditCost$ is quadratic with regard to the number of nodes. This is because the algorithm has to sum the cost of matching all the nodes and edges. Note the maximum number of edges is also quadratic with regard to the number of nodes. Nevertheless, this algorithm has the advantage of deducing the *DmaxDB* bound through the process of generating both graphs with a negligible cost, and therefore it does not need the specific computation

of the $EditCost$ function. At the end of the method, nodes of G^q are reordered to avoid knowing in advance the best considered correspondence.

2.2.1 The algorithm

Algorithm *Random_Graph_pair* (algorithm 3) returns two graphs and the upper and lower bound distance. A Matlab implementation is available in [47]. To properly define $EditCost$ (Equation 1.1), the same order is needed in both graphs, although this is not necessary from the algorithmic point of view. For this reason, the generated graphs have different orders.

The input parameters of the algorithm are:

- n^p :the number of nodes of the larger graph.
- D_v :the number of deleted nodes.
- D_p: the number of deleted edges in both graphs without considering the edges deleted (because their adjacent nodes have been deleted).
- S_v: number of modified nodes.
- S_e: number of modified edges.
- K_v: cost of deleting or inserting a node.
- K_e: cost of deleting or inserting an edge.

The output parameters of the algorithm are:

- G^p and G^q: randomly generated graphs.
- *DminDB* and *DmaxDB*: lower and upper bounds

Random_Graph_Pair function generates a graph of order n^p. Some parameters could be added to this function, such as the number or type of attributes, or the degree of the graph. The maximum computational cost of

Algorithm 3 Random_Graph_Pair

1: $G^p \leftarrow Random\_Graph(n^p)$
2: $(G^q, StructureCost) \leftarrow Delete(G^p, D_v, D_e)$
3: $(G^q, SubstituteCost) \leftarrow Modify(G^q, S_v, S_e)$
4: $(G^p, G^q) \leftarrow Reorder(G^p, G^q)$
5: $DminDB \leftarrow |n^p - n^q| \cdot K_v + |\check{n}^p - \check{n}^q| \cdot K_e$
6: $DmaxDB \leftarrow StructureCost + SubstituteCost$

this function is linear regarding to the number of nodes. Note that generating a node means also adding its adjacent edges.

The delete function generates another graph G^q as a copy of G^p, but the D_v nodes that have the highest indices and their adjacent edges are deleted. In this step $\check{D}_e$ is defined as the number of deleted edges. In other words, they are deleted because they were connected to the deleted nodes. The order of this new graph is $n^q = n^p - D_v$. Moreover, in a second step, D_e extra edges are deleted in G^p and also in G^q. In this second step, the function imposes that the deleted edges in one of the graphs do not coincide on the equivalent edges of the other graph (the second graph is a sub-graph of the first one). Thus, each edge deletion implies an increase in the edit cost of K_e (Equation 1.1) since the other graph has the deleted edge. The function also returns *StructureCost* (Equation 2.4), which is the edit cost generated by the whole deletions of nodes and edges:

$$StructureCost = D_v \cdot K_v + (\hat{D}_e + 2 \cdot D_e) \cdot K_e \tag{2.4}$$

The modify function randomly alters the attributes of S_v nodes and S_e edges in G^q. Again, other parameters could be added, such as the model of the introduced noise. Moreover, this function returns *SubstituteCost*, which is the cost of consiering only the differences between attributes.

$$SubstituteCost = \sum_{i=1}^{n^q} C_{vs}(v_i^p, v_i^q) + \sum_{i=1}^{n^q} C_{es}(e_{i,j}^p, e_{i,j}^q) \tag{2.5}$$

SubstituteCost is the edit cost generated by these modifications and it is computed as the sum of the distances between the old attributes and the updated ones for each modified node or edge C_{vs} and C_{es}, respectively. The implicit correspondence between these graphs is $\breve{f}(p,q)$ (Equation2.1).

The upper bond *DmaxDB* computed in the algorithm holds Equation 2.3. This is because *SubstituteCost* (Equation 3.13) obtains the same value than the first two terms of Equation 1.1 and *StructureCost* (Equation 2.4) obtains the same value than the other four terms of Equation 1.1.

Finally, the function *Reorder* performs two actions. First, it randomly swaps the indices of nodes to prevent the $\breve{f}(p,q)$ correspondence to be the identity. And second, it swaps randomly both graphs to avoid the first graph to always have the larger order.

2.3 Experimental Results

The aim of the practical experimentation is twofold. On one hand, we want to validate the distance deduced by algorithms that compute the graph edit distance in a sub-optimal way, which have the lowest runtimes. Thus, we have compared the bipartite graph matching that has a cubic cost, the greedy graph matching that has a quadratic cost and the Belief algorithm (section 3.2), which has a linear computational cost. On the other hand, we want to show that our method can generate a pair of large graphs with their upper and lower bounds in a reasonable time. And therefore, we can test the fastest algorithm, which is the Belief algorithm, with large graphs. Algorithms have been implemented in Matlab and they have been run in a MacPro i5. The Matlab code and the experiments are publicly available at [47].

Belief 3.2 algorithm needs an initial *seed* that is composed only of one node to node mapping. The mapping of the first node is deduced through the implementation of the graph generation algorithm.

The experiments have been set up as follows: graphs have an order of n and a degree of $0.2 \cdot n$. Nodes have only one attribute, a natural number between 0 and 99. Edges are unattributed. The number of distorted and deleted nodes is $S_v = 0.1 \cdot n$ and $D_v = 0.3 \cdot n$, respectively. The number of inserted and deleted edges is $D_e = 0.2 \cdot n^2$. Since edges are unattributed, $S_e = 0$. Finally, $K_v = K_e = 50$ (it is one-half of the maximum value [46]). Since these values have been selected randomly, experiments can be easily rerun with different parameters given the code provided in [47].

Figure 2.2 shows the normalised distances by the average orders of both graphs computed by the three algorithms. Note that the obtained values are the approximated *GEDs*, as commented in section 1.2.1. That means that the value is the addition of the mapped local structures called *Stars*. The lines are the upper and lower bounds returned by the graph generation algorithm.

As commented in Section 2.1, the the upper and lower bounds are known instead of the true distance. The three algorithms calculated (*BP* [36], *Greedy* [37] and Belief [38]) algorithms are between these values. The Belief propagation is the closest to the middle of these bounds although that does not mean that it is the closest to the real distance. The larger the graphs are, the longer will be the gap between these algorithms. This is because the gap $DmaxDB - DminDB$ tends to increase since it can be easily deduced that it is $SubstituteCost + 2 \cdot D_e \cdot K_e$, where $SubstituteCost$ increases when the number of modified nodes augments.

Figure 2.3 compares the average runtime spent by the three algorithms to compute the graph edit distance. The $Bipartite$ graph matching and the $Greedy$ algorithm have comparable runtimes although the computational cost of the first one is $O(s \cdot n^2 + n^3)$ and the computational cost of the second one is $O(s \cdot n^2 + n^2)$. This is because the cost of compute the first step of these algorithms, $s \cdot n^2$, is much larger than the cost of compute the second step, n^3 or n^4, given the orders n of the analysed graphs.

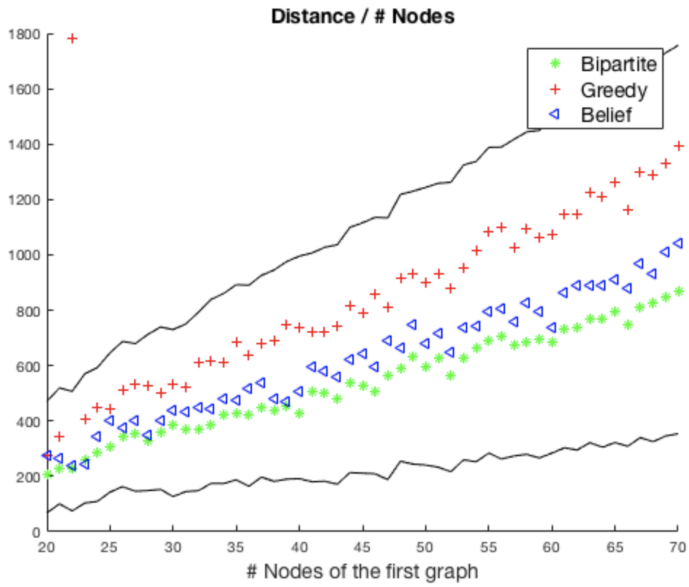


Fig. 2.2 The dots represent the graph edit distance deduced by the three algorithms divided by the number of nodes of the first graphs. Each value is the average of 3 runs. The lines are the upper and lower bounds deduced by the graph generation algorithm.

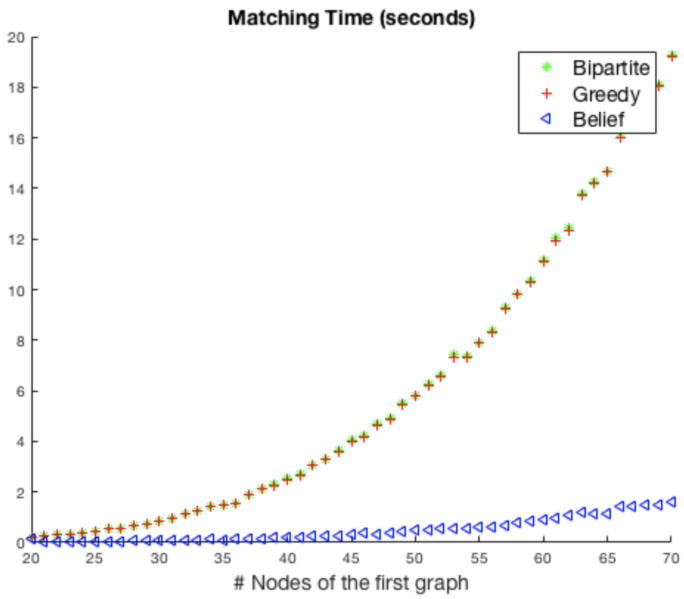


Fig. 2.3 Runtime spent by the three graph-matching algorithms.

Figure 2.4 displays the average distance of three runs normalised by the order of the first graph. This is computed by the Belief algorithm, which is the fastest algorithm. It is the first time that the graph edit distance computed by a sub-optimal algorithm has been tested with graphs that have 300 nodes considering the two bound distances. This algorithm keeps computing a distance inside the gap of both bounds in a linear computational cost.

Figure 2.5 shows the runtime of the Belief algorithm. Although the computational cost of the algorithm is linear, the runtime is almost linear. It is important to remark that the comparison of two graphs of 300 nodes only takes 25 seconds.

Finally, Figure 2.6 presents the runtime spent to generate the pair of synthetic graphs and the two bounds. In this case, the graphs generated have 600 nodes. Note that the algorithm only spends less than 7 seconds to generate a 600-node pair of graphs. As commented before, this code could be speeded up by being analysed in depth or by coding in C language.

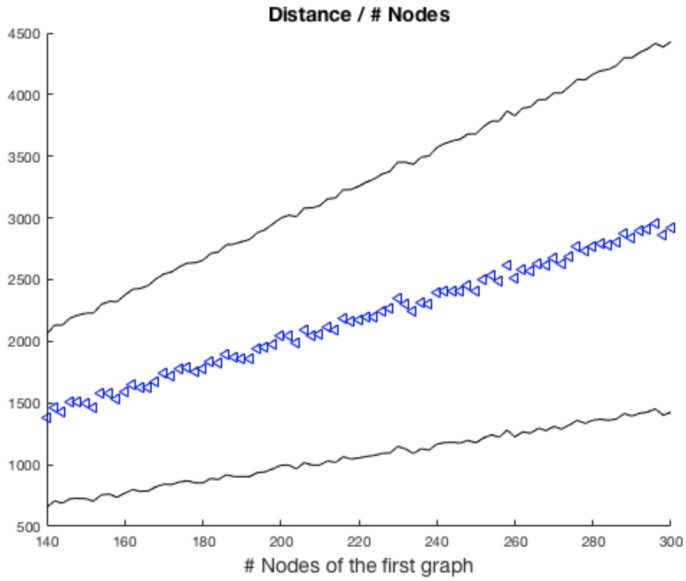


Fig. 2.4 The dots are the graph edit distance deduced by the Belief algorithm. Each value is the average of 3 runs. Lines are the upper and lower bounds deduced by the graph generation algorithm.

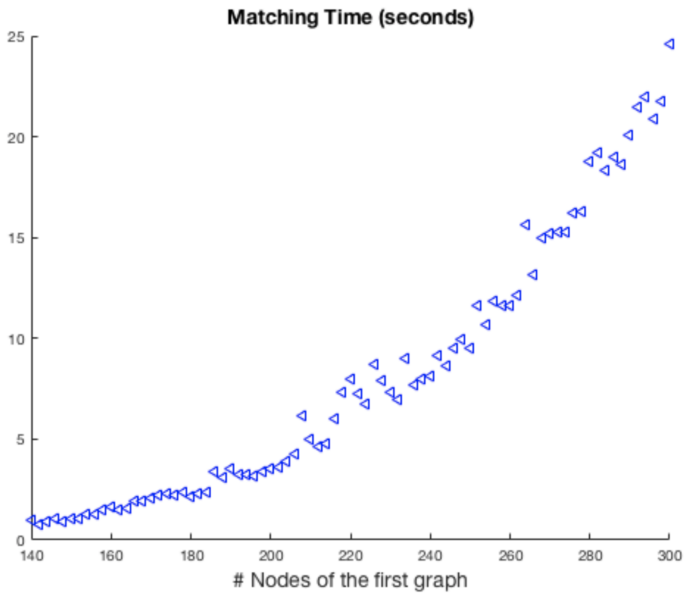


Fig. 2.5 Runtime spent by the Belief algorithm.

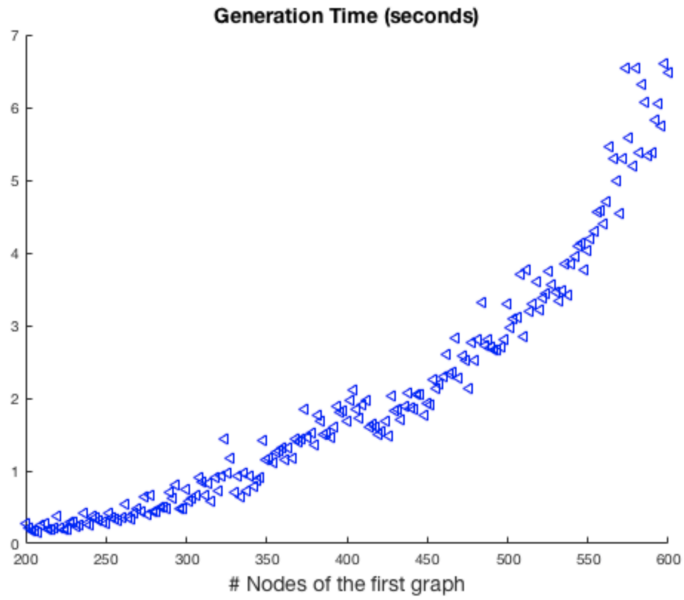


Fig. 2.6 Runtime spent to generate the pair of graphs and the upper and lower bounds, with regard to the number of nodes of the generated graphs. Each value is the average of three runs.

Comparing the generation algorithm and the believe propagation algorithm, it can be concluded that the generation of a pair of graphs is approximately 20 times faster than matching it with the fastest algorithm. This result means that, in a graph-matching testing process, the temporal effort to generate the synthetic graphs is negligible regarding the temporal effort needed to perform the graph matching test.

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

Chapter 3

Error tolerant graph matching in linear computational cost

3.1 Introduction

In recent years, there has been an increase in the number of people registered in the social networks and also in the number of different social networks. In some applications, for instance, personalised advertising, it would be interesting to locate people from one network on another network, in order to increment the knowledge about these people. It is worth noting that in some cases, the nodes in each network that represent the same person are known, since this knowledge is available from other sources of information. However, this is not the most common case, given that several people could have the same name in the net, or people could use different aliases in each network. These few mappings between both networks will be called *Seeds* and they represent crucial information in the model presented here. Figure 3.1 shows a simple example of these *Seeds*.

When applications handle huge graphs (composed by more than 10,000 nodes), the matching process becomes an important handicap, not only from the runtime point of view, but also because of the storage space. The graph-

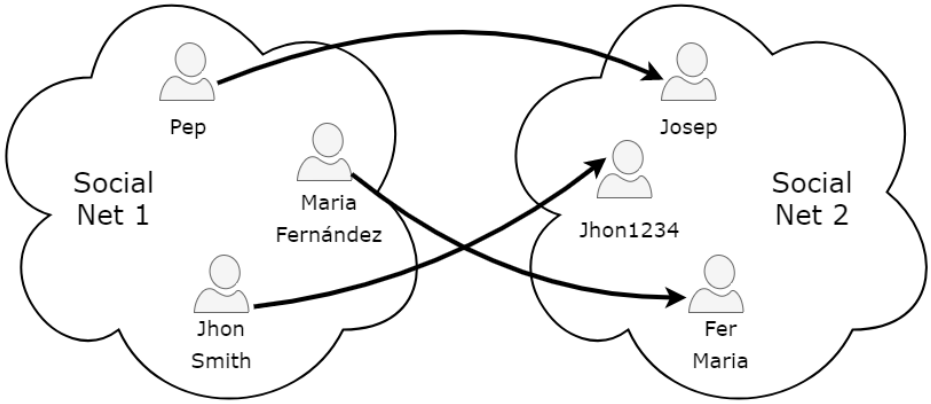


Fig. 3.1 Two social networks and three initial mappings called *Seeds*.

matching algorithm presented in this thesis is designed to match huge graphs. Thus, is considered that the computational costs of both steps in these algorithms are too high. Therefore, this highlights the need to define an algorithm that, on one hand, merges both steps and, on the other hand, computes the mappings between local structures of specific selected pairs of nodes. To do this, the computational cost has to be linear with regard to the order of the graphs. This aim is achieved by spreading the knowledge of the correct mapping through the local structures of both graphs and, therefore, the algorithm computes the matching between *Stars* that will be more than likely to be in the final correspondence.

In Section 3.2.2, it is empirically deduced that the computational cost of the algorithm presented in this thesis is $O(s \cdot \sqrt{d} \cdot n)$, where s is the computational cost of mapping the local structures, d is the number of the output edges per node and n is the order of the graphs. Typically, when the *Star* is the local structure and the matching between them is done by the *Bipartite* graph matching, [36, 44], the final computational cost of this algorithm is $O(d^3 \cdot \sqrt{d} \cdot n)$.

3.2 The method

3.2.1 The Belief Propagation Algorithm

Algorithm 4 shows the pseudo-code of the error-tolerant graph-matching algorithm presented in this thesis, called Belief Algorithm. The input of the algorithm is the same as any error-tolerant graph-matching algorithm that computes the Graph edit distance in addition to the initial *Seeds*. In other words, the input is composed of a pair of graphs, the edit cost functions and a set of *Seeds*. The output is the deduced node-to-node correspondence between both graphs. The Belief algorithm assumes that the initial knowledge is composed of the *Seeds*, and this knowledge is spread or propagated through both graphs.

The core of the algorithm is the distance between *Stars* of both graphs and, for this reason, there is always an ordered and updated set of correspondences between the *Stars* of both graphs. Thus, in each iteration, the algorithm selects the node-to-node mapping from this set whose *Star* distance is the minimum one as a correct and definitive mapping between two nodes. Afterwards, the algorithm computes the whole *Star* distances between the mapped neighbour nodes and introduces them into this set.

More specifically, the algorithm uses the following four sets, which are summarised in Table 3.1.

Seeds: Set of initial mappings between nodes of both graphs that are supposed to be ground-truth mappings. Each initial correspondence is represented by $[Seed, Seed']$ where $Seed \in \Sigma_v$ and $Seed' \in \Sigma'_v$.

Matching: The output of the program. Each element is a correspondence between a node of each graph $[v, v']$, $Seed \in \Sigma_v$ and $Seed' \in \Sigma'_v$ and it represents a bijective function. During the execution of the algorithm, this set always increases, given that any pair of nodes is never deleted from *Matching*. It represents the current partial matching.

Table 3.1 Sets used by Algorithm 4

Sets	Data in each register
Seeds	$[v_a, v'_i]$
Pending	$[v_a, v'_i], D, f$
Computing	$[v_a, v'_i]$
Matching	$[v_a, v'_i]$

Pending: A Set of registers composed of three elements: a pair of mapped nodes $[v, v']$, $Seed \in \Sigma_v$ and $Seed' \in \Sigma'_v$; the Graph edit distance D and the correspondence f between the *Stars* that are the central nodes. This distance and matching are computed through the function $(D, f) = Match\_Star([S, S'])$, where S and S' are the *Stars* of v and v', respectively. In other words, $D = EditDist(S_a, S'_i)$ and $f = f_{sub}(S_a, S'_i)$. For each iteration of the algorithm, a mapping with the minimum distance is extracted and erased from *Pending*. The algorithm finishes when *Pending* is empty, meaning that the algorithm has explored all the mappings introduced.

Computed: A Set of pairs of nodes where $Match\_Star([S, S'])$ has been computed. It is necessary in order to not compute this function several times with the same pair of nodes. Note that this set always increases, since a pair of nodes is never deleted from it.

The Belief algorithm is detailed in Algorithm 4. It is composed of three main parts:

In the first part (lines 1-9), the imposed mappings called *Seeds* are introduced into *Computed* and *Pending*. To do so, the *Match_Star* computation is applied to all of them. This is because the mappings are known but not the distance between them. It must be highlighted that an imposed *Seed* could not be considered a final *Matching*. This is because it may happen that another

option is costless, or it could be a contradiction between *Seeds* (for instance, not being a bijective mapping).

In the second part (the rest of the lines), the algorithm iteratively extracts from *Pending* the mappings $[v,v']$ which have the minimum *Star* distance (Line 12). These mappings are always considered part of the final matching and therefore they are inserted in *Matching* (Line 13). Any mappings in *Pending* that have one of the nodes v or v' are selected in Line 12, and are deleted from *Pending* to force the matching to be bijective (Line 14). Symbol ~means any value. Line 15 selects each mapping, $[w,w']$ of the matching between *Stars* obtained in Line 12. The aim of the loop in lines 15 - 25 is to compute the distance between *Stars* of the mapped neighbourhood nodes, $[w,w']$ and insert them into *Computed* and *Pending*. This action is performed only if they have not been previously computed (Line 16) and if the involved nodes of both graphs are not part of the partial current matching (Line 17). This ensures the obtention of a bijective mapping. The latter establishes the partial correspondence to spread through the connected nodes. Per each pair of mapped nodes, their *Star* is obtained (lines 18 and 19) and *Match_Star* is computed (line 20) to deduce their distance and node-to-node mapping. Then, the mapped nodes $[w,w']$ are inserted in *Computed* and *Pending* (lines 21 and 22).

In the third part, the non-considered nodes of both graphs are added into the final correspondences considering that in some cases, not all the nodes of both graphs are included in *Matching* when the algorithm completes the execution in Line 26. For this reason, in Line 27 the non-included nodes in G are included as a deletion and the non-included nodes in G' are included as an insertion.

Algorithm 4 Belief Propagation Graph Matching (G,G',*Seeds*)

Initialization:
Initialize *Pending*, *Matching* and *Computed* to the empty set
for each register in *Seeds*: [Seed,Seed'] **do**
    $S \leftarrow Star(G, Seed)$
    $S' \leftarrow Star(G', Seed')$
    $(D, f) \leftarrow Match\_Star(S, S')$
    Insert [Seed,Seed'] into *Computed*
    Insert {[Seed,Seed'], D, f} into *Pending*
end for
Spreading:
while *Pending* not empty **do**
    $\{[v, v'], D, f\} \leftarrow Min\_DistancePending$
    Insert [v,v'] into *Matching*
    Delete [v, ~], ~, ~ and [~, v'], ~, ~ from *Pending*
    for all mapping [w,w'] so that w=f(w') **do**
        if [w,w'] not in *Computed* **then**
            if not ([w, ~] or [~,w']) in *Matching* **then**
                $S \leftarrow Star(G, w)$
                $S' \leftarrow Star(G', w')$
                $(D, f) \leftarrow Match\_Star\ (S, S')$
                Insert [w,w'] into *Computed*
                Insert [w,w'], D, f into *Pending*
            end if
        end if
    end for
end while
Insert in *Matching* the nodes not considered until now.
return *Matching*

3.2.2 Computational cost

In this section, the computational cost of the algorithm is empirically deduced as $O(s \cdot \sqrt{d} \cdot n)$. To do so, the number of times that function *Match_Star* is executed is $\sqrt{d} \cdot n$ (s is the computational cost of *Match_Star*).

In the following experiments, two pairs of synthetic graphs are generated using the method described in section 2.1. Figure 3.2 shows the number of times that the function has been executed normalised by the order of the graphs. This value is called k. Distortion = 0 means that both graphs are the same; Distortion = 1 means that they are completely different. Degree = 0 means no edges; Degree = 1 means that the whole nodes are connected. The Distortion represents the variability of the Gaussian noise applied to the attributes with mean=0.

Firstly, k increases when n also increases (the same colour represents the same order). This behaviour is analysed in depth later, since the algorithm has to be used with huge graphs.

Secondly, the more different the graphs are, the more k increases when the percentage of distortion increases. Suppose two *Stars* in both graphs that share some neighbouring nodes and the function *Mach_Star* is executed in both pairs of *Stars*. On the left, Figure 3.3 shows the case where both graphs are similar. In this case, the two correspondences generated by *Mach_Star* tend to be similar. On the right, Figure 3.3, shows the case where both graphs are different (the structure is the same but the attributes are different). In this latter case, the two correspondences generated by *Mach_Star* tend to be different. In the left case, after running both comparisons between *Stars*, the elements $[v_1, v'_1]$, $[v_4, v'_4]$ and $[v_5, v'_5]$ have been added to the *Computed* set. However, in the right case, the new elements are $[v_1, v'_1]$, $[v_4, v'_4]$, $[v_5, v'_5]$, $[v_1, v'_4]$, and $[v_4, v'_1]$. Therefore, two extra *Star* comparisons have been made due to the difference between the attributes of the nodes.

Thirdly, k increases when the ratio of the graph degree tends to be 0.5. When the degree is small, a few times *Mach_Star* is computed in the loop in

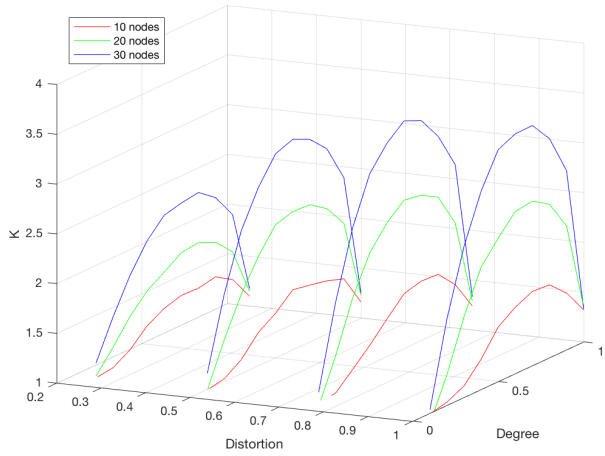


Fig. 3.2 k: Number of times *Match_Star* is executed normalised by the order of the graphs. The aspects considered here are the several degree ratios of the graphs, the distortions applied while generating the second graphs, and the orders of both graphs.

lines 15 - 25. When the degree is large, *Mach_Star* is computed several times in the loop in lines 15 - 25. Consequently, this causes the condition in Line 16 to discard the following computations of *Mach_Star* in the next executions of the loop in lines 11 - 26.

Figure 3.4 displays the evolution of k with regard to the graphs order when Distortion=1, and with different values of d (the number of output edges per node). It could be considered that k is independent of the order of the graphs and also, that it could be approximated by $k = \sqrt{d}$. Thus, it can be concluded that the computational cost of the Belief algorithm presented here is $O(s \cdot \sqrt{d} \cdot n)$. As commented in sections 1.1 and 1.2, *Mach_Star* is usually solved by the Bipartite graph matching algorithm [36], [44] applied only to the *Stars*. Considering that, it has a cubic cost with respect to the number of nodes, then $s = d^3$. In this case, the final cost of the Belief algorithm is $O(d^3 \cdot \sqrt{d} \cdot n) = O(d^{3.5} \cdot n)$. It is important to emphasise that it is the first

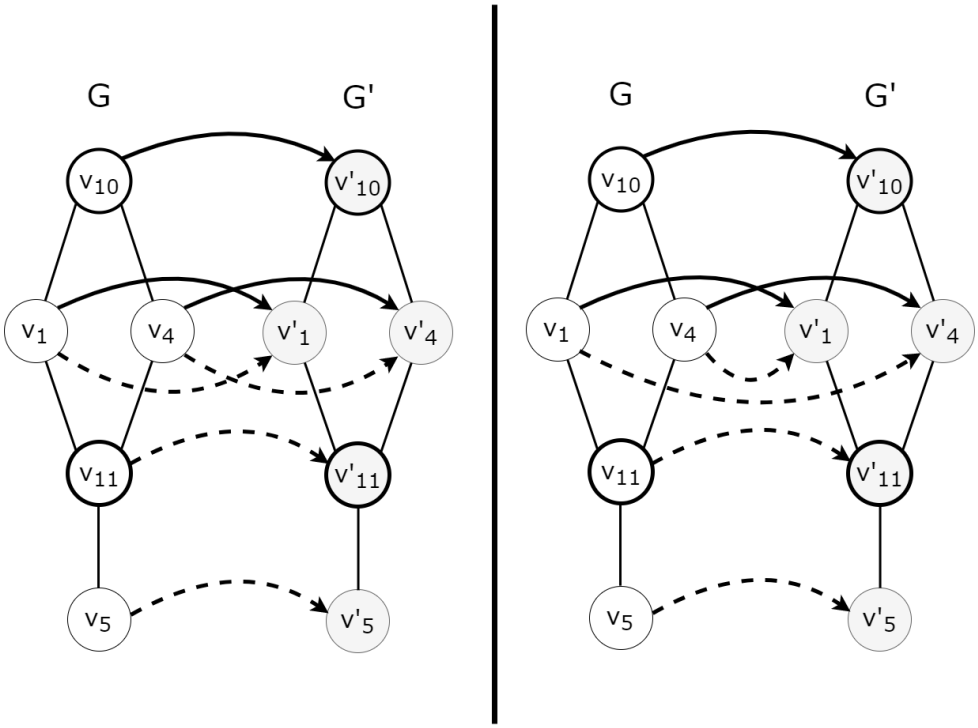


Fig. 3.3 Correspondences generated by *Mach_Star* given two pairs of stars. Plain arrows represent the correspondence generated by the *Star* when the mapping $[v_{10}, v'_{10}]$ is the central node. Dashed arrows represent the correspondence generated by the *Star* when the mapping $[v_{11}, v'_{11}]$ is the central node. In the left case the graphs are similar, and in the right case the attributes of the graphs are different. *Star* correspondences between similar graphs tend to share node-to-node mappings, but this is not the case when graphs are different.

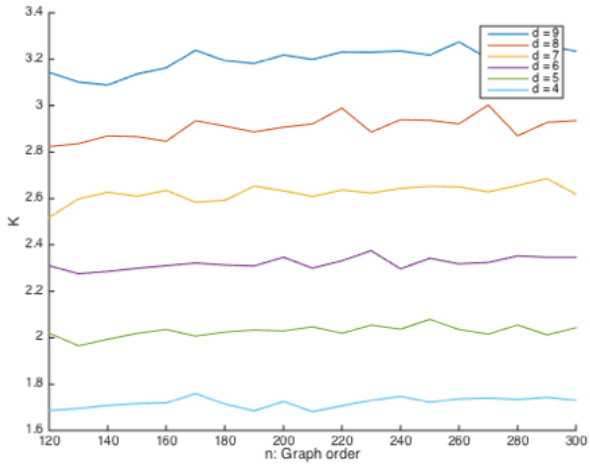


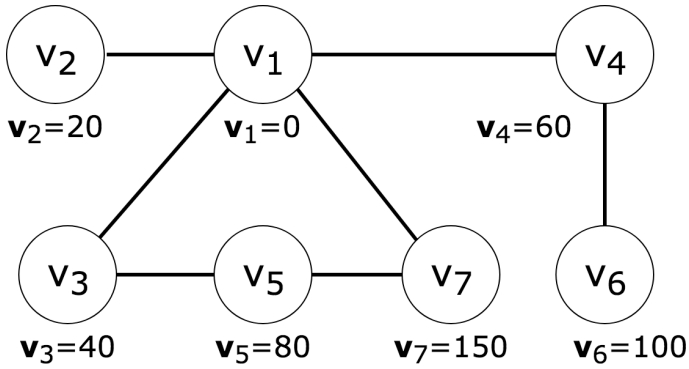
Fig. 3.4 k with regard to the graph order when the compared graphs are completely different, and with various values of d (the number of output edges per node). It can be deduced that k can be approximated by $\sqrt{d}$.

time that an error-tolerant graph matching algorithm has been applied to a graph with 5000 nodes.

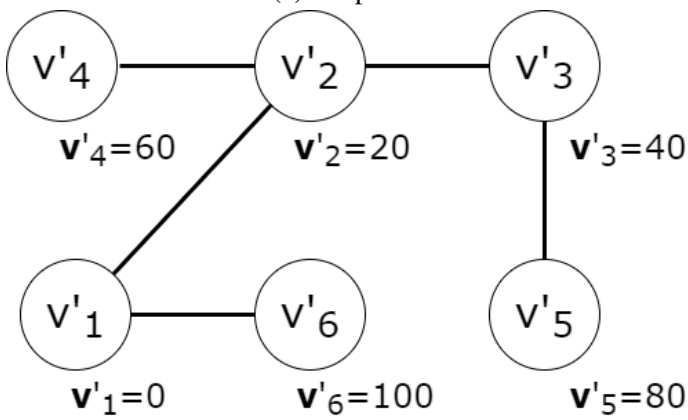
3.2.3 A toy example

In this section, the Algorithm 4 is debugged using the graphs shown in Figure 3.5 with the aim of explaining properly how it works. The nodes have only one attribute which is a Natural number, and the edges are unattributed. The value of the variables considered are $C_{vd} = C_{ed} = C_{vi} = C_{ei} = 25$ and $C_{vs}(v_a, v'_i) = |v_a - v'_i|$ (remember that $\gamma_v(v_a) = v_a$ and $\gamma'_v(v'_i) = v'_i$). Moreover, it is assumed that the initial set of *Seeds* is composed of only one pair, which is $[v_1, v'_1]$.

Line 1 initializes the sets *Pending*, *Matching* and *Computed* to an empty set. Then, from Line 2 to Line 8, the algorithm imposes the node correspondence between the two graphs contained in the *Seeds* set. Table 3.2



(a) Graph G.



(b) Graph G'.

Fig. 3.5 Graph examples

Table 3.2 Debug of Algorithm 4

Sets	Data in Line 8
Seeds	$[v_1, v'_1]$
Pending	$[v_1, v'_1], 0, f_{sub}(S_1, S'_1)$
Computing	$[v_1, v'_1]$
Matching	$[\varnothing]$

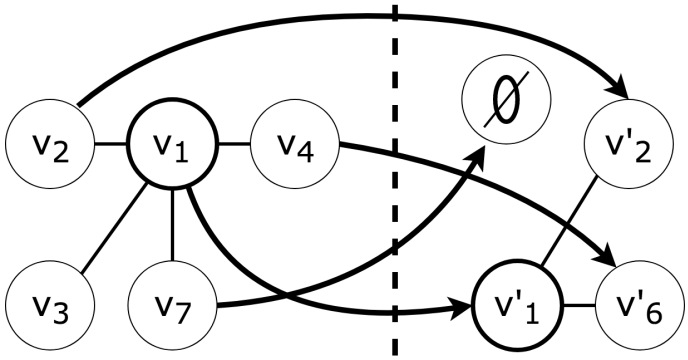


Fig. 3.6 Correspondence $f_{sub}(S_1, S'_1)$ between S_1 and S'_1. Central nodes are highlighted in bold.

shows the content of each set at this initialisation and Figure 3.5b shows the correspondence $f_{sub}(S_1, S'_1)$.

In the first iteration of the **While** (from Line 11 to Line 26), the algorithm extracts the pair $[v_1, v'_1]$ from *Pending* and inserts it into *Matching*. Then, it computes *Match_Star* for all neighbours that are currently mapped by the correspondence $f_{sub}(S_1, S'_1)$ (from Line 15 to Line 25). Table 3.3 shows the content of each set at this point. Figure 3.7, Figure 3.8 and Figure 3.9 provide the correspondences $f_{sub}(S_2, S'_2)$, $f_{sub}(S_3, \varnothing)$, $f_{sub}(S_4, S'_6)$ and $f_{sub}(S_7, \varnothing)$, respectively.

In the second iteration of the **While**, the mapping $[v_2, v'_2]$ in *Pending* has the minimum *Star* distance. Therefore, this pair of nodes is selected, deleted

Table 3.3 Debug of Algorithm 4 in 1^{st} iteration

Sets	**Data in the first iteration of While**
Seeds	$[v_1, v'_1]$
Pending	$[v_2, v'_2], 12.5, f_{sub}(S_2, S'_2)$ $[v_3, \varnothing], 41.7, f_{sub}(S_3, \varnothing)$ $[v_4, v'_6], 22.5, f_{sub}(S_4, S'_6)$ $[v_7, \varnothing], 41.7, f_{sub}(S_7, \varnothing)$
Computed	$[v_1, v'_1]$ $[v_2, v'_2]$ $[v_3, \varnothing]$ $[v_4, v'_6]$
Matching	$[v_1, v'_1]$

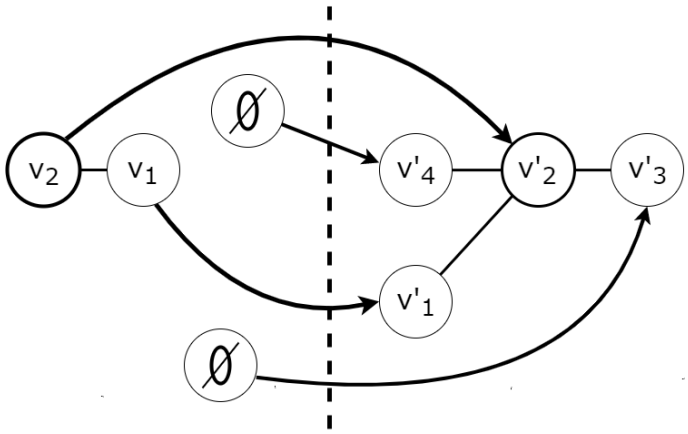


Fig. 3.7 Correspondence $f_{sub}(S_2, S'_2)$.

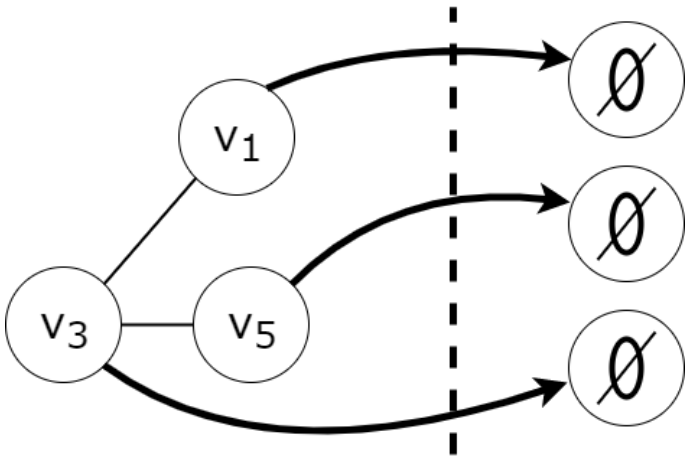


Fig. 3.8 Correspondence $f_{sub}(S_3, \varnothing)$.

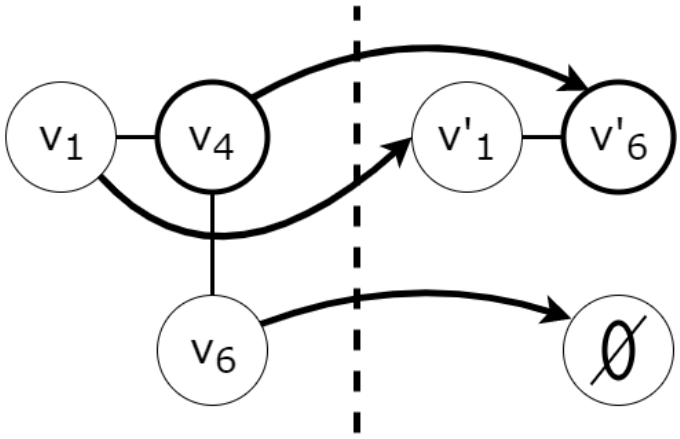


Fig. 3.9 Correspondence $f_{sub}(S_4, S'_6)$.

Table 3.4 Debug of Algorithm 4 in 2^{nd} iteration

Sets	**Data in the second iteration of While**
Seeds	$[v_1, v'_1]$
Pending	$[v_3, \varnothing], 41.7, f_{sub}(S_3, \varnothing)$ $[v_4, v'_6], 22.5, f_{sub}(S_4, S'_6)$ $[v_7, \varnothing], 41.7, f_{sub}(S_7, \varnothing)$ $[\varnothing, v'_3], 41.7, f_{sub}(\varnothing, S'_3)$ $[\varnothing, v'_4], 37.5, f_{sub}(\varnothing, S'_4)$
Computed	$[v_1, v'_1]$ $[v_2, v'_2]$ $[v_3, \varnothing]$ $[v_4, v'_6]$ $[v_7, \varnothing]$ $[\varnothing, v'_3]$ $[\varnothing, v'_4]$
Matching	$[v_1, v'_1]$ $[v_2, v'_2]$

from *Pending* and inserted in *Pending* and *Matching*. Moreover, $f_{sub}(\varnothing, S'_3)$ and $f_{sub}(\varnothing, S'_4)$ are computed, and pairs $[\varnothing, v'_2]$ and $[\varnothing, v'_2]$ are added into *Pending*. Table 3.4 presents the sets after the second iteration.

In the third iteration, the mapping $[v_4, v'_6]$ has the minimum *Star* distance in *Pending* (Table 3.4) and therefore, this mapping is selected and added into *Matching* (Table 3.5).

In the fourth iteration, the mapping $[\varnothing, v'_4]$ has the minimum *Star* distance in *Pending* (Table3.5). In this case, all the mappings generated by *Match_Star* have been previously considered. Accordingly, the only difference between the new situation (Table 3.6) and the previous one (Table 3.5) is the deletion of the element $[\varnothing, v'_4]$ in *Pending* and its inclusion in *Matching* (Table 3.6).

In the same way as in the fourth iteration, in the fifth, sixth and seventh iterations the selected element in *Pending* does not generate new mappings,

Table 3.5 Debug of Algorithm 4 in 3^{rd} iteration

Sets	**Data in the third iteration of While**
Seeds	$[v_1, v'_1]$
Pending	$[v_3, \varnothing], 41.7, f_{sub}(S_3, \varnothing)$ $[v_7, \varnothing], 41.7, f_{sub}(S_7, \varnothing)$ $[\varnothing, v'_3], 41.7, f_{sub}(\varnothing, S'_3)$ $[\varnothing, v'_4], 37.5, f_{sub}(\varnothing, S'_4)$ $[v_6, \varnothing], 37.5, f_{sub}(S_6, \varnothing)$
Computed	$[v_1, v'_1]$ $[v_2, v'_2]$ $[v_3, \varnothing]$ $[v_4, v'_6]$ $[v_7, \varnothing]$ $[\varnothing, v'_3]$ $[\varnothing, v'_4]$ $[v_6, \varnothing]$
Matching	$[v_1, v'_1]$ $[v_2, v'_2]$ $[v_4, v'_6]$

Table 3.6 Debug of Algorithm 4 in 4^{th} iteration

Sets	**Data in the fourth iteration of While**
Seeds	$[v_1, v'_1]$
Pending	$[v_3, \varnothing], 41.7, f_{sub}(S_3, \varnothing)$ $[v_7, \varnothing], 41.7, f_{sub}(S_7, \varnothing)$ $[\varnothing, v'_3], 41.7, f_{sub}(\varnothing, S'_3)$ $[v_6, \varnothing], 37.5, f_{sub}(S_6, \varnothing)$
Computed	$[v_1, v'_1]$ $[v_2, v'_2]$ $[v_3, \varnothing]$ $[v_4, v'_6]$ $[v_7, \varnothing]$ $[\varnothing, v'_3]$ $[\varnothing, v'_4]$ $[v_6, \varnothing]$
Matching	$[v_1, v'_1]$ $[v_2, v'_2]$ $[v_4, v'_6]$ $[\varnothing, v'_4]$

Table 3.7 Debug of Algorithm 4 in 5^{th} iteration

Sets	Data in the fifth iteration of While
Seeds	$[v_1, v'_1]$
Pending	$[v_3, \varnothing], 41.7, f_{sub}(S_3, \varnothing)$ $[v_7, \varnothing], 41.7, f_{sub}(S_7, \varnothing)$ $[\varnothing, v'_3], 41.7, f_{sub}(\varnothing, S'_3)$
Computed	$[v_1, v'_1]$ $[v_2, v'_2]$ $[v_3, \varnothing]$ $[v_4, v'_6]$ $[v_7, \varnothing]$ $[\varnothing, v'_3]$ $[\varnothing, v'_4]$ $[v_6, \varnothing]$
Matching	$[v_1, v'_1]$ $[v_2, v'_2]$ $[v_4, v'_6]$ $[\varnothing, v'_4]$ $[v_6, \varnothing]$

since all of them have been previously computed and, consequently, the number of elements in *Pending* decreases. Table 3.7, Table 3.8 and Table 3.9 show the sets after these iterations.

At this point of the algorithm's execution, the set *Pending* is empty (Table 3.10) and hence the execution of the algorithm jumps to Line 27. In this line, nodes v_5 and v'_5 are included in *Matching* as a deletion $[v_5, \varnothing]$ and also as an insertion $[\varnothing, v'_5]$. Finally, the algorithm finishes returning the correspondence in *Matching*. Figure 3.10 shows the final correspondence. In this example, the classical sub-optimal algorithms [20], [36], [44], [37], [10] would compute 6 x 6 = 36 times *Match_star* but the Belief algorithm has only computed

Table 3.8 Debug of Algorithm 4 in 6^{th} iteration

Sets	**Data in the sixth iteration of While**
Seeds	$[v_1, v'_1]$
Pending	$[v_3, \varnothing], 41.7, f_{sub}(S_3, \varnothing)$ $[v_7, \varnothing], 41.7, f_{sub}(S_7, \varnothing)$
Computed	$[v_1, v'_1]$ $[v_2, v'_2]$ $[v_3, \varnothing]$ $[v_4, v'_6]$ $[v_7, \varnothing]$ $[\varnothing, v'_3]$ $[\varnothing, v'_4]$ $[v_6, \varnothing]$
Matching	$[v_1, v'_1]$ $[v_2, v'_2]$ $[v_4, v'_6]$ $[\varnothing, v'_4]$ $[v_6, \varnothing]$ $[\varnothing, v'_3]$

Table 3.9 Debug of Algorithm 4 in 7^{th} iteration

Sets	**Data in the seventh iteration of While**
Seeds	$[v_1, v'_1]$
Pending	$[v_3, \varnothing], 41.7, f_{sub}(S_3, \varnothing)$
Computed	$[v_1, v'_1]$ $[v_2, v'_2]$ $[v_3, \varnothing]$ $[v_4, v'_6]$ $[v_7, \varnothing]$ $[\varnothing, v'_3]$ $[\varnothing, v'_4]$ $[v_6, \varnothing]$
Matching	$[v_1, v'_1]$ $[v_2, v'_2]$ $[v_4, v'_6]$ $[\varnothing, v'_4]$ $[v_6, \varnothing]$ $[\varnothing, v'_3]$ $[v_7, \varnothing]$

Table 3.10 Debug of Algorithm 4 in 8^{th} iteration

Sets	**Data in the eighth iteration of While**
Seeds	$[v_1, v'_1]$
Pending	$\varnothing$
Computing	$[v_1, v'_1]$ $[v_2, v'_2]$ $[v_3, \varnothing]$ $[v_4, v'_6]$ $[v_7, \varnothing]$ $[\varnothing, v'_3]$ $[\varnothing, v'_4]$ $[v_6, \varnothing]$
Matching	$[v_1, v'_1]$ $[v_2, v'_2]$ $[v_4, v'_6]$ $[\varnothing, v'_4]$ $[v_6, \varnothing]$ $[\varnothing, v'_3]$ $[v_7, \varnothing]$ $[v_3, \varnothing]$

it three times. Specifically, to compute the pairs $[v_1, v'_1]$, $[v_2, v'_2]$ and pairs $[v_4, v'_5]$.

3.3 Experimental validation

In the first part of this section, the Belief algorithm is validated and analised using synthetic graphs, whereas in the second part a real application is shown hereof. We have used small graphs to compare the Belief algorithm against other non-linear algorithms, and also we have used a larger one to show its runtime.

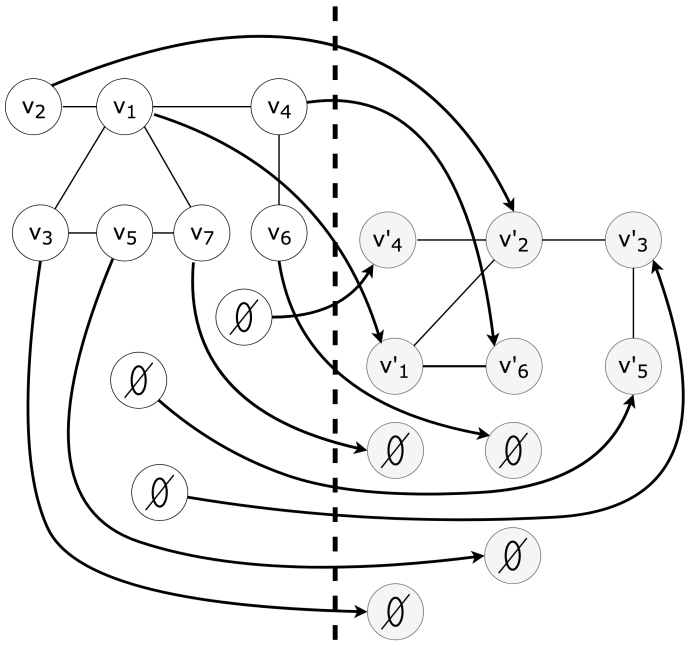


Fig. 3.10 Final correspondence between G and G'.

3.3.1 Validation using synthetic graphs

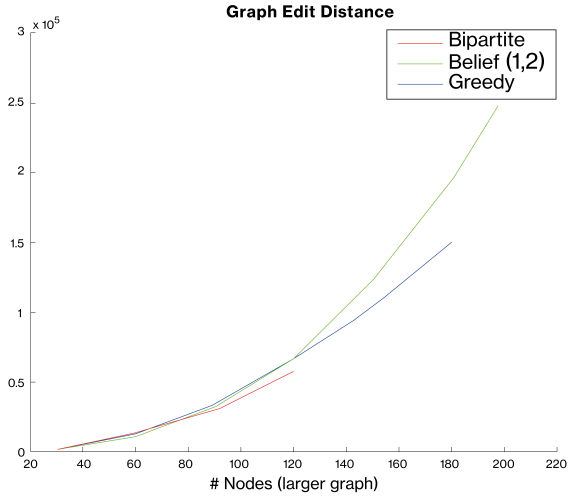
The aim of this section is to validate the Belief algorithm from the matching, quality and runtime points of view. As can be deduced from Equation 1.2, the lower the edit cost, the better is considered the matching. Thus, we have compared our method with two graph matching algorithms: the Bipartite graph matching [36], [44] which has a cubic cost, and the Greedy graph matching [37], [10] which has a quadratic cost. As a reminder, our method has a linear computational cost, but it needs an initial mapping, which we have called *Seed*. We have not computed the optimal matching through an $A*$ algorithm [13] due to runtime reasons. Therefore, we do not know which is the optimal distance given a pair of graphs. Algorithms are implemented in Matlab and they have been executed in Windows 10 i7. The software is publicly available at [47].

The experiments have been set up as follows: First, we have randomly generated an attributed graph with only one attribute on the nodes (a value between 0 and 99), and a degree of approximately $0.2 \cdot n$, being n the order of graphs. Starting from this graph, we have generated another graph by copying it and then deleting and inserting the 10% of the nodes and edges, to later modify the attribute value of another 10% of the nodes. We assume that the optimal matching is the identity (the process of generating the synthetic graphs used in this experiments is described in depth in section 2.1). Accordingly, the Belief algorithm imposing the $Seed = [1,1]$ and $Seed = [2,2]$ is computed. Clearly, it may be another matching with lower cost due to the noise added to the second graph, so the mappings $[1,1]$ and $[2,2]$ may not be the best option. This fact can be considered as part of the noise that our algorithm has to deal with. The cost of deleting and inserting nodes and edges has been set at 25 (it is a $\frac{1}{4}$ of the maximum value [49]).

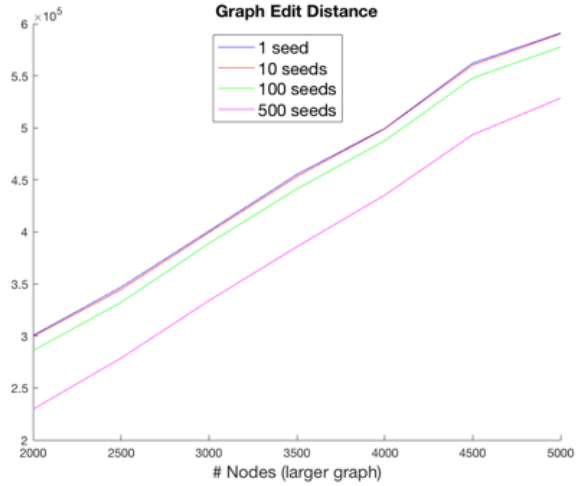
Figure 3.11 displays the average Graph Edit Distance of 100 runs. When graphs are small (first plot), the three algorithms deduce similar distances. Nevertheless, the tendency is to increase the gap between these algorithms when the order of the graphs increases. This behaviour was also reported in [50] but considering only Bipartite graph matching and Greedy algorithms.

In Figure 3.11a, the difference between one and two *Seeds* is too small to be appreciated. In Figure 3.11b, the Graph Edit Distance computed by our algorithm given different number of *Seeds* is shown. As supposed, when the number of *Seeds* increases, the deduced correspondences are better and consequently the Graph Edit Distance is lower.

Figure 3.12 shows the runtime of the three algorithms. The Belief algorithm is the fastest followed by the Greedy, and finally the Bipartite. The Belief algorithm with two *Seeds* is slightly faster than the Belief algorithm with one *Seed*, nevertheless, the gap is too small to be appreciated in the plot. This is because the correspondences generated by *Match_Star* tend to be more optimal with more *Seeds*, since there are more options in the *Pending*



(a) Comparison of the Graph Edit Distance between Bipartite, Greedy and Belief algorithms.



(b) Graph Edit Distance of the Belief algorithm.

Fig. 3.11 Graph Edit Distance obtained by the Bipartite, Greedy and Belief algorithms with one and two *Seeds*.

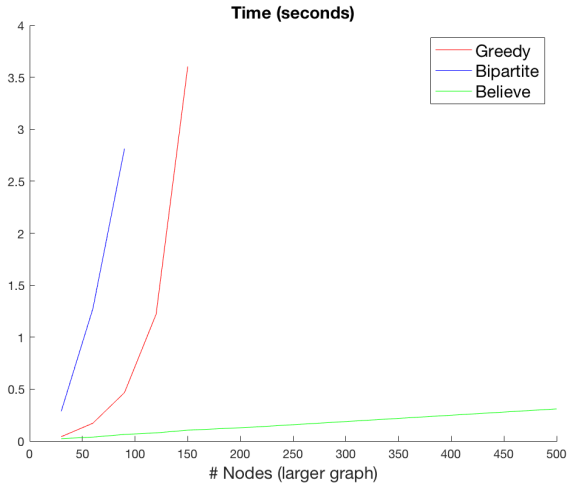


Fig. 3.12 Runtime in seconds spent by the Bipartite, Greedy and Belief algorithms with one and two *Seeds*.

set. The cost of the Believe algorithm is linear, whereas the Bipartite and Greedy algorithms have cubic and quadratic costs, respectively.

Figure 3.13 presents the runtime of the experiment shown in Figure 3.4. The runtime is almost linear with respect to the number of nodes, although it depends on the number of output edges per node, d. Note that mapping a graph with 5000 nodes and 9 neighbours per node takes a runtime of only 9 seconds (Matlab, MacBookPro I5). Moreover, we analysed a different number of *Seeds* and we deduced that the runtime do not depends on the number of *Seeds*.

3.3.2 Real applications

We were asked to determine the variability of friendships in a social network. That is, given a specific individual, how many friends has added or deleted. To do so, the social network is sampled time to time. As commented in the introduction, attributed graphs are a natural way for representing social

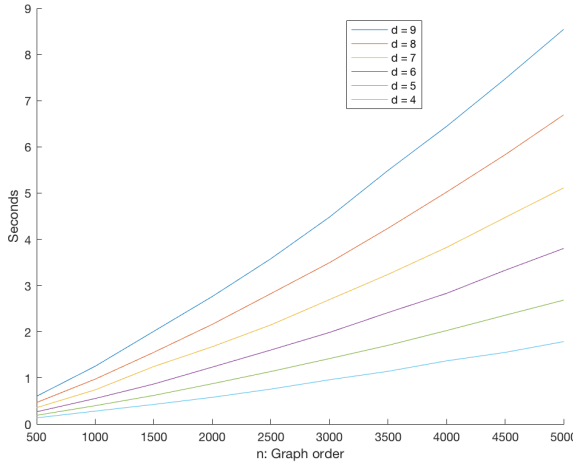
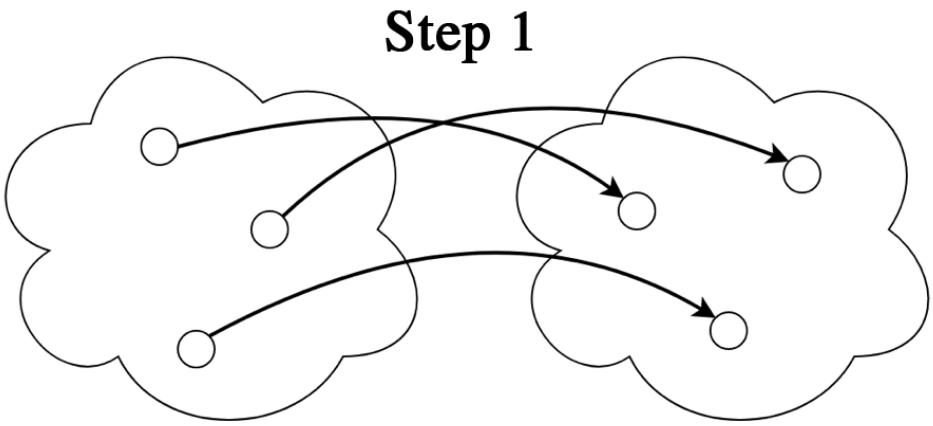


Fig. 3.13 Runtime in seconds of matching two graphs given several graph orders and numbers of output edges per node, d.

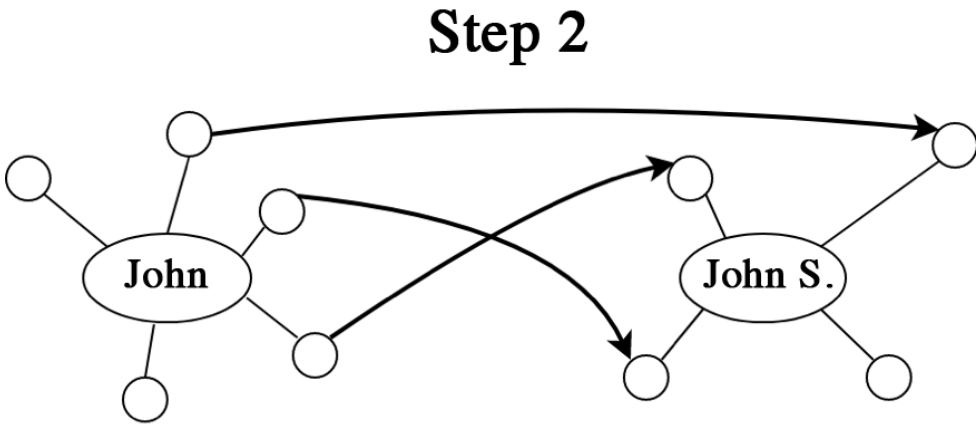
networks, where nodes are the users and edges are the friendships between users. Given two temporal samples of the graph, the friendship variability of someone is modelled as the number of new edges that are inserted and connected to its node, plus the ones deleted that were connected to its node.

Figure 3.9 graphically displays the two steps of the process needed to compute the variability. In the first step, the mapping between nodes of the two graphs that represent the social network in two different temporal samples is computed. In the second step, the number of inserted and removed edges per node is counted. Note that the mapping between neighbours is part of the mapping deduced in the first step. In the example of Figure 3.9, the variability of John is three, since two neighbours from the first net are not mapped and one neighbour of the second net is also not mapped.

Due to the anonymisation, it is not possible to directly deduce the node-to-node mapping between different temporal samples of the social networks. First, it is necessary to compute the correspondence between consecutive samples of the social network through the Belief algorithm, and then it is



(a) Finding the mapping.



(b) Step 2.

Fig. 3.14 Counting different neighbours.

necessary to count the number of neighbour nodes that diverge in both samples for each node. Moreover, we received only some mappings between nodes of consecutive samples used as *Seeds*. The method to deduce these node-to-node mappings is unknown.

The original social network is not publicly available. To guarantee the reproducibility of the experiments, we used the database Tarragona-Facebook generated by ourselves. It is available in [47]. To do so, we used the database ego-Facebook [27] modifying certain samples in order to generate the *Seeds*. In this database, nodes represent people and edges are friendships. Facebook data has been anonymised by replacing the Facebook internal identifications of each user with a new value. Also, the interpretation of the features has been obscured. For instance, where the original dataset may have contained a feature "political = Democratic Party", the new data would simply contain "political = anonymised feature 1". Thus, using the anonymised data, it is possible to determine whether two users have the same political affiliations, but not what their individual political affiliations represent.

Figure 3.15 shows the normalised histogram of the friendship variation on the database Tarragona-Facebook. As a conclusion almost the 60% of the population kept exactly the same friends between the two samples; 15% of the population have a variation of one friend and 7% of the population have a variation of two friends.

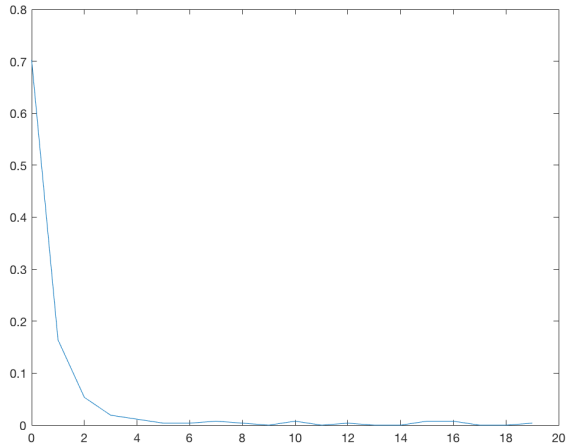


Fig. 3.15 Normalised variation of the Tarragona-Facebook social network friendships.

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

Chapter 4

A general framework to learn the edit cost based on an embedded model

4.1 Introduction

In this chapter, a general framework is presented to learn the *StarEditDistance* (section 1.2.2). Although this framework could be concretised into different methods, only two different examples are presented in section 4.2.1.

After learning the *StarEditDistance*, several graph-matching algorithms could be adapted to use these edit functions in the classification process. In the experimental evaluation, the *GED* is computed through the Bipartite algorithm [36]. In this case, adapting the algorithm means how C^S, C^D and C^I are defined in it. In the original definition of the algorithm [36], these costs were computed considering that *Stars* are graphs with a concrete structure.

4.2 The method

4.2.1 Learning Error Tolerant Graph Matching

For simplicity, the local structure is defined as a *Star* (section 4.2.1.2). The *Stars* are composed of a central node, the adjacent edges and also the nodes connected to these edges (figure 1.2c).

4.2.1.1 General Problem setting

Figure 4.1 shows the basic scheme of the learning method presented here. The main aim is to learn the substitution, insertion and deletion costs of *Stars* C^S, C^D and C^I through a supervised learning method. Initially, the learning database needs to have a specific format. Usually, the database's registers are composed of a pattern and their class. Contrarily, the database's registers needed here are composed of a pair of graphs (G^p, G'^p) and their ground-truth correspondences $\hat{f}^p$, (p represents the register number). These ground-truth correspondences $\hat{f}^p$ have been deduced by an external system (human or artificial) and are considered to be the best mappings for learning purposes. Also, the ground-truth correspondences are independent of the definition of the edit costs. The aim of the learning method is to define these edit costs as functions, so the deduced correspondences become close to the ground-truth correspondences $\hat{f}^p$ for all pairs of graphs (G^p, G'^p).

Fingerprint matching could be a good example of the generation of these ground-truth correspondences. Given two fingerprints, a specialist decides which is the best mapping between minutiae of these fingerprints. Thus, the specialist knows nothing about the GED nor the edit costs. Therefore, the correspondence decided by the specialist is not influenced by these parameters.

In the first step of the scheme in section 4.2.1.2, six different sets of *Stars* are defined, given the database registers composed of two graphs and their ground-truth correspondence in the learning database: the *Stars* that are mapped by the ground-truth correspondences and also the ones that are

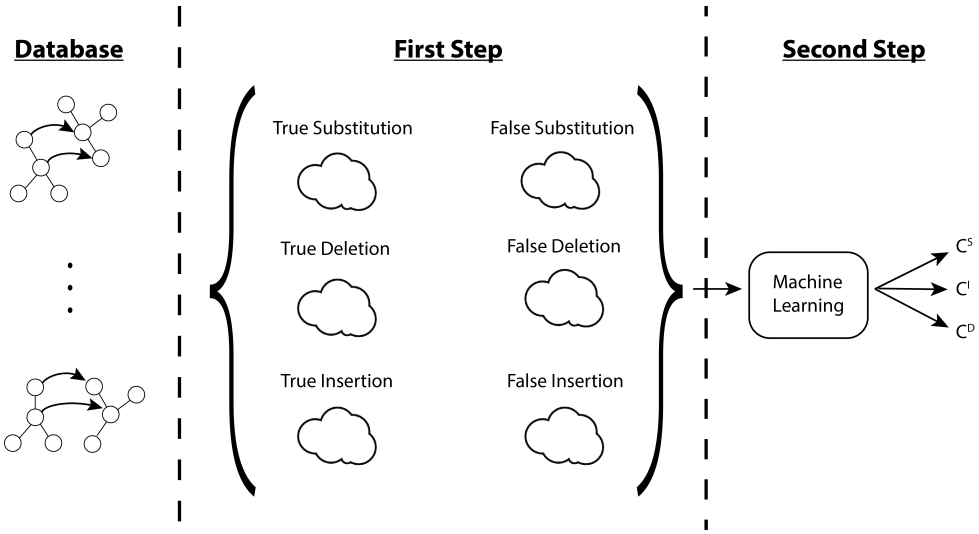


Fig. 4.1 General machine learning scheme composed of two steps.

not mapped; the *Stars* that the ground-truth correspondences impose to be deleted and also the ones that do not have to be deleted; and finally, the *Stars* that the ground-truth correspondences impose to be inserted and also the ones that do not have to be inserted. Each *Star* can appear in several sets.

In the second step of the scheme presented here (section 4.2.1.2), the machine learning model deduces the C^s, C^D and C^I functions given the six sets of *Stars*. It is based on embedding the *Stars* into vectors and then applying classical machine learning algorithms on these vectors.

Figure 4.2 displays the general graph matching scheme. It is a classical scheme where the graph matching module incorporates the information of the edit costs learned in the learning process C^s, C^D and C^I. The input of the module is composed of a pair of attributed graphs G and G' and the output is the *GED*. Given the previously commented aim of the learning method, the graph matching algorithm has to deduce node-to-node correspondences that would be close to the ground truth node-to-node correspondences.

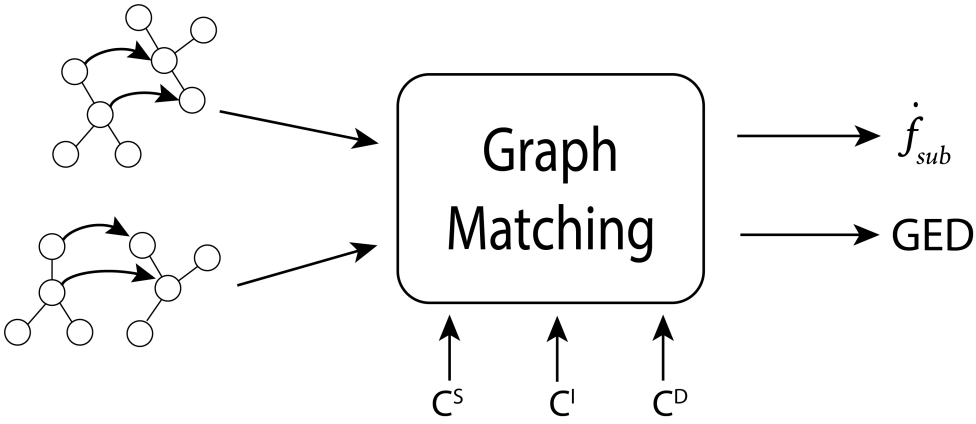


Fig. 4.2 General graph matching scheme.

4.2.1.2 Definition of the stars' sets

The model is based on the following statement: if the ground truth correspondence $\hat{f}^p$ imposes that two nodes have to be substituted, then it may hold that the substitution cost of the involved *Stars* might be lower than the substitution costs of the combinations of the other *Stars*. Moreover, if the ground truth correspondence $\hat{f}^p$ imposes that a node has to be deleted, then it may hold that the deletion cost of the involved *Star* might be lower than the deletion costs of the *Stars* that the ground truth correspondence imposes they have to be substituted. Similarly, this also occurs with the node insertions.

Figure 4.3 presents an example of a ground truth correspondence $\hat{f}^p$. Nodes v_1^p and v_2^p are substituted by $v_1'^p$ and $v_2'^p$, nodes v_3^p and $v_4'^p$ are deleted, and finally, nodes $v_3'^p$ and $v_4'^p$ are inserted. It may happen that $C^s(S_1^p, S_1'^p)$ would have to be lower than $C^s(S_1^p, S_2'^p)$ and $C^s(S_2^p, S_1'^p)$. This also happens with $C^s(S_2^p, S_2'^p)$. Moreover, it may happen that $C^D(S_3^p)$ would have to be lower than $C^D(S_1^p)$ and $C^D(S_2^p)$. Similarly, this also occurs with $C^D(S_4^p)$. Finally, it also may happen that $C^I(S_3'^p)$ would have to be lower than $C^I(S_1'^p)$ and $C^I(S_2'^p)$. This also happens with $C^I(S_2'^p)$.

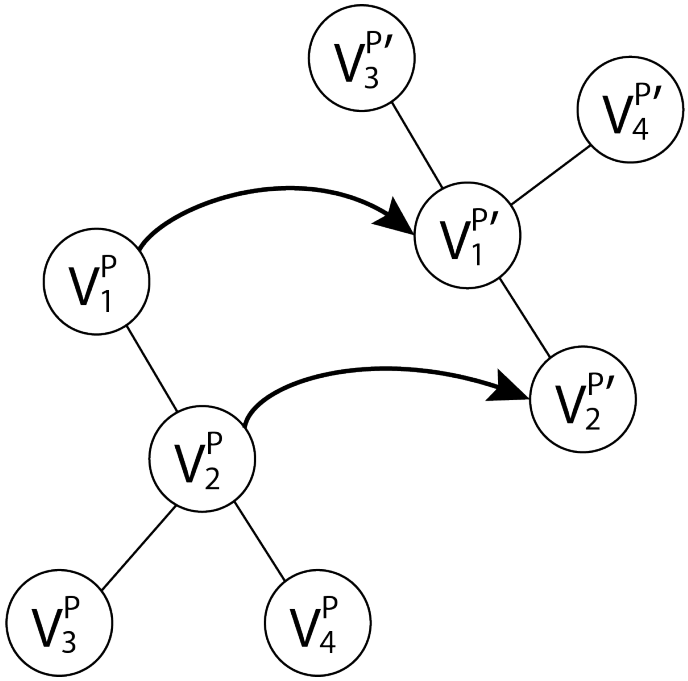


Fig. 4.3 Ground-truth correspondence $\hat{f}^p$ from G^p to G'^p.

To fix these initial ideas into a learning model, two classes of mappings in the substitution cases, two classes of mappings in the deletion cases, and two classes of mappings in the insertion cases are defined. If a ground-truth correspondence $\hat{f}^p$ defines a substitution, $\hat{f}^p(v_a^p) = v_i'^p$, then the pair of *Stars* $\left\{S_a^p, S_i'^p\right\}$ belongs to the class $True\_Substitution$.

Contrarily, all combinations of pairs $\left\{S_a^p, S_j'^p\right\}$ that $j \neq i$ and also all combination of pairs $\left\{S_b^p, S_i'^p\right\}$ that $b \neq a$ between $non-null$ nodes belong to the class $False\_Substitution$. Furthermore, if the ground-truth correspondence $\hat{f}^p$ imposes that the node v_a^p has to be deleted, then the *Star* S_a^p belongs to class $True\_Deletion$. Oppositely, all *Stars* S_b^p whose central nodes v_b^p are substituted, belong to the class $False\_Deletion$. Something similar occurs with the insertion operations. If the ground-truth correspondence $\hat{f}^p$ imposes that node $v_i'^p$ has to be inserted, then the *Star* $S_i'^p$ are considered that they belong to the class $True\_Insertion$. Contrarily, all *Stars* $S_j'^p$ whose central nodes $v_j'^p$ are substituted belong to the class $False\_Insertion$. It should be noted that some *Stars* or pairs of *Stars* can be included in different sets.

Figure 4.4 shows the classes of pairs of *Stars* previously defined, given the substitutions, deletions and insertions of the example in Figure 4.3.

4.2.1.3 Embedding stars into vectors

This section presents a model to learn the costs C^s, C^D and C^I based on a classical machine-learning method. To do so, these costs have to be modelled as functions, where the domain is a point in a vector space and the codomain is a Real number. Therefore, an embedding of the *Stars* to points in a suitable vector space is needed. This embedding has to encode the *Stars* in vectors of equal size and produce one vector per *Star*. Mathematically, for a given *Star* S, the *Star* embedding is a function Φ, which maps S_a to a point E_a in a T dimension space $\mathbb{R}^T$. It is given as $\Phi(S_a) = E_a$. The value T is concretised above.

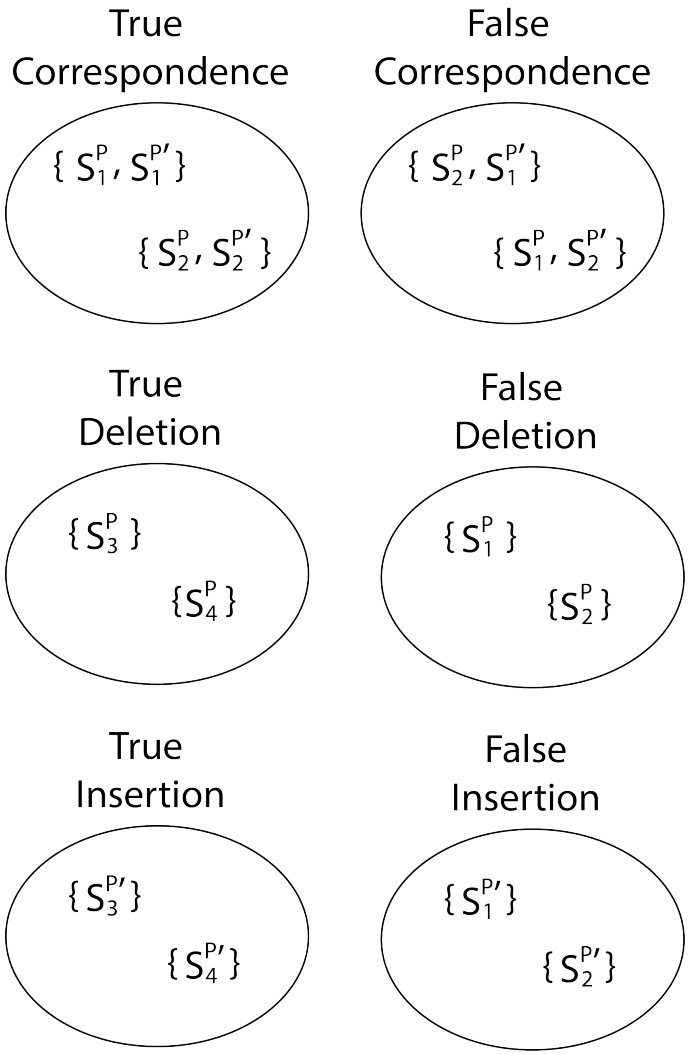


Fig. 4.4 Classes and mappings given the example of Figure 4.3

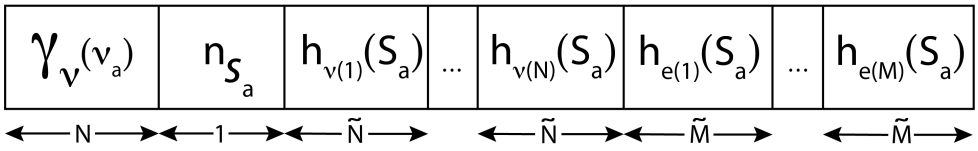


Fig. 4.5 The E_a embedding of *Star* S_a.

Figure 4.5 graphically provides the embedding of the *Star* S_a . The first N elements are the attributes of the nodes and the next one is the number of nodes of the *Star*, n_{S_a}. The next cells are filled with the histograms generated by the attributes of the external nodes and the attributes of the external edges. Histograms $h_{v(i)}$ and $h_{e(i)}$ represent the histograms generated by the i^{th} attribute of the nodes and edges, respectively. N and M are the number of attributes on the nodes and edges, respectively. Finally, $\tilde{N}$ and $\tilde{M}$ are the number of bins of the node and edge histograms, respectively. This representation has been inspired by the one presented in [28]. In that case, the model embedded a whole graph into a vector. Since we want to embed a *Star*, which is a concrete structure of a graph, we have somehow concretised their embedding model. Thus, $T = N + 1 + \tilde{N} * N + \tilde{M} * M$.

Then, given the six sets defined in the previous section, the present method describes three matrices as shown in Figure 4.6.

- The *Substitution Matrix* has three main columns. The first one has the vectors E_a calculated by the embedding function, given the first *Stars* S_a of the pairs of *Stars* in the sets *True Substitution* or *False Substitution*. Similarly, the second column has the vectors E_i' calculated by the embedding function, given the second *Stars* S_i' of the pairs of *Stars* in the same sets. Finally, there is a value equal to 1 in the third column if the pair of *Stars* belongs to the *True Substitution* set, and a 0 value if it belongs to the *False Substitution* set.

- The *Deletion Matrix* has two main columns: The first one has the vectors E_a calculated by the embedding function, given the the *Stars* S_a in the sets *True Substitution* or *False Substitution*. There is a value equal to 1 in the

second column if S_a belongs to the $TrueDeletion$ set and there is a 0 value if it belongs to the $FalseDeletion$ set.

- In the same way, this occurs with the *Insertion Matrix*, but considering the *Stars* S'_i in the sets $True\ Insertion$ or $FalseInsertion$.

Deletion Matrix, *Insertion Matrix* and *Substitution Matrix* are used to learn the edit functions C^D, C^I and C^S, respectively. The first column in the *Deletion Matrix* and *Insertion Matrix* and the first two columns in the *Substitution Matrix* are the input data of the machine learning method, whereas the last column is the class to be learned. Is worth to mention that these functions are learned independently.

4.2.2 Neural Network

The C^S is modeled by a regression function learned through an artificial neural network, NN^S. In the learning phase, the machine learning is fed by the *Substitution Matrix* (the last column is the ideal output of the substitution function). When the neural network NN^S has learned the regression function, the substitution cost $C^S(S_a, S'_i)$ is computed as the output of this neural network, NN^S, as follows:

$$C^S(S_a, S'_i) = Output(NN^S, [E_a, E'_i]) \tag{4.1}$$

The C^D is modeled by another regression function based on an artificial neural network, NN^D, in a similar way than C^S. Nevertheless, in this case, the machine learning is fed by the *Deletion Matrix*:

$$C^D(S_a) = Output(NN^D, [E_a]) \tag{4.2}$$

The same applies with the insertion cost C^I but using the information of *Insertion Matrix*:

$$C^I(S'_i) = Output(NN^I, [E'_i]) \tag{4.3}$$

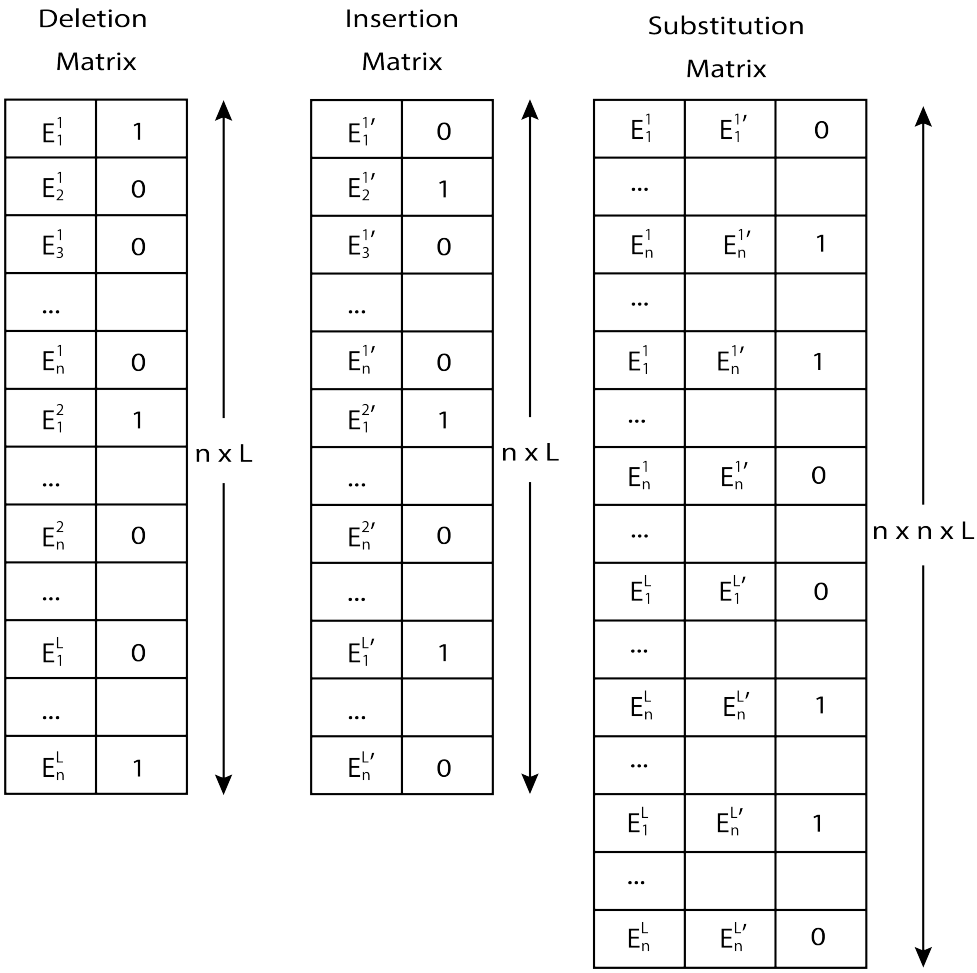


Fig. 4.6 The E_a embedding of *Star* S_a.

4.2.3 Probabilistic Density Distribution

In this method C^S is defined by two probability density functions based on a mixture of Gaussians, Pdf_True^S and Pdf_False^S. The first density function is modelled by columns that have the information of E^a and E'_i in the *Substitution Matrix*, but only using the rows that have a value of 1 in the last column. The second density function is modelled likewise but only using the rows that have a 0 value in the last column.

Thus, the substitution cost $C^S(S_a, S'_i)$ is defined as the subtraction of the probabilities obtained from these probability density functions (Equation 4.4). The constant 1 is needed to assure that the cost is always positive. The cost has to be low if the probability obtained from the set *True_Substitution* is high or the probability obtained from the set *False_Substitution* is low.

$$C^S(S_a, S'_i) = 1 - Prob(PdfTrue^S, [E_a, E'_i]) + Prob(PdfFalse^S, [E_a, E'_i]) \tag{4.4}$$

Functions C^D and C^I are modelled in a similar way. Nevertheless, matrices *Deletion_Matrix* and *Insertion_Matrix* are used. Thus:

$$C^D(S_a) = 1 - Prob(Pdf\_True^D, [E_a]) + Prob(Pdf\_False^D, [E_a]) \tag{4.5}$$

$$C^I(S'_i) = 1 - Prob(Pdf\_True^I, [E'_i]) + Prob(Pdf\_False^I, [E'_i]) \tag{4.6}$$

4.3 Experimental Results

The presented method has been validated using fourteen databases available in [47]. All of them were used to test other learning methods and were previously

published. For example, some belong to the public graph repository [29]. The main characteristic of these databases is that their registers are not only composed of a graph and its class, but they are composed of a pair of graphs and a ground-truth correspondence between them, as well as their class. This register structure is useful to analyse and develop graph matching algorithms and to learn their parameters in a broad manner. Table 4.1 shows the main features of the fourteen databases.

We have measured the quality of the learning algorithms through the accuracy of the returned correspondence. We have experimentally validated our learning method with a methodology composed of two steps. In the first one, we have learned the edit costs considering our method and other published ones. In the second step, we have applied the general scheme presented in Figure 4.1, where the graph matching was the Bipartite graph matching [45], and we have deduced the accuracies of the returned correspondences. The accuracy is defined as the inverse of the normalised hamming distance between the returned correspondence and the ground-truth correspondences $\hat{f}^p$ for all pairs of graphs $(G^p, G^{p\prime})$. The returned accuracy of a learning method is the mean of all the accuracies computed in a database.

Our method has been tested through four different specifications: Neural network (NN), Probability density function (PDF), Neural network without histograms (NNnoHis) and Probability density function without histograms (PDFnoHis). In the last two options, the embedded domain does not have the histograms. That is, the vector shown in Figure 4.5 is only composed of the first $N+1$ bins. Table 4.4 displays the technical specifications that have achieved the best results. Taking into account the Neural network experiments, the symbol $-$ means that this experiment was no tested. The whole Neural networks have three layers. The three numbers in a cell represent the number of neurons of the input layer, the hidden layer and the output layer. Considering the Probability density distributions, we only could synthesize them in the case of $PDFnoHis$ in the second round of experiments.

Table 4.1 Main features of the fourteen databases. The first row shows in which experiment the databases have been used.

		First Experiment						
		House 90	***Hotel 90***	***House 1***	***Hotel 1***	***Noise***	***Rotate***	***Shear***
Graph	**Learn**	8	8	37	68	68	68	68
	Test	6	6	36	66	66	66	66
	Val.	8	8	74	66	66	66	66
Corr.	**Learn**	4	4	37	34	34	34	34
	Test	3	3	36	33	33	33	33
	Val.	4	4	37	33	33	33	33
Num. of classes		1	1	1	1	1	1	1
Num. of attrib		60	60	60	60	60	60	60
Description					*SIFT*			
Avg. num. nodes		30	30	30	30	35	35	35
Avg. num. edges		156	154	156	152.7	180.6	183.8	185.7
Max. nodes		30	30	30	30	35	35	35
Max. edges		158	156	158	158	184	184	186
Max. D./I.		0	0	0	0	0	0	0
Avg. null D./I.		0	0	0	0	0	0	0

Table 4.2 Main features of the fourteen databases. The first row shows in which experiment the databases have been used.

		Second Experiment		
		Letter High	***Letter Med***	***Letter Low***
Graph	**Learn**	750	750	750
	Test	750	750	750
	Val.	750	750	750
Corr.	**Learn**	37500	37500	37500
	Test	37500	37500	37500
	Val.	37500	37500	37500
Num. of classes		15	15	15
Num. of attrib		2	2	2
Description			*(x,y)*	
Avg. num. nodes		4.6	4.6	4.6
Avg. num. edges		6.2	6.4	9
Max. nodes		8	9	9
Max. edges		12	14	18
Max. D/I.		4	5	5
Avg. null D./I.		0.4	0.4	0.4

Table 4.3 Main features of the fourteen databases. The first row shows in which experiment the databases have been used.

		Third Experiment			
		Boat	***East Park***	***East South***	***Resid***
Graph	**Learn**	50	50	50	50
	Test	50	50	50	50
	Val.	-	-	-	-
Corr.	**Learn**	25	25	25	25
	Test	25	25	25	25
	Val.	-	-	-	-
Num. of classes		1	1	1	1
Num. of attrib		64	64	64	64
Description		*SURF*			
Avg. num. nodes		50	50	50	50
Avg. num. edges		278.4	276	278.8	276.4
Max. nodes		50	50	50	50
Max. edges		282	280	282	278
Max. D/I.		50	47	50	50
Avg. null D./I.		32	34	37	32

Table 4.4 Configuration of the Neural networks and Probability density functions in the four specifications of our method: NN, NNnoHis, PDF and PDFnoHis.

Number of neurons per layer				
Algorithm		***First Experiment***	***Second Experiment***	***Third Experiment***
NN	NN^s	1321-500-2	45-45-2	1409-500-2
	NN^d	-	23-23-2	705-352-2
	NN^i	-	23-23-2	705-352-2
NNnoHis	NN^s	122-122-2	5-5-2	130-130-2
	NN^d	-	3-3-2	65-65-2
	NN^i	-	3-3-2	65-65-2

Table 4.5 shows the accuracy of the databases in Tables 4.1, 4.2 and 4.3, given the four particularisations of our method and the methods in [50, 6, 25, 9, 3, 8, 32, 31]. The values in these methods have been extracted from the experimental sections of those papers, for this reason, some cells are not filled. We have also added the results presented in [29]. In that paper, the authors present some accuracy results where the edit costs have been manually imposed as constants, given two different configurations of the Bipartite graph matching algorithm: one configuration where the edit costs are computed through the *Stars*, and another where the edit costs are computed through the Degree (a *Star* without the external nodes).

Note that our method implemented as a Probability density function without the histograms in the embedded vector is equivalent to the method presented in [31], except for some implementation details not explained in the original publication. This also applies to our method implemented as a Neural network without the histograms in the embedded vector and the method presented in [32].

We have performed three rounds of experiments. The difference between them is the databases type of ground-truth correspondences.

- In the first round of experiments, we have used the first seven databases of Table 4.1. In these databases, the ground-truth correspondences do not have deletion nor insertion operations. Then, it is not possible for the learning methods to learn these edit functions. Thus, they are useful to analyse how the learning algorithms deduce the substitution cost C^S without influence of the insertion and deletion costs, C^D and C^I. This first round of experiments is useful to fairly compare our framework to the methods [6, 25, 9, 32, 31], which only learn the substitution costs.

When the substitution costs have been learned, the pattern recognition model applies the graph matching algorithm with the learned substitution costs C^S and the insertion and deletion costs $C^D = Inf$ and $C^I = Inf$. These values are set to force the graph matching algorithm to return the correspondences

without the insertion and deletion operations, in the same way as the ground-truth correspondences.

Note that our method with the Neural network (NN) and the one in [32] (which is the same as our method but without the histograms) obtains the maximum accuracy (all the node-to-node mappings are properly assigned). Moreover, the algorithm that computes the Probability density method is not able to deduce the multimodal Gaussian function and returns "ill conditioned". We believe that it is not possible to deduce the Gaussian functions because the graph nodes have a high number of attributes (thus the multimodal Gaussian has 1321 and 122 dimensions in the first experiment, and 1409 and 130 in the third experiment) but the number of correspondences per class is low (see Table 4.4).

- In the second round of experiments, we analyse the complete method (learning the substitution, deletion and insertion functions), but using graphs with less attributes to avoid the "ill conditioned" returned in the Probability density function. In this case, we have used the Letter databases presented in Table 4.2, where their graphs have only two attributes in the nodes and their correspondences have insertions and deletions. Our method has been compared to the methods that learn the substitution, deletion and insertion costs, which are [3, 8, 32, 31, 29].

Similar to the first round of experiments, the Neural network with histograms achieves the highest accuracy. Moreover, our method with the Probability density functions is not able to deduce the Gaussian function. Nevertheless, our method without the information of the histograms (which is the same than [31]) is able to synthesize the Gaussian functions and return some results similar to other methods. In these databases, it seems that the histograms positively contribute to the learning process, since our method with Neural networks returns a higher accuracy than the one in [32]. Finally, authors of the paper [29] claim that their presented results are the best ones computed by them, given different combinations of edit costs. Our method

obtains better or similar results in the three databases, although it does not differ much.

- In the third round of experiments, we analyse the complete method (learning the substitution, deletion and insertion functions), as we did in the second round of experiments, but now the graphs have a larger number of attributes. We used the databases in the last four columns of Table 4.3, where the graphs have 64 attributes. We compared our method to the ones presented in [50, 8, 32, 31].

Our method based on Neural networks but without using the information of the histograms obtains the best results. Since the number of neighbors is very low (5 in average) and the number of attributes is 64, we suppose that the histograms become too sparse to be useful for the learning algorithm (although fuzzy methods were tested on the histograms). As it happened in the first experiments, the learning algorithm is not able to deduce the Probability density functions because the number of dimensions is too high and the number of correspondences is too low. In this experiment, the manually imposed edit costs (the last two rows in Table 4.5) achieved better results than the learning method presented in [8].

With these three rounds of experiments, we conclude that our method with Neural networks is useful to deduce the edit costs in the databases that have substitution, insertion and deletion operations, and also in the databases that only have substitution operations. Nevertheless, it is important to consider that the histogram does not always positively contribute to the learning algorithm. It seems that the histograms do not have to be included in the embedded vector when they are too sparse. Our method with the Probability density functions only could be synthesised when the histograms were not included.

Table 4.5 Accuracies deduced by the methods referenced in the first column given the fourteen databases. The cells marked with a "−" are values not given in the original papers. "i.c" means "ill conditioned" (the learning method is not able to generate the Gaussian function). The first four rows show the accuracies of our method considering four adaptations (NN and PDF are Neural network and Probability density function methods: NNnoHis and PDFnoHis are the same adaptations but without the histograms in the embedding vector).

	First Experiment						
Algorithm	***House 90***	***Hotel 90***	***House 1***	***Hotel 1***	***Noise***	***Rotate***	***Shear***
NN	1	1	1	1	1	1	1
PDF	i.c.	i.c.	i.c.	i.c.	i.c.	i.c.	i.c.
PDFnoHis[31]	i.c.	i.c.	i.c.	i.c.	i.c.	i.c.	i.c.
NNnoHis[32]	1	1	1	1	1	1	1
[6]	0.85	0.9	-	-	0.67	0.98	0.57
[25]	-	-	1	0.98	-	-	-
[9]	0.77	0.79	1	1	0.81	0.45	0.82
[3]	1	1	0.88	0.97	0.99	1	1
[29] Star	-	-	-	-	-	-	-
[8] Degree	-	-	-	-	-	-	-
[8]	-	-	-	-	-	-	-
[50] $C_{vd} = 1$ $C_{ve} = 1$	-	-	-	-	-	-	-
[50] $C_{vd} = .5$ $C_{ve} = .5$	-	-	-	-	-	-	-

Algorithm	Second Experiment		
	Letter High	***Letter Med***	***Letter Low***
NN	0.91	0.90	0.98
PDF	i.c.	i.c.	i.c.
PDFnoHis[31]	0.83	0.76	0.93
NNnoHis[32]	0.87	0.87	0.97
[6]	-	-	-
[25]	-	-	-
[9]	-	-	-
[3]	0.82	0.70	0.85
[29] Star	0.89	0.90	0.97
[8] Degree	0.87	0.85	0.97
[8]	-	-	0.71
[50] $C_{vd} = 1$ $C_{ve} = 1$	-	-	-
[50]$C_{vd} = .5$ $C_{ve} = .5$	-	-	-

	Third Experiment			
Algorithm	***Boat***	***East Park***	***South Park***	***Resid***
NN	0.14	0.15	0.18	0.17
PDF	i.c.	i.c.	i.c.	i.c.
PDFnoHis[31]	i.c.	i.c.	i.c.	i.c.
NNnoHis[32]	0.44	0.56	0.4	0.55
[6]	-	-	-	-
[25]	-	-	-	-
[9]	-	-	-	-
[3]	0.16	0.13	0.07	0.15
[29] Star	-	-	-	-
[8] Degree	-	-	-	-
[8]	0.22	0.21	0.2	0.2
[50] $C_{vd} = 1$ $C_{ve} = 1$	0.32	0.31	0.29	0.36
[50] $C_{vd} = .5$ $C_{ve} = .5$	0.34	0.34	0.29	0.45

Chapter 5

Graph Edit distance applied to define muscle mechanics

5.1 Introduction

The simulation of the human muscular system has multiple applications in biomechanics, biomedicine and motion study in general. This is because mechanical alterations of the normal functioning of muscles seem to be involved in several pathologies. Finite-element models are commonly used to simulate the muscle behaviour to deeply understand the anatomical mechanisms of specific muscles.

This chapter presents a new model to simulate the muscular human system. This model is called structural finite-element model and is composed of two main parts: the first part is based on a classical finite-element algorithm and the second part is based on a graph-matching algorithm. In this way, a numerical method and a structural pattern recognition method are put together. It is known that, in some cases, the high computational cost of the structural pattern recognition algorithms can discourage their use in real applications. For this reason, the use of the Belief algorithm is incorporated [38] because it returns a correspondence between the nodes of two graphs in linear cost, with

regard to the number of nodes. The restrictions of the algorithm (an initial small set of node-to-node correspondence) are naturally incorporated into the model presented here. This new model simulates the human muscular system. It increases the accuracy of the classical finite-state models by incorporating structural pattern recognition methodologies at the expense of an insignificant increase of runtime.

5.2 The method

5.2.1 Simulation of muscle mechanics

Several differential models for the simulation of muscular mechanics have been developed. Between the most popular models, there is the one proposed by [21, 22] and there is also the viscoelastic model proposed by [4]. The latter uses the Hill-Maxwell mechanical model based on the Huxley theory and the proposals of [53]. In these models, the state of the muscle in a concrete time is represented as a mesh, and the muscular mechanics (defined as the control of muscular contraction in each point of the muscle) is simulated by a differential model that updates the mesh in the next discrete time.

Figure 5.1 displays the basic scheme of these models. In each iteration, each point of the mesh updates its features, such as position, speed, acceleration, elasticity, and so on, given the finite element model such as [21, 22] or [4], among others. It is supposed to be an initial state of the mesh and also an initial perturbation that only affects the first iteration. The perturbation is usually defined as a pressure in a specific point. Since the model returns the mesh in discrete times, a memory and a clock to impose the iterations is needed.

The main data in the finite element model is a hexahedral mesh where each node (that represents a point in the muscle) is connected to three other nodes, so quadrilateral faces are set between nodes. Nodes in the mesh have two main attributes: one is the 3D position of the node, which represents the position

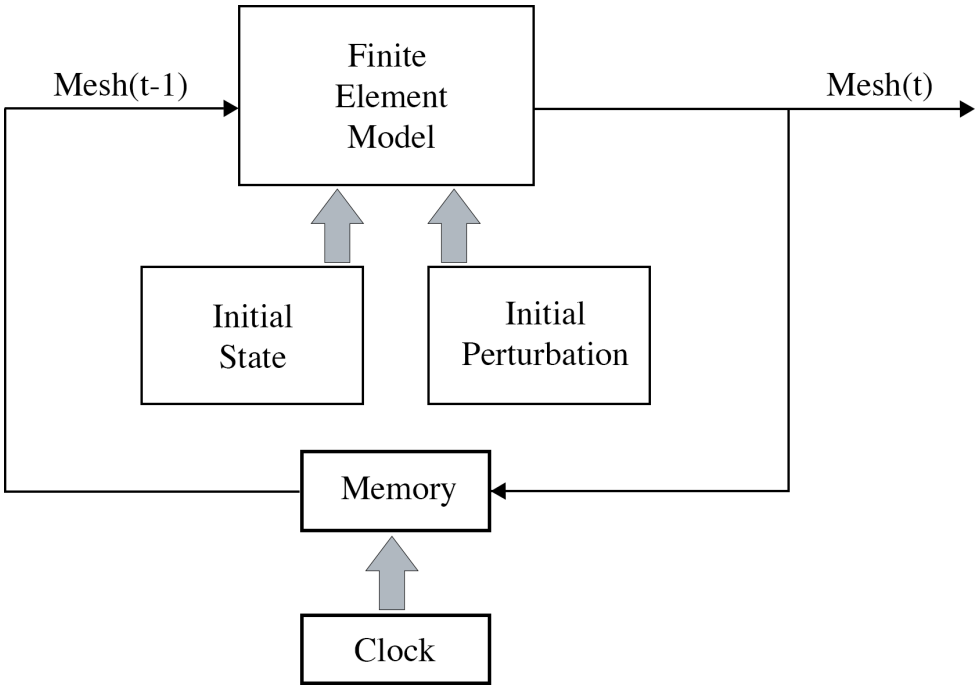


Fig. 5.1 Classical iterative model for the muscle mechanics that uses a finite-element model.

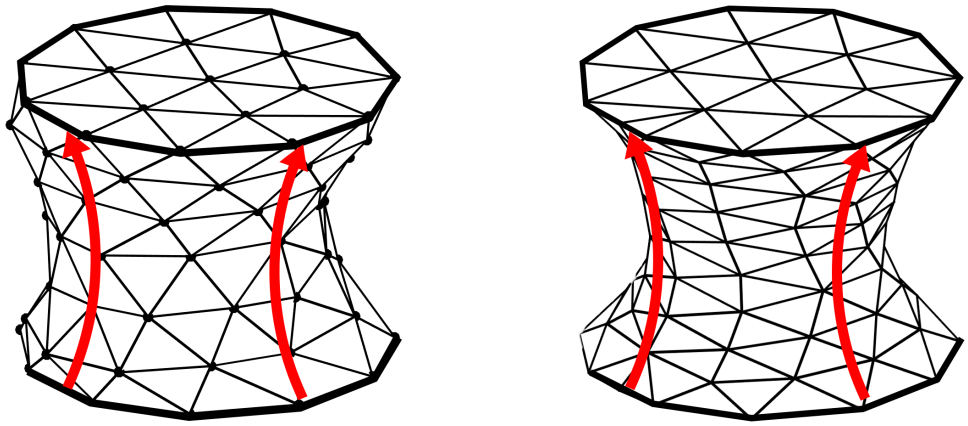


Fig. 5.2 Representation of a mesh based on hexahedrons. The right one is more regular than the left one.

of a muscle point in the space. The other attribute is the physical information of the muscle point, as for example elasticity, speed or acceleration. Note that edges do not have attributes. The finite element model accuracy depends on how regular are the hexahedral faces and also how similar are their areas [21, 22]. Nevertheless, the initial perturbation on the mesh moves the nodes from their original positions and therefore, its faces begin to be more irregular in each iteration. Thus, the finite element model accuracy is negatively influenced by the nodes movement. Figure 5.2 shows two meshes. On the right side, the faces are more regular between them than in the left one. In this case, the prediction accuracy of the mesh state in the next iteration is supposed to be higher in the right mesh than in the left one. Although the right mesh is the same as the left but recomputed, the general shape is almost the same.

The method presented here (Figure 5.3) computes again the mesh in each iteration, reducing the degradation of the finite element model accuracy. That is, the Mesh recomputing module is added to the classical scheme on Figure 5.1. Recomputing the mesh means finding a more regular mesh and, therefore,

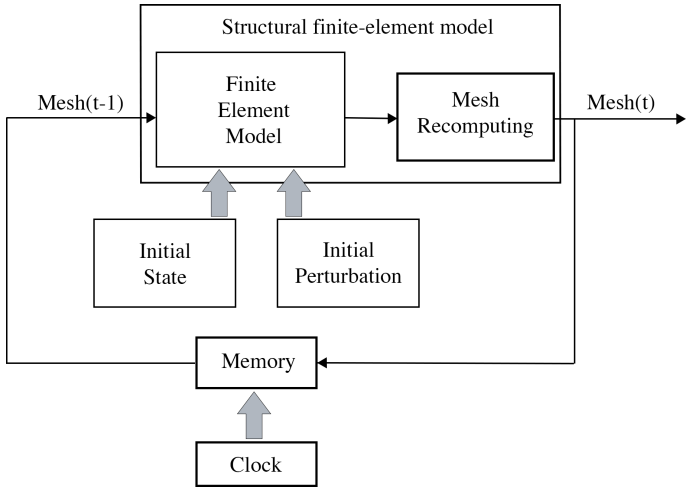


Fig. 5.3 The iterative model for the muscle mechanics that uses the structural finite-element model instead of the finite-element model. The mesh is recomputed in each iteration.

the differential model becoming more accurate while computing the next state of the mesh. Nevertheless, the main problem arises when new nodes appear in the new mesh, since the physical properties of these new positions have to be deduced.

Figure 5.4 provides an example of a three node mesh performed by the new model given an iteration. The finite differential model deduces the new positions of the nodes and their physical information (the physical information is not shown in the figure). The mesh recomputing module deduces that it is worth to add a new node to make the triangles more equilateral. Note that the 3D position and the physical information of this new node has to be deduced by the mesh recomputing module. Thus, in each iteration the 3D positions and the physical information of the "old nodes" are deduced by the finite differential module, but the 3D positions and the physical information of the "new nodes" are deduced by the mesh recomputing module.

Clearly, recomputing the mesh has a temporal cost to be considered while simulating muscles, since meshes can have up to 10.000 nodes. In this

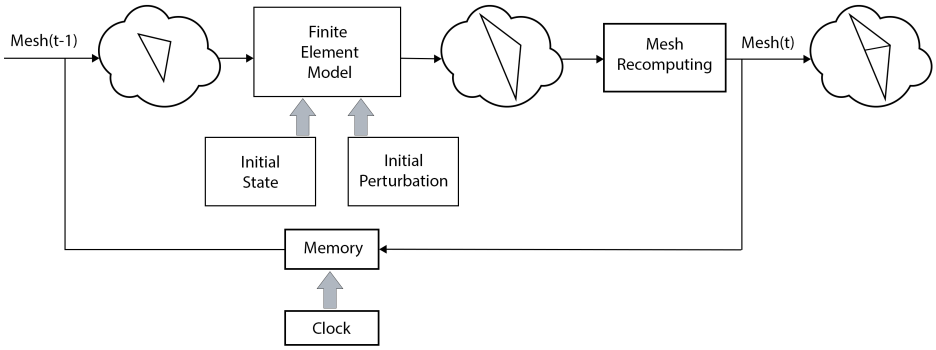


Fig. 5.4 An example of the evolution of a mesh, represented as a cloud with three nodes in the new model.

proposal this is done in linear computational cost with respect to the number of the mesh nodes, because of the Belief algorithm [38].

In Section 5.2.2, the new model is explained in detail. It is important to point out that the finite differential models for the simulation of muscular mechanics are not explained in detail here because they outreach the scope of this thesis. For more information about the differential models see [21, 22, 4, 53]. The three graph-matching algorithms previously seen (Bipartite graph matching [44], Greedy graph matching [37] and Belief propagation graph matching [38]) can be used in the Mesh recomputing module as we present in Section 5.3.

5.2.2 Differential model with mesh recomputing

The proposed model for the muscle mechanics simulation is composed of two main parts: the first part is a classical finite-element model and the second one is a structural pattern recognition model, as shown in Figure 5.3. As commented in Section 5.1, any finite-state model can be used, so the present section is primarily focused on the structural pattern recognition model. Figure 5.5 displays the basic scheme of this second part called

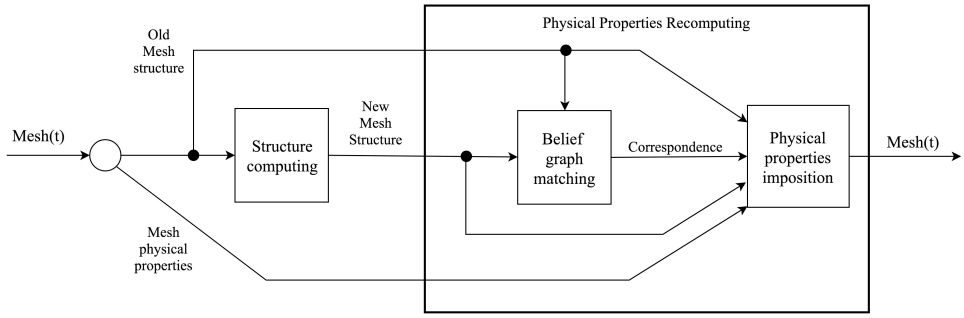


Fig. 5.5 General scheme of the Mesh recomputing model.

Mesh recomputing model. The first circumference is not a real process because it simply splits the mesh into its two main components: the structure (the set of nodes with their 3D position and also the set of edges) and the physical properties (the set of nodes with their physical properties). Therefore, the general scheme of the Mesh recomputing model has two main steps: the Structure recomputing module and the Physical properties recomputing module.

The Structure recomputing module returns the best hexahedral mesh from the initial mesh structure. The new mesh is based on the minimisation of the square factor of its area faces using the *Delaunay* algorithm [11]. As commented in the introduction, the new nodes do not have physical properties since they are located in different positions from those of the original mesh. The computation of these properties occurs in the second part of the process.

The Physical properties recomputing module consists of two main processes. In the first process, the graph correspondence between the old and the new mesh is deduced through the Belief propagation graph matching [38]. This graph correspondence is deduced by considering the structure of the graphs, that is, the node 3D positions and the edges. As commented in section 3.1, this matching algorithm needs an initial proportionally small set of node-to-node mappings called *Seeds*. These *Seeds* are composed of the nodes forming the surface of the mesh (external nodes). Most of this surface

nodes do not modify their position, except for those receiving the initial perturbation. This is because they are little affected by the elasticity of the muscle. Then, it can be assumed that the surface nodes in the initial and final mesh are mapped only considering their identification, unaltered from the initial mesh to the final one. The Belief propagation graph matching handles the surface nodes that modify their position due to the initial perturbation as noise elements.

When the mesh correspondence is computed, the model is ready to impose the physical properties to the new mesh. The nodes in the new mesh mapped by the correspondence (a substitution operation in the graph edit distance nomenclature) receive the physical properties of the mapped node in the old mesh. Moreover, the non-mapped nodes in the new mesh (that is, the ones inserted in the graph edit distance nomenclature) receive the average of the physical properties of the surrounding nodes (the neighbour nodes). Figure 5.6 shows a simple example of the three node-to-node mapping options: *Seeds* (nodes on the surface), substitutions (inner nodes mapped by the graph matching algorithm) and insertions (inner nodes non-mapped by the graph matching algorithm). Node deletions are not considered here since this implies that there is no node in the new mesh and therefore no properties to compute.

5.3 Experimental validation

We have organised the experimental validation in three sections: - In the first section, we compare the behaviour of the classical method with the method presented in this thesis.

To do so, we show the evolution of an artificial muscle that initially has a cylindrical shape given the classical (Figure 5.1) and the new scheme (Figure 5.3).

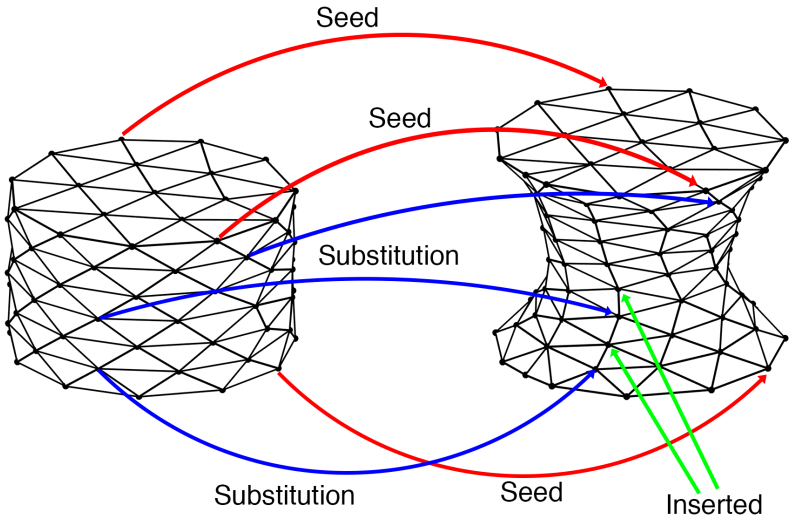


Fig. 5.6 General scheme of the physical property imposition through three options: *Seeds*, substitutions and insertions.

- In the second section, we analyse the effect of using different graph-matching algorithms in the mesh recomputing module. For this reason, we compare the evolution of this cylinder given three sub-optimal graph-matching algorithms used on the mesh recomputing module. Thus, we substitute the belief propagation graph matching for other graph matching algorithms that have higher computational cost.

- Finally, in the third section we apply our method to simulate the evolution of the shape of a human heart, given an initial pressure. This application constitutes the leitmotiv of the new method. As commented before, the classical methods tend to be less precise when the number of iterations increase, because the triangles of the mesh tend to be less smoother.

5.3.1 Comparing graph matching algorithms

Figure 5.7 (left column) presents the evolution of a synthetic deformable muscle with a cylindrical shape, given the classical scheme presented in Figure 5.1 (no recomputing the mesh) when an external perturbation is imposed in a side. Initially, the cylinder has a radius of 100 millimetres and is 300 millimetres height. The advantage of using this cylinder is that we can know the real evolution when a perturbation is imposed, assuming some elasticity properties. Then, we can analyse how much accurate the simulation is. For instance, we know that the perturbation modifies the cylinder 25 millimetres in height and 100 millimetres wide at its centre. The modelled cylinder is originally composed of a 100 nodes mesh, but most of them are inner nodes and therefore are occluded.

Figure 5.7 (right column) shows the evolution of the cylinder given the new scheme presented in Figure 5.3 (the mesh is recomputed in each step). In this case, the Belief graph matching is used to deduce the mesh correspondence. The nodes on the top and bottom sides of the cylinder are the *Seeds* used in the Belief algorithm. Comparing both evolutions, the simulation is more accurate in the right column than in the left one. The shape of the top and bottom sides are the same in both columns because the perturbation applied does not affect them. Due to a larger number of triangles and a better organisation of them, the mesh can represent the deformation in a more realistic way in each iteration when the differential model is applied.

5.3.1.1 Comparing the sub-optimal graph matching algorithms

We have compared the three graph matching algorithms commented in Section 1.3: Bipartite [36], Greedy [37] and Belief [38]. These are the most used algorithms when the applications need to compare huge graphs, or the runtime is an important restriction. Calculate the mesh correspondences becomes an important handicap not only from the runtime point of view but also from the storage space. As commented, the model presented here can

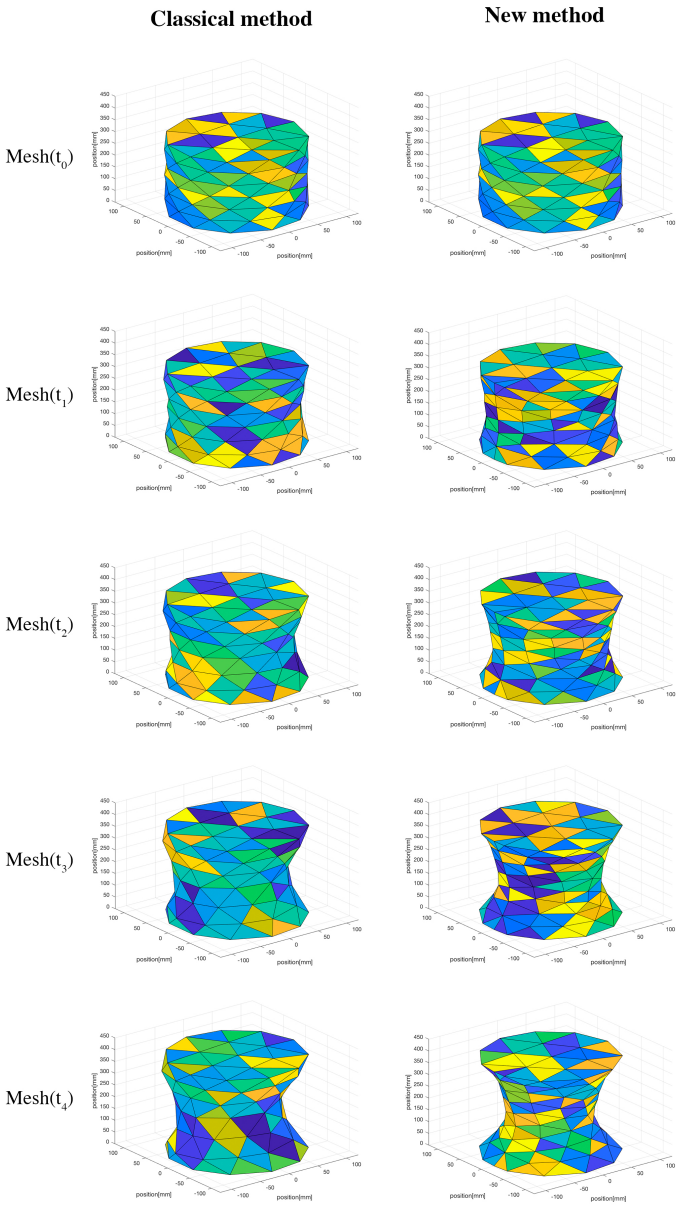


Fig. 5.7 Evolution of the cylinder given an initial perturbation. Left column: the classical method. Right column: the new method where the mesh is recomputed at each iteration.

Table 5.1 Normalised Graph edit distance between meshes for each iteration and computed by three algorithms.

Algorithm	Greedy	Bipartite	Belief
Mesh(t1), Mesh(t0)	395	351	407
Mesh(t2),Mesh(t1)	215	183	230
Mesh(t3),Mesh(t2)	480	453	441
Mesh(t4),Mesh(t3)	351	313	418

handle huge graphs (more than 100.000 nodes). We have to consider that the correspondence between the nodes of two meshes is computed in each iteration of the modelling algorithm.

The aim of this experiment is to show that the graph matching algorithm with the lowest cost (linear computational cost) achieves a similar correspondence quality compared to the two other algorithms with higher computational cost. Table 5.1 displays the normalised Graph edit distance between the input mesh and the output mesh in Figure 5.3, for each iteration. The minimum distances per column are indicated in bold. The Bipartite algorithm tends to deduce the minimum distance compared to the other algorithms.

In the experiments with the Belief algorithm, the *Seeds* are the nodes on the top and bottom sides, since these nodes tend to be less influenced by the external perturbation.

Table 5.2 shows the runtime spent by the three graph matching algorithms to deduce the graph correspondence. As expected, the Belief algorithm is the fastest, followed by the Greedy and the Bipartite algorithms. More concretely, in this experiments the Belief algorithm is forty times faster than the Bipartite and the Greedy is approximately six times faster than the Bipartite. Thus considering this simple shape composed of a cylinder, we conclude that it is worth to use the Belief algorithm since it achieves accuracies similar to the Bipartite but with a huge reduction of the runtime. In this case, we do not

Table 5.2 Runtime in seconds of the graph matching algorithms when computing the distance between meshes.

Algorithm	Greedy	Bipartite	Belief
Mesh(t1), Mesh(t0)	1.05	8.27	0.22
Mesh(t2),Mesh(t1)	1.22	8.47	0.19
Mesh(t3),Mesh(t2)	2.96	12.79	0.3
Mesh(t4),Mesh(t3)	2.66	12.49	0.21

recommend the use of the Greedy algorithm because the achieved accuracy is similar to the Belief algorithm but much slower. Note that the computational cost of these algorithms depends on the number of nodes. When the number of nodes in the meshes increases, so it does the difference in the runtime.

5.3.1.2 Simulating the dynamics of a human heart

In this section we have used a real data set to test the system proposed in the present thesis. Figure 5.8 shows the mesh representation of a real human heart with 37599 nodes, where the maximum axis length in each dimension is 120 mm. We have chosen to model the human heart because it is a muscular organ, so it fits the theoretical model introduced in section 5.1. In Figure 5.9 the heart is zoomed to show in detail the polygons created by the mesh. The colour represents in both figures the position in z axis of each polygon: from 0 (the darkest colour) to the maximum value (the brightest colour).

We have recalculated the mesh of the heart T times during the simulation after applying a force to the heart model. T is determined by the clock of the system to obtain a better representation of the muscle's movement. The mathematical model increased the accuracy of the results due to the precision of the mesh.

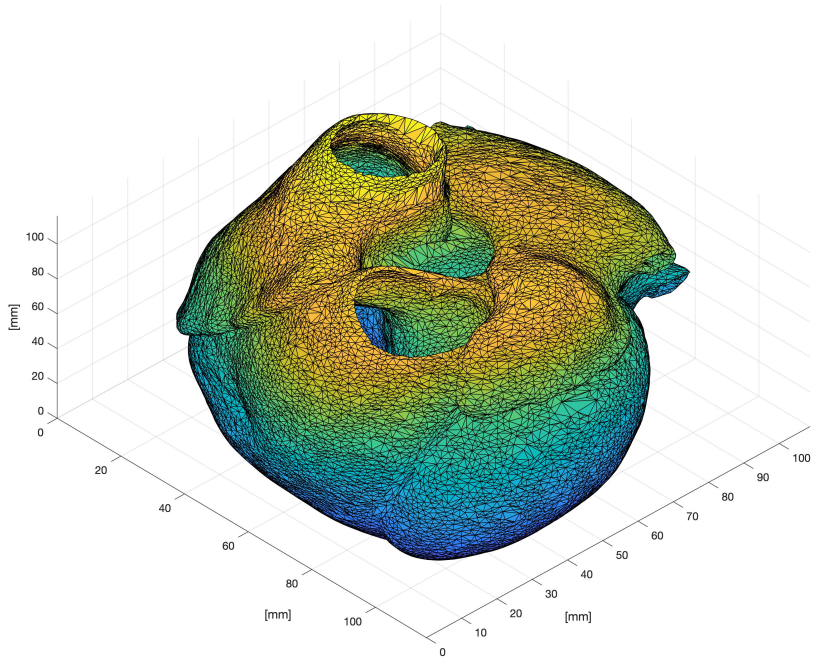


Fig. 5.8 Mesh representation of a real human heart.

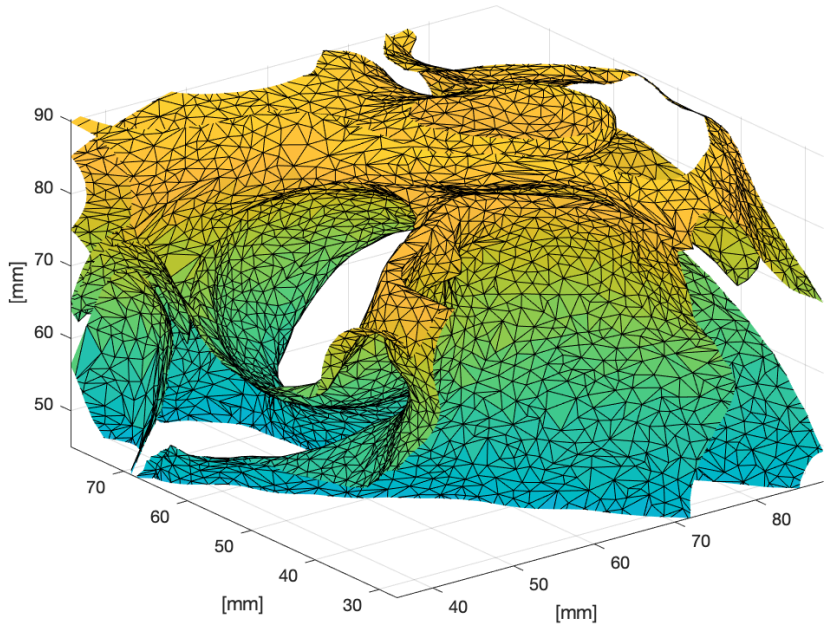


Fig. 5.9 Zoomed mesh of a real human heart.

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

Chapter 6

Conclusions

6.1 Graph Edit Distance Testing through Synthetic Graphs Generation

This thesis has presented a method to generate synthetic graphs with upper and lower bounds. In the practical experimentation, we realised that the time spent to generate the pairs of graphs is negligible with regard to the time spent to compare the pair of graphs with the fastest algorithm. This means that our synthetic graph generation method opens the door to test the current and future graph-matching algorithms with large graphs. It is important to emphasize that the bottle neck is the runtime of the graph-matching algorithms but not the generation of the graphs. In this sense, the runtime of generating a pair of graphs and their bounds is linear with regard to the order of the graphs. Moreover, if we use an actual computer with a non-optimised Matlab code, the runtime of the generation algorithm is lower than six seconds, considering graphs of 600 nodes. This means that it is feasible to generate a large graph database, with large graphs and their upper and lower bounds in a reasonable time.

6.2 Error-tolerant graph matching in linear computational cost using an initial small partial matching

For the first time, we have presented an error-tolerant graph-matching algorithm with linear computational and linear space costs with respect to the nodes. Specifically, the computational cost is $O(d^{3.5} \cdot n)$, where d is the number of output edges per node and n the order of the graphs. This algorithm is useful for computing the correspondence between huge graphs or social networks.

To achieve this low computational cost, the Belief algorithm needs an initial node-to-node mapping to begin to spread the knowledge of the matching between the graphs, called *Seeds*. The experimental validation demonstrates that this algorithm is clearly faster than the two of the most used algorithms, although the average distance seems to be more sub-optimal in this case. Moreover, we have deduced that the more *Seeds* given to the algorithm, the better the deduced correspondence.

It is also the first time that the matching between two large social networks has been deduced for two main reasons: the linear runtime with respect to the number of nodes, and in addition, the fact that a bi-dimensional matrix (where the number of rows or columns is the graph order) does not need to be defined. Thus, we have achieved the aim of analysing the temporary friendship of a social network that could not be achieved with the other graph matching algorithms presented in the literature.

6.3 A general framework to learn the graph edit costs based on an embedded model

Edit costs functions are application dependent and usually manually set based on maximising the accuracy in the recognition process. We have proposed a general framework to learn the substitution, deletion and insertion costs based on minimising the Hamming distance between the deduced correspondences and the ground-truth correspondences. Moreover, we have concretised our framework on two models, one based on neural networks and the other based on multimodal probability density functions. We have tested our framework on fourteen public databases with different characteristics, and we have empirically deduced that the neural network achieves the highest accuracies. Nevertheless, we conclude that the inclusion or exclusion of the histograms information in the embedded vector depends on the sparsity of this vector. The Probability density function method has obtained poor results. A possible explanation is that the number of correspondences in the database is too low and the number of dimensions too high. Further research is needed to test our model with huge graphs and with a larger number of graphs in the databases. To do so, we could use the algorithm presented in [39] that generates pairs of graphs with a ground truth in almost linear cost. With these graphs, we could construct a large database of huge graphs and then analyse the performance of the Probability density function. Moreover, using this new database, the classical neural network method could be converted into a deep neural network.

6.4 Incorporating a graph matching algorithm into a muscle mechanics model

We propose a new discrete model for the simulation of muscle mechanics where the mesh grid is recomputed in each iteration. Recomputing the mesh requires to find the correspondence between nodes of the previous mesh and the recomputed one. This is a very costly process and maybe that is the reason why recomputing the mesh was not considered in the literature models. Nevertheless, our model solves this problem by using a graph matching algorithm that deduces a sub-optimal correspondence in linear cost. This algorithm needs an initial small set of node-to-node mappings. We have incorporated this mappings into the method by selecting the external nodes of the mesh that tend to be more static than the inner ones. The aim of developing a new method was to increase the accuracy of the muscle dynamics, considering that when the number of iterations increase, the accuracy tends to decrease. To conclude, our method presents higher accuracy at the expense of a linear and low increase of runtime.

Chapter 7

List of Publications

7.1 Journals

1. Santacruz, P. and Serratosa, F. (2018a). Error-tolerant graph matching in linear computational cost using an initial small partial matching. *Pattern Recognition Letters, Online*. **Quartile Q1.**

2. Santacruz, P. and Serratosa, F. (2019a). A general framework to learn the graph edit costs based on an embedded model. *Neural Processing Letters,* ***Under review***.

3. Santacruz, P. and Serratosa, F. (2019b). Graph edit distance applied to define muscle mechanics. *Medical Engineering and Physics,* ***Under review***.

7.2 Congresses

1. Algabli, S., Santacruz, P., and Serratosa, F. (2019). Learning the graph edit distance parameters for point-set image registration. *Computer Analysis of Images and Patterns*. **Core B - Under review**.

2. Santacruz, P. and Serratosa, F. (2018b). Graph edit distance testing through synthetic graphs generation. In *24th International Conference on Pattern Recognition 2018, Beijing, China, August 20-24, 2018*, pages 572–577. **Core B**.

3. Santacruz, P. and Serratosa, F. (2018c). Learning the sub-optimal graph edit distance edit costs based on an embedded model. In *Structural, Syntactic, and Statistical Pattern Recognition - Joint IAPR International Workshop, S+SSPR 2018, Beijing, China, August 17-19, 2018, Proceedings*, pages 282–292. **Core A**.

References

[1] Abu-Aisheh, Z., Raveaux, R., and Ramel, J. (2016). Anytime graph matching. *Pattern Recognition Letters*, 84:215–224.

[2] Algabli, S., Santacruz, P., and Serratosa, F. (2019). Learning the graph edit distance parameters for point-set image registration. *Computer Analysis of Images and Patterns*.

[3] Algabli, S. and Serratosa, F. (2018). Embedding the node-to-node mappings to learn the graph edit distance parameters. *Pattern Recognition Letters*, 112:353–360.

[4] Bestel, J. (2000). *Modèle différentiel de la contraction musculaire contrôlée. Application au système cardio-vasculaire*. PhD thesis, Université Paris-Dauphine. Thèse de doctorat: Automatique Paris.

[5] Blumenthal, D. B. and Gamper, J. (2018). On the exact computation of the graph edit distance. *Pattern Recognition Letters, Online*.

[6] Caetano, T. S., McAuley, J. J., Cheng, L., Le, Q. V., and Smola, A. J. (2009). Learning graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(6):1048–1058.

[7] Conte, D., Foggia, P., Sansone, C., and Vento, M. (2004). Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(03):265–298.

[8] Cortés, X. and Serratosa, F. (2015). Learning graph-matching edit-costs based on the optimality of the oracle's node correspondences. *Pattern Recognition Letters*, 56:22–29.

[9] Cortés, X. and Serratosa, F. (2016). Learning graph matching substitution weights based on the ground truth node correspondence. *International Journal of Pattern Recognition and Artificial Intelligence*, 30(02):1650005.

[10] Cortés, X., Serratosa, F., and Riesen, K. (2016). On the relevance of local neighbourhoods for greedy graph edit distance. In *S+SSPR*, volume 10029 of *Lecture Notes in Computer Science*, pages 121–131.

[11] De Loera, J. A., Rambau, J., and Santos, F. (2010). *Triangulations: Structures for Algorithms and Applications*. Springer Publishing Company.

[12] Fankhauser, S., Riesen, K., and Bunke, H. (2011). Speeding up graph edit distance computation through fast bipartite matching. In *Graph-Based Representations in Pattern Recognition - 8th IAPR-TC-15 International Workshop, GbRPR 2011, Münster, Germany, May 18-20, 2011. Proceedings*, pages 102–111.

[13] Ferrer, M., Serratosa, F., and Riesen, K. (2015). Improving bipartite graph matching by assessing the assignment confidence. *Pattern Recognition Letters*, 65:29–36.

[14] Fischer, A., Suen, C. Y., Frinken, V., Riesen, K., and Bunke, H. (2015). Approximation of graph edit distance based on Hausdorff matching. *Pattern Recognition*, 48(2):331–343.

[15] Foggia, P., Percannella, G., and Vento, M. (2014). Graph matching and learning in pattern recognition in the last 10 years. *International Journal of Pattern Recognition and Artificial Intelligence*, 28(01):1450001.

[16] Gallagher, B. (2006). Matching structure and semantics: A survey on graph-based pattern matching. In *Capturing and Using Patterns for Evidence Detection, Papers from the 2006 AAAI Fall Symposium, Washington, DC, USA, October 13-15, 2006.*, pages 45–53.

[17] Gao, X., Xiao, B., Tao, D., and Li, X. (2010). A survey of graph edit distance. *Pattern Analysis Applications*, 13(1):113–129.

[18] Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York.

[19] Gill, P. E., Murray, W., and Wright, M. H. (1981). *Practical optimization*. Academic Press Inc., London.

[20] Gold, S. and Rangarajan, A. (1996). A graduated assignment algorithm for graph matching. *IEEE Transactions on Pattern Analysis and Maching Intelligence*, 18(4):377–388.

[21] Huxley, A. (1957). Muscle structure and theories of contraction. In Butler, J. and Katz, B., editors, *Progress in biophysics and biophysical chemistry*, chapter 7, pages 257–318. Pergamon Press.

[22] Huxley, A. (1974). Muscular contraction. *The Journal of Physiology*, 243:1–43.

[23] Justice, D. and III, A. O. H. (2006). A binary linear programming formulation of the graph edit distance. *IEEE Transactions on Pattern Analysis and Maching Intelligence*, 28(8):1200–1214.

[24] Kuhn, H. W. and Yaw, B. (1955). The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, pages 83–97.

[25] Leordeanu, M., Sukthankar, R., and Hebert, M. (2012). Unsupervised learning for graph matching. *International Journal of Computer Vision*, 96(1):28–45.

[26] Lerouge, J., Abu-Aisheh, Z., Raveaux, R., Héroux, P., and Adam, S. (2017). New binary linear programming formulation to compute the graph edit distance. *Pattern Recognition*, 72:254–265.

[27] Leskovec, J., Kleinberg, J. M., and Faloutsos, C. (2007). Graph evolution: Densification and shrinking diameters. *Naval Research Logistics Quarterly*, 1(1):2.

[28] Luqman, M. M., Ramel, J., Lladós, J., and Brouard, T. (2013). Fuzzy multilevel graph embedding. *Pattern Recognition*, 46(2):551–565.

[29] Moreno-García, C. F., Cortés, X., and Serratosa, F. (2016). A graph repository for learning error-tolerant graph matching. In *Structural, Syntactic, and Statistical Pattern Recognition*, pages 519–529.

[30] Munkres, J. (1957). Algorithms for the assignment and transportation problems. *Journal of the Society of Industrial and Applied Mathematics*, 5(1):32–38.

[31] Neuhaus, M. and Bunke, H. (2005). Self-organizing maps for learning the edit costs in graph matching. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 35(3):503–514.

[32] Neuhaus, M. and Bunke, H. (2007). Automatic learning of cost functions for graph edit distance. *Information Sciences*, 177(1):239–247.

[33] Rebagliati, N., Solé-Ribalta, A., Pelillo, M., and Serratosa, F. (2012). Computing the graph edit distance using dominant sets. In *International Conference on Pattern Recognition*, pages 1080–1083. IEEE Computer Society.

[34] Riesen, K. (2015). *Structural Pattern Recognition with Graph Edit Distance - Approximation Algorithms and Applications*. Advances in Computer Vision and Pattern Recognition. Springer.

[35] Riesen, K. and Bunke, H. (2008). IAM graph database repository for graph based pattern recognition and machine learning. In *SSPR/SPR*, volume 5342 of *Lecture Notes in Computer Science*, pages 287–297. Springer.

[36] Riesen, K. and Bunke, H. (2009). Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision computing*, 27(7):950–959.

[37] Riesen, K., Ferrer, M., Dornberger, R., and Bunke, H. (2015). Greedy graph edit distance. In *Proceedings of the 11th International Conference on Machine Learning and Data Mining in Pattern Recognition - Volume 9166*, MLDM 2015, pages 3–16. Springer.

[38] Santacruz, P. and Serratosa, F. (2018a). Error-tolerant graph matching in linear computational cost using an initial small partial matching. *Pattern Recognition Letters, Online*.

[39] Santacruz, P. and Serratosa, F. (2018b). Graph edit distance testing through synthetic graphs generation. In *24th International Conference on Pattern Recognition 2018, Beijing, China, August 20-24, 2018*, pages 572–577.

[40] Santacruz, P. and Serratosa, F. (2018c). Learning the sub-optimal graph edit distance edit costs based on an embedded model. In *Structural, Syntactic, and Statistical Pattern Recognition - Joint IAPR International Workshop, S+SSPR 2018, Beijing, China, August 17-19, 2018, Proceedings*, pages 282–292.

[41] Santacruz, P. and Serratosa, F. (2019a). A general framework to learn the graph edit costs based on an embedded model. *Neural Processing Letters,* ***Under review***.

[42] Santacruz, P. and Serratosa, F. (2019b). Graph edit distance applied to define muscle mechanics. *Medical Engineering and Physics,* ***Under review***.

[43] Santo, M. D., Foggia, P., Sansone, C., and Vento, M. (2003). A large database of graphs and its use for benchmarking graph isomorphism algorithms. *Pattern Recognition Letters*, 24(8):1067–1079.

[44] Serratosa, F. (2014a). Fast computation of bipartite graph matching. *Pattern Recognition Letters*, 45:244–250.

[45] Serratosa, F. (2014b). Speeding up fast bipartite graph matching through a new cost matrix. *International Journal of Pattern Recognition and Artificial Intelligence*, 29:1550010.

[46] Serratosa, F. (2015a). Computation of graph edit distance: Reasoning about optimality and speed-up. *Image Vision Computing*, 40:38–48.

[47] Serratosa, F. (2015b). Graph databases. http://deim.urv.cat/~francesc.serratosa/databases/.

[48] Serratosa, F. (2018). A methodology to generate attributed graphs with a bounded graph edit distance for graph-matching testing. *International Conference on Pattern Recognition*, 32(11):1850038.

[49] Serratosa, F. and Cortés, X. (2015a). Graph edit distance. *Pattern Recognition Letters*, 65(C):204–210.

[50] Serratosa, F. and Cortés, X. (2015b). Graph edit distance: Moving from global to local structure to solve the graph-matching problem. *Pattern Recognition Letters*, 65:204–210.

[51] Solé-Ribalta, A., Serratosa, F., and Sanfeliu, A. (2012). On the graph edit distance cost: Properties and applications. *International Conference on Pattern Recognition*, 26(5).

[52] Vento, M. (2015). A long trip in the charming world of graphs for pattern recognition. *Pattern Recognition*, 48(2):291–301.

[53] Zahalak, G. I. and Ma, S. P. (1990). Muscle activation and contraction: constitutive relations based directly on cross-bridge kinetics. *Journal of Biomechanical Engineering*, 112(1):52–62.

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

UNIVERSITAT ROVIRA I VIRGILI
ERROR-TOLERANT GRAPH MATCHING ON HUGE GRAPHS AND LEARNING STRATEGIES ON THE EDIT COSTS
Jose Luis Santacruz Muñoz

UNIVERSITAT
ROVIRA i VIRGILI