International Master of Research in Computer Science: Computer Aided Decision Support

Graph for Pattern Recognition

Author:

Romain Raveaux

Zeina Abu-Aisheh

Jean-Yves Ramel

in the

RFAI groups
at the University of Tours

October 2016

Abstract

Attributed graphs are powerful data structures for the representation of complex entities. In a graph-based representation, vertices and their attributes describe objects (or part of objects) while edges represent interrelationships between the objects. Due to the inherent genericity of graph-based representations, and thanks to the improvement of computer capacities, structural representations have become more and more popular in the field of Pattern Recognition (PR). PR problems can take advantage of graph in two ways :

- through Graph Matching
- through Graph Embedding

Keywords Graph matching, Graph embedding, Pattern Recognition

Part I

State of the Art

Chapter 2

Strength and Weakness of Actual Graph Matching Methods

The art challenges the technology, and the technology inspires the art. John Lasseter (Director)

Contents

Abstract

In this chapter, we present an overview of the definitions and the concepts that underlie the presented works in the thesis. We also dig into the details of existing Graph Matching problems as well as methods dedicated to solving them. A particular focus on Graph Edit Distance problem and techniques is made in the last section of this chapter.

2.1 Definitions and Notations

2.1.1 Graph

Graphs are an efficient data structure and the most general formalism for object representation in structural Pattern Recognition (PR). They are basically composed of a finite or infinite set of vertices V, that represents parts of objects, connected by a set of edges $E \subseteq VXV$, that represents the relations between these two parts of objects, where each edge connects two vertices in the graph. Formally saying, $e(u_i, u_j)$, or e_{ij}, where both u_i and u_j are vertices that belong to the set V.

Definition 1 *Graph*
$G = (V, E)$
V is a set of vertices
E is a set of edges such that $E \subseteq V \times V$

2.1.2 Subgraph

A subgraph G_s is a graph whose set of vertices V_s and set of edges E_s form subsets of the sets V and E of graph G. A subgraph G_s of graph G is said to be induced (or full) if, for any pair of vertices u_i and u_j of G_s, $e(u_i, u_j)$ is an edge of G_s if and only if $e(u_i, u_j)$ is an edge of G. In other words, G_s is an induced subgraph of G if it has exactly the edges that appear in G over the same vertices set, i.e., $E_s = E \cap V_s \times V_s$

Definition 2 *Subgraph*
$V_s \subseteq V$
$E_s \subseteq E \cap V_s \times V_s$

Figure 2.1 shows a subgraph G_s in a graph G.

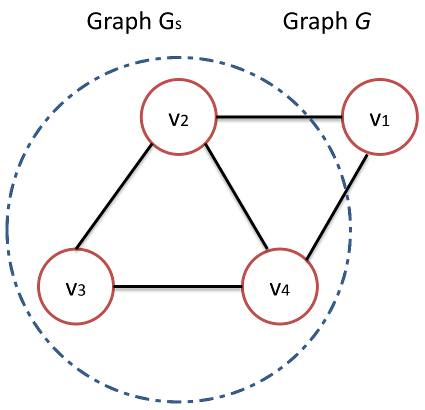


Figure 2.1: A subgraph G_s: $e(v_2, v_3)$, $e(v_3, v_4)$, $e(v_4, v_2)$ of graph G.

2.1.3 Directed and Undirected Graphs

A graph G is said to be *undirected* when each edge e_{ij} of the set E has no direction. This kind of graphs represents a symmetric relation. Mathematically saying: $e(u_i, u_j) \in E \Leftrightarrow e(u_j, u_i) \in E$. In contrast to the *directed* graphs which respect the direction that is assigned to each edge e_{ij}. Thus, for the *directed* graphs $e(u_i, u_j) \neq e(u_j, u_i)$.

2.1.4 Attributed Graphs

Non-attributed graphs are only based on their neighborhood structures defined by edges, *e.g.* molecular graphs where the structural formula is considered as the representation of a chemical substance. Thus, no attributes can be found on neither the edges nor the vertices of graphs. Whereas in attributed, or labelled, graphs (AG), significant attributes can be found on edges, vertices or both of them which efficiently describe objects (in terms of shape, color, coordinate, size, etc.) and their relations.

In AGs, two extra parameters have been added (μ, ζ) where vertices' attributes and edges' attributes are represented successively.

Mathematically speaking, AG is considered as a set of 6 tuples (V,E,L_V,L_E,μ,ζ) such that:

Definition 3 *Attributed Graph*
G = (V,E,L_V,L_E,μ,ζ)
V is a set of vertices
E is a set of edges such as $E \subseteq V \times V$
L_V is a set of vertex attributes
L_E is a set of edge attributes
$\mu : V \rightarrow L_V$. μ is a vertex labeling function which associates the label l_{u_i} to a vertex u_i
$\zeta : E \rightarrow L_E$. ζ is an edge labeling function which associates the label $l_{e_{ij}}$ to an edge e_{ij}

Definition 3 allows to handle arbitrarily structured graphs with unconstrained labeling functions. For example, attributes of both vertices and edges can be part of the set of integers $L = \{1, 2, 3, \cdots\}$, the vector space $L = \mathbb{R}^n$ and/or a finite set of symbolic attributes $L = \{x, y, z, \cdots\}$.

In PR, a combination of both symbolic and numeric attributes on vertices and edges is required in order to describe the properties of vertices and their relations. For notational convenience, directed attributed relational graphs are simply referred to as graphs in the rest of the thesis.

2.1.5 Graph Size

The graph order $|V|$ refers to the number of vertices of the given graph G.

2.1.6 Vertex Degree

The degree of vertex u_i refers to the number of edges connected to u_i. Note that when the graph G is directed then one should consider u_i's in-degree and out-degree where the in-degree refers to the number of incoming edges and the out-degree refers to the number of outgoing edges of vertex u_i.

2.1.7 Graph Density

Graph Density is the ratio of the number of edges divided by the number of edges of a complete graph with the same number of vertices. Dense graphs represent graphs with large vertices' degrees (i.e., large number of edges connected to each vertex u_i in the graph G) while sparse graphs represent graphs with low vertices' degrees.

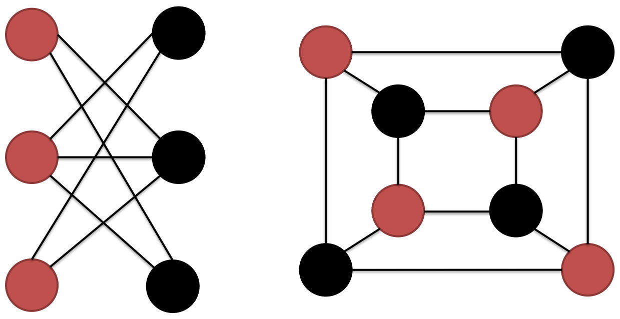


Figure 2.2: Two examples of bipartite graphs

2.1.8 Special Graphs

Planar Graph This term refers to any graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints.

Weighted Graph This kind of graphs is a particular case of attributed graphs where $L_E = \mathbb{R}$ such that edge attribute $l_{e_{ij}}$ represents the weight of an edge $e(u_i, u_j)$.

Directed Acyclic Graph A directed acyclic graph is a directed graph with no directed cycles, such that there is no way to start at some vertex u_i and follow a sequence of edges that eventually loops back to u_i again.

Bipartite Graph This term refers to any graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V; that is, U and V are each independent sets. Figure 2.2 illustrates two examples of a bipartite graph.

Simple Graph A simple graph is a graph that does not contain self-loops or multi-edges (i.e., two or more edges connecting the same two vertices in a graph).

2.1.9 Graph Matching

Graph matching (GM) is the process of finding a correspondence between the vertices and the edges of two graphs that satisfies some (more or less stringent) constraints ensuring that similar substructures in one graph are mapped to similar substructures in the other. Matching problems are divided into two broad categories: the first category contains exact GM problems that require a strict correspondence among the two objects being matched or at least among their subparts. The second category defines error-tolerant GM problems, where a matching can occur even if the two graphs being compared are structurally different to some extent. GM, whether exact or error-tolerant, is applied on patterns that are transformed into graphs. This approach is called structural in the sense of using the structure of the patterns to compare them.

In the following sections we focus on graph-based matching problems in Pattern Recognition. For the sake of clarity, we start from the easiest problem to express up to the hardest one and then we shed light on the problem that is tackled in the thesis.

2.1.10 Exact Graph Matching

In this type of problems and at the aim of matching two graphs, significant part of the topology together with the corresponding vertex and edge attributes in the graphs G_1 and G_2 have to be identical. Exact GM methods can only be efficiently applied on attributed graphs whose attributes are symbolic or non-attributed graph. For any algorithm proposed for solving an exact GM problem, a yes or no answer is outputted. In other words, the output of each exact GM algorithm indicates whether or not a (sub)graph is found in another graph. When a (sub)graph is found, it will be identified in both of the involved graphs. This problem is not directly tackled in the thesis. However, it is considered as the basis of GM problems thanks to its easiness to be explained. Thus, in this section, formal introductions of exact GM problems is given.

2.1.10.1 Graph isomorphism

The mapping, or matching, between the vertices of the two graphs must be edge-preserving in the sense that if two vertices in the first graph are linked by an edge, they are mapped to two vertices in the second graph that are linked by an edge as well. This condition must be held in both directions, and the mapping must be bijective. That is, a one-to-one correspondence must be found between each vertex of the first graph and each vertex of the second graph. When graphs are attributed, attributes have to be identical. More formally, when comparing two graphs $G_1 = (V_1,E_1,L_{V_1},L_{E_1},\mu_1,\zeta_1)$ and $G_2 = (V_2,E_2,L_{V_2},L_{E_2},\mu_2,\zeta_2)$ we are looking for a bijective function $f : V_1 \rightarrow V_2$ which maps each vertex $u_i \in V_1$ onto a vertex $v_k \in V_2$ such that certain conditions are fulfilled:

Definition 4 *Graph isomorphism*
A bijective function $f : V_1 \rightarrow V_2$ is a graph isomorphism from G_1 to G_2 if:

1. $\forall u_i \in V_1,\ \mu_1(u_i) = \mu_2(f(u_i))$
2. $\forall (u_i, u_j) \in V_1 \times V_1, e(u_i, u_j) \in E_1 \Leftrightarrow e(f(u_i), f(u_j)) \in E_2$
3. $\forall e(u_i, u_j) \in E_1,\ \zeta_1(e(u_i, u_j)) = \zeta_2(e(f(u_i), f(u_j)))$

Figure 2.3 depicts an instance of the graph isomorphism problem. Note that G_1 and G_2 can be called source and target graphs, respectively. Both G_1 and G_2 are simple graphs. In this thesis, we also consider matching of simple graphs.

Graph isomorphism is one of the problems for which it has not yet been demonstrated if it belongs to NP-complete or not. However, there is still no algorithm that can solve the problem in polynomial time. Yet, readers who are aware of the recent rise of graph isomorphism might have heard about the claim of Babai in [11] of solving graph isomorphism in quasipolynomial time.

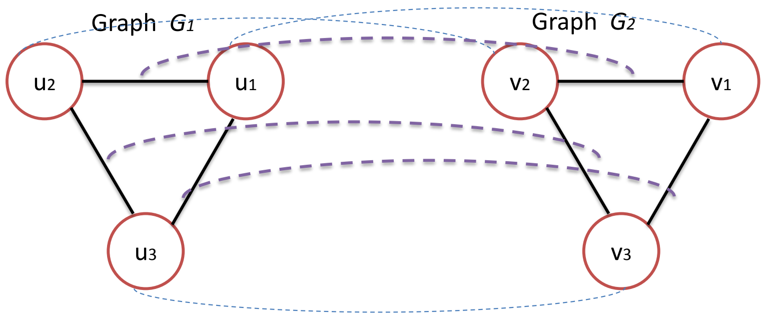


Figure 2.3: Graph isomorphism between G_1 and G_2.

2.1.10.2 Induced Subgraph Isomorphism (SGI)

It requires that an isomorphism holds between one of the two graphs and a vertex-induced subgraph of the other. More formally, when comparing two graphs $G_1 = (V_1,E_1,L_{V_1},L_{E_1},\mu_1,\zeta_1)$ and $G_2 = (V_2,E_2,L_{V_2},L_{E_2},\mu_2,\zeta_2)$ we are looking for a function $f : V_1 \rightarrow V_2$ which maps each vertex $u_i \in V_1$ onto a vertex $u_j \in V_2$ such that certain conditions are fulfilled :

Definition 5 *Induced subgraph isomorphism*
An injective function $f : V_1 \rightarrow V_2$ is a subgraph isomorphism from G_1 to G_2 if:

1. $\forall u_i \in V_1,\ \mu_1(v) = \mu_2(f(u_i))$
2. $\forall (u_i, u_j) \in V_1 \times V_1, e(u_i, u_j) \in E_1 \Leftrightarrow e(f(u_i), f(u_j)) \in E_2$
3. $\forall e(u_i, u_j) \in E_1,\ \zeta_1(e(u_i, u_j)) = \zeta_2(e(f(u_i), f(u_j)))$

In its exact formulation, the subgraph isomorphism must preserve the labeling, i.e., $\mu_1(u_i) = \mu_2(v_k)$ and $\zeta_1(e(u_i, u_j)) = \zeta_2(e(v_k, v_z))$ where $u_i, u_j \in V_1,\ v_k, v_z \in V_2,\ e(u_i, u_j) \in E_1$ and $e(v_k, v_z) \in E_2$.

Figure 2.4 depicts an instance of the graph isomorphism problem. The NP-completeness proof of subgraph isomorphism can be found in [55].

2.1.10.3 Monomorphism

Monomorphism, also known as partial subgraph isomorphism, is a light form of induced subgraph isomorphism. It also drops the condition that the mapping should be edge-preserving in both directions. It requires that each vertex of the source graph is mapped to a distinct vertex of the target graph, and each edge of the source graph has a corresponding edge in the target graph. However, the target graph may have both extra vertices and extra edges.

The subgraph monomorphism problem between a pattern graph G_1 and a target graph G_2 is defined by:

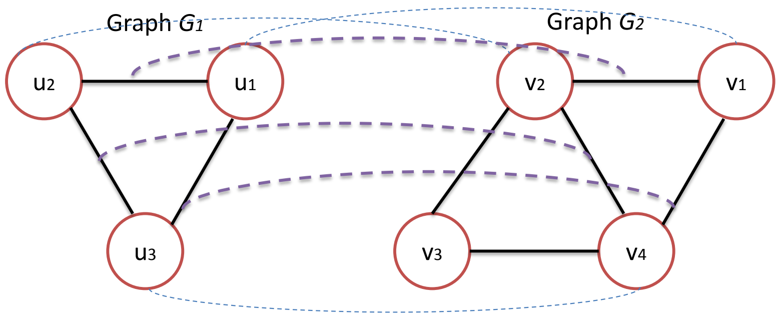


Figure 2.4: Induced subgraph isomorphism.

Definition 6 *Monomorphism*
An injective function $f : V_1 \rightarrow V_2$ is a subgraph isomorphism from G_1 to G_2 if:

1. $\forall u_i \in V_1,\ \mu_1(u_i) = \mu_2(f(u_i))$
2. $\forall e(u_i, u_j) \in E_1, (f(u_i), f(u_j)) \in E_2$
3. $\forall e(u_i, u_j) \in E_1,\ \zeta_1(e(u_i, u_j)) = \zeta_2(e(f(u_i), f(u_j)))$

As in SGI, in the exact formulation of subgraph monomorphism problem, the subgraph isomorphism must preserve the labeling, i.e., $\mu_1(u_i) = \mu_2(v_k)$ and $\zeta_1(e(u_i, u_j)) = \zeta_2(e(v_k, v_z))$ where $u_i, u_j \in V_1$, $v_k, v_z \in V_2$, $e(u_i, u_j) \in E_1$ and $e(v_k, v_z) \in E_2$.

2.1.10.4 Maximum Common Subgraph (MCS)

Maximum Common Subgraph is the problem of mapping a subgraph of the source graph to an isomorphic subgraph of the target graph. Usually, the goal is to find the largest subgraph for which such a mapping exists. Actually, there are two possible definitions of the problem, depending on whether vertex-induced subgraphs or partial subgraphs are used. In the first case, the maximality of the common subgraph refers to the number of vertices, while in the second one it is the number of edges that is maximized.

Definition 7 *Maximum Common Subgraph (MCS)*
Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. A graph $G_s = (V_s, E_s)$ is said to be a common subgraph of G_1 and G_2 if there exists subgraph isomorphism from G_s to G_1 and from G_s to G_2. The largest common subgraph is called the maximum common subgraph, or MCS, of G_1 and G_2.

2.1.11 Error-Tolerant Graph Matching Problems

The stringent constraints imposed by exact GM are, in some circumstances, too rigid for the comparison of two graphs. So the matching process must be tolerant: it must accommodate the differences by relaxing, to some extent, the constraints that define the matching type.

2.1.11.1 Problem Transformation: from Exact to Error-Tolerant

Error-tolerant matching is generally needed when no significant identical part of the structure together with the corresponding vertex and edge attributes in graphs G_1 and G_2 can be found. Instead, matching G_1 and G_2 is associated to a penalty cost. For example, this case occurs when vertex and edge attributes are numerical values (scalar or vectorial). The penalty cost for the mapping can then be defined as the sum of the distances between label values. A first solution to tackle such problems relies on a discretization or a classification procedure to transform the numerical values into nominal/symbolic attributes. The main drawback of such approaches is their sensitivity to frontier effects of the discretization or misclassification. A subsequent exact GM algorithm would then be unsuccessful. A second solution consists in using exact GM algorithms and customizing the compatibility function for pairing vertices and edges. The main drawback of such approaches is the need to define thresholds for these compatibilities. A last way consists in using an error-tolerant GM procedure that overcomes this drawback by integrating the numerical values during the mapping search. In this case, the matching problem turns from a decision one to an optimization one.

2.1.11.2 Substitution-Tolerant Subgraph Isomorphism

Substitution-Tolerant Subgraph Isomorphism [80] aims at finding a subgraph isomorphism of a pattern graph G_s in a target graph G. This isomorphism only considers label substitutions and forbids vertex and edge insertion in G. This kind of subgraph isomorphism is often needed in PR problems when graphs are attributed with real values and no exact GM can be found between attributes due to noise. A subgraph isomorphism is said to be substitution-tolerant when the mapping does not affect the topology. That is, each vertex and each edge of the pattern graph has a one-to-one mapping into the target graph, however, two vertices and/or edges can be matched (or substituted) even if their attributes are not similar. A substitution-tolerant mapping is generally needed when no exact mapping between vertex and/or edge attributes can be found, but when the mapping can be associated to penalty cost. For example, this case occurs when vertex and edge attributes are numerical values (scalar or vectorial) resulting from a feature extraction step as often in pattern analysis. See Figure 2.5.

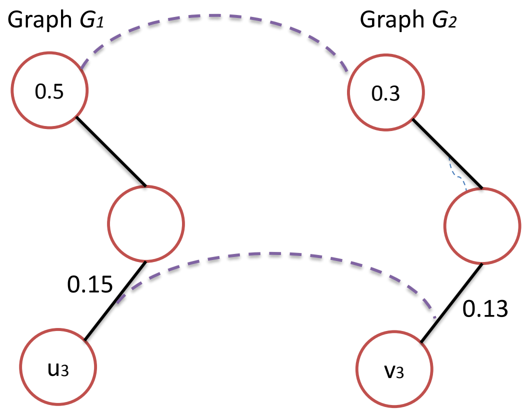


Figure 2.5: Substitution-tolerant subgraph isomorphism problem. G_1 and G_2 are attributed graphs, the mapping takes the difference between the attributes into account. Mappings are also edge-preserving.

Definition 8 *Substitution-Tolerant Subgraph Isomorphism*
An injective function $f : V_1 \rightarrow V_2$ is a subgraph isomorphism of $G_1 = (V_1,E_1,L_{V_1},L_{E_1},\mu_1,\zeta_1)$ and $G_2 = (V_2,E_2,L_{V_2},L_{E_2},\mu_2,\zeta_2)$ if the following conditions are satisfied:

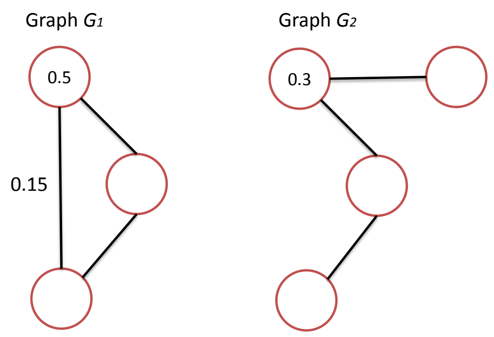
1. $\forall u_i \in V_1,\ \mu_1(u_i) \approx \mu_2(f(u_i))$
2. $\forall (u_i, u_j) \in V_1 \times V_1, e(u_i, u_j) \in E_1 \Leftrightarrow e(f(u_i), f(u_j)) \in E_2$
3. $\forall e(u_i, u_j) \in E_1, \zeta_1(e(u_i, u_j)) \approx \zeta_2(e(f(u_i), f(u_j)))$

In PR applications, where vertices and edges are labeled with measures which may be affected by noise, a substitution-tolerant formulation which allows differences between attributes of mapped vertices and edges is mandatory. However, these differences are associated to costs where the objective is to find the mapping corresponding to the minimal global cost, if one exists. i.e., $\mu_1(u_i) \approx \mu_2(v_k)$ and $\zeta_1(e(u_i, u_j)) \approx \zeta_2(e(v_k, v_z))$.

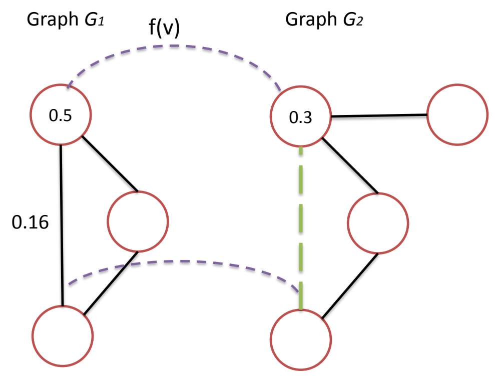
Figure 2.5 depicts the substitution-tolerant subgraph isomorphism problem.

2.1.11.3 Error-Tolerant Subgraph Isomorphism

Error-Tolerant Subgraph Isomporphism [95] takes into account the difference in topology as well as attributes. Thus, it requires that each vertex/edge of graph G_1 is mapped to a distinct vertex/edge of graph G_2 or to a dummy vertex/edge. This dummy elements can absorb structural modifications between the two graphs.



(a) Graphs G_1 and G_2



(b) Error-Tolerant Subgraph Isomorphism of the graphs G_1 and G_2. Note that the green dashed line on G_2 represents an edge deletion.

Figure 2.6: Error-Tolerant Subgraph Isomorphism of G_1 and G_2

Definition 9 *Error-Tolerant Subgraph Isomorphism*
An injective function $f : V_1 \rightarrow V_2$ is an error-tolerant subgraph isomorphism from $G_1 = (V_1,E_1,L_{V_1},L_{E_1},\mu_1,\zeta_1)$ to $G_2 = (V_2,E_2,L_{V_2},L_{E_2},\mu_2,\zeta_2)$ if the following conditions are satisfied:

1. Δ_{V_2} is a set of dummy vertices
2. Δ_{E_2} is a set of dummy edges
3. $\forall u_i \in V_1, f(u_i) \in V_2 \cup \Delta_{V_2}$
4. $\forall e(u_i,u_j) \in E_1, e(f(u_i), f(u_j)) \in E_2 \cup \Delta_{E_2}$
5. $\forall u_i \in V_1, \mu_1(u_i) \approx \mu_2(f(u_i))$ *and* $\forall e(u_i,u_j) \in E_1, \xi_1(e(u_i,u_j)) \approx \xi_2(e(f(u_i), f(u_j)))$

The error-tolerant subgraph isomorphism of graphs G_1 and G_2 is depicted in Figure 2.6.

2.1.11.4 Error-Tolerant Graph Matching

A significant number of error-tolerant GM algorithms base the definition of the matching cost on an explicit model of the errors (deformations) that may occur (i.e., missing vertices, etc.), assigning a possibly different cost to each kind of error. These algorithms are often denoted by error-correcting or error-tolerant [35, 92].

Definition 10 *Error-Tolerant GM*
A function $f : V_1 \cup \{\epsilon\} \rightarrow V_2 \cup \{\epsilon\}$ is an error-tolerant GM from $G_1 = (V_1,E_1,L_{V_1},L_{E_1},\mu_1,\zeta_1)$ to $G_2 = (V_2,E_2,L_{V_2},L_{E_2},\mu_2,\zeta_2)$ where ϵ refers to the empty vertex. Considering only nonempty vertices, f is bijective. However, when taking into account ϵ, several vertices from V_1 can be mapped to ϵ. Such an operation is referred to as deletion of these vertices from V_1. Similarly, ϵ can be mapped to several vertices from V_2 representing the insertion of these vertices in v_1. Formally, f must fulfill certain conditions:

1. $f(u_1) \neq \epsilon \Rightarrow f(u_1) \neq f(u_2 \; \forall u_1, u_2 \in V_1$
2. $f^{-1}(v_1) \neq \epsilon \Rightarrow f^{-1}(v_1) \neq f^{-1}(v_2) \; \forall v_1, v_2 \in V_2$

Figure 2.7 depicts the error-tolerant graph isomorphism problem.

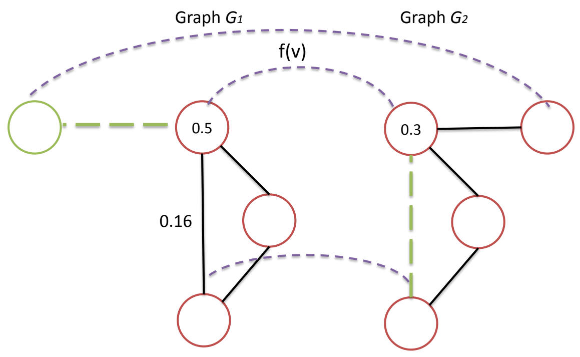


Figure 2.7: Error-Tolerant Graph Isomorphism of the graphs G_1 and G_2 presented in Figure 2.6(a). Note that the dashed vertex and the dashed line on G_1 represent vertex insertion and edge insertion operations respectively. The Dashed line on G_2 depicts an edge insertion operation.

In the thesis, the term *source* graph refers to graph G_1 while *target* graph refers to G_2.

2.1.11.5 Error-Tolerant Matching Cost

As mentioned before, error-tolerant GM has an advantage over exact GM which lies in error and noise tolerance in the matching process. In exact GM, when comparing two vertices or two edges, the evaluation answer is yes or no. That is, the matching result tells whether the vertices or edges are equal. In error-tolerant GM, a measurement of the strength of matching vertices and/or edges is called *cost*. This cost is applicable on both graph structures and attributes. The basic idea is to assign a penalty cost to each edit operation according to the amount of distortion that it introduces in the transformation. When (sub)graphs differ in their attributes or structures, a high cost is added in the matching process. Such a cost prevents dissimilar (sub)graphs from being matching since they are different. Likewise, when (sub)graphs are similar, a small cost is added to the overall cost. This cost includes matching two vertices and/or edges, inserting a vertex/edge or deleting a vertex/edge. Deletion and insertion operations are transformed to assignments of a non-dummy vertex to a dummy vertex one. Substitutions simply indicate to vertex-to-vertex and edge-to-edge assignments.

Formally, error-tolerant GM $f : V_1 \rightarrow V_2$ is a minimization problem where the goal is to minimize the overall cost c of matching G_1 and G_2.

Definition 11 *Matching Cost Function*

$$c(f) = \overbrace{\sum_{\substack{u_i \in V_1 \\ f(u_i) \in V_2}} c(u_i, f(u_i))}^{\text{vertex substitutions}} + \overbrace{\sum_{\substack{u_i \in V_1 \\ f(u_i) = \epsilon}} c(u_i, \epsilon)}^{\text{vertex deletions}} + \overbrace{\sum_{\substack{u_j \in V_2 \\ f^{-1}(u_j) = \epsilon}} c(\epsilon, u_j)}^{\text{vertex insertions}} +$$

$$\overbrace{\sum_{\substack{e(u_i,u_j) \in E_1 \\ e(f(u_i),f(u_j)) \in E_2}} c(e(u_i,u_j), e(f(u_i), f(u_j)))}^{\text{edge substitutions}} + \overbrace{\sum_{\substack{e(u_i,u_j) \in E_1 \\ e(f(u_i),f(u_j)) = \epsilon}} c(e(u_i,u_j), \epsilon)}^{\text{edge deletions}} +$$

$$\overbrace{\sum_{\substack{e(v_k,v_z) \in E_2 \\ e(f^{-1}(v_k),f^{-1}(v_z)) = \epsilon}} c(\epsilon, e(v_k, v_z))}^{\text{edge insertions}} \tag{2.1}$$

2.1.11.6 Graph Edit Distance

The graph edit distance (GED) was first reported in [137, 118, 60]. GED is a dissimilarity measure for graphs that represents the minimum-cost sequence of basic editing operations to transform a graph into another graph by means classically included operations: insertion, deletion and substitution of vertices and/or edges. Therefore, GED can be formally represented by the minimum cost edit path transforming one graph into another. Edge operations are taken into account in the matching process when substituting, deleting or inserting their adjacent vertices. From now on and for simplicity, we denote the substitution of two vertices u_i and v_k by $(u_i \rightarrow v_k)$, the deletion of vertex u_i by $(u_i \rightarrow \epsilon)$ and the insertion of vertex v_k by $(\epsilon \rightarrow v_k)$. Likewise for edges $e(u_i, u_j)$ and $e(v_k, v_z)$, $(e(u_i, u_j) \rightarrow e(v_k, v_z))$ denotes edges substitution, $(e(u_i, u_j) \rightarrow \epsilon)$ and $(\epsilon \rightarrow e(v_k, v_z))$ denote edges deletion and insertion, respectively. The structures of the considered graphs do not have to be preserved in any case. Structure violations are also subject to a cost which is usually dependent on the magnitude of the structure violation [111]. And so, the meta parameters of each of deletion, insertion and substitution affect the matching process. The discussion around the selection of cost functions and their parameters is beyond the topic of this thesis and will not be discussed in this thesis.

Let $\gamma(G_1, G_2)$ denote the set of edit paths that transform G_1 into G_2. To select the most promising edit path among all the edit paths of $\gamma(G_1, G_2)$, a cost, denoted by $c(ed)$, is introduced, see Definition 11. Thus, for each operation (edge/vertex substitutions, edge/vertex deletions and edge/vertex insertions) a penalty cost is added. GED tries to find the minimum overall cost ($d_{\lambda_{min}}(G_1, G_2)$) among all generated costs.

Formally saying, GED is based on a set of edit operations ed_i where $i = 1 \dots k$ and k is the number of edit operations. This set is referred to as *Edit Path*.

Definition 12 *Edit Path*
A set $\{ed_1, \cdots, ed_k\}$ of k edit operations ed_i that transform G_1 completely into G_2 is called a (complete) edit path $\lambda(G_1, G_2)$ between G_1 and G_2. A partial edit path refers to a subset of $\{ed_1, \cdots, ed_q\}$ that partially transforms G_1 into G_2.

Formally saying, the edit distance of two graphs is defined as follows.

Definition 13 *(Graph Edit Distance)*
Let $G_1 = (V_1,E_1,L_{V_1},L_{E_1},\mu_1,\zeta_1)$ and $G_2 = (V_2,E_2,L_{V_2},L_{E_2},\mu_2,\zeta_2)$ be two graphs, the graph edit distance between G_1 and G_2 is defined as:

$$d_{\lambda_{min}}(G_1, G_2) = \min_{\lambda \in \gamma(G_1,G_2)} \sum_{ed_i \in \lambda} c(ed_i) \tag{2.2}$$

Where $c(ed_i)$ denotes the cost function measuring the strength of an edit operation ed_i and $\gamma(G_1, G_2)$ denotes the set of all edit paths transforming G_1 into G_2. The exact correspondence, λ_{min}, is one of the correspondences that obtains the minimum cost (i.e., $d_{\lambda_{min}}(G_1, G_2)$).

Generally speaking, Definition 13 is constrained by vertices and so vertices of the involved graphs are privileged during the matching process. That is, edge operations are taken into account in the matching process when substituting, deleting or inserting their underlying or corresponding vertices.

In GED and so error-tolerant GM, each vertex of G_1 can be either matched with a vertex in G_2 or deleted (in this case it will be matched with ϵ). Similarly, each vertex of G_2 can be either matched with a vertex in G_1 or inserted in G_1 (in this case it will be matched with ϵ). Likewise, edges of G_1 can be either matched with edges of G_2 or deleted while edges of G_2 can be either inserted in G_1 or matched with edges in G_1. However, the decision of whether an edge is inserted, substituted, or deleted is done regarding the matching of their adjacent vertices. That is, the neighborhood of edges dominates their matching. For better understanding, see Figure 2.8. Note that in the given scenarios, u_i and u_j of G_1 are matched with v_k and v_z of G_2, respectively. Formally, $f(u_i) = v_k$ and $f(u_j) = v_z$. Based on these scenarios, three cases can be identified:

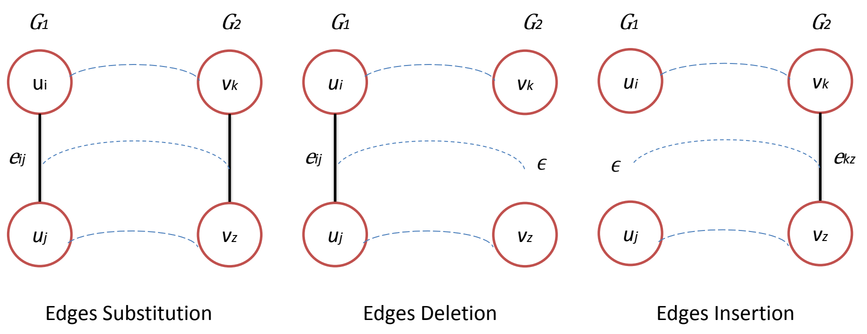


Figure 2.8: Edge mappings based on their adjacent vertices and whether or not an edge between two vertices can be found

- If there is an edge $e_{ij} = e(u_i, u_j) \in E_1$ and an edge $e_{kz} = e(v_k, v_z) \in E_2$, edges substitution between $e(u_i, u_j)$ and $e(v_k, v_z)$ is performed (i.e., $e(f(u_j), f(u_j)) = e(v_k, v_z)$).
- If there is an edge $e_{ij} = e(u_i, u_j) \in E_1$ and there is no edge between v_k and v_z (i.e., $e(v_u, v_k) = \epsilon$), edge deletion of $e(u_i, u_j)$ is performed (i.e., $e(f(u_j), f(u_j)) = \epsilon$).
- If there is no edge between u_i and u_j (i.e., $e_{ij} = e(u_i, u_j) = \epsilon$) and there is an edge between and an edge $e_{kz} = e(v_k, v_z) \in E_2$, edge insertion of $e(v_k, v_z)$ is performed (i.e., $e(f^{-1}(v_k), f^{-1}(v_z)) = \epsilon$).

An example of an edit path between two graphs G_1 and G_2 is shown in Figure 2.9, the following operations have been applied in order to transform G_1 into G_2: three edge deletions, one vertex deletion, one vertex insertion, one edge insertion and three vertex substitutions.

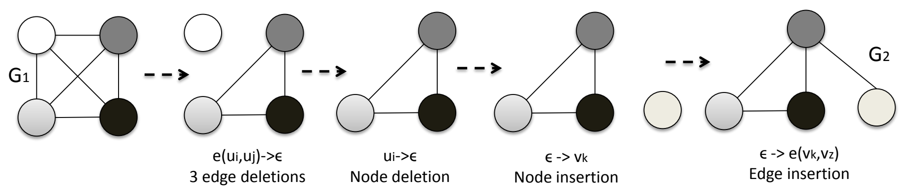


Figure 2.9: Transforming G_1 into G_2 by means of edit operations. Note that vertices attributes are represented in different gray scales

A cost function is associated with each edit operation indicating the change strength an edit operation had done. In fact, GED directly corresponds to the definition of error-tolerant GM, see Definition 10. Thus, GED has also been shown to be NP-hard [155].

Conditions on Cost Functions If no conditions are put on the cost functions for deleting, inserting or substituting vertices and/or edges, then one can have infinite number of complete edit paths λ. For example, one can insert any ed_i (*e.g.* $\epsilon \rightarrow u_i$) and then remove it (i.e., $u_i \rightarrow \epsilon$) and thus by doing so with the other edit operations one can end up having infinite number of solutions for $GED(G_1, G_2)$. In order to overcome this problem, some constraints have to be defined for any cost function proposed for solving GED. In [98], three constraints are illustrated. The first constraint, referred to as positivity, is defined as follows:

$$c(ed_i) \geq 0 \text{ s.t. } ed_i \text{ is an edit operation on vertices or edges.} \tag{2.3}$$

By adding this condition, any cost function has to be non-negative.

In order not to allow inserting a vertex or edge and then deleting it, a second condition or constraint is defined. This condition only allows substitutions to have a zero cost. That is, the cost of deletions and insertions have to be always greater than zero. Formally:

$$c(ed_i) > 0 \text{ s.t. } ed_i \text{ can be an insertion or a deletion of a vertex or an edge.} \tag{2.4}$$

However, when the attributes of two edges or two vertices that are matched differ, a distance between attributes should be defined. Such a distance depends on the graph database. Later in the thesis, some graph databases along with their cost functions will be presented.

From the aforementioned constraint, one can see that substitutions are always privileged. In order to prevent some expensive substitutions, deletions or insertions from being included in the edit path, a third constraint, referred to as triangle inequality, is initialized:

$$\begin{aligned} c(u_i \rightarrow v_k) &\leq c(u_i \rightarrow v_z) + c(v_z \rightarrow v_k) \\ c(u_i \rightarrow \epsilon) &\leq c(u_i \rightarrow v_z) + c(v_z \rightarrow \epsilon) \\ c(\epsilon \rightarrow v_k) &\leq c(\epsilon \rightarrow v_z) + c(v_z \rightarrow v_k) \end{aligned} \tag{2.5}$$

Where u_i, v_k and v_z are vertices that are included in an edit path. For example, a deletion $(\epsilon \rightarrow v_k)$ is performed if it is less expensive or equal to adding a vertex $(\epsilon \rightarrow v_z)$ followed by $(v_z \rightarrow v_k)$ (see line 3 of the third constraint). While this constraint only talks about vertex operations, it has to be applied on not only vertices but also edges.

It has been shown by Neuhaus and Bunke [98] that for GED to be a metric, each of its elementary operations has to satisfy not only the aforementioned properties but also one more property, referred to as Symmetry. The Symmetry constraint is defined as follows:

$$c(ed_i) = c(ed_i^{-1}) \tag{2.6}$$

The property includes vertices and edges' operation ed_i. For instance, $(u_i \rightarrow v_k)$ has to be equal to $(v_k \rightarrow u_i)$. Likewise, deleting a vertex $(u_i \rightarrow \epsilon)$ is equal to inserting it (i.e., $\epsilon \rightarrow u_i$).

2.1.11.7 Multivalent Matching

All the aforementioned matching problems, whether exact or error-tolerant ones, belong to the univalent family in the sense of allowing one vertex or one edge of one graph to be substituted with one and only one vertex or edge in the other graph.

In many real-world applications, comparing patterns described at different granularity levels is of great interest. For instance, in the field of image analysis, an over-segmentation of some images might occur whereas an under-estimation occurs in some other images resulting in allowing several regions of one image to be correspondent, or related to, a single region of another image. Based on this fact, multivalent matching problem emerged to be one of the interesting problems in graph theory [29]. Multivalent matching drops the condition that vertices in the source graph are to be mapped to distinct vertices of the target graph. Thus, in multivalent matching, vertex in the first graph can be matched with an empty set of vertices, one vertex or even multiple vertices in the other graph. This matching problem is also called relational matching since GM is no longer a function but rather a relation $m \subseteq V_1 \times V_2$. The objective of this kind of matching is to minimize the number of split vertices (i.e., vertices that are matched with more than one vertex).

Mathematically, the relation m associating a vertex of one graph to a set of vertices of the other graph can be defined as follows:

Definition 14 *Multivalent Matching*
A relation $m \subseteq V_1 \times V_2$ is a multivalent matching from G_1 to G_2 if:

1. $\forall u_i \in V_1,\ m(u_i) \approx \{v_k \in V_2 | (u_i, v_k) \in m\}$
2. $\forall v_k \in V_2,\ m(v_k) \approx \{u_i \in V_1 | (u_i, v_k) \in m\}$

where $m(v_*)$ denotes the set of vertices that are associated with a vertex v_* by the relation m.

Figure 2.10 illustrates an example of two objects (object 1 and object 2). At a first glance, one may think that both objects are similar, however, while there is only one front wall in object 2 (i.e., wall 5), there are two front walls (e and f) in object 2. And thus when both objects are transformed into relational graphs, wall 5 is matched to both walls f and e in graph G_1. Based on the aforementioned definition of multivalent, m of the previous example is defined as: $\{(a, 1), (b, 2), (c, 3), (d, 4), (e, 5), (f, 5)\}$. Thus, in this mapping, the set of vertices mapped with 5 in G_2 is referred to as $m(5) = \{e, f\}$. In another scenario (m= $\{(a, 1), (b, 2), (a, 3), (b, 4), (e, 5)\}$), one can remark that f in G_1 is not mapped to any vertex and thus $m(f) = \emptyset$. Since each vertex can be matched to zero, one or many vertices, the complexity of multivalent matching dramatically increases when compared to the aforementioned GM problems in this chapter.

2.1.11.8 Modeling Graph Matching by Hard and Soft Constraints

In [81], GM is defined by constraint-based modeling language. By using such constraints, any GM problem can be expressed. The constraint-based language, or so-called synthesizer, is designed on top of Comet [140]. Once a user defines the characteristics of the selected problem, a Comet program is automatically created. This program has two modes: Constraint Programming (CP) and Constraint-Based Local Search (CBLS). One of these modes is automatically used depending on the problem's characteristics. For

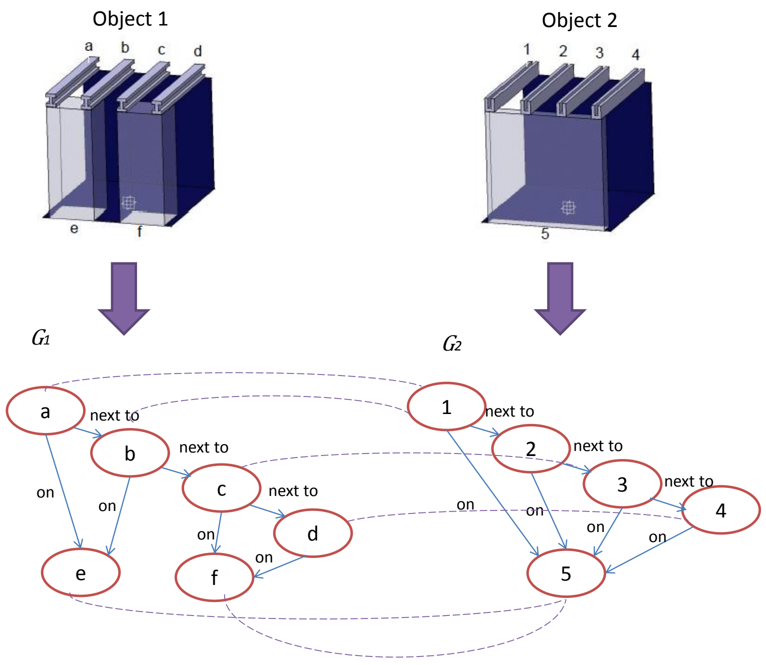


Figure 2.10: Multivalent matching example (taken from [29])

instance, CP is used for computing exact measures while CBLS is suited for computing error-tolerant measures.

Let $G_1 = (V_1,E_1,L_{V_1},L_{E_1},\mu_1,\zeta_1)$ and $G_2 = (V_2,E_2,L_{V_2},L_{E_2},\mu_2,\zeta_2)$ be two graphs. To match these graphs, a list of constraints can be used to specify the considered matching problem and thus GM is turned into satisfying these selected constraints. The main constraints family is divided into 4 sets. The first set allows to specify the minimum and maximum number of vertices a vertex is matched to. The second set ensures that a set U of vertices is injective. The third set permits to identify clearly that a couple of vertices must be matched to a couple of vertices connected by an edge. The fourth set ensures that the labels of matched vertices or edges must be equal. Each constraint can be either a hard or a soft one. Hard constraints cannot be violated while soft ones may be violated at some given cost. Thus, for each soft constraint, a violation cost is needed such that the similarity is maximized.

Figure 2.11 illustrates two graphs G_1 and G_2. One can model the problem of whether one of the two graphs is included into the other using hard and soft constraints. For instance, if all the four constraints are hard, the problem is turned to be Induced Subgraph Isomorphism, see Section 2.1.10.2. On the other hand, if the constraints are a mixture of hard and soft constraints, the problem becomes graph Partial Subgraph Isomorphism, see Section 2.1.10.3.

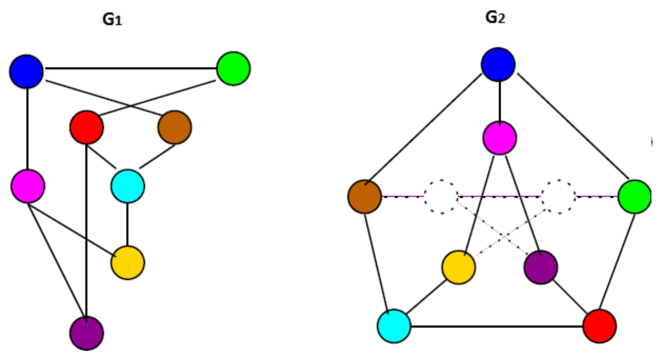


Figure 2.11: An example of two graphs G_1 and G_2. The objective is to decide whether one graph is included into the other. If constraints on edges are hard, the problem is Induced Subgraph Isomorphism. If constraints are both hard and soft ones, the problem is graph Partial Subgraph Isomorphism. (Taken from http://contraintes.inria.fr/∼fages/SEMINAIRE/Solnon09.pdf)

2.1.11.9 Graph Edit Distance as Quadratic Assignment Problem

GED can be reformulated as a quadratic assignment problems (QAPs) [14]. QAPs belong to the class of NP-hard problems. Over last three decades, extensive research has been done on QAPs. In [116], Sahni and Gonzalez have shown that the QAP is NP-hard and that even finding a suboptimal solution within some constant factor from the optimal solution cannot be done in polynomial time unless P=NP.

As a well-known quadratic assignment application, we mention the flow matrix. The objective is to find an assignment of all facilities to all locations (i.e., a permutation $p \in \Pi_N$), such that the total cost of the assignment is minimized. Given a set $N = \{1, 2, \cdots, n\}$ and $n \times n$ matrices $F = (f_{ij})$ and $D = (d_{p(i)p(j)})$, the quadratic assignment problem (QAP) can then be defined as follows:

$$\min_{p \in \Pi_N} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{p(i)p(j)} + \sum_{i=1}^{n} c_{ip(i)} \tag{2.7}$$

where $F = (f_{ij})$ is the flow of materials from facility i to facility j whereas $D = (d_{p(i)p(j)})$ is the matrix whose elements d_{kl} represent the distance from location $p(i)$ to location $p(j)$. The cost of simultaneously assigning facility i to location $p(i)$ and facility j to location $p(k)$ is $f_{ij}d_{p(i)p(j)}$. Finally, $c_{ip(i)}$ is the cost of placing facility i at location $p(i)$. For a comprehensive survey of QAPs, we refer the interested reader to [24].

In order to reformulate GED as QAP, two challenging points have been considered. First, having equal cardinality matrices taking into account the unequal cardinality of vertices and edges in the involved graphs of GED. Second, GED is more general than QAP since it does not necessarily assign each vertex or edge in G_1 to a vertex or an edge in G_2. That is, GED also allows the deletion of vertices and edges of G_1 as well as the insertion of vertices and edges of G_2.

These issues have been solved by adding empty vertices and so edges in the list of vertices, as stated here:

$$V_1^{\Delta} = V_1 \cup \{\epsilon_1, \epsilon_2, \cdots \epsilon_m\}$$
$$V_2^{\Delta} = V_2 \cup \{\epsilon_1, \epsilon_2, \cdots \epsilon_n\}$$

where $n = |V_1|$ and $m = |V_2|$. Therefore, the adjacency matrices of G_1 and G_2 (i.e., A and B respectively) are defined as follows:

$$A_{(n+m)\times(n+m)} = \left|\begin{array}{cccc|cccc}
a_{11} & ... & ... & a_{1n} & \epsilon & \epsilon & ... & \epsilon \\
... & ... & ... & ... & \epsilon & \epsilon & ... & \epsilon \\
... & ... & ... & ... & ... & ... & ... & ... \\
a_{n1} & ... & ... & a_{nn} & \epsilon & ... & \epsilon & \epsilon \\
\hline
\epsilon & \epsilon & ... & \epsilon & 0 & ... & ... & 0 \\
\epsilon & \epsilon & ... & \epsilon & ... & ... & ... & ... \\
... & ... & ... & ... & ... & ... & ... & ... \\
\epsilon & ... & \epsilon & \epsilon & 0 & ... & ... & 0
\end{array}\right|$$

$$B_{(n+m)\times(n+m)} = \left|\begin{array}{cccc|cccc}
b_{11} & ... & ... & b_{1m} & \epsilon & \epsilon & ... & \epsilon \\
... & ... & ... & ... & \epsilon & \epsilon & ... & \epsilon \\
... & ... & ... & ... & ... & ... & ... & ... \\
b_{m1} & ... & ... & b_{mm} & \epsilon & ... & \epsilon & \epsilon \\
\hline
\epsilon & \epsilon & ... & \epsilon & 0 & ... & ... & 0 \\
\epsilon & \epsilon & ... & \epsilon & ... & ... & ... & ... \\
... & ... & ... & ... & ... & ... & ... & ... \\
\epsilon & ... & \epsilon & \epsilon & 0 & ... & ... & 0
\end{array}\right|$$

The elements of the matrices A and B indicate whether an edge can be found between vertices. For instance, if there is an edge between u_i and u_j in G_1, $a_{ij} \in A$ will refer to that edge. Note that in A and B there is no edge between any vertex u_i and an empty vertex, the contrary is also true. That is, ϵ is found for the impossible cases.

Based on V_1^{Δ} and V_2^{Δ}, the cost matrix C can be established as follows:

The left upper corner of the matrix contains all possible vertex substitutions (i.e., $u_i \to u_j$), the diagonal of the right upper matrix represents the cost of all possible vertex deletions (i.e., $u_i \to \epsilon$) and the diagonal of the bottom left corner contains all possible vertex insertions (i.e., $\epsilon \to u_j$). The bottom right corner elements cost is set to zero which concerns the substitution of $\epsilon \to \epsilon$.

Now that all elements are ready (i.e., the adjacency matrices A and B and the cost matrix C), equation 2.7 can be rewritten as follows:

$$d_{min} = \min_{(\varphi_1,\varphi_2,\cdots\varphi_{n+m})\in\Pi_{n+m}} \sum_{i=1}^{n+m} c_{i\varphi(i)} + \sum_{i=1}^{n+m}\sum_{j=1}^{n+m} c(a_{ij} \to b_{\varphi_i\varphi_j}) \tag{2.8}$$

$$C_{(n+m)\times(n+m)} = \left|\begin{array}{cccc||cccc}
c_{1,1} & \cdots & \cdots & c_{1,m} & c_{1,\epsilon} & \infty & \cdots & \infty \\
\cdots & \cdots & \cdots & \cdots & \infty & c_{2,\epsilon} & \cdots & \infty \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
c_{n,1} & \cdots & \cdots & c_{n,m} & \infty & \cdots & \infty & c_{n,\epsilon} \\
\hline
c_{\epsilon,1} & \infty & \cdots & \infty & 0 & \cdots & \cdots & 0 \\
\infty & c_{\epsilon,2} & \cdots & \infty & \cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\infty & \cdots & \infty & c_{\epsilon,m} & 0 & \cdots & \cdots & 0
\end{array}\right|$$

where Π_{n+m} refers to the set of all $(n+m)!$ possible permutations of the integers $1, 2, \cdots, (n+m)$. The first linear term $\sum_{i=1}^{n+m} c_{i\varphi(i)}$ is dedicated to the sum of vertex edit costs while the second quadratic term $\sum_{i=1}^{n+m}\sum_{j=1}^{n+m} c(a_{\varphi_i\varphi_j} \rightarrow b_{\varphi_i\varphi_j})$ refers to the underling edge cost resulted from the permutation $(\varphi_1, \varphi_2, \cdots \varphi_{n+m})$. For instance, if vertex $u_i \in V_1^{\Delta}$ is matched with vertex $v_k \in V_2^{\Delta}$ and vertex $u_j \in V_1^{\Delta}$ is matched with vertex $v_z \in V_2^{\Delta}$, then edge $e(u_i, u_j)$ has to be matched with edge $e(v_k, v_z)$. These edges are kept in a_{ij} and b_{kz}, respectively. As previously mentioned, edges might be empty.

2.1.12 Matching Difficulties

In this section a revision of all the aforementioned problems, whether univalent or multivalent, is conducted. Figure 2.12 summarizes all the discussed problems and highlights two properties:

- Difficulty property: one can see that the difficulty increases when looking at Figure 2.12 from top to bottom. That is, the difficulty of multivalent matching is the highest while the one of exact GM is the lowest.
- Constraint property: Unlike exact and error-tolerant matching which belong to the univalent class, multivalent matching has the least constraints since it allows the matching of one to none, one to one, one to many and many to many.

2.2 Synthesis of Error-Tolerant Graph Matching Methods

2.2.1 Motivation

Restricting applications to exact GM is obviously not recommended. In reality, objects suffer from the presence of both noise and distortions, due to the graph extraction process. Thus, exact GM algorithms fail to answer whether two graphs G_1 and G_2 are, not identical, but similar. In addition, when describing non-discrete properties of an object, graph vertices and edges are attributed using continuous attributes (i.e., $L \subseteq \mathbb{R}$). Such objects (i.e., with non-discrete labels) are likely to be nonidentical. In this thesis, as a first step and since the complexity of multivalent error-tolerant matching is even harder than univalent error-tolerant matching, multivalent matching is not tackled.

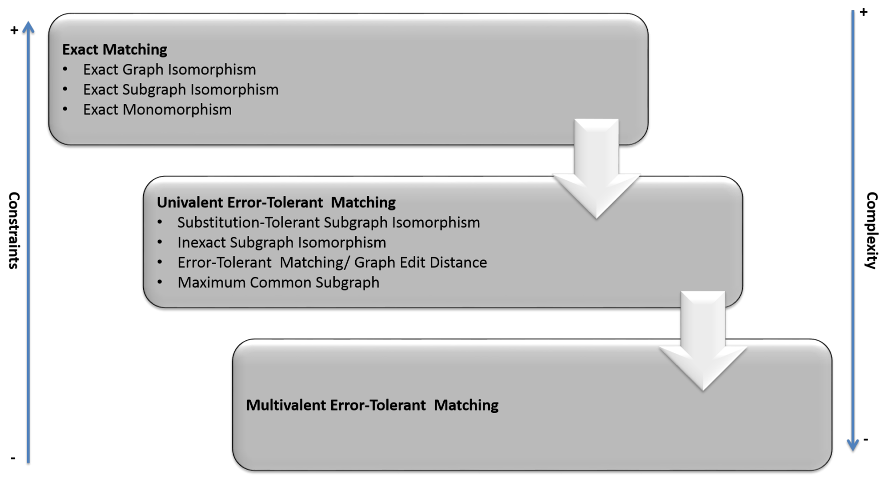

+
Constraints
-
Exact Matching
• Exact Graph Isomorphism
• Exact Subgraph Isomorphism
• Exact Monomorphism
Univalent Error-Tolerant Matching
• Substitution-Tolerant Subgraph Isomorphism
• Inexact Subgraph Isomorphism
• Error-Tolerant Matching/ Graph Edit Distance
• Maximum Common Subgraph
Multivalent Error-Tolerant Matching
+
Complexity
-

Figure 2.12: Graph matching difficulties according to constraints

Consequently and for all the aforementioned arguments, we focus on univalent error-tolerant GM taking into account the applicability of its proposed methods in various real-world applications. Error-tolerant GM, aims at relaxing, to some extent, the constraints of the extract matching process in such a way that a similarity answer/score is given for matching an attributed source graph with an attributed target graph while penalizing the structure of one or both them. Error-tolerant GM techniques have been widely proposed in the literature. In this section, we survey the methods presented in the literature to solve error-tolerant GM. Since we cannot review all the methods, the list of methods is considered as a non-exhaustive one. We refer the interested reader to two surveys that focused on applications using graphs at different levels [92, 36].

2.2.2 Error-Tolerant Methods in the Literature

In our synthesis, we focus on error-tolerant GM methods that are learning-free (i.e., methods that are not based on a machine learning step). The reason for which we have not focused on such methods is because there are few graph databases with ground truths. Moreover, ground truths constructed by humans cannot be always achieved for some specific structures such as chemical structures. Thus, methods that are based on neural networks (*e.g.* [54, 131, 77]) and Expectation-Maximization (EM) (*e.g.* [41, 7, 89]) are not detailed in the synthesis.

We divide the methods in the literature into two big families: deterministic and non-deterministic methods.

2.2.2.1 Deterministic Methods

Formalization by means of relaxation labeling is another type of GM formalization that has been proposed in the literature. The very first work has been proposed in [50]. Labels of target graphs are presented as a discrete set, each vertex of the source graph is assigned to one label of the target graph set. The selection of a label candidate is done using Gaussian Probability Distribution. The vertex-matching procedure is iteratively conducted. In the beginning, an initial labeling is selected which is dependent on the vertex attributes, vertex connectivity, etc. Afterwards, the labeling is enhanced until a satisfying labeling is found. Such a satisfying labeling is considered as the best matching of two graphs.

Some enhancements have been proposed in the relaxation labeling domain. [61, 74] are the first works applying probability theory to GM where an iterative approach is conducted using a method called probabilistic relaxation. In these works, both a Gaussian error and the use of binary relations are justified to be enough for fully defining the whole structure. The main drawback of the initial formulation of this technique, namely the fact that vertex and edge labels are used only in the initialization of the matching process. Such a drawback was overcome in [142]. A Bayesian perspective is used for both unary and binary vertex attributes in [142]; [59]; and [147]. In [65] this method is further improved by also taking edge labels into account in the evaluation of the consistency measure for matching hierarchical relational models. Bayesian graph [96] has been built up on the idea of probabilistic relaxation. The concept of Bayesian graph has also been successfully applied to trees [136].

Spectral Graph Theory is considered as an approach of great interests for solving GM problems [90, 34, 88, 115]. In this approach, graphs' topology is characterized using eigenvalues and eigenvectors of the adjacency matrix or Laplacian matrix ([148]). The computation of eigenvalues/eigenvectors is a well studied problem that can be solved in polynomial time. However, despite the benefits achieved using all algorithmic tools, spectral methods suffer from two main limitations. First, the weakness in their representation of vertex and/or edge attributes in PR applications, some spectral methods can deal only with unlabeled or labeled graphs under constraints (*e.g.* only constrained label alphabets or real weights assigned to edges [138].). Second, the sensitivity of eigen-decomposition towards structural errors as they can not cope with graphs affected by noise, such as missing or extra vertices [22].

In the literature, researchers also focused on GED and proposed lots of approaches for solving it. In [23] a distance measure based on the maximal common subgraph of two graphs is proposed. This defined distance is a metric that does not depend neither on the edit distance nor on the costs of the edit operations. The well-known A^* algorithm [102] has been employed for solving GED. A^* along with a lower bound can prune off the underlying search tree by decreasing the number of explored nodes and edges. However, because of the combinatorial explosion of the search tree of GED, the problem is still known to be NP-hard. A linear programming formulation of GED has been reported in [71], such a formulation is applicable only on graphs whose edges are unattributed.

An algorithm in [82], referred to by the Integer Projected Fixed Point Method (IPFP), is proposed for GM. IPFP is based on QAP of GM, see Section 2.1.11.9. This algorithm is an efficient approach, which iterates between maximizing linearly approximated objective in discrete domain and maximizing along the gradient direction in continuous domain. Thus, IPFP guarantees convergence properties. However, in general, this approach often stops early with bad local maximum. In fact, since IPFP is a greedy approach which is based on discretization, during its execution it can find a bad local maximum.

Apart from the aforementioned methods in the section, there are also approximate algorithms that work directly on the adjacency matrix of the graphs relaxing the combinatorial optimization to a continuous one like the Path Following algorithm proposed in [153]. Path Following is based on convex-concave relaxations of the initial integer programming problem. The reason of choosing convex/concave relaxations is to approximate in the best way the objective function on the set of permutation matrices. In order to achieve the convex-concave programming formulation, the weighted GM problem is also reformulated as QAP over the set of permutation matrices where the quadratic term encodes the structural compatibility and the linear term encodes local compatibilities, see Section 2.1.11.9. Then this problem is relaxed to two different optimization problems: a quadratic convex and a quadratic concave optimization. Path Following allows to integrate the alignment of graph structural elements with the matching of vertices with similar attributes. Finally, a Graduated NonConvexity and Graduated Concavity Procedure (GNCGCP) [86] has been proposed as a general optimization framework to suboptimally solve the combinatorial optimization problems. One of the special cases in the paper is solving error-tolerant GM. This proposal has two steps. First, graduated nonconvexity which realizes a convex relaxation. Second, graduated concavity which realizes a concave relaxation.

2.2.2.2 Non-Deterministic Methods

The formulation of complex GM problems as combinatorial optimization has been proposed in the literature. Genetic algorithms are considered as examples of such a formulation. Matching is formalized as states (chromosomes) of a search space with a corresponding fitness in [3, 145, 127, 132, 10]. Genetic algorithms start with an initial pool of chromosomes, considered as matching, which evolves iteratively into other generations of matching. Despite the randomness of genetic algorithms in exploring the search space, promising chromosomes can be chosen if one carefully designs genetic algorithms. These chromosomes can then be improved during specific genetic operations. In fact, low cost matching is prioritized which guarantees to have, though not optimal, low cost matching. Furthermore, genetic algorithms are able to efficiently overcome the problem of both huge search spaces and the local minima proposed for approximating GED. However, genetic algorithms are non-deterministic in the sense of having different solutions when running its algorithms several times.

Some generic algorithms have been proposed for solving a variety of GM problems. A greedy algorithm has been proposed in [29]. In the beginning, an empty mapping m is given. Then in order to fill up this set with vertex-to-vertex and edge-to-edge mappings, a greedy way is used to choose mappings that maximize the similarity. To choose the best vertex-to-vertex mappings, vertices that share the same in and out-edges are always privileged. The step of finding the best candidates is repeated until finding a local optimum solution. This algorithm is non-deterministic and thus one need to run it several times and choose the best solution after. In order to improve this algorithm, local search algorithms can be employed [57, 73]. Local search algorithms aim at improving solutions by locally exploring their neighborhoods. In GM, neighborhoods can be obtained by adding or removing vertex-to-vertex mappings in m. To choose the best neighbor to be explored, Tabu search is used [57, 64]. Tabu search prevents backward moves by memorizing the last k moves, such a step overcomes the problem of local optimum from which greedy search algorithms suffer. The authors of [130], inspired by [13], have proposed a reactive tabu search as a generic GM algorithm where k is dynamically adapted. A hashing key is given for each explored path. When the same mapping is explored twice (i.e. when a collision happens in the hash table), the search must be diversified. However, when no collision happens for a certain number of iterations, that is an indicator of the diversity of the mappings and thus the size of k can be decreased. Finally, an Iterated reactive tabu search is proposed in [117] where k executions of reactive search, each of which has $maxMoves/k$ allowed moves, are launched. In the end, the best matching found during the k executions is kept.

2.2.3 Synthesis

In Table 2.1, we synthesize the error-tolerant GM methods, presented in the literature, taking into account the following criteria:

- Optimality: Whether an algorithm is able to find a global minimum solution (i.e., the best solution among all the existing solutions or so-called optimal solution) or a local minimum one (i.e., not necessarily an optimal solution but a suboptimal one).
- Maximum graphs size: What is the number of vertices on which an algorithm was tested and on which it can perfectly work.
- Graphs type: Symbolic, numeric, weighted, cyclic/acyclic graphs or a mixture of them.
- Parallelism ability: The ability of an algorithm to be run on several machines.
- Popularity: the importance that an algorithm has taken in the literature.

As depicted in Table 2.1, one can remark that exact methods cannot match graphs whose sizes are more than 15 vertices while approximate methods can cope with graphs of larger sizes (i.e., up to 250 vertices in the literature). This is due to the fact that exact methods are computationally expensive. Indeed, exact methods are CPU and Memory consuming. Not only number of vertices can make a problem hard to solve but also graphs density and attributes. For example, matching non-attributed graphs is more difficult than matching attributed ones since attributes can help in quickly finding the optimal or a good near-optimal solution.

2.2.3.1 Large graphs

Based on the graph sizes reported in Table 2.1, we define the term ”large graphs” by dividing it into two categories: exact and approximate large graphs, as shown in Table 2.2. These categories are used as a definition of large graphs in the rest of the thesis. The maximum size of PR graphs is taken from the largest database dedicated to PR graphs, to the best of our knowledge, it is the webpage database [106].

Approximate methods can be grouped into two families:

Name	**Optimality**	**Maximum Graphs Size**	**Graphs type**	**Parallelism Ability**	**Popularity**	**References**
String based methods	Suboptimal	thousands of vertices $\rightarrow$ Hundreds of Regions	Cyclic/acyclic symbolic graphs	++	+	[66, 8]
Spectral Theory(1)	Optimal	Up to 10	Weighted graphs "same size"	++	++	[138]
Spectral Theory(2)	Suboptimal	Not precised	Directed Acyclic Graphs (DAG)	++	++	[46, 126]
Vertex Assignment	Suboptimal	Up to 128 vertices	Directed attributed graphs	+	++	[105, 69, 107, 58]
Genetic algorithm	Suboptimal	Up to 100 vertices	Various types depending on the application	++	-	[3, 10, 53, 67]
Tabu search based algorithms	Suboptimal	Up to 250 vertices	Various types depending on the application	++	+	[117, 130]
Probabilistic Relaxation	Suboptimal	100 vertices	Various types of graphs	–	?	[142, 47],
GED-A^*	Optimal	Up to 10	Various types depending on the application	+	++	[23]
GED-ILP	Optimal	up to 10	Graphs with unattributed edges	+	++	[71]
Approx GED	Suboptimal	Up to hundreds of vertices	Various types depending on the application	++	++	[91, 107]
IPFP	Suboptimal	Up to 50	Weighted graphs	++	+	[82]
Path Following	Suboptimal	Up to 100	Weighted graphs	++	+	[153]
GNCGCP	Suboptimal	up to 50	Unlabeled graphs	++	+	[86]

Table 2.1: Comparison between error-tolerant GM methods in terms of their optimality, maximum graphs size, graphs type, parallelism ability and popularity.

Category	largest size of PR graphs	large graphs
Exact Methods	843 vertices	larger than 15 vertices
Approximate Methods		larger than 250 vertices

Table 2.2: Defining the term "large graphs" in both exact and approximate GM methods.

1. Reformulation of the problem: The GM problem is truncated into a simpler problem (*e.g.* reducing the GM problem to an assignment problem [107]). However, solving an approximated or a constraints-relaxed formulation does not lead to an optimal solution of the original problem.

2. Approximate optimization: Solving the GM problem can be done by approximate algorithms that reduce the size of the search space and thus lead to find near-optimal or so-called approximate solutions (*e.g.* genetic algorithms [3] and EM algorithms [7]).

Approximate GM methods are way faster than exact GM methods where lots of them can be run in polynomial time and with high classification rates. However, they do not guarantee to find nearly optimal solutions, specially when graphs are *complex*. Furthermore, none of the approximate methods present a formalism showing that approximate methods provide lower or upper bounds for the reference graph edit distance within a fixed factor (*e.g.* a factor of 4 in [87]). We believe that the larger the error-tolerant GM problem (i.e., the more complex the graphs), the less accurate the distance and the lower the precision. In other words, matching two large graphs using an approximate error-tolerant GM method leads to a large divergence when comparing it with an exact method. In approximate problem reformulation, approximate methods only compare graphs superficially without regard to the global coherence of the matched vertices and edges. Whereas in approximate optimization, parts of the solution space of such methods are left behind or ignored. However, sometimes approximate algorithms are efficient specially when attributes help in quickly finding the optimal or a good near-optimal solution.

A significant remark one may notice is that the size of the graphs, involved in the experiments of all GM methods, is a hundred or so. Based on this remark, we raise the following questions:

- Why cannot GM methods, whether exact or not, cope with graphs larger than hundreds of vertices?
- Why referenced and publicly available datasets do not exceed hundreds of vertices? Why they do not contain graphs with different densities and/or attributes?
  - Is it because there is no need to work on larger or more complex graphs in the PR domain?
  - Is it because of the limitations of the algorithms proposed for solving GM problems?
  - Or is it always because of the complexity of such problems?

These raised questions are still open research questions.

2.3 A Focus on Graph Edit Distance

In this chapter, we give some arguments for which we have given a focus on GED.

First, GED is an error-tolerant problem that has been widely studied and largely applied to PR. Its flexibility comes from its generality as it can be applicable on unconstrained attributed graphs. Moreover, it can be dedicated to various applications by means of specific edit cost functions.

Second, it has been shown by Neuhaus and Bunke [98] that GED can be a metric if each of its elementary operations satisfies the three properties of metric spaces (i.e., positivity, symmetry and triangular inequality), see Section 2.1.11.6.

Third, few research papers have discussed the direct relation between GED and MCS ([21]; and [19]). Indeed, with the metric constraints explained above, GED can also pass through a maximum unlabeled common subgraph of two graphs (G_1 and G_2). MCS(G_1, G_2) is not necessarily unique for the two graphs G_1 and G_2.

Last, but not least, GED has been used in different applications such as Handwriting Recognition (i.e., [48]), Word Spotting (*e.g.* [109, 144]) and Palm-print classification (*e.g.* [124]).

For all the above-mentioned arguments, we mainly focus on GED as a basis of this thesis.

2.3.1 Graph Edit Distance Computation

The methods of the literature can be divided into two categories depending on whether they can ensure the exact matching to be found or not. For the sake of clarity in the rest of the thesis, the term *vertex* refers to an element of a graph while the term *node* represents an element of the search tree.

2.3.1.1 Exact Graph Edit Distance Approaches

A*-based GED A widely used method for edit distance computation is based on the A^* algorithm [102]. This algorithm is considered as a foundation work for solving GED. Algorithm 1 recalls the main steps of the A^* method.

The A^*-based method for exact GED computation proceeds to an implicit enumeration of all possible solutions without explicitly evaluating all of them [112]. This is achieved by means of an ordered tree. Such a search tree is constructed dynamically at run time by iteratively creating successor vertices.

Only leaf vertices correspond to feasible solutions and so complete edit paths. For a node p in the search tree, $g(p)$ represents the cost of the partial edit path accumulated so far, and $h(p)$ denotes the estimated costs from p to a leaf node representing a complete solution. The sum $g(p) + h(p)$ is the total cost assigned to a node in the search tree and is referred to as a lower bound $lb(p)$. Given that the estimation of the future costs $h(p)$ is lower than, or equal to, the real costs, an exact path from the root node to a leaf node is guaranteed to be found [107].

Algorithm 1 Astar ($A*$)

Input: initial state s_i and a goal state s_g Output: a shortest path $path_{ig}$ from s_i to s_g

```
1: $OPEN \leftarrow \Phi$
2: $OPEN$.add($s_i$)
3: while $OPEN \neq \Phi$ do
4:     $s_b \leftarrow OPEN$.minimumNode()          ▷ Remove the best node from $OPEN$
5:     if $s_b$.equals($s_g$) then                 ▷ If $s_b$ is the goal state $s_g$
6:         $path_{ig} \leftarrow s_b$.backtrackPath()    ▷ through recorded parents backtrack from $s_g$ until reaching $s_i$
7:         Return $path_{ig}$
8:     end if
9:     successors$_{s_b}$ ← successors($s_b$)     ▷ Create the successors of $s_b$ and evaluate them
10:    $OPEN$.add(successors$_{s_b}$)   ▷ add successors to $OPEN$ and record their parent $s_b$
11: end while
```

Algorithm 2 depicts the pseudo code of A^*. This algorithm is taken from [112]. However, in Algorithm 2 we illustrate it differently. The set *OPEN* of partial edit paths contains the search tree nodes to be processed in the next steps. p refers to the current node that will be explored (lines 1 and 2). In the beginning, the first level of the search tree is constructed and inserted in $OPEN$ (lines 3 to 6). To construct the first level of the search tree, u_1 in G_1 is substituted with each v_i in G_2 in addition to the deletion of u_1 (i.e., $u_1 \to \epsilon$). Algorithm 3 represents how the children of node p are generated. The substitution (lines 3 to 6) or the deletion of a vertex (lines 7 and 8) are considered simultaneously, which produces a number of successor nodes in the search tree. These successors are then saved in a list *Listp*. Back to Algorithm 2, the most promising partial edit path $p \in$ *OPEN*, i.e., the one that minimizes $lb(p)$, is always chosen first (lines 8 and 9). This procedure guarantees that the complete edit path outputted by the algorithm is always optimal, i.e., its cost is the minimum among all possible competing paths. If all vertices of G_1 have been processed (line 11), the remaining vertices of the second graph are inserted in a single step (lines 15 to 19). Finally, when all the branches of the tree have been pruned, if p, whose cost is the minimum, is a complete node, p and its cost $g(p)$ are outputted as an optimal solution of $GED(G_1, G_2)$ (line 13). Note that pendingV_1 and pendingV_2 represent a set of V_1 and V_2 that has not yet been matched.

The edit operations on edges are implied by edit operations on their adjacent vertices. For instance, whether an edge is substituted, deleted, or inserted, depends on the edit operations performed on its adjacent vertices. Figure 2.13 recalls the notations of vertices and edges of G_1 and G_2.

Algorithms 4, 5 and 6 represent vertex insertions, deletions and substitutions, respectively. In the case of vertex insertions of $v_k \in V_2$ in V_1, one also has to insert its adjacent edges (i.e., each e_{kz}), see Algorithm 4. On the other hand for vertex deletion of $v_i \in V_1$, one also has to delete its adjacent edges (i.e., each e_{ij}), see Algorithm 5. Note that pendingE_1 and pendingE_2 refer to a set of E_1 and E_2 that has not yet been matched.

As for vertex substitutions ($u_i \to v_k$), depicted in Algorithm 6, several cases should be taken into account:

Algorithm 2 Astar GED algorithm (*A**)

Input: Non-empty attributed graphs $G_1 = (V_1,E_1,L_{V_1},L_{E_1},\mu_1,\zeta_1)$ and $G_2 = (V_2,E_2,L_{V_2},L_{E_2},\mu_2,\zeta_2)$ where $V_1 = \{u_1,...,u_{|V_1|}\}$ and $V_2 = \{v_1,...,v_{|V_2|}\}$
Output: A minimum cost edit path (p_{min}) from G_1 to G_2 *e.g.*, $\{u_1 \to v_3,\ u_2 \to \epsilon\ ,\ \epsilon \to v_2\}$

```
 1: $p \leftarrow$ root node of the tree with all vertices and edges of $g_1$ and $g_2$ as pending lists
 2: $OPEN \leftarrow \Phi$
 3: $Listp \leftarrow$ GenerateChildren($p$)
 4: for $p \in Listp$ do
 5:     $OPEN$.AddFirst($p$)
 6: end for
 7: while true do
 8:     $OPEN \leftarrow$ SortAscending($OPEN$)          ▷ according to $l(b)=g(p)+h(p)$
 9:     $p \leftarrow OPEN$.PopFirst()          ▷ Take first element and remove it from $OPEN$
10:     $Listp \leftarrow$ GenerateChildren($p$)
11:     if $Listp = \Phi$ then
12:         if pending$V_2(p) = \Phi$ then
13:             Return($g(p)$,$p$)     ▷ Return $p$ and its cost (distance) as the optimal solution
    of $GED(G_1, G_2)$
14:         else
15:             for $v_i \in$ pending$V_2(p)$ do
16:                 $q \leftarrow$ insertion($q$,$v_i$)          ▷ i.e., $\{\epsilon \to v_i\}$
17:                 $p$.AddFirst($q$)
18:             end for
19:             $OPEN$.AddFirst($p$)
20:         end if
21:     else
22:         for $p \in Listp$ do
23:             $OPEN$.AddFirst($p$)
24:         end for
25:     end if
26: end while
```

Algorithm 3 GenerateChildren

Input: A tree node p
Output: A list *Listp* whose elements are the children of p

```
1: $Listp \leftarrow \Phi$
2: $u_1 \leftarrow pendingV_1(p)$.PopFirst()
3: for $v_i \in$ pending$V_2(p)$ do
4:     $q \leftarrow$ substitution($p$,$u_1$,$v_i$)          ▷ i.e., $\{u_1 \to v_i\}$
5:     $Listp$.AddFirst($q$)          ▷ $q$ is a tree node
6: end for
7: $q \leftarrow$ deletion($p, u_1$)          ▷ i.e., $\{u_1 \to \epsilon\}$
8: $Listp$.AddFirst($q$)
9: Return($Listp$)
```

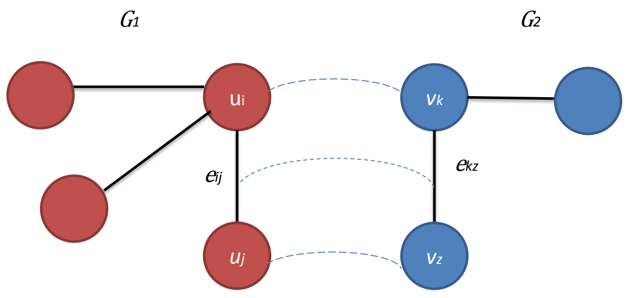


Figure 2.13: Notations recall: G_1 and G_2 are two graphs, e_{ij} is an edge between two vertices v_i and v_j in G_1 while e_{kz} is an edge between two vertices v_k and v_z in G_2

- If vertex u_j is already deleted (i.e., $u_j \to \epsilon$), then e_{ij} has to be deleted (line 9).
- If vertex u_j is already substituted to vertex u_z, but there is no edge e_{kz} between vertices v_k and v_z then e_{ij} has to be deleted (line 13).
- If vertex u_j is already substituted to vertex u_z, an there is an edge e_{kz} between vertices v_k and v_z then e_{ij} is substituted to e_{kz} (line 15).
- If vertex u_i is already substituted to vertex u_z, but there is no edge e_{ij} between vertices v_i and v_j then e_{kz} is inserted (line 28).

Note that the complexity of the substitution $u \to v$ is $\mathcal{O}(|adj(u)| + |adj(v)|)$. where $adj(u)$ and $adj(v)$ are the adjacent edges of u and v, respectively.

Algorithm 4 Insertion

Input: A tree node q and a vertex v_k in G_2
Output: A tree node q

```
1: $q$.add($\epsilon \to v_k$)
2: $ListE_{v_k}$ = edges($v_k$)
3: for $e_{kz} \in ListE_{v_k}$ do
4:     $q$.add($\epsilon \to e_{kz}$)
5:     pending$E_2$(q).remove($e_{kz}$)        ▷ remove $e_{kz}$ from the pending$E_2$ of $q$
6: end for
7: pending$V_2$(q).remove($v_k$)
8: Return($q$)
```

Note that Algorithms 4, 5 and 6 are also used in all the methods proposed in the thesis.

Algorithm 1 presents a Best-First algorithm which ends up having tremendous number of unnecessary nodes in memory, such a fact is considered as a drawback of this approach. In the worst case, the space complexity can be expressed as $O(|\gamma|)$ where $|\gamma|$ is the cardinality of the set of all possible edit paths [38]. Since $|\gamma|$ is exponential in the number of vertices involved in the graphs, the memory usage is still an issue.

There are lots of ways to solve the problem of estimating $h(p)$ for the costs from the current node p to a leaf node. In Section 5.3.3.1, we will study the effect of different $h(p)$'s

Algorithm 5 deletion

```
Input: A tree node $q$ and a vertex $u_i$ in $G_1$
Output: A tree node $q$
1: $q$.add($u_i \rightarrow \epsilon$)
2: $ListE_{u_i}$ = edges($u_i$)
3: for $e_{ij} \in ListE_{u_i}$ do
4:     $q$.add($e_{ij} \rightarrow \epsilon$)
5:     pending$E_1$(q).remove($e_{ij}$)
6: end for
7: pending$V_1$(q).remove($u_i$)
8: Return($q$)
```

on A^*.

Binary Linear Programming A binary linear programming formulation of GED is proposed in [71]. GED between graphs is treated as finding a subgraph of a larger graph referred to as the edit grid. The edit grid only needs to have as many vertices as the sum of the total number of vertices in the graphs being compared. The edit grid is a complete graph $G_\Omega = (\Omega, \Omega X \Omega, \mu_\Omega)$ where Ω denotes a set of vertices with N elements. Accordingly, $\Omega X \Omega$ is the set of undirected edges connecting all pairs of vertices. GED between $G_1 = (V_1,E_1,L_{V_1},L_{E_1},\mu_1)$ and $G_2 = (V_2,E_2,L_{V_2},L_{E_2},\mu_2)$ can be expressed by:

$$GED(G_1, G_2) = \min_{P \in \{0,1\}^{NXN}} \sum_{i=1}^{N} \sum_{j=1}^{N} c(l(A_1^i), l(A_2^j))P^{ij} + \frac{1}{2} c(0,1) |A_1 - PA_2P^T|^{ij} \quad (2.9)$$

where P is a permutation matrix representing all possible permutations of the elements of edit grid. $A_n \in \{0,1\}^{NXN}$ is the adjacency matrix corresponding to G_n in the edit grid. P represents N^2 boolean variables. N needs to be no larger than $|V_1| + |V_2|$ where $|V_1|$ and $|V_2|$ are the numbers of vertices of the involved graphs. $l(A_n^i)$ is the attribute assigned to the i^{th} row/column of A_n. Finally, the function c is a metric between two vertex attributes.

Formulation 2.9 is quadratic since it holds the product of P variables (PA_2P^T). In order to linearize it, two matrices, S and T, are introduced (inspired by [6]), and thus a binary linear formulation is obtained:

$$GED(G_1, G_2) = \min_{P,S,T \in \{0,1\}^{NXN}} \sum_{i=1}^{N} \sum_{j=1}^{N} c(l(A_1^i), l(A_2^j))P^{ij} + \frac{1}{2} c(0,1)(S+T)^{ij} \quad (2.10)$$

$$s.t. \begin{cases} (A_1P - PA_2 + S - T)^{ij} = 0, \ \forall i,j & (2.10.1) \\ \sum_i P^{ik} = \sum_j P^{kj} = 1, \forall k & (2.10.2) \end{cases}$$

P, S and T represent $3 \times N^2$ boolean variables where N is the number of vertices $|V_1| + |V_2|$. Two types of constraints are applied to the objective function. In the first

Algorithm 6 substitution

Input: a tree node q, a vertex u_i in G_1 and a vertex v_k in G_2

Output: a tree node q

```
1: $q$.add($u_i \rightarrow v_k$)
2: $ListE_{u_i}$ = edges($u_i$)
3: $ListE_{v_k}$ = edges($v_k$)
4: **for** $e_{ij} \in ListE_{u_i}$ **do**
5:     $u_j$ = $e_{ij}$.getOtherEndVertex()
6:     **if** $q$.contains($u_j$) **then**    ▷ check whether $q$ has an edit operation that contains $u_j$
7:         $v_z$ = $u_j$.getMatchedVertexG2();
8:         **if** $v_z = \epsilon$ **then**
9:             $q$.add($e_{ij} \rightarrow \epsilon$)
10:        **else**
11:            $e_{kz}$ = getEdgeBetween($v_k, v_z$)
12:            **if** $e_{kz} = \phi$ **then**
13:                $q$.add($e_{ij} \rightarrow \epsilon$)
14:            **else**
15:                $q$.add($e_{ij} \rightarrow e_{kz}$)
16:                pending$E_2$(q).remove($e_{kz}$)    ▷ remove $e_{kz}$ from the pending$E_2$ of $q$
17:            **end if**
18:        **end if**
19:        pending$E_1$(q).remove($e_{ij}$)
20:    **end if**
21: **end for**
22: **for** $e_{kz} \in ListE_{v_k}$ **do**
23:     $v_z$ = $e_{kz}$.getOtherEndVertex()
24:     **if** $q$.contains($v_z$) **then**
25:         $u_j$ = $v_z$.getOtherVertexG1()
26:         $e_{ij}$ = getEdgeBetween($u_i$,$u_j$)
27:         **if** $e_{ij} = \phi$ **then**
28:             $q$.add($\epsilon \rightarrow e_{kz}$)
29:             pending$E_2$(q).remove($e_{kz}$)
30:         **end if**
31:     **end if**
32: **end for**
33: pending$V_1$(q).remove($v_i$)
34: pending$V_2$(q).remove($v_k$)
35: Return($q$)
```

type of constraints, the GM problem is formulated as the minimization of the difference in adjacency matrix norms for unattributed graphs with the same number of vertices. The second constraints limit the set of acceptable permutation where one element of grid (i.e., vertex) must be permuted with exactly one element. There are N^2 constraints of type 1 and $2N$ constraints of type 2. Finally, the model is solved by a mathematical programming solver (lpsolve). One drawback of this method is that it does not take into account attributes on edges which limits the range of application.

2.3.1.2 Approximate Graph Edit Distance Approaches

The main reason that motivates researchers to work on approximate GED comes from the combinatorial explosion of exact GED methods. Therefore, numerous variants of approximate GED algorithms are proposed for making GED computation substantially faster. In this section, we dig into the details of the approximate methods.

Beam Search A modification of *A**, called Beam-Search (*BS*), has been proposed in [91]. The purpose of *BS*, is to prune the search tree while searching an exact edit path. Instead of exploring all edit paths in the search tree, a parameter x is set to an integer x which is in charge of keeping the x most promising partial edit paths in the *OPEN* set. Such an algorithm cannot always ensure the exact matching to be found. When $x = 1$, *BS* becomes a greedy search algorithm.

Bipartite Matching As previously mentioned in Section 2.1.11.9, GED has been reformulated as an instance of QAP. In [107], Riesen et al have also reformulated the assignment problem as finding an exact matching in a complete bipartite GM. However, they have reduced the QAP of GED computation to an instance of a Linear Sum Assignment Problem (LSAP). LSAPs as well as QAPs formulate an assignment problem of entities. Unlike QAPs, LSAPs are able to optimize the permutation $(\varphi_1, \cdots, \varphi_{n+m})$ with respect to the linear term $(i.e \sum_{i=1}^{n+m} c_{i\varphi(i)})$, see Section 2.1.11.9. That is, the matrix C is the only matrix that is considered and the quadratic term (i.e., $\sum_{i=1}^{n+m} \sum_{j=1}^{n+m} c(a_{\varphi_i\varphi_j} \rightarrow b_{\varphi_i\varphi_j})$) is omitted from the objective function. By doing so, the edges between vertices are neglected since that the matrices A and B are the ones that refer to the edges of G_1 and G_2, respectively. In order to reduce GED into a LSAP, local rather than global relationships are considered.

Formally saying, let $G_1 = (V_1, E_1, \mu_1, \xi_1)$ and $G_2 = (V_2, E_2, \mu_2, \xi_2)$ with $V_1 = (u_1, \ldots, u_{|V_1|})$ and $V2 = (v_1, \ldots, v_{|V_2|})$ respectively. A square cost matrix C_{ve} is constructed between G_1 and G_2 as follows:

$$C_{ve} = \left|\begin{array}{cccc||cccc}
c_{1,1} & \dots & \dots & c_{1,|V_2|} & c_{1,\epsilon} & \infty & \dots & \infty \\
\dots & \dots & \dots & \dots & \infty & c_{2,\epsilon} & \dots & \infty \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
c_{|V_1|,1} & \dots & \dots & c_{|V_1|,|V_2|} & \infty & \dots & \infty & c_{|V_1|,\epsilon} \\
\hline
c_{\epsilon,1} & \infty & \dots & \infty & 0 & \dots & \dots & 0 \\
\infty & c_{\epsilon,2} & \dots & \infty & \dots & \dots & \dots & \dots \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
\infty & \dots & \infty & c_{\epsilon,|V_2|} & 0 & \dots & \dots & 0
\end{array}\right|$$

At a first glance, this matrix looks similar to the matrix C in Section 2.1.11.9. However, each element c_{ij} in the matrix C_{ve} corresponds to the cost of assigning the i^{th} vertex of G_1 to the j^{th} vertex of G_2 in addition to assigning the edges of the i^{th} vertex of G_1 to the edges of the j^{th} vertex of G_2. For the edge edit operation costs, the minimum sum is selected. Thus, the problem of GM is reduced to finding the minimum assignment cost C_{ve} such that $p = p_1, \dots, p_n$ is a matrix permutation. In the worst case, the maximum number of operations needed by the algorithm is $O((n+m)^3)$ where n and m denote $|V_1|$ and $|V_2|$, respectively. In the rest of the thesis, this algorithm is referred to as BP.

As in the matrix C, the left upper corner of the matrix contains all possible vertex substitutions, the diagonal of the right upper matrix represents the cost of all possible vertex deletions and the diagonal of the bottom left corner contains all possible vertex insertions. The bottom right corner elements cost is set to zero which concerns the substitution of $\epsilon - \epsilon$.

Recently, two new versions of BP to compute GED, called Fast Bipartite method (*FBP*) and Square Fast Bipartite method (*SFBP*), have been published in [122] and [123], respectively.

In, *FBP*, the cost matrix is composed of only one quadrant. When $|V_1| \neq |V_2|$, the unused cells in the matrix are filled with zeros so as to be a square matrix which is the case of linear assignation methods [17].

$$C_{ve} = \left|\begin{array}{cccc}
c_{1,1} - (c_{1,\epsilon} + c_{\epsilon,1}) & \dots & \dots & c_{1,|V_2|} - (c_{1,\epsilon} + c_{\epsilon,|V_2|}) \\
\dots & \dots & \dots & \dots \\
\dots & \dots & \dots & \dots \\
c_{|V_1|,1} - (c_{|V_1|,\epsilon} + c_{\epsilon,1}) & \dots & \dots & c_{|V_1|,|V_2|} - (c_{|V_1|,\epsilon} + c_{\epsilon,|V_2|})
\end{array}\right|$$

On the other hand, in *SFBP*, two square matrices are defined. One of them is used depending on the order of the involved graphs. When $|V_1| \leq |V_2|$, C_{ve} is represented as:

$$C_{ve} = \left|\begin{array}{cccc} c_{1,1} & \dots & \dots & c_{1,|V_2|} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ c_{|V_1|,1} & \dots & \dots & c_{|V_1|,|V_2|} \\ \hline \hline c_{\epsilon,1} & \infty & \dots & \dots \\ \infty & c_{\epsilon,2} & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \infty & \dots & \infty & c_{\epsilon,|V_2|} \end{array}\right|$$

Whereas when $|V_2| \leq |V_1|$, C_{ve} is represented as:

$$C_{ve} = \left|\begin{array}{cccc||cccc} c_{1,1} & \dots & \dots & c_{1,|V_2|} & c_{1,\epsilon} & \infty & \dots & \infty \\ \dots & \dots & \dots & \dots & \infty & c_{2,\epsilon} & \dots & \infty \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ c_{|V_1|,1} & \dots & \dots & c_{|V_1|,|V_2|} & \infty & \dots & \infty & c_{|V_1|,\epsilon} \end{array}\right|$$

FBP and *SFBP* have a restriction since edit costs have to be defined such that the edit distance is a distance function [123] that is equal to *BP*. Thus three restrictions have to be satisfied:

- Insertion and deletion costs have to be symmetric.
- $C_{vs}(u_i, v_j)$ and $C_{es}(e_{ij}, e_{kz})$ have to be defined as a distance measure.
- $c(u_i, v_j) \leq 2.K_v$ and $c(e_{ij}, e_{kz}) \leq 2.K_e$ where $2.K_v$ and $2.K_e$ are the costs of inserting and deleting vertices and edges, respectively.

In *BP*, *FBP* and *SFBP* when substituting, deleting or inserting vertices, their local substructures are taken into account. Among local substructures, the *degree centrality* and the *clique centrality* are considered as the two most used ones:

- Degree Centrality (λ_i^{deg}) of each node $u_i \in V_1$ is set to k_i where k_i refers to the number of edges connected to u_i.
- Clique Centrality (λ_i^{clique}) of each node $u_i \in V_1$ is set to k_{ij} where k_{ij} refers to the number of edges connected to u_i as well as their neighboring vertices $\{u_j\}$.

Figure 2.14 illustrates an example of λ_i^{deg} and λ_i^{clique}. When matching graphs whose edges are unattributed, the vertices cost includes counting the number of their neighboring edges. In [39], in addition to λ_i^{deg} and λ_i^{clique}, three other vertex centralities have been proposed (the *planar centrality*, the *eigenvector centrality* [16] and the *Google's PageRank centrality* [18]) and their effect on *SFBP* is studied.

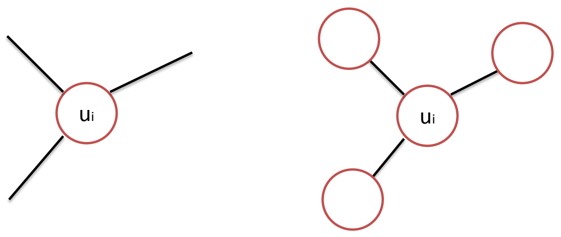


Figure 2.14: Examples of the local substructures: λ_i^{deg} and λ_i^{clique}. Note that $\lambda_i^{deg} = 3$ and $\lambda_i^{clique} = 6$

Improvements via Search Strategies Since *BP* considers local structures rather than global ones, an overestimation of the exact GED cannot be neglected. Recently, few works have been proposed for overcoming such a problem. Researchers have observed that *BP*'s overestimation is very often due to a few incorrectly assigned vertices. That is, only few vertex substitutions from the next step are responsible for additional (unnecessary) edge operations in the step after and thus resulting in the overestimation of the exact edit distance.

In [110], a greedy swap GED is proposed, see Algorithm 7. As a first step, *BP* is used to compute a first distance d_{best} as well as a mapping m (line 1). Then for each pair of vertex assignments (i.e., $(u_i \rightarrow u_{p_i})$ and $(u_j \rightarrow u_{p_j})$) in m, the partial cost $cost_{orig} = cost(u_i \rightarrow u_{p_i}) + cost(u_j \rightarrow u_{p_j})$ is calculated. This cost is compared with the new mapping cost $cost_{swap}$ which is based on swapping the former vertex assignments (i.e., $(u_i \rightarrow u_{p_j})$ and $(u_j \rightarrow u_{p_i})$) (lines 7 and 8). In order to decide whether the new swapped mappings are beneficial or not, a parameter θ multiplied by $cost_{orig}$ is compared with the absolute value of $cost_{orig}$-$cost_{swap}$ (line 9). If this value is smaller than the defined threshold, the swapping is performed, a new mapping $\bar{m}$ is generated and a new distance $d_{(\bar{m})}$ is derived (lines 10 and 11). If $d_{(\bar{m})}$ is smaller than d_{best}, d_{best} is updated and replaced by $d_{(\bar{m})}$. Moreover, m is replaced by $\bar{m}$. The steps from line 5 to 18 are repeated searching for a better m and so d_{best}. The parameter *swapped* is used as an indicator that tells if there were some changes that have been made when executing the two for loops. If *swapped* equals *true*, the steps are re-executed on the new m. Once m becomes stable (i.e., when *swapped* equals *false*), d_{best} and m are outputted as a best answer that can be found by Greedy-Swap. Note that d_{best} is not necessarily the optimal solution since *BP* is based on local search and that's why the Greedy-Swap algorithm belongs to approximate GED methods.

Based on the same idea of [110], a Beam-Search version of *BP*, called *BP-Beam*, is proposed in [113]. This work focuses on investigating the influence of the order in which the assignments are explored, such a process is considered as a post search process on the distance quality. As in [110], the original node assignment $d_{(m)}(G_1, G_2)$ is systematically varied by swapping $(u_i \rightarrow v_{p_i})$ and $(u_j \rightarrow v_{p_j})$. For each swap it is verified whether (and to what extent) the derived distance approximation stagnates, increases or decreases. For a systematic variation of mapping m, a tree search is used. As usual, in tree-search based methods, a set *OPEN* is employed that holds all of the unprocessed tree nodes. Each tree node is a triple ($\bar{m}$,q,$d_{(\bar{m})}(G_1, G_2)$) where $\bar{m}$ is the matching, q is its depth in the search

Algorithm 7 Greedy-Swap (G_1,G_2)

Input: Non-empty attributed graphs $G_1 = (V_1, E_1, \mu_1, \zeta_1)$ and $g_2 = (V_2, E_2, \mu_2, \zeta_2)$ where $V_1 = \{u_1, ..., u_{|V_1|}\}$ and $V_2 = \{v_1, ..., v_{|V_2|}\}$ and a parameter θ
Output: A minimum cost edit path (d_{best}) from G_1 to G_2

```
1: $d_{best}$,$m = d_{(m)}$($G_1$,$G_2$)
2: $swapped = true$
3: while swapped do
4:     $swapped$=$false$
5:     for $i = 1...(m + n - 1)$ do
6:         for $j = i + 1...(m + n)$ do
7:             $cost_{orig} = cost(u_i \rightarrow v_{p_i})+ cost(u_j \rightarrow v_{p_j})$
8:             $cost_{swap} = cost(u_i \rightarrow v_{p_j})+ cost(u_j \rightarrow v_{p_i})$
9:             if —$cost_{orig} - cost_{swap}| \leq \theta.cost_{orig}$ then
10:                $\bar{m} = m - \{u_i \rightarrow v_{p_i}, u_j \rightarrow v_{p_j}\} \cup \{u_i \rightarrow v_{p_j}, u_j \rightarrow v_{p_i}\}$
11:                Derive approximate edit distance $d_{(\bar{m})}(G_1, G_2)$
12:                if $d_{(\bar{m})}(G_1, G_2) < d_{best}$ then
13:                    $d_{best}$,$m = d_{(\bar{m})}(G_1, G_2)$
14:                    $swapped = true$
15:                end if
16:            end if
17:        end for
18:    end for
19: end while
20: return $d_{best}$ and $m$
```

tree and $d_{(\bar{m})}(G_1, G_2)$ is the distance obtained from the mapping $\bar{m}$. Since *BP* is used as a first step, it is considered as a first node in *OPEN* and thus its depth q is equal to zero. Nodes are kept sorted in ascending order according to their depth in the search tree. As long as *OPEN* is not empty, the triple $(\bar{m}, q, d_{(\bar{m})}(G_1, G_2))$ located at the first position in *OPEN* is retrieved and removed from *OPEN*. The successors of $(\bar{m}, q, d_{(\bar{m})}(G_1, G_2))$ are generated by swapping $(u_i \rightarrow v_{p_i})$ and $(u_j \rightarrow v_{p_j})$ (i.e., $u_i \rightarrow v_{p_j}$) and $(u_j \rightarrow v_{p_i})$ where $i = q$ and $j = \{(q), \cdots, (|V_1| + |V_2|)\}$. These successors are inserted at the end of the list *OPEN* where $q = q + 1$. When each successor is inserted, a systematic verification is performed to verify whether the derived distance $d_{(\bar{m})}(G_1, G_2)$ is smaller than the best distance found so far (i.e. d_{best}). If this is the case, d_{best} is modified. After successors insertion and d_{best} verification, the best x solutions are kept in *OPEN* and the next node is retrieved and removed and so on. The algorithm stops when *OPEN* becomes empty. d_{best} is then outputted as a final solution of *BP-Beam*(G_1, G_2).

Recently, an iterative version of *BP-Beam*, referred to as *IBP-Beam*, has been proposed in [45]. This algorithm starts by computing a first distance $d_m(G_1, G_2)$ using *BP*. A randomization step is then applied to change the order of the matched vertices. Afterwards, *BP-Beam* is applied on the new-ordered matching. The randomization and *BP-Beam* are repeated k times. This algorithm has two parameters (x and k). Results showed that this algorithm takes much longer time than *BP-Beam*. However, it improves the distance quality.

In [114], a search procedure based on a genetic algorithm referred to as *BPGA* is proposed for improving the accuracy of *BP*. After calculating a first upper bound $(m, d_m(G_1, G_2))$ using *BP*, an initial population $P(0)$ is build by computing N random order variations of m (i.e., $m_1^{(0)}, \cdots, m_N^{(0)}$). Each variation $m_i^{(0)} \in P(0)$ is computed by assigning a mutation probability to each vertex-to-vertex mapping (i.e., $u_i \rightarrow v_{p_i}$). This probability tells whether a mapping can be deleted or not. In the case of deleting a vertex-to-vertex mapping, an infinity cost is assigned to it (i.e., c$(u_i \rightarrow v_{p_i})$=$\infty$). Given this modification in a matrix entity, a new mapping $m_i^{(0)}$ and its underlying distance are generated. Note that $m_i^{(0)}$ does not contain $u_i \rightarrow v_{p_i}$ anymore. As mentioned before, N mappings are generated through mutation procedure and thus the aforementioned procedure is repeated N times to obtain an initial population $P(t) = \{m_1^{(t)}, \cdots, m_N^{(t)}\}$. Afterwards, a subset E, called parents, of $P(0)$ is created aiming at obtaining a second population $P(t+1)$. Parents' selection is achieved by selecting the best K approximations whose distances are the minimum. These parents are inserted into $P(t+1)$ without any modification. Then, to generate the other $N - |E|$ mappings, the following strategy is repeated $N - |E|$ times. Two mappings $\bar{m}$, $\bar{\bar{m}} \in E$ are randomly selected and combined in one mapping m where $C_m = \max(\{c_{i,j}^{-}, c_{i,j}^{=}\})$ where $\bar{c}$ and $\bar{\bar{c}}$ are the cost matrices of $\bar{m}$ and $\bar{\bar{m}}$, respectively. Any prevented mapping (i.e., a mapping whose cost is infinity) in $\bar{m}$ and $\bar{\bar{m}}$ is also prevented in the merged mapping m. The selection of parents $E \subseteq P(t)$ and the generation of $N - |E|$ new mappings are repeated again and again. The algorithm is stopped when the best distance has not been modified for ι iterations where $\iota \leq t$. Since any genetic algorithm is non-deterministic, the computation of *BPGA* is repeated s times. Afterwards, the best distance along with its matching are outputted. Thus, when comparing *BPGA* to *BP*, *BPGA* increases the run time by parameters $s.t.N$.

Compared to the original BP, the improvements proposed in [45, 113, 110, 114] increase run times. However, they improve the accuracy of the BP solution.

Hausdorff Edit Distance In [49], the authors propose a novel modification of the Hausdorff distance that takes into account not only substitution, but also deletion and insertion cost. We refer to the modified version of the Hausdorff distance as H. This approach allows multiple vertex assignments, consequently, the time complexity is reduced to quadratic (i.e., $O(n^2)$) with respect to the number of vertices of the involved graphs.

Definition 15 *The Modified Hausdorff (H)*

$$H(G_1, G_2) = \sum_{u \in V_1} \min_{v \in V_2} \bar{c_1}(u,v) + \sum_{v \in V_2} \min_{u \in V_1} \bar{c_2}(u,v) \tag{2.11}$$

where V_1 corresponds to the vertices of G_1 and V_2 to the vertices of G_2. The cost functions $\bar{c_1}(u,v)$ and $\bar{c_2}(u,v)$ for matching vertex u with vertex v are:

$$\bar{c_1}(u,v) = \begin{cases} \frac{c(u,v)}{2}, & \text{if } c(u,v) < c(u,\epsilon) \\ c(u,\epsilon), & \text{otherwise} \end{cases} \tag{2.12}$$

$$\bar{c_2}(u,v) = \begin{cases} \frac{c(u,v)}{2}, & \text{if } c(u,v) < c(\epsilon,v) \\ c(\epsilon,v), & \text{otherwise} \end{cases} \tag{2.13}$$

To compute $\bar{c_1}(u,v)$, among all the possible substitutions $\frac{c(u,v)}{2}$, the one with the smallest cost is chosen. Otherwise, the deletion cost $c(u,\epsilon)$ is returned. The same thing for $\bar{c_2}(u,v)$ where the minimum substitution $\frac{c(u,v)}{2}$ is chosen. Otherwise, the insertion cost $c(\epsilon,v)$ is returned (i.e., if $c(u,v) > c(\epsilon,v)$).

For both $\bar{c_1}(u,v)$ and $\bar{c_2}(u,v)$, the estimated implied edge cost is included with each $c(u,v)$ as well as $c(u,\epsilon)$ and $c(\epsilon,v)$ such that:

$$c(i,j) = c(i,j) + \frac{\text{estimated implied edge cost}}{2}$$

Unlike the approximate GED methods explained in this section, $H(G_1, G_2)$ is a lower bound GED method.

2.4 Performance Evaluation of Graph Matching

In this section, a focus on the existing benchmarks for GM methods is given. Table 2.3 synthesizes the graph databases presented in the literature. One may notice that exact GM methods have been evaluated at matching level. However, error-tolerant GM methods have been evaluated at the classification level rather than the matching one. Yet, in the literature, some approximate GED methods have been evaluated at the matching level but unfortunately on graphs whose sizes are not bigger than 16 vertices [108]. One may ask the following question: why error-tolerant GM algorithms have not been tested at the matching level? Is it because there is really no need for that and that the only need for error-tolerant GM methods is to classify graphs? For instance, one of the databases repository called CMU [2] is devoted to error-tolerant GM with ground truth information. However, graphs have the same number of vertices and thus the scalability measure cannot be assessed. Indeed, one may clearly see that there is a lack of performance comparison measures dedicated to the scalability of error-tolerant GM methods, whether exact or approximate ones.

Ref	**Problem Type**	**Graph Type**	**Database Type**	**Metrics Type**	**Purpose**
[119]	Exact GM	Non-attributed	Synthetic	Accuracy and scalability	Matching
[2]	Error-tolerant GM	Attributed	Real-world	Memory consumption, accuracy and matching quality	Matching
[106]	Error-tolerant GM	Attributed	Real-world	Accuracy and running time	Classification
[37, 52]	Exact GM	Attributed	Synthetic	Accuracy and scalability	Matching
[27]	Exact GM	(Non)attributed	Real-world	Scalability	Matching

Table 2.3: Synthesis of graph databases

2.5 Conclusion on the State-of-the-Art Methods

After having explored GM methods in general and GED methods in particular, we spot light and emphasize on several facts.

GED is the most flexible and generic GM problem since it can be applied on any type of graphs by changing the cost functions of both vertices and edges. Moreover, it can be transformed into an exact GM problem by means of metric constraints. Unlike statistical approaches, GED methods provide both a matching and a distance of the two involved graphs. It is also the most studied problem in the literature.

Table 2.4 synthesizes the GED methods on which we shed light in this chapter. Exact methods have not been intensely studied in the literature. In fact, exact GED methods are guaranteed to find the exact matching but have a run time and/or memory usage that is exponential in the size of the input graphs, such a fact limits the exact methods to work on relatively small graphs. For instance, A^* has shown to be a memory consuming method as it is based on a Best-First search algorithm.

Method	**Reference**	**Problem type**	**Graphs type**	**Distributed?**
*A**	[112]	Exact GED	symbolic and numeric attributes	NO
BLP	[71]	Exact GED	symbolic and numeric attributes only on vertices	NO
BS	[99]	Approximate GED	symbolic and numeric attributes	NO
BP	[107]	Approximate GED	symbolic and numeric attributes	NO
SWAP-BP	[113]	Approximate GED	symbolic and numeric attributes	NO
BP-Beam	[110]	Approximate GED	symbolic and numeric attributes	NO
IBP-Beam	[45]	Approximate GED	symbolic and numeric attributes	NO
BPGA	[114]	Approximate GED	symbolic and numeric attributes	NO
FBP	[122]	Approximate GED	symbolic and numeric attributes	NO
SFBP	[123]	Approximate GED	symbolic and numeric attributes	NO
H	[48]	Approximate GED	symbolic and numeric attributes	NO

Table 2.4: Synthesis of the GED methods presented in the thesis

On the other hand, approximate GED methods often have a polynomial run time in the size of the input graphs and thus are much faster, but do not guarantee to find the exact matching in addition to the fact that the quality of their provided answers (i.e., distance and matching) have not been studied. In addition, approximate methods have not been experimented on large or dense graphs. We, authors, believe that the more *complex* the graphs, the larger the error committed by the approximate methods. Graphs are generally more complex in cases where neighborhoods and attributes do not allow to easily differentiate between vertices. In addition to the lack of diversity of graph datasets, there is a lack in metrics for deeply evaluating error-tolerant GM methods since only classification rates have been evaluated. For instance, resources consumption of each method has not been deeply studied. Moreover, in GED, the impact of cost functions always remains a question.

Based on the aforementioned facts and conclusions, we believe that it is highly important to study and propose new solutions that can be listed as follows:

- Defining an optimized and exact GED method that can match larger graphs and consume less memory than A^*.
- Introducing a new kind of GM methods that could adapt themselves to give a trade-off between available resources (time and memory) and the quality of the provided solution.
- Putting forward a distributed or a parallel version of a GED method in order to handle larger graphs and to get more precised solutions (i.e., mappings and distance).
- Proposing new metrics and datasets with different types of graphs to better characterize GED methods in terms of precision of the provided solution (i.e., instead of only classification rates).

Bibliography

[1] Open-mp. *http://www.openmp.org.*

[2] Cmu house and hotel datasets. *http://vasc.ri.cmu.edu/idb/html/motion.*, 2013.

[3] R.C. Wilson A.D.J. Cross and E.R. Hancock. Inexact graph matching using genetic search. *Pattern Recognition*, 30(6):953–970, 1997.

[4] Cinque L. Member S. Tanimoto S. Shapiro L. Allen, R. and D. Yasuda. A parallel algorithm for graph matching and its maspar implementation. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):490–501, 1997.

[5] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele, and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, 2008.

[6] H. A. Almohamad and S. O. Duffuaa. A linear programming approach for the weighted graph matching problem. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 15(5):522–525, 1993.

[7] Edwin R. Hancock. Andrew D. J. Cross. Graph matching with a dual-step em algorithm. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20:1236–1253, 1998.

[8] Josep Llados Anjan Dutta and Umapada Pal. A symbol spotting approach in graphical documents by hashing serialized graphs. *Pattern Recognition*, 46(3):752–768, 2012.

[9] Mikhail J. Atallah and Susan Fox, editors. *Algorithms and Theory of Computation Handbook.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1998.

[10] Surapong Auwatanamongkol. Inexact graph matching using a genetic algorithm for image recognition. *Pattern Recognition Letters*, 28(12):1428 – 1437, 2007.

[11] László Babai. Graph isomorphism in quasipolynomial time. *CoRR*, abs/1512.03547, 2015.

[12] Lucio Barreto and Michael Bauer. Parallel branch and bound algorithm - a comparison between serial, openmp and mpi implementations. *journal of Physics: Conference Series*, 256(5):012–018, 2010.

[13] R. Battiti and M. Protasi. Reactive local search for the maximum clique problem 1. *Algorithmica*, 29(4):610–637, 2001.

[14] M. Beckman and T.C. Koopmans. Assignment problems and the location of economic activities. *Econometrica*, 25:53–76, 1957.

[15] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.

[16] Phillip Bonacich. Power and centrality: A family of measures. *American journal of Sociology*, 92(5):1170–1182, 1987.

[17] François Bourgeois and Jean-Claude Lassalle. An extension of the munkres algorithm for the assignment problem to rectangular matrices. *Commun. ACM*, 14:802–804, 1971.

[18] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.

[19] Luc Brun. Relationships between graph edit distance and maximal common structural subgraph. *¡hal-00714879v3¿*, 2012.

[20] Mihai Budiu, Daniel Delling, and Renato Fonseca F. Werneck. DryadOpt: Branch-and-Bound on Distributed Data-Parallel Execution Engines. *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 1278–1289, 2011.

[21] H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letter.*, 18:689–694, 1997.

[22] Horst Bunke and Kaspar Riesen. Towards the unification of structural and statistical pattern recognition. *Pattern Recognition Letters*, 33(7):811–825, 2012.

[23] Horst Bunke and Kim Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 2:255–259, 1998.

[24] Rainer E. Burkard, Eranda ela, Panos M. Pardalos, and Leonidas S. Pitsoulis. The quadratic assignment problem, 1998.

[25] D. Butenhof. *Programming with Posix Threads*. Addison-Wesley, 1997.

[26] Peter Cappello, Bernd Christiansen, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser, and Daniel Wu. Javelin: Internet-Based Parallel Computing Using Java, 1997.

[27] Vincenzo Carletti, Pasquale Foggia, and Mario Vento. Performance comparison of five exact graph matching algorithms on biological databases. volume 8158, pages 409–417, 2013.

[28] Imen Chakroun and Nordine Melab. Operator-level gpu-accelerated branch and bound algorithms. In *ICCS*, volume 18, 2013.

[29] Pierre-Antoine Champin and Christine Solnon. Measuring the similarity of labeled graphs. In *ICCBR*, volume 2689 of *Lecture Notes in Computer Science*, pages 80–95. Springer, 2003.

[30] Vijay Chandru and M. R. Rao. Algorithms and theory of computation handbook. pages 30–30. 2010.

[31] Barbara Chapman, Gabriele Jost, Ruud Van der Pas, and David J. Kuck. *Using OpenMP : portable shared memory parallel programming.* MIT Press, 2008.

[32] C.L. Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, 275:314 – 347, 2014.

[33] Chia-Shin Chung, James Flynn, and Janche Sang. Parallelization of a branch and bound algorithm on multicore systems. *journal of Software Engineering and Applications*, 5:12–18, 2012.

[34] F. R. K. Chung. *Spectral Graph Theory.* American Mathematical Society, 1997.

[35] D Conte, P Foggia, C Sansone, and M Vento. Thirty years of Graph Matching. *IJPRAI*, 18(3):265–298, 2004.

[36] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. How and why pattern recognition and computer vision applications use graphs. In *Applied Graph Theory in Computer Vision and Pattern Recognition*, pages 85–135. 2007.

[37] Foggia Pasquale Vento Mario Conte, Donatello. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *journal of Graph Algorithms and Applications*, 11(1):99–143, 2007.

[38] Thomas H. Cormen et al. *Introduction to Algorithms.* The MIT Press, 3rd edition, 2009.

[39] Xavier Cortés, Francesc Serratosa, and Carlos Francisco Moreno-García. On the influence of node centralities on graph edit distance for graph classification. In *GbRPR 2015 Proceedings*, pages 231–241, 2015.

[40] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[41] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *journal of the Royal Statistical Society, Series B*, 39(1):1–38, 1977.

[42] I. Dorta, C. Leon, and C. Rodriguez. A comparison between mpi and openmp branch-and-bound skeletons. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 66–73, 2003.

[43] Maciej Drozdowski. *Scheduling for Parallel Processing.* Springer Publishing Company, Incorporated, 1st edition, 2009.

[44] P. Erdős and A. Rényi. On random graphs. I. *Publ. Math. Debrecen*, 6:290–297, 1959.

[45] Miquel Ferrer, Francesc Serratosa, and Kaspar Riesen. A first step towards exact graph edit distance using bipartite graph matching. In *Graph-Based Representations in Pattern Recognition - 10th IAPR-TC-15 International Workshop*, pages 77–86, 2015.

[46] Miquel Ferrer, Francesc Serratosa, and Alberto Sanfeliu. Synthesis of median spectral graph. In *IbPRIA (2)*, pages 139–146, 2005.

[47] Andrew M. Finch, Richard C. Wilson, and Edwin R. Hancock. An energy function and continuous edit process for graph matching. *Neural Computation*, 10(7):1873–1894, 1998.

[48] Andreas Fischer, Ching Y. Suen, Volkmar Frinken, Kaspar Riesen, and Horst Bunke. A fast matching algorithm for graph-based handwriting recognition. *Graph-Based Representations in Pattern Recognition*, pages 194–203, 2013.

[49] Andreas Fischer, Ching Y. Suen, Volkmar Frinken, Kaspar Riesen, and Horst Bunke. Approximation of graph edit distance based on hausdorff matching. *Pattern Recognition*, 48(2):331–343, 2015.

[50] M.a. Fischler and R.a. Elschlager. The Representation and Matching of Pictorial Structures. *IEEE Transactions on Computers*, 22(1):67–92, 1973.

[51] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[52] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *IAPR TC-15*, pages 188–199, 2001.

[53] G.P. Ford and J. Zhang. A structural graph matching approach to image understanding. *Intelligent Robots and Computer Vision X: Algorithms and Techniques*, pages 559–569, 1991.

[54] Paolo Frasconi, Marco Gori, and Alessandro Sperduti. A general framework for adaptive processing of data structures. *IEEE Transactions on Neural Networks*, 9:768–786, 1998.

[55] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[56] B. Gendron et al. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, (7-8), 1994.

[57] Fred Glover and Manuel Laguna. *Tabu Search.* Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[58] S. Gold and A Rangarajan. A graduated assignment algorithm for graph matching. *Workshops SSPR and SPR*, pages 377–388, 1996.

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

[60] G. Allermann. H. Bunke. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters.*, 1:245–253, 1983.

[61] E.R. Hancock and J. Kittler. Edge-labeling using dictionary-based relaxation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2):165–181, 1990.

[62] Eric A. Hansen and Rong Zhou. Anytime heuristic search. *J. Artif. Int. Res.*, 28(1):267–297, 2007.

[63] Peter Hart et al. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, (2):100–107, 1968.

[64] A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987.

[65] Benoit Huet and Edwin R. Hancock. Shape recognition from large image libraries by inexact graph matching. *Pattern Recognition Letters*, 20:1259 – 1269, 1999.

[66] E. Marti J. Llados and J. J. Villanueva. Symbol recognition by error-tolerant subgraph matching between region adjacency graphs. *Patt. Anal. Mach.*, pages 1137–1143, 2001.

[67] X. Jiang and H. Bunke. Marked subgraph isomorphism of ordered graphs. *Workshops SSPR and SPR*, pages 122–131, 1998.

[68] Salim Jouili. *Indexation de masses de documents graphiques : approches structurelles.* Theses, Université Nancy II, 2011.

[69] Salim Jouili and Salvatore Tabbone. Graph matching based on node signatures. In *GBR*, pages 154–163, 2009.

[70] Flavio Junqueira et al. *Zookeeper: Distributed Process Coordination.* 2013.

[71] Hero A Justice D. A binary linear programming formulation of the graph edit distance. *IEEE Trans Pattern Anal Mach Intell.*, 28:1200–1214, 2006.

[72] J. Kazius et al. Derivation and validation of toxicophores for mutagenicity prediction. *journal of Medicinal Chemistry*, 48(1):312–20, 2005.

[73] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[74] Josef Kittler, William J. Christmas, and Maria Petrou. Probabilistic relaxation for matching problems in computer vision. In *ICCV*, pages 666–673. IEEE, 1993.

[75] Andreas L. Köll and Hermann Kaindl. A new approach to dynamic weighting. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 16–17, 1992.

[76] Richard E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41 – 78, 1993.

[77] C. Kotropoulos, a. Tefas, and I. Pitas. Morphological elastic graph matching applied to frontal face authentication under well-controlled and real conditions. *Pattern Recognition*, 33(12):1935–1947, 2000.

[78] V. Kumar, V.N. Rao, and University of Texas at Austin. Department of Computer Sciences. *Parallel Depth First Search: Part II ; Analysis.* 1988.

[79] Vipin Kumar, Ananth Y. Grama, and Nageshwara Rao Vempaty. Scalable load balancing techniques for parallel computers. *J. Parallel Distrib. Comput*, 22:60–79, 1994.

[80] Pierre Le Bodic, Pierre HéRoux, SéBastien Adam, and Yves Lecourtier. An integer linear program for substitution-tolerant subgraph isomorphism and its use for symbol spotting in technical drawings. *Pattern Recognition.*, 45:4214–4224, 2012.

[81] Vianney Le clment de saint-Marcq, Yves Deville, and Christine Solnon. Constraint-based Graph Matching. In *15th International Conference on Principles and Practice of Constraint Programming*, LNCS, pages 274–288. Springer, September 2009.

[82] Marius Leordeanu, Martial Hebert , and Rahul Sukthankar. An integer projected fixed point method for graph matching and map inference. In *Proceedings Neural Information Processing Systems*, pages 1114–1122, 2009.

[83] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.

[84] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, and Sebastian Thrun. Anytime search in dynamic graphs. *Artif. Intell.*, 172:1613–1643, 2008.

[85] Wenyin Liu, Josep Llados i Canet, and International association for pattern recognition, editors. *Graphics recognition : Ten years review and future perspectives : 6th international workshop, GREC 2005.* Springer, 2006.

[86] Zhiyong Liu and Hong Qiao. GNCCP - graduated nonconvexityand concavity procedure. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36:1258–1267, 2014.

[87] Daniel P. Lopresti and Gordon T. Wilfong. A fast technique for comparing graph representations with applications to performance evaluation. *IJDAR*, 6(4):219–229, 2003.

[88] Bin Luo, Richard C. Wilson, and Edwin R. Hancock. Spectral embedding of graphs. *Pattern Recognition*, 36(10):2213–2230, 2003.

[89] Bin Luo and Edwin R. Hancock. Structural graph matching using the em algorithm and singular value decomposition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(10):1120–1136, October 2001.

[90] Bin Luo and Edwin R. Hancock. Structural graph matching using the em algorithm and singular value decomposition. *IEEE Trans. PAMI*, 23:1120–1136, 2001.

[91] K. Riesen M. Neuhaus and H. Bunke. Fast suboptimal algorithms for the computation of graph edit distance. *Proceedings of 11th International Workshop on Structural and Syntactic Pattern Recognition.*, 28:163–172, 2006.

[92] P. Foggia M. Vento. *Graph-Based Methods in Computer Vision: Developments and Applications*, chapter Graph Matching Techniques for Computer V, pages 1–41. 2013.

[93] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons, Inc., New York, NY, USA, 1990.

[94] Ciaran McCreesh and Patrick Prosser. A parallel branch and bound algorithm for the maximum labelled clique problem. *Optimization Letters*, 9(5):949–960, 2015.

[95] B.T. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(5):493–504, 1998.

[96] Richard Myers, Richard C. Wilson, and Edwin R. Hancock. Bayesian graph edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22:628–635, 2000.

[97] Michael O. Neary and Peter R. Cappello. Advanced eager scheduling for java-based adaptive parallel computing. *Concurrency - Practice and Experience*, 17:797–819, 2005.

[98] M. Neuhaus and H. Bunke. Bridging the gap between graph edit distance and kernel machines. *Machine Perception and Artificial Intelligence.*, 68:17–61, 2007.

[99] M. Neuhaus et al. Fast suboptimal algorithms for the computation of graph edit distance. *Proceedings of 11th International Workshop on Structural and Syntactic Pattern Recognition.*, 28:163–172, 2006.

[100] N. J. Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1980.

[101] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2010. Version 3.2.

[102] B. Raphael. P. Hart, N. Nilsson. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems, Science, and Cybernetics.*, 28:100–107, 2004.

[103] Peter S. Pacheco. *Parallel Programming with MPI.* 1996.

[104] V N Rao and V Kumar. Parallel depth-first search on multiprocessors part i: Implementation. *International journal on Parallel Programming*, 16(6):479–499, 1987.

[105] Burie J. Raveaux, R. and Ogier. A graph matching method and a graph matching distance based on subgraph assignments. *Pattern Recognition Letters*, 31:394–406, 2010.

[106] Bunke H.. Riesen, K. Iam graph database repository for graph based pattern recognition and machine learning. *Pattern Recognition Letters.*, 5342:287–297., 2008.

[107] Bunke H. Riesen, K. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing.*, 28:950–959, 2009.

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

[109] Kaspar Riesen, Darko Brodić, Zoran N. Milivojević, and Čedomir A. Maluckov. *Digital Heritage. Progress in Cultural Heritage: Documentation, Preservation, and Protection: 5th International Conference*, pages 724–731. 2014.

[110] Kaspar Riesen and Horst Bunke. Improving approximate graph edit distance by means of a greedy swap strategy. In *ICISP*, pages 314–321, 2014.

[111] Kaspar Riesen et al. *Graph Classification and Clustering Based on Vector Space Embedding.* 2010.

[112] Kaspar Riesen, Stefan Fankhauser, and Horst Bunke. Speeding up graph edit distance computation with a bipartite heuristic. In *MLG*, 2007.

[113] Kaspar Riesen, Andreas Fischer, and Horst Bunke. Combining bipartite graph matching and beam search for graph edit distance approximation. In *Artificial Neural Networks in Pattern Recognition*, pages 117–128, 2014.

[114] Kaspar Riesen, Andreas Fischer, and Horst Bunke. Improving approximate graph edit distance using genetic algorithms. In *SSSPR14*, pages 63–72, 2014.

[115] Antonio Robles-Kelly and Edwin R. Hancock. A Riemannian approach to graph embedding. *Pattern Recognition*, 40(3):1042–1056, 2007.

[116] S. Sahni and T. Gonzalez. P-complete approximation problems. *journal of the Association of Computing Machinery*, 23:555–565, 1976.

[117] Olfa Sammoud, Sbastien Sorlin, Christine Solnon, and Khaled Ghedira. A Comparative Study of Ant Colony Optimization and Reactive Search for Graph Matching Problems. In *EvoCOP 2006*, pages 287–301, 2006.

[118] A. Sanfeliu and K.S. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:353–362, 1983.

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

[120] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Multimedia Computing and Networking (MMCN)*, 2002.

[121] Anna Sciomachen, Giovanni Felici, Raffaele Cerulli, Francesco Carrabs, Raffaele Cerulli, and Paolo Dell?Olmo. Operational research for development, sustainability and local economies a mathematical programming approach for the maximum labeled clique problem. *Procedia - Social and Behavioral Sciences*, 108:69 – 78, 2014.

[122] Francesc Serratosa. Fast computation of bipartite graph matching. *Pattern Recognition Letters*, 45:244–250, 2014.

[123] Francesc Serratosa. Speeding up fast bipartite graph matching through a new cost matrix. *IJPRAI*, 29(2), 2015.

[124] Francesc Serratosa and Xavier Cortés. *Structural, Syntactic, and Statistical Pattern Recognition: Joint IAPR International Workshop*, chapter Edit Distance Computed by Fast Bipartite Graph Matching, pages 253–262. 2014.

[125] Masashi Shimbo and Toru Ishida. Controlling the learning process of real-time heuristic search. *Artificial Intelligence*, 146(1):1 – 41, 2003.

[126] Bretzner L. Macrini D. Fatih Demirci M. Jnsson C. Shokoufandeh, A. and S. Dickinson. The representation and matching of categorical shape. *Computer Vision and Image Understanding*, pages 139–154, 2006.

[127] Montek Singh, Amitabha Chatterjee, and Santanu Chaudhury. Matching structural shape descriptions using genetic algorithms. *Pattern Recognition*, 30(9):1451 – 1462, 1997.

[128] Alok Sinha. Client-server computing. *Commun. ACM*, 35(7):77–98, 1992.

[129] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, 1998.

[130] Sébastien Sorlin and Christine Solnon. Reactive tabu search for measuring graph similarity. In *GBR2015*, pages 172–182, 2005.

[131] Alessandro Sperduti, Ro Sperduti, and Antonina Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8:714–735, 1997.

[132] P. N. Suganthan. Attributed relational graph matching by neural-gas networks. *IEEE Signal Processing Society Workshop on Neural Networks for Signal Processing X*, pages 366–374, 2000.

[133] E. Taillard. Benchmarks for basic scheduling problems. *http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html.*

[134] Andrew S. Tanenbaum and Robbert Van Renesse. Distributed operating systems. *ACM Comput. Surv.*, 17:419–470, 1985.

[135] Crainic Teodor Gabriel, Le Cun Bertrand, and Roucairol. Catherine. Parallel branch-and-bound algorithms. pages 1–28. 2006.

[136] Andrea Torsello and Edwin R. Hancock. Computing approximate tree edit distance using relaxation labeling. *Pattern Recognition Letters*, 24(8):1089 – 1097, 2003.

[137] Wen-hsiang Tsai, Student Member, and King-sun Fu. Pattern Deformational Model and Bayes Error-Correcting Recognition System. *IEEE Transactions on Systems, Man, and Cybernetics*, 9:745–756, 1979.

[138] S. Umeyama. An eigendecomposition approach to weighted graph matching problems. *IEEE Pattern Anal. Mach. Intel*, pages 695–703, 1988.

[139] UPC Consortium. Upc language specifications, v1.2. Tech Report LBNL-59208, 2005.

[140] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

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

[142] J. Kittler W. Christmas and M. Petrou. Structural matching in computer vision using probabilistic relaxation. *IEEE Trans. PAMI,*, 2:749–764, 1995.

[143] Robert A. Wagner and Roy Lowrance. An extension of the string-to-string correction problem. *J. ACM*, 22:177–183, 1975.

[144] P. Wang, V. Eglin, C. Garcia, C. Largeron, J. Llads, and A. Forns. A novel learning-free word spotting approach based on graph representation. In *Document Analysis Systems (DAS), 2014 11th IAPR International Workshop*, pages 207–211, 2014.

[145] Y. Wang and N. Ishii. A genetic algorithm and its parallelization for graph matching with similarity measures. *Artificial Life and Robotics*, 2:68–73, 1998.

[146] Tom White. *Hadoop: The Definitive Guide*. first edition edition, 2009.

[147] R.C. Wilson and E.R. Hancock. Structural matching by discrete relaxation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(6):634–648, 1997.

[148] Richard C Wilson, Edwin R Hancock, and Bin Luo. Pattern vectors from algebraic graph theory. *IEEE transactions on pattern analysis and machine intelligence*, 27:1112–1124, 2005.

[149] Stephen Wolfram. *The Mathematica Book*. Wolfram Media, Incorporated, 5 edition, 2003.

[150] Chengzhong Xu and Francis C. Lau. *Load Balancing in Parallel Computers: Theory and Practice.* Kluwer Academic Publishers, 1997.

[151] K BHOSLIB Xu. Benchmarks with hidden optimum solutions for graph problems. *http://www.nlsde.buaa.edu.cn/?kexu/benchmarks/graph-benchmarks.htm.*

[152] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 1–14, 2008.

[153] M. Zaslavskiy, F. Bach, and J.P. Ver. A path following algorithm for the graph matching problem. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31:2227–2242, 2009.

[154] Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. Comparing stars: On approximating graph edit distance. *Proc. VLDB Endow.*, 2:25–36, 2009.

[155] Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. Comparing stars: On approximating graph edit distance. *PVLDB*, 2(1):25–36, 2009.

[156] Weixiong Zhang. Complete anytime beam search. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, pages 425–430, 1998.

[157] Feng Zhou and Fernando De la Torre. Factorized graph matching. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 127–134, 2012.

[158] Rong Zhou and Eric A. Hansen. Multiple sequence alignment using anytime A*. In *Eighteenth National Conference on Artificial Intelligence*, pages 975–976, 2002.

[159] S Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73, 1996.

[160] Shlomo Zilberstein and Stuart J. Russell. Approximate reasoning using anytime algorithms. In *Imprecise and Approximate Computation*, pages 43–43. 1995.