UNIVERSITÉ DE TOURS

ÉCOLE DOCTORALE MIPTIS

Laboratoire d'Informatique Fondamentale
et Appliquée de Tours (EA 6300)

THÈSE présentée par :

Mostafa DARWICHE

pour obtenir le grade de : Docteur de l'université de Tours

Discipline/ Spécialité : INFORMATIQUE

When Operations Research meets Structural Pattern Recognition: on the solution of Error-Tolerant Graph Matching problems

Acknowledgment

There is a known controversial saying that says “Behind every successful man there is a woman”. Regardless of what I think, I will borrow it and make it “Behind every successful thesis there are talented supervisors”. It is at the moment of writing the acknowledgment, where I look back over the three years of the Ph.D. and I realize that I had my back covered and I was never alone. I am deeply grateful and thankful for all the help and the support that I got from my dear supervisors: Pr. Vincent T’Kindt, Dr. Romain Raveaux and Dr. Donatello Conte. It is thanks to you that I was able to reach the end of the thesis. To Vincent, you were there from the first until the last day, all I can say is thanks and I hope that I lived up to your expectations. To Romain, it has been a pleasure working with someone passionate like you and thank you for all your kind support and guidance. To Donatello, I thank you for being there all the time ready to listen and to discuss. I truly admired working with each one of you and I can say you taught me a lot and on many levels.

I also want to thank Pr. Christine Solnon and Pr. Federico Della Croce, first for accepting to be part of the jury and second for taking the time to read the thesis. As well, thanks to Pr. Sebastien Adam and Pr. Michel Habib and I am honored to have you in the jury of my thesis.

Special thanks to ROOT and RFAI teams which I was part of, and to all the professors and the colleagues.

Finally, I hope our paths will cross in the future.

Abstract

This thesis is focused on *Graph Matching* (GM) problems and in particular the *Graph Edit Distance* (GED) problems. There is a growing interest in these problems due to their numerous applications in different research domains, e.g. biology, chemistry, computer vision, etc. However, these problems are known to be complex and hard to solve, as the GED is a $\mathcal{NP}$-hard problem. The main objectives sought in this thesis, are to develop methods for solving GED problems to optimality and/or heuristically. *Operations Research* (OR) field offers a wide range of exact and heuristic algorithms that have accomplished very good results when solving optimization problems. So, basically all the contributions presented in thesis are methods inspired from OR field. The exact methods are designed based on deep analysis and understanding of the problem, and are presented as *Mixed Integer Linear Program* (MILP) formulations. The proposed heuristic approaches are adapted versions of existing MILP-based heuristics (also known as matheuristics), by considering problem-dependent information to improve their performances and accuracy.

The first contribution consists in a new and adapted *Local Branching* (LocBra) matheuristic, designed to solve a sub-problem of the general GED problem. This heuristic is based on solving small MILP formulations to perform local searches in defined neighborhoods. Mechanisms such as neighborhood definition, intensification and diversification, are all modified to consider problem/instance-dependent information. Experimentally, this heuristic has proven to outperform existing good heuristics and its capacity in computing near optimal solutions. More application-oriented experiments are executed, where their results have successfully shown this version of local branching is very suitable for real applications.

The second contribution is focused on providing a good MILP formulation that performs better than existing ones and deals with the general GED problem. After several attempts, a new formulation, denoted by F3, is proposed with a nice feature that its constraints are independent from the number of edges in the graphs. The results of the experiments have shown that this formulation is better than the best existing one, especially on dense and very dense graphs.

Since a new and good formulation has been founded, the LocBra heuristic designed in the first contribution, is then modified to use F3, so the heuristic can deal with the general GED problem. This is the third main contribution. Evaluating the heuristic on different reference databases, has shown very positive results when compared to existing heuristics.

After accomplishing good results with LocBra matheuristic, another matheuristic, called *Variable Partitioning Local Search* (VPLS) is considered, following the same concept as in LocBra. An adapted VPLS is designed based on extracting problem-dependent information to solve the GED problem. It is also based on F3 formulation. The proposed mechanism to extract useful information turned out to be very interesting and helped improving the performance of the heuristic.

Other contributions and realizations are mentioned in the thesis, that were essentials and helped a lot in accomplishing the aforementioned contributions.

Keywords: Pattern Recognition, Operations Research, Graph Matching, Graph Edit Distance, Mixed Integer Linear Program, Matheuristic, Local Branching, Variable Partitioning Local Search.

Contents

List of Tables

List of Figures

Abbreviations and notations

Abbreviations used along the manuscript

GM	Graph matching
EGM	Exact graph matching
ETG	Error-tolerant graph matching
Matching and assignment	Compute the correspondences between vertices
PR	Pattern recognition
MILP	Mixed integer linear program
ILP	Integer linear program
LP	Linear program
GED	Graph edit distance
QAP	Quadratic assignment problem
BP	Bipartite graph matching
FBP	Fast BP
SFBP	Square FBP
SBPBeam	Sorted BP-beam
IPFP	Integer projected fixed point
GNCCP	Graduated non convexity and concavity procedure
GED^{EnA}	Graph edit distance, edges no attributes
JH	Justice and Hero (2006) MILP formulation
F1	Fist MILP formulation by Lerouge et al. (2017)
F2	Second MILP formulation by Lerouge et al. (2017)
B&B	Branch-and-bound
LS	Local search
ILS	Iterated local search
VLS	Variable neighborhood search
GA	Genetic algorithms
VPLS	Variable partitioning local search

Notations used along the manuscript

$G = (V, E, \mu, \zeta)$	An attributed graph
$G = (V, E)$	An unattributed graph
u_i or i	Refers to vertices in a graph
$e = (i, j)$	Refers to an edge in a graph
$\mu(i)$	Returns the attributes assigned to the vertex
$\zeta(e)$	Returns the attributes assigned to the edge
$d(G, G')$	Stands for the distance between the graphs
$[a_{ij}]$	Refers to a matrix
$c(.)$	Refers to a function

Chapter 1

Introduction

Context

A very convenient and efficient way to model objects and patterns is to use graph-based representations. Graphs provide a satisfactory structural representation of an object, by defining the main components that form the object using vertices, and drawing the relations between them using edges. More information and characteristics can be stored in the graph by assigning labels/attributes to vertices and edges. These attributes can be of numerical or nominal type (or both), giving more flexibility to graphs. Due to the importance of graphs, there is a whole field of study devoted for graphs in mathematics, known as *graph theory*. It turned out to be very suitable for numerous practical problems that come from different field, such as biology, chemistry, neuro-science, computer vision and computer science. In *Pattern Recognition* (PR) field, graph-based representations appear to have a great interest, because they form the core of structural pattern recognition. The ultimate target in this field is to recognize objects and patterns inside images or videos, but to do so these objects must be modeled and represented somehow. Graphs, by definition, are flexible and do not have restrictions on the number of vertices/edges, and the attributes may be of different sizes and types. Moreover, graphs are resilient to rotation and translation of the objects inside the images, which makes them the potent representation of objects. But a question arises at this point which is: how to compare the graphs?

Graph Matching (GM) problems are there to answer this question. Solving these problems provide a way to compare graphs, by computing a similarity or a dissimilarity measure between two graphs. This measure is computed after determining the correspondences between vertices and edges of the graphs. Typically, the matching can be done in different forms: strictly match the graphs (same number of vertices/edges, same structure), or tolerate some differences in the graphs. Therefore, GM problems are split into two main categories (Conte et al., 2004): *Exact Graph Matching* (EGM) and *Error-Tolerant Graph Matching* (ETGM). The second form is more familiar, because the matching must be tolerant to the differences in the topology and attributes of the graphs, since it is unlikely to have isomorphism between graphs in real-life scenarios.

A well-known problem that belongs to the category of ETGM problems is the *Graph Edit Distance* (GED) problem. Solving this problem implies minimizing a dissimilarity measure that stands for the cost needed to transform one graph into another through a series of edit operations (Bunke and Allermann, 1983). The available edit operations are substitution, insertion and deletion for vertices or edges, and a cost is associated to each operation. The dissimilarity measure is then defined by the sum of the costs of the edit operations realized. In the past years, the GED problem has gained more attention, mainly because it has been shown to generalize other GM problems such as maximum common subgraph, graph and subgraph isomorphism (Bunke, 1997, 1999). However, and since graphs are flexible and can be large and complex, in the case of modeling complex patterns and objects, solving to optimality the GED problem becomes difficult and intractable in practice. In fact, GED has been proven to be in the class of $\mathcal{NP}$-hard problems (Zeng et al., 2009). Many algorithms have been proposed to solve the problem. Most of them are heuristic approaches to compute sub-optimal solutions, due to the high complexity of the problem. Few exact approaches exist to solve the problem to optimality, but they fail to deal with big instances of graphs. Some of the exact approaches are *Mixed Integer Linear Program* (MILP) formulations that turn out to be more efficient than *Branch & Bound* (B&B) based methods. On the other hand, the heuristic approaches are more focused on the speed, so they converge fast towards feasible solutions. Of course, there is a compromise here between the speed of the heuristic and the quality of the solution. This is the big dilemma in the GED problem, the ability to develop a heuristic that is fast enough but at the same time guarantee computing close-to-optimal solutions.

The best place to find answers and ideas when it comes to complex and $\mathcal{NP}$-hard optimization problems, is the *Operations Research* (OR) field. OR touches many domains and disciplines, notably industrial engineering and operations management, and computer science. Optimization problems and their complexities are well established in OR, where also various methods and techniques can be found to model optimization problems, such as mathematical programming. Mathematical programming provides techniques to express optimization problems mathematically in the form of *Linear Programming* models, MILP models, etc. These models can, then, be solved to optimality using existing powerful black-box solvers, e.g. CPLEX, Gurobi and Xpress. OR, as well, provides very advanced heuristic techniques, metaheurstics and matheuristics to solve hard optimization problems.

Positioning of the Thesis

Somehow to enforce the relations between PR and OR fields, and to prove that OR can offer very good techniques to solve PR optimization problems, this thesis is going to be focused on developing solution techniques to solve the GED problem that are inspired from OR methods. When reviewing the state-of-the-art exact methods of the GED problem, it can be seen that the best methods are the MILP formulations, in particular JH (Justice and Hero, 2006) and F2 (Lerouge et al., 2017) formulations. They perform better than B&B-based methods (Lerouge et al., 2017). It will be, then, interesting to develop new MILP formulations that can compete with the existing ones. And why MILP formulations? The answer is simply because MILP solvers capability in solving those formulations is increasing quickly. The solvers nowadays are able to solve bigger and harder instances. Hence, if designing a good MILP formulation and solving it today with MILP solvers, yield good results, then better results are expected in the next coming years without the need to modify the formulation because the solvers are going to be improved.

Additionally, OR can potentially contribute to heuristic methods. Many good and fast heuristics already exist in the literature to solve the GED problem (e.g. Bipartite Graph Matching (Riesen et al., 2007a), SBPBeam (Ferrer et al., 2015), IPFP (Bougleux et al., 2017), etc.). Some of these heuristics are based on metaheuristic approaches, such as beam-search based methods (BeamSearch (Neuhaus et al., 2006) and SBPBeam (Ferrer et al., 2015)). Beam-search is a very basic metaheuristic approach, where in OR more complex and effective ones exist that are not yet tried on the GED problem. One could think of MILP-based heuristics, which are also known as matheuristics. Matheuristics combine MILP formulations, MILP solvers and heuristic techniques (e.g. neighborhood exploration, diversification, etc.) in a procedure that aims at computing near-optimal solutions. They have shown great capability in solving optimization problems. Such techniques have not yet been tested on the GED problem.

The focus of this thesis is summarized in the following two points:

- Developing exact approaches based on MILP formulations to solve the GED problem, that are better than the existing methods. This gives a push to the state-of-the-art exact methods, by scaling up to bigger and more complex instances.
- Design matheuristics to solve the GED problem, that can live up to the performance of existing heuristics. Also, they must assure a good trade-off between speed and accuracy, so they can be used in final GED applications.

The contributions of this thesis, if good results are achieved, may be very important because they will benefit two research communities: PR community in introducing new solution methods like matheuristics, which has not yet been done. As well, OR community in bringing attention of researchers to GM and GED problems that have a wide range of applications in machine learning and patter recognition fields.

Outline

The thesis is organized in two main parts:

I- The first part consists of two chapters:

- Chapter 2 covers the state-of-the-art of the GM and GED problems. It starts with a panoramic review to GM problems and categories, plus the general sub-problems in each category. Next, it gives a detailed review to the GED problem (definition, applications, challenges). Later, it presents the most known and efficient exact and heuristic methods for solving the GED problem. Finally, it summarizes all the reviews, by highlighting the problematic of existing methods, and declaring the important contributions sought by this thesis.
- Chapter 3 is dedicated to review the basics of OR. First, it discusses the importance of OR techniques in modeling and solving optimization problems. It

shows how OR studies and classifies optimization problems based on their complexities. Next, it provides a review to the most known optimization methods, including exact and heuristic approaches. Lastly, it concludes with shedding the light on potential matheuristic methods and the possibility of applying them to solve the GED problem.

II- The second part covers the contributions and it is divided into two chapters:

- Chapter 4 handles the GED^{EnA} (Edges no Attributes) sub-problem of the main GED problem. It discusses the importance of making such distinction. Also, it provides a experimental comparison to existing MILP formulations, with and without the pre-processing technique adopted from OR field. Next, based on the analyses and the results of the experiments, it presents a proposition of an adapted local branching matheuristic to solve the GED^{EnA} problem. The results of exhaustive general and application-oriented experiments are presented afterwards. This chapter ends with concluding remarks and summarizing all the contributions.
- Chapter 5 is dedicated to the GED problem. It is split into two main sections: propositions of MILP formulations and propositions of adapted local branching and variable partitioning local search matheuristics. It provides experimental studies and evaluations to all the proposed methods. Finally, it sums up all the contributions and comments on the results achieved.

At last, a general conclusion summarizing all the contributions and works presented in the thesis, is given in Chapter 6. This chapter ends with perspectives and possible directions for future research.

Part I

State of the art

Chapter 2

Graph Matching and Graph Edit Distance problems: State of the art

Contents

2.1 Graph definitions, types and applications

A graph is a structural representation that has different types and classes. There is a lot of history to graphs and their applications that has started a long time ago. They have gained popularity in the field of mathematics at the beginning and then spread to computer science. With that, many challenges have come across and a new section/field of study, known as *graph theory*, has started. In this section, the main definitions and types of graphs are given, followed by the different classes of graphs and, then applications in which they appear.

2.1.1 Graph definitions and types

Here is a review of the common types of graphs that can be encountered in fields of mathematics and computer science.

Graph. A graph is a mathematical structure that models pairwise relations between abstract objects. Those objects are called *vertices* and the relations between them are expressed by *edges*. A basic and arbitrarily structured graph mainly consists of two sets: vertices V and edges E.

Definition 1 (Graph)**.** *A graph $G = (V, E)$ is such that:*
V is the set of vertices,
$i \in V$ denotes a vertex,
E is the set of edges, with $E \subseteq V \times V$,
and $e = (i, j) \in E$ refers to an edge, with $i, j \in V$.

Subgraph. A graph G_s is said to be a subgraph of G if V_s and E_s are, respectively, parts of V and E. A subgraph is explicitly considered induced, which means that every edge $e \in E_s$ is an edge if and only if e is in E.

Definition 2 (Subgraph)**.** *A subgraph $Q = (V_s, E_s)$ is such that:*
$V_s \subseteq V$,
and $E_s \subseteq E \cap V_s \times V_s$.

Directed and undirected graphs. A graph is explicitly undirected unless it is said to be directed. In the former case, writing $e = (i, j)$ or $e = (j, i)$ does not matter, because (i, j) is the same edge as (j, i). However, in a directed graph $(i, j) \neq (j, i)$. Definition 1 is valid for undirected graphs, while Def. 3 below introduces directed graphs.

Definition 3 (Directed graph)**.** *A graph $G = (V, E)$ is such that:*
V is the set of vertices,
$i \in V$ denotes a vertex,
E is the set of edges, with $E \subseteq V \times V$,
$e = (i, j) \in E$ refers to an edge,
and $(i, j) \neq (j, i)$

Attributed graph. One advantage to use graphs when representing patterns and objects is that more information and characteristics can be stored and assigned to vertices and edges. Such information are called *attributes* or *labels*. The attributes can be of type numerical, i.e. $L = \mathbb{R}^n$, or nominal i.e. $L = \{\alpha, \beta, \gamma, ...\}$ or even a combination of both.

Definition 4 (Attributed graph)**.** *An attributed graph is a 4-tuple $G = (V, E, \mu, \xi)$, with:*
V the set of vertices,
$E \subseteq E \cap V \times V$ the set of edges,
$\mu : V \to L_V$ the function that assigns attributes to vertices,
$\xi : E \to L_E$ the function that assigns attributes to edges,
L_V the set of all possible attributes for vertices,
and L_E the set of all possible attributes for edges.

Unattributed graph Unattributed graphs are graphs in which vertices and edges have no attributes. It can be seen as a particular case where $L_V = L_E = \{\phi\}$, based on Def. 4.

2.1.2 Graph classes

There are many defined classes of graphs based on shared properties and structures. Graphs belonging to such classes are considered as special cases and are explored in graph

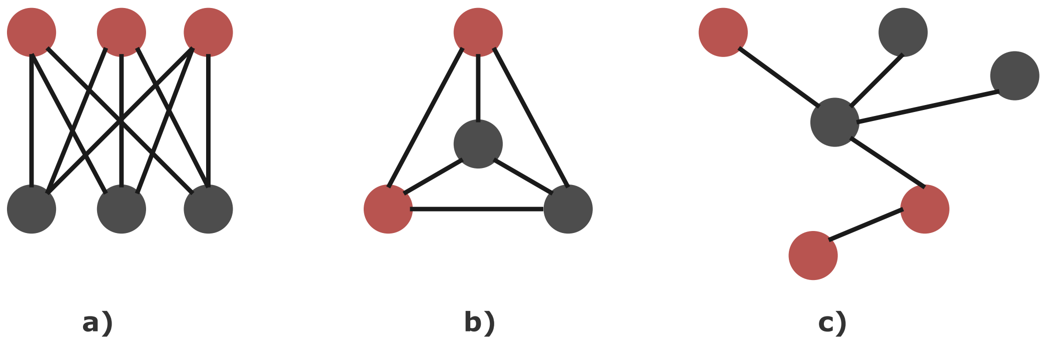


Figure 2.1: Example of graphs: a) bipartite, b) planar and c) tree

theory field to model various mathematical and geometrical problems. Here is a list of common classes.

Regular graphs. G is said a regular graph if all vertices $i \in V$ have the same degree. The degree of a vertex is the number of edges emanated from i.

Simple graphs. G is a single graph if it has no loops and no multi-edges. That is, there is no edge that starts and ends with the same vertex and no two edges connecting the same vertices.

Complete graphs. A complete graph is a graph in which every two vertices are connected by an edge. The number of edges is $\frac{1}{2} \times |V| \times (|V| - 1)$ if the graph is undirected and $|V| \times (|V| - 1)$ otherwise.

Connected graphs. A graph is said connected if there exists a path between each pair of vertices. In the case of directed graphs, if there exist two paths (in both directions) between each pair of vertices, then the graph is strongly connected. A path in a graph is a sequence of edges connecting multiple distinct vertices.

Bipartite graphs. A graph is said to be a bipartite graph in case the set of vertices can be split into two sets V_1 and V_2, such that every edge in the graph connects one vertex in V_1 with another vertex in V_2 ($V_1 \cap V_2 = \{\phi\}$). There are no edges between the vertices of each set. Figure 2.1-a shows an example of bipartite graph.

Planar graphs. If a graph can be drawn in a plane in a such a way, there are no intersections between edges, then it is a planar graph. An example of this graph class is shown in Figure 2.1-b.

Cycle graphs. Cycle graphs are graphs in which there exists one or more vertices where a closed walk can be done. The walk starts at vertex i and ends at the same vertex. A walk is defined as a sequence of alternating vertices and edges such as $v_0, e_1, v_1, e_2, ..., e_k, v_k$ where each edge $e_i = \{v_i - 1, v_i\}$. The length of this walk is k.

Tree. A tree is both an acyclic (cycle free) and connected graph. The graph in Figure 2.1-c depicts an example of a tree.

Weighted graphs. Such graph is a special case of the attributed graph. Instead of assigning multiple attributes to every edge, only one numeric attribute is associated to edges i.e. $\xi : E \rightarrow \mathbb{R}$.

2.1.3 Graph applications

There is a whole field of study devoted for graphs in mathematics, known as *graph theory*. Since graphs are powerful in modeling structural relations, they are applied to several practical problems. These problems come from different fields, such as:

- **Biology**: in the simple case, graphs can model proteins and enzymes, where vertices represent aminoacids and edges draw the adjacencies between them. In a bigger picture, they model networks of relations between proteins of different species. The vertices in such graphs represent proteins, and the edges represent interactions, which depict the activities of different proteins (Carletti et al., 2013). The task is then to determine groups of proteins that perform similar activities.

- **Chemistry**: graphs (attributed graphs in particular) form a natural representation of the atom-bond structure of chemical molecules. Each vertex in the graph represents an atom, while an edge represents a molecular bond (Raymond and Willett, 2002). Then, a common problem is to compare and find similarities between complex molecules.

- **Computer science**: the most famous field involving graphs. They are used to model communication networks, big data organization, websites link structure, road maps, social media, and many others (Noel and Jajodia, 2005; Grandjean, 2016; Wu et al., 2014). Some of common problems in this field: vertex cover, maximum clique, shortest path between two vertices, graph matching, etc.

- **Computer vision**: attributed graphs are used widely in this field to perform mainly Pattern Recognition tasks, such as object recognition and detection (Wiskott et al., 1997), object tracking Gomila and Meyer (2003), supervised and unsupervised classifications (Raveaux et al., 2007), etc. It is one the fields dedicated to solve graph-based problems.

- **Physics**: a graph is used to model the interaction between parts of a system like quantum mechanical motion of electrons (Estrada, 2013). The problem of graph coloring appears in this field.

- **Neuro-science**: graphs are called *Bayesian networks* and they represent the inference between brain neurons (Lee and Mumford, 2003). Graphs help visualizing and determining the active neurons or clusters of neurons.
- Other fields that use graphs such as mathematics, sociology, computational linguistics.

This list is short and does not cover all the fields, even the mentioned ones employ graphs in different forms and actually involve many tasks and problems, further to the examples presented. However, it is presented briefly to show the importance of graph-based representations and the diversity of application fields.

2.1.4 Graph notations

Here are general notations that will be used later in the rest of the thesis.

Graph size. $|G|$ denotes the size of the graph and is equal to $|V|$.

Vertex degree. d_i is the degree of a vertex $i \in V$. The degree is the number of edges emanated from i.

Graph density. For a graph $G = (V, E, \mu, \xi)$, the density is computed by Equation 2.1 for undirected graphs and 2.2 for directed graphs.

$$D = \frac{2|E|}{|V|(|V|-1)} \tag{2.1}$$

$$D = \frac{|E|}{|V|(|V|-1)} \tag{2.2}$$

Adjacency matrix. The adjacency matrix is a common approach to represent a graph. It is generally a $(0,1) - matrix$ of size $|V| \times |V|$, where the presence of a 1 indicates that there is an edge connecting the two vertices placed on rows and columns of the matrix.

Definition 5 (Adjacency matrix)**.** *Let $G = (V, E, \mu, \xi)$ be a graph. The adjacency matrix $A = [a_{ij}]$ of size $|V| \times |V|$ is defined by*

$$a_{ij} = \begin{cases} 1 \textit{ if } (i,j) \in E \\ 0 \textit{ otherwise} \end{cases}$$

Incidence matrix. Another (less common) approach to represent a graph consists in using the incidence matrix, which is a $(0,1) - matrix$ of size $|V| \times |E|$. It has vertices row wise and edges column wise. The values in the matrix are set to 1 when the vertex on a row is part of the edge on a column, and 0 otherwise.

Definition 6 (Incidence matrix)**.** *Let $G = (V, E, \mu, \xi)$ be a graph. The incidence matrix $B = [b_{ij}]$ of size $|V| \times |E|$ is defined by*

$$b_{ij} = \begin{cases} 1 \text{ if vertex } i \text{ and edge } e \text{ are incident} \\ 0 \text{ otherwise} \end{cases}$$

2.2 Graph matching

2.2.1 Concept and categories

In *Pattern Recognition* (PR) field, there is a distinction between statistical and structural pattern recognition. The former consists in computing vectors of features to represent objects, while the latter uses structural data and in particular graphs. In both approaches, a common task is to come up with methods to compare two objects or an object with a defined model. Statistical methods have yielded very good results, due to the easiness of exploiting vectors to extract information and characteristics that describe and distinguish an object. However, few limitations exist with such approach: features vectors must have the same size in a defined application, and it is not evident how to model the relations between the features especially when dealing with complex objects (Riesen, 2015). On the other hand, the use of graphs overcomes those problems, because graphs are flexible by definition, i.e. there are no restrictions on the number of vertices/edges, and attributes may be of different sizes and types. Moreover, graphs are resilient to rotation and translation of objects inside images, which makes them the potent representation of objects. To this, graphs have become more known and the interest of researches in studying them has grown in the past decades.

Graph Matching (GM) (aka graph comparison) is the core of structural pattern recognition. Generally, it defines a similarity or dissimilarity measure for structural graphs. For simplicity, and since similarity and dissimilarity are related (i.e. when similarity value increases, dissimilarity value decreases), only dissimilarity term is used in the rest of the thesis. Given two graphs $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$, solving a GM problem relates to mapping vertices V to V' and edges E to E', while satisfying a set of defined topological constraints (Conte et al., 2004). The mapping/matching between two sets of vertices and two sets of edges is also known as correspondences. The dissimilarity measure can be computed after determining those correspondences. However, the constraints may or may not be flexible, depending on the goal of the comparison. Accordingly, there are two main categories of GM problems: *Exact graph Matching* (EGM) problems and *Error-tolerant graph matching* (ETGM) problems. Exact GM approach is a very strict, in the sense that when comparing two (sub-)graphs, they must be identical (e.g. number of vertices/edges, structure, attributes, etc). It is not the case in ETGM, where it is possible to map two vertices that don't carry the same attributes, and also it tolerates differences in structures e.g. extra vertices or edges. The second category is considered more general than EGM, which can be considered as a special case of ETGM. Both categories are discussed in the following sub-sections starting with explaining the different problems that fall into each category and the important methods to tackle them.

GM problems have spread to many domains and have a large number of applications. Below is a listing of such applications:

- 2D and 3D image analysis (Eshera and Fu, 1986; Suganthan et al., 1995; Perchant et al., 1999; Wilson and Hancock, 1998)
- Document processing: OCR and handwritten recognition (Rocha and Pavlidis, 1994), string recognition (Filatov et al., 1995), symbol and graphics recognition (Lladós et al., 1996)
- Biometric identification: Face authentication and recognition (Wiskott, 1997; Wiskott et al., 1997), fingerprint recognition (Maio and Maltoni, 1996)
- Image database: Indexing and retrieval (Petrakis and Faloutsos, 1997; Park et al., 1997)
- Video analysis: Annotation and retrieval from databases (Shearer et al., 2001), object tracking (Gomila and Meyer, 2001), motion estimation (Salotti and Laachfoubi, 2001)
- Biomedical and biological applications (Wang et al., 1998; Dumay et al., 1992; Fischer et al., 2002)

2.2.2 Exact graph matching

The main constraint to meet in this category of GM problems is that the mapping between vertices of graphs G and G' must be edge-preserving. Matching two couples of vertices impels matching the edges induced by them. In the case of attributed graphs, two matched vertices or edges must carry the same attributes. The matching category is very strict and require a bijective function. Numerous problems fall into this category and they are all detailed in the survery about graphs by Conte et al. (2004). Some of EGM problems are reviewed in the next sub-sections.

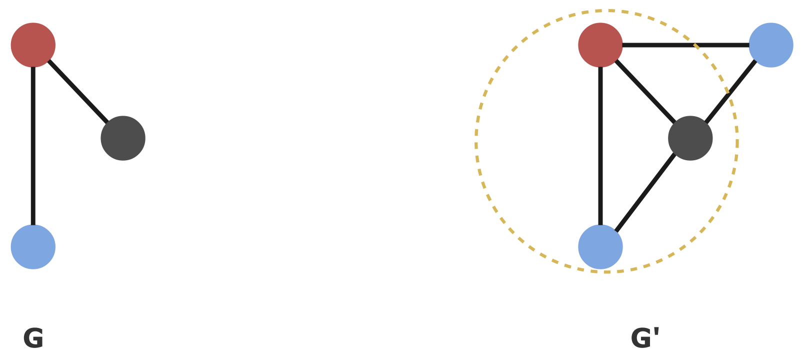


Figure 2.2: Graph G is a subgraph isomorphic to G', the subgraph is highlighted in yellow. This is not induced, because in the subgraph of G' there is an extra edge.

2.2.2.1 Graph isomorphism

It is the problem of finding the exact matching between two graphs, with the additional constrain that it is edge-preserving.

Definition 7 (Graph isomorphism). *Let $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$ be two attributed graphs. A graph isomorphism is a bijective function $\mathcal{F} : V \rightarrow V'$, satisfying:*

1. $\mu(i) = \mu'(\mathcal{F}(i))$ $, \forall i \in V$,
2. $e = (i, j) \Rightarrow f = (\mathcal{F}(i), \mathcal{F}(j)) \in E'$, *with* $\xi(e) = \xi'(f)$, $\forall e \in E$,
3. $f = (k, l) \Rightarrow e = (\mathcal{F}^{-1}(k), \mathcal{F}^{-1}(l)) \in E$, *with* $\xi'(f) = \xi(e)$, $\forall f \in E'$.

G is said isomorphic to G' if there exists a bijective function $\mathcal{F}$ satisfying the three rules above. Graph isomorphism problem is shown to be in $\mathcal{NP}$ class (Kun, 2015). However, it is known yet whether the problem is in $\mathcal{P}$ or $\mathcal{NP}$-complete. Recently there was a very interesting work by Babai (2016) that proposed a quasi-polynomial time algorithm to solve this problem. A quasi-polynomial time algorithm has a worse running time of $2^{O((log\ n)^c)}$, with $c > 1$. Such algorithm is runs slower than polynomial time, yet faster than the exponential time. The work is inspired by methods and analysis from group theory domain.

2.2.2.2 Subgraph isomorphism

The problem is similar to graph isomorphism problem, except that it looks if G is a subgraph of G' in the strict sense (G is contained in G'). It is also called *Monomorphism.*

Definition 8 (Subgraph isomorphism). *Let $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$ be two attributed graphs. A subgraph isomorphism is a injective function $\mathcal{F} : V \rightarrow V'$, satisfying:*

1. $\mu(i) = \mu'(\mathcal{F}(i))$ $, \forall i \in V$,
2. $e = (i, j)$, *there exists a* $f = (\mathcal{F}(i), \mathcal{F}(j)) \in E'$, *with* $\xi(e) = \xi'(f)$, $\forall e \in E$.

An example of monomorphism between graphs is given in Figure 2.2. The presence of an extra edge in the subgraph highlighted in yellow is acceptable and prevents it from becoming induced subgraph isomorphism. This problem is $\mathcal{NP}$-complete following the same proof of the induced case as in Garey and Johnson (1990). There exist many heuristic and approximation methods to compute the monomorphism, which can be found in (Conte et al., 2004).

2.2.2.3 Induced subgraph isomorphism

It is a stronger form of Subgraph isomorphism. It requires edge-preserving and the matched part in G' must be exactly as G, i.e. it does not permit having extra vertices or edges.

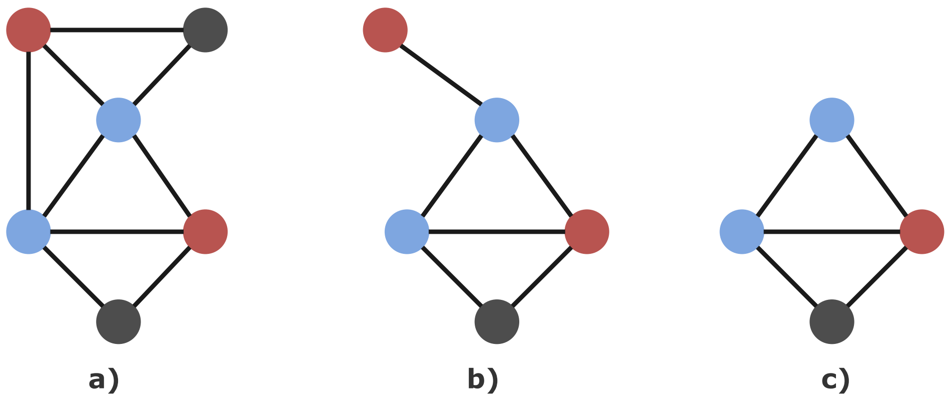


Figure 2.3: Graph (c) is the maximum common subgraph of (a) and (b)

Definition 9 (Induced subgraph isomorphism). *Let $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$ be two attributed graphs. An induced subgraph isomorphism is a injective function $\mathcal{F} : V \to V'$, satisfying:*

1. $\mu(i) = \mu'(\mathcal{F}(i)), \ \forall i \in V$,
2. $e = (i, j) \Rightarrow f = (\mathcal{F}(i), \mathcal{F}(j)) \in E'$, *with* $\xi(e) = \xi'(f), \ \forall e \in E$,
3. $f = (k, l)$, *there exists a* $e = (\mathcal{F}^{-1}(k), \mathcal{F}^{-1}(l)) \in E$, *with* $\xi'(f) = \xi(e), \ \forall f \in E'$.

This problem is $\mathcal{NP}$-complete as proved by Garey and Johnson (1990). This form of matching is even harder than graph isomorphism. There are several works proposing algorithms to solve this problem based on decision trees (Messmer and Bunke, 1999) and random walks (Gori et al., 2005). In Figure 2.3, graph (c) is an induced subgraph isomorphism for graphs (a) and (b).

2.2.2.4 Maximum common subgraph

Another stringent problem of EGM problems is the maximum common subgraph problem. It is the problem of finding the maximal part that is common in terms of vertices, edges and attributes, between two graphs.

Definition 10 (Maximum common subgraph). *Let $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$ be two attributed graphs. Graph g is common subgraph of G and G', if there exists a subgraph isomorphism from g to G and from g to G'. g is called maximum common subgraph, if there exists no bigger common subgraph.*

Figure 2.3 shows an example of a maximum common subgraph. The problem of finding the maximum common subgraph is $\mathcal{NP}$-hard (Akutsu and Tamura, 2012) and the prove has been done by reduction of a maximum clique instance to a maximum common subgraph. There were several attempts, inspired from existing algorithms that solve the maximum clique problem, in order to solve the maximum common subgraph as the works by Levi (1973) and McGregor (1982). The problem is also exploited and well studied, which has led to compute a dissimilarity measure between two graphs, once the maximum common subgraph is computed (Bunke and Shearer, 1998). The measure is defined by:

$$d_{MCS}(G,G') = 1 - \frac{|mcs(G,G')|}{max(|G|,|G'|)} \tag{2.3}$$

with $mcs(G,G')$ is the size of the maximum common subgraph g. The distance gets smaller as the maximum common subgraph gets larger. When the distance is 0, the graphs are then isomorphic.

2.2.3 Error tolerant graph matching

This is the second and most important category of GM problems. EGM problems impose rigid constraints on the matching, since it must be strict in terms of number of vertices, number of edges and having equivalent attributes. For instance, when matching a couple of vertices in the first graph to another couple of vertices in the second graph, if there is an edge between one couple then there must be an edge in the second, otherwise matching these two couples cannot be done. Moreover, to match two vertices (or edges), they must have the same attributes values. However, when graphs comes from real-life scenarios, e.g. when extracting graphs from images, they might contain extra vertices and edges. This could be caused by the presence of noise in the images, or by a poor quality of the images. In such cases, EGM will fail in comparing graphs as they are not exact. To overcome this, and perform matching between graphs in a more flexible fashion, some of EGM constraints can be relaxed. This is the family of ETGM problems, where solving an ETGM problem leads to computing a matching between graphs that tolerates some differences in the topologies and the attributes. The matching obtained, in this case, comes at a cost, which is considered as a similarity or dissimilarity score, by using a function or distance between the attributes of two vertices/edges. In the case of dissimilarity costs, the smaller the cost, the more similar the vertices are (it is the opposite in the case of similarity costs). EGM problems can be seen a special case of ETGM problems, such that the aim is to find the matching with all costs are null. Therefore, ETGM problems are a more generic form of graph matching, that is also harder than EGM problems. Allowing matching to be tolerable and flexible, increases the complexity of the problem. In addition, the matching problem, in the context of error-tolerant, has become an optimization problem, while it was a decision problem in the case of exact graph matching. The goal is find the least cost matching possible between two graphs. This category has gained more attention and there are many interesting problems defined in this context. The most important ones are reviewed in the following sub-sections.

2.2.3.1 Substitution-tolerant subgraph isomorphism

This type of GM problems is based on only substitution of vertices and edges, and it is "tolerant", so it allows differences in the attributes. However, the tolerance is limited to the attributes but sharing the same topology must be met. All vertices and edges of the first graph must be matched with other vertices and edges of the second graph, with the possibility of having differences in the attributes of the matched items. This difference is measured by a defined function or distance measure, that is called the cost of substitution. The rest of the rules and matching constraints of subgraph isomorphism are still valid in this type of matching.

Definition 11 (Substitution-tolerant subgraph isomorphism)**.** *Let $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$ be two attributed graphs. A substitution-tolerant subgraph isomorphism is a injective function $\mathcal{F} : V \to V'$, satisfying:*

1. $\mu(i) \approx \mu'(\mathcal{F}(i)),\ \forall i \in V,$
2. $e = (i, j) \Rightarrow f = (\mathcal{F}(i), \mathcal{F}(j)) \in E'$, *with* $\xi(e) \approx \xi'(f),\ \forall e \in E,$
3. $f = (k, l)$, *there exists a* $e = (\mathcal{F}^{-1}(k), \mathcal{F}^{-1}(l)) \in E$, *with* $\xi'(f) \approx \xi(e),\ \forall f \in E'.$

This problem differs from the exact subgraph isomorphism by dropping the equality between labels. They should be approximately equal, or the distance between them is very small. This problem is suitable for cases where graphs are extracted from images and there is noise introduced in graphs. This ETGM type is $\mathcal{NP}$-hard minimization problem, and there exists a Mixed Integer Linear Program (MILP) formulation proposed in the literature to solve it (Le Bodic et al., 2012).

2.2.3.2 Error-tolerant subgraph isomorphism

Error-tolerant sugraph isomorphism problem is similar to substitution-tolerant subgraph isomorphism problem, but it also enables matching two graphs with different topologies (e.g. number of vertices, number of edges, ...). Basically, in this problem, a set of dummy vertices and edges is introduced, which will be used to match vertices and edges of graph G. A vertex in G can be then matched with a vertex in G' or a dummy vertex, depending on the (smallest) matching cost. So, in this problem, there is not only substitution as in the first one, but also deletions, which are represented by matching a vertex/edge with dummy vertice/edges.

Definition 12 (Error-tolerant subgraph isomorphism)**.** *Let $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$ be two attributed graphs. An error-tolerant subgraph isomorphism is a injective function $\mathcal{F} : V \cup \Delta_V \to V' \cup \Delta_{V'}$, satisfying:*

1. $\Delta_{V'}$ *is the set of dummy vertices,*
2. $\Delta_{E'}$ *is the set of dummy edges,*
3. $\mu(i) \approx \mu'(\mathcal{F}(i)),\ \forall i \in V, \mathcal{F}(i) \in V' \cup \Delta_{V'},$
4. $e = (i, j) \Rightarrow f = (\mathcal{F}(i), \mathcal{F}(j)) \in E' \cup \Delta_{E'}$, *with* $\xi(e) \approx \xi'(f),\ \forall e \in E$
5. $f = (k, l)$, *there exists a* $e = (\mathcal{F}^{-1}(k), \mathcal{F}^{-1}(l)) \in E$, *with* $\xi'(f) \approx \xi(e),\ \forall f \in E' \cup \Delta_{E'}.$

This problem is a $\mathcal{NP}$-hard minimization problem, which requires a matching cost to be defined. The algorithms designed to tackle the problem are denoted as error-correcting or error-tolerant matching, and such an algorithm can be found in (Messmer and Bunke, 1998).

2.2.3.3 Error-tolerant graph isomorphism

Error-tolerant graph isomorphism problem is a generalization of the error-tolerant problem, where the whole graphs must be matched and there are dummy vertices and edges on both sides. It is completely tolerant to graphs: structures, topologies and the attributes on vertices and edges.

Definition 13 (Error-tolerant graph isomorphism)**.** *Let $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$ be two attributed graphs. An Error-tolerant graph isomorphism is a injective function $\mathcal{F} : V \to V' \cup \Delta_{V'}$, satisfying:*

1. *Δ_V and $\Delta_{V'}$ are the set of dummy vertices,*
2. *Δ_E and $\Delta_{E'}$ is the set of dummy edges,*
3. *$\mu(i) \approx \mu'(\mathcal{F}(i))$, $\forall i \in V \cup \Delta_V, \mathcal{F}(i) \in V' \cup \Delta_{V'}$,*
4. *$e = (i, j) \Rightarrow f = (\mathcal{F}(i), \mathcal{F}(j)) \in E' \cup \Delta_{E'}$, with $\xi(e) \approx \xi'(f)$, $\forall e \in E \cup \Delta_E$,*
5. *$f = (k, l)$, there exists a $e = (\mathcal{F}^{-1}(k), \mathcal{F}^{-1}(l)) \in E \cup \Delta_E$, with $\xi'(f) \approx \xi(e)$, $\forall f \in E' \cup \Delta_{E'}$.*

This is one of the most general form of error-tolerant GM problems, which turns out to be $\mathcal{NP}$-hard. The problem has been studied in the literature and there exists a heuristic algorithm to solve it by Bunke and Allermann (1983). Other heuristic approaches can be found in the literature as well. Figure 2.4 depicts an example of error-tolerant GM.

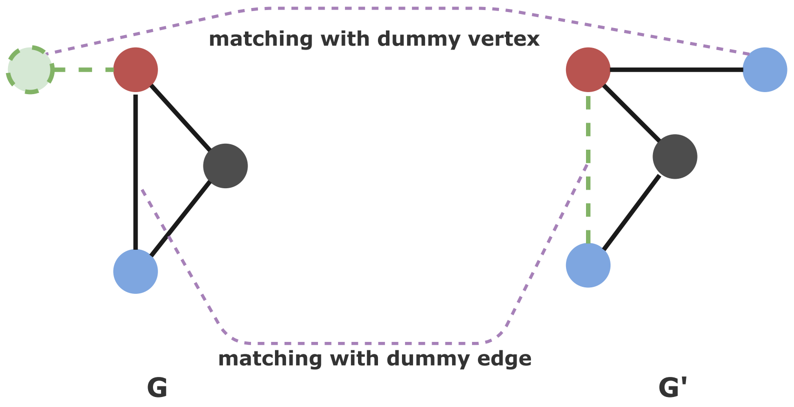


Figure 2.4: Error-tolerant graph isomorphism example. The green dashed lines and circles are the dummy vertices and edges added to graphs. The left unmatched vertex in G' is matched with the dummy vertex in G.

2.2.3.4 Graph edit distance

In the same spirit, the *graph edit distance* (GED) problem is an error-tolerant graph matching problem (Sanfeliu and Fu, 1983; Bunke and Allermann, 1983). Solving the GED problem leads to computing a distance or dissimilarity measure between two graphs. It is tolerant, without any restrictions to differences in topologies, structures and attributes on vertices and edges. GED consists in finding the minimum cost needed to transform one graph into another, through a series of edit operations. The possible edit operations are substitution, insertion and deletion of vertices or edges, with a cost associated to each operation. The GED problem has been proven to be very flexible and can be seen as a generalization to other graph matching problems, by modifying some properties (Bunke, 1997). Furthermore, it is able to cope with any type of graphs: attributed or non-attributed, directed or undirected. The problem has been well studied by many researchers and has been involved in many application domains. For all these reasons, GED is considered the most important and versatile problem in ETGM. The section 2.3 is dedicated to the GED problem, starting by defining the problem, the application domains and then the most efficient methods to solve it. Solving the GED problem is the subject of this thesis.

2.2.3.5 Multivalent matching

This is a general type of matching and less common than the above matching types, though it has interesting applications in graph theory (Champin and Solnon, 2003). The multivalent matching can be emerged in both exact or error-tolerant matching. It rather permits matching one vertex in G with one or multiple vertices in G'. More generally, multivalent matching can be of the following types:

- One to many,
- Many to one,
- Many to many.

This problem involves special operations such as vertex merging and splitting, with a constraint of non-overlapping between the matched group of vertices. To formalize the multivalent matching, a relation m, that associates a vertex to multiple vertices, is introduced.

Definition 14 (Multivalent matching)**.** *Let $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$ be two attributed graphs. A relation $m \subset V \times V'$ is a multivalent matching between G and G', if:*

1. $m(i) = \{k \in V' : (i, k) \in m\}, \forall i \in V,$
2. $m(k) = \{i \in V : (i, k) \in m\}, \forall k \in V'.$

The work by Sorlin et al. (2007) proposes a generic distance measure based on multivalent matching. As well, in the same reference this problem is proven to be $\mathcal{NP}$-hard, by applying reduction from the GED problem, which is also $\mathcal{NP}$-hard.

Table 2.1: Summary of GM problems and their complexities

	Matching problem	Complexity
Exact GM	Graph isomorphism	$\mathcal{NP}$
	Subgraph isomorphism	$\mathcal{NP}$-complete
	Induced subgraph isomorphism	$\mathcal{NP}$-complete
	Maximum common subgraph	$\mathcal{NP}$-hard
Error-tolerant GM	Substitution-tolerant subgraph isomorphism	$\mathcal{NP}$-hard
	Error-tolerant subgraph isomorphism	$\mathcal{NP}$-hard
	Error-tolerant graph isomorphism	$\mathcal{NP}$-hard
	Graph edit distance	$\mathcal{NP}$-hard
	Multivalent matching	$\mathcal{NP}$-hard

2.2.4 Summary and prospects for moving forward

So far, most of the important GM problems that can be found in the literature, have been reviewed by giving their definitions and complexities. The problems are numerous and each one has its own properties and restrictions for matching graphs. Table 2.1 sums up the problems and their complexities. Most of them are $\mathcal{NP}$-hard problems, which means that finding an optimal solution is difficult and may not be possible in reasonable time. Despite their importance, dealing with such problems is a serious challenge.

The present thesis's main objective is to study and develop methods to solve ETGM problems. It is meant then to deal specifically with ETGM problems and not EGM. Those problems are more general and less restrictive: they tolerate differences in the topology of the graphs and the attributes. Such tolerance helps accommodating to deformations, presence of extra vertices, wrong feature values in the graphs. They appear due to the existence of noise or non-deterministic elements during the acquisition process of the graph. Another important factor to favor ETGM problems is that they cover the exact problems because they can compute the exact matching if it is possible to have it. But of course, this tolerance increases the complexity of the problem.

Among the ETGM problems, there is the GED problem that is more appealing and most encountered in the literature. Of course, there are convincing reasons as to why this problem is important, which are:

- The GED problem was studied and applied to many application fields, such as: Pattern Recognition, Chem and Bio-informatics, Knowledge and Process Management, etc. There is a taxonomy of the applications detailed by Stauffer et al. (2017). The applications are discussed in details in the following sections.
- The GED problem is considered as the most important problem in the structural pattern recognition approaches. The book by Riesen (2015) discusses in details and provide a comparison between statistical and structural pattern recognition.
- The GED problem is considered as a distance measure and called the dissimilarity measure between graphs. It computes the number and the strength of distortions that have to be applied to transform one graph into a another.

- The GED problem is very flexible and can operate with any kind of costs functions. In particular, if the costs on edit operations satisfy the properties of metric space, then the distance computed by solving the GED problem can be considered as a metric between graphs. The properties of a metric space are: non-negativity, identity of indiscernibles and triangle inequality.

- The GED problem has been shown to be a generalization of other graph matching problems (e.g. maximum common subgraph or graph and subgraph isomorphism), by simply changing the cost metric properties. This was shown by Bunke (1997, 1999). Then, developing techniques and methods to solve this problem will help in contributing to other matching problems.

All these reasons have led to select the GED as the ETGM problem that this thesis will be focusing on. The objectives are mainly to study it in particular, in order to extract important properties to help developing efficient methods. The methods can be either exact i.e. they provide optimal solutions at the cost of an exponential time, or heuristics i.e. they compute sub-optimal solutions in reasonable time. The rest of this chapter is devoted to explain the GED problem and to review the most important and recent methods introduced in the literature.

2.3 Graph Edit Distance

This section discusses the graph edit distance problem in details. The definition of the problem is given first, followed by the application fields where it is involved. Next, a review of the most important methods that solve the problem, by distinguishing two main categories: exact and heuristic methods.

2.3.1 Problem concept and definition

The graph edit distance is, by definition, considered as a dissimilarity measure between two graphs (Eq. 2.4).

$$\begin{gathered} d : \mathcal{G} \times \mathcal{G}' \to \mathbb{R}^+, \\ \textit{with } \mathcal{G} \textit{ the graph space.} \end{gathered} \tag{2.4}$$

It computes the number and the strength of the distortions that have to be applied to transform one graph into another. In other words, given two attributed graphs $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$, solving the GED problem consists in transforming the source graph G into the target graph G' (Bunke and Allermann, 1983). To accomplish such a transformation, the following set of edit operations are introduced:

- $i \to k$: substitution of two vertices $i \in V$ and $k \in V'$, i.e. i is also said to be matched with k,

- $i \to \epsilon$: deletion of vertex $i \in V$ from G,

- $\epsilon \to k$: insertion of vertex $k \in V'$ into G,

- $e \to f$: substitution of two edges $e = (i,j) \in E$ and $f = (k,l) \in E'$, i.e. e is also said to be matched with f,
- $e \to \epsilon$: deletion of edge $e = (i,j) \in E$ from G,
- $\epsilon \to f$: insertion of edge $f = (k,l) \in E'$ into G.

with ϵ referring to the *null* vertex or edge and is used to represent deletion and insertion operations.

In addition, each edit operation has an associated cost, computed by a function $c(.)$. Defining such a cost function is, in most cases, problem-dependent (e.g. graphs extracted from images or representing chemical molecules) and graph-type dependent (e.g. attributed or unattributed graphs). For the sake of clarity and generality, the cost function is defined to take an elementary edit operation as a parameter, and the output is based on the type of the operation. For example, the cost function for attributed graphs can be written as follows:

- $c(i \to k) = Sub_v(\mu(i), \mu'(k)), i \in V, k \in V'$,
- $c(i \to \epsilon) = Del_v(\mu(i)), i \in V$,
- $c(\epsilon \to k) = Ins_v(\mu'(k)), k \in V'$,
- $c(e \to f) = Sub_e(\xi(e), \xi'(f)), e \in E, f \in E'$,
- $c(e \to \epsilon) = Del_e(\xi(e)), e \in E$,
- $c(\epsilon \to f) = Ins_e(\xi'(f)), f \in E'$.

So, in the case of attributed graphs, costs are dependent on the attributes assigned to vertices and edges. For instance, the cost of substituting vertex i with vertex k is computed by the function $Sub_v(.,.)$, which is a defined distance between the two sets of attributes assigned to vertices i and k. $Del_v(.)$ and $Ins_v(.)$ are also functions to compute a cost based on the attributes of the vertex when deleting or inserting: such a cost is also seen as a penalty. The same logic is applied to edges as well when designing cost functions. Eq. 2.5 lists the domain definition of costs functions for vertices and edges edit operations.

$$\begin{aligned} Sub_v &: \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^+, \\ Del_v &: \mathbb{R}^n \to \mathbb{R}^+, \\ Ins_v &: \mathbb{R}^m \to \mathbb{R}^+, \\ Sub_e &: \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^+, \\ Del_e &: \mathbb{R}^n \to \mathbb{R}^+, \\ Ins_e &: \mathbb{R}^m \to \mathbb{R}^+. \end{aligned} \tag{2.5}$$

On the other hand, if the graphs are unattributed, the matching is mainly concerned about the topology and the structure in the graphs. Therefore, the function becomes as follows:

- $c(i \to k) = 0, i \in V, k \in V'$,

- $c(i \to \epsilon) = \tau, i \in V$,
- $c(\epsilon \to k) = \tau, k \in V'$,
- $c(e \to f) = 0, e \in E, f \in E'$,
- $c(e \to \epsilon) = \kappa, e \in E$,
- $c(\epsilon \to f) = \kappa, f \in E'$.

with τ and $\kappa \in \mathbb{R}^+$. The absence of attributes on vertices and edges leads to fixing the costs of edit operations.

Other cost function properties and restrictions are discussed in section 2.3.2.

Definition 15 (Topology constraint). *The topology constraint implies that matching (substituting) two edges $(i, j) \in E$ and $(k, l) \in E'$ is valid if and only if their incident vertices are matched:*

- *$i \to k$ and $j \to l$ OR $i \to l$ and $j \to k$ in case of undirected graphs,*
- *$i \to k$ and $j \to l$ in the case of directed graphs.*

The topology constraint imposes restrictions on edges matching and draws the relation between vertices matching and edges matching. It is not possible to randomly match edges of both graphs, instead edges matching must satisfy the topology constraint.

A sequence of edit operations that transforms G into G' is called an *edit path*. There is also the notion of *complete edit path*, which is a feasible solution to the GED problem.

Definition 16 (Edit path). *An edit path is a sequence of elementary edit operations applied on G to obtain G'. A valid edit path must consider the following:*

- *deleting a vertex implies deleting all its incident edges,*
- *inserting an edge is possible only if the two vertices already exist or have been inserted,*
- *inserting an edge must not create more than one edge between two vertices or self-loops.*

Definition 17 (Complete edit path). *A complete edit path is basically an edit path of the form $\lambda(G, G') = \{o_1, ..., o_k\}$ that considers the following properties:*

- *o_i is an elementary edit operation on a vertex or an edge,*
- *k is a positive integer,*
- *a vertex/edge can have at most one edit operation applied on it.*

Subsequently, solving the graph edit distance problem consists in computing the complete edit path with the least total cost.

Definition 18 (Graph edit distance)**.** *The graph edit distance between two graphs G and G' is defined by:*

$$d_{min}(G, G') = min_{\lambda \in \Gamma(G,G')} \sum_{o_i \in \lambda(G,G')} c(o_i) \tag{2.6}$$

where $\Gamma(G, G')$ is the set of all complete edit paths, d_{min} is the minimal cost obtained by a complete edit path $\lambda(G, G')$, and $c(.)$ is the function that assigns costs to elementary edit operations o_i satisfying all the above constraints (Eq. 2.9 till 2.16).

It is obvious that the GED problem depends on some defined cost functions, but they are not involved in GED's definition. Another way to say it, the cost functions must be defined and designed beforehand, apart from the GED solution. There is a general and easy way to store the costs of edit operations using martices. Matrix $[c_v]$ (Eq. 2.7) represents the costs of edit operations for vertices: $\forall(i, k) \in V \times V'$, the substitution costs are stored as shown in the equation 2.7. Then, a column labeled ϵ is added to store the costs of deletions of the vertices in V. For insertion operations, the row ϵ in the matrix contains the insertion costs for vertices in V'. The values inside the matrix are expressed by $c_{u,v}$, that is the result of calling the cost function $c(u \to v)$ for the edit operation. For edge edit operations, matrix $[c_e]$ (Eq. 2.8) is computed for every $((i, j), (k, l)) \in E \times E'$, plus the row and column ϵ for deletions and insertions of edges.

$$c_v = \begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & \cdots & v_{|V'|} & \epsilon \end{array} \\ \left[\begin{array}{ccccc} c_{1,1} & c_{1,2} & \cdots & c_{1,|V'|} & c_{1,\epsilon} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,|V'|} & c_{2,\epsilon} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{|V|,1} & c_{|V|,2} & \cdots & c_{|V|,|V'|} & c_{|V|,\epsilon} \\ c_{\epsilon,1} & c_{\epsilon,2} & \cdots & c_{\epsilon,|V|} & 0 \end{array}\right] \end{array} \begin{array}{c} \\ u_1 \\ u_2 \\ \vdots \\ u_{|V|} \\ \epsilon \end{array} \tag{2.7}$$

$$c_e = \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & \cdots & e_{|E|} & \epsilon \end{array} \\ \left[\begin{array}{ccccc} c_{1,1} & c_{2,1} & \cdots & c_{|E|,1} & c_{\epsilon,1} \\ c_{1,2} & c_{2,2} & \cdots & c_{|E|,2} & c_{\epsilon,2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{1,|E'|} & c_{2,|E'|} & \cdots & c_{|E|,|E'|} & c_{\epsilon,|E'|} \\ c_{1,\epsilon} & c_{2,\epsilon} & \cdots & c_{|E|,\epsilon} & 0 \end{array}\right] \end{array} \begin{array}{c} \\ f_1 \\ f_2 \\ \vdots \\ f_{|E'|} \\ \epsilon \end{array} \tag{2.8}$$

Property 1. *The edges matching are driven by the vertices matching.*

This is an important property in the GED problem and it is inferred from defintion 15. It states that edges matching are dependent on vertices matching. For example, to substitute edges $(i, j) \to (k, l)$, the incident vertices must be matched i.e. $i \to k$ and $k \to l$. Evidently, if vertices matching ($i \to k$ and $k \to l$) are determined before, it could be evinced that edges (i, j) and (k, l) are going to be substituted. Another example, deleting a vertex i, the incident edges (originated from i) will have to be deleted. Accordingly, it could be generalized that once vertices matching are fixed, edges matching can be deduced.

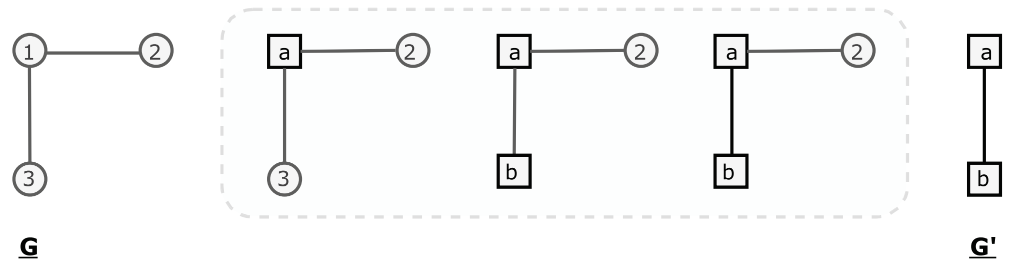


Figure 2.5: Transformation a graph G into graph G'. Edit operations: Substitution of $1 \to a$, $3 \to b$ and $(1,3) \to (a,b)$; deletion of vertex 2 and edge $(1,2)$

This property is important and helpful when designing methods to solve the problem, by focusing more on determining vertices matching, considering it the difficult part, and then deducing edges matching.

Figure 2.5 depicts an example of the transformation of a graph G into a graph G'. The complete edit path is composed of the following edit operations: $1 \to a$, $3 \to b$, $(1,3) \to (a,b)$, $2 \to \epsilon$ and $(1,2) \to \epsilon$. This is a valid complete edit path, so it is a feasible solution, but is it the best one? As stated before, the solution obtained after solving the GED problem represents the sequence of edit operations with the least total cost. Based on Definition 18, if somehow all complete edit paths in $\Gamma(G,G')$ can be enumerated, then it is only a matter of calculating the costs of all paths and select the cheapest one. However it is not the case, as enumerating all complete edit paths in $\Gamma(G,G')$ is not very easy and straightforward. It involves determining all valid paths that satisfy all the conditions and restrictions, evaluating them and selecting the best/optimal one with the least total cost. This is the reason of the high combinatorial complexity of the GED problem. Hence, the problem is difficult and its complexity grows exponentially with the growth of the graph sizes i.e. number of vertices. In fact, the GED problem was proven to be $\mathcal{NP}$-hard by Zeng et al. (2009). So, unless $\mathcal{P} = \mathcal{NP}$, solving the problem to optimality cannot be done in polynomial time of the size of the input graphs. Regarding the complexity proof, Zeng et al. (2009) have used a reduction of an induced subgraph isomorphism instance to a GED instance, in the special case where $|V| \leq |V'|$ and $|E| \leq |E'|$. The reduction, of course, was done in polynomial time. Therefore, the GED problem is $\mathcal{NP}$-hard since the sugraph isomorphism problem is known to be $\mathcal{NP}$-complete (Garey and Johnson, 1990).

Due to the high complexity of the problem, most of the researches were focused on developing heuristic and approximation algorithms to solve it. There are plenty of those methods in the literature, and some of them are really effective and can compute the matching between graphs in a very short amount of time. There are also exact algorithms and *Mixed Integer Linear Program* (MILP) formulations that were designed to compute the optimal solutions to the GED problem. However, such methods suffer from the problem of scalability, i.e. they are not applicable on large graphs. In the following sub-sections, the most recent and important heuristic and exact algorithms are reviewed and discussed in details, highlighting their pros and cons.

2.3.2 Cost functions

Two main criteria are discussed in this section related to cost functions definition for the GED problem.

2.3.2.1 Cost function restrictions

In addition to the above restrictions on the edit path in the GED problem definition, there are certain constraints that the cost function $c(.)$ must meet (Neuhaus and Bunke, 2007). These constraints give more sense to solutions by making the cost function a metric, obeying the metric conditions: positivity, identity of indiscernibles, symmetry and triangle inequality. The first constraint (Eq. 2.9) ensures that deletion and insertion operations have positive costs, and substitution also is positive or zero.

$$c(i \to k) \geq 0 \ and \ c(i \to \epsilon) > 0 \ and \ c(\epsilon \to k) > 0, \ \forall i \in V, k \in V'. \tag{2.9}$$

Those constraints favor substitution over deletion and insertion operations, by allowing substitution to cost zero. Of course, the same constraints exist on edges.

$$c(e \to f) \geq 0 \ and \ c(e \to \epsilon) > 0 \ and \ c(\epsilon \to f) > 0, \ \forall e \in E, f \in E'. \tag{2.10}$$

However, this does not give priority to substitutions over insertions and deletions, in the case where substitution has a non-zero cost. Thus, the equations 2.11, also known as the triangle inequality, are imposed.

$$\begin{gathered} c(i_1 \to k_1) \leq c(i_1 \to \epsilon) + c(\epsilon \to k_1), \\ c(i_1 \to \epsilon) \leq c(i_1 \to k_2) + c(k_2 \to \epsilon), \\ c(\epsilon \to k_1) \leq c(\epsilon \to k_2) + c(k_2 \to k_1), \\ \forall i_1 \in V \ and \ \forall k_1, k_2 \in V'. \end{gathered} \tag{2.11}$$

Equations 2.11 simply force a substitution operation between two vertices to cost less than deleting the first vertex and inserting the second vertex. Deleting one vertex must always be cheaper than substituting that vertex and then delete the replacement. Lastly, inserting a vertex must cost less than inserting a different vertex and then substituting it with that vertex. The triangle inequality is applied on edges as well.

$$\begin{gathered} c(e_1 \to f_1) \leq c(e_1 \to \epsilon) + c(\epsilon \to f_1), \\ c(e_1 \to \epsilon) \leq c(e_1 \to f_2) + c(f_2 \to \epsilon), \\ c(\epsilon \to f_1) \leq c(\epsilon \to f_2) + c(f_2 \to f_1), \\ \forall e \in E \ and \ \forall f_1, f_2 \in E'. \end{gathered} \tag{2.12}$$

The function has to satisfy the identity of indiscernibles condition, which states that substituting two vertices (resp. edges) costs zero if and only if all their attributes are equal. The condition is imposed on vertices by Eq. 2.13 and on edges by Eq. 2.14.

$$c(i \to k) = 0 \iff \mu(i) = \mu'(k), \forall i \in V \ and \ k \in V' \tag{2.13}$$

$$c(e \rightarrow f) = 0 \iff \xi(e) = \xi'(f), \forall e \in E \text{ } and \text{ } f \in E' \tag{2.14}$$

This is however not yet a metric, because it still needs the symmetry constraints to be meet. They are defined as follows:

$$\begin{gathered} c(i \rightarrow k) = c(k \rightarrow i), \\ c(i \rightarrow \epsilon) = c(\epsilon \rightarrow i), \\ c(\epsilon \rightarrow k) = c(k \rightarrow \epsilon), \\ \forall i \in V \text{ } and \text{ } k \in V'. \end{gathered} \tag{2.15}$$

Likewise, the symmetry constraints are applied on edges costs.

$$\begin{gathered} c(e \rightarrow f) = c(f \rightarrow e), \\ c(e \rightarrow \epsilon) = c(\epsilon \rightarrow e), \\ c(\epsilon \rightarrow f) = c(f \rightarrow \epsilon), \\ \forall e \in E \text{ } and \text{ } f \in E'. \end{gathered} \tag{2.16}$$

The aforementioned constraints are crucial and must be satisfied when designing a cost function for the edit operations. This implies that GED becomes a distance function, and the identity property can be inferred:

$$d(G, G') = 0 \iff G = G' \tag{2.17}$$

2.3.2.2 Objective driven cost functions

Another aspect to consider when talking about the GED is the cost of edit operations. Solving the GED problem implies minimizing the edit operations cost to transform one graph into another. Each edit operation has an associated cost function as in Eq. 2.5. Cost functions can take into account vertex or edge attributes. As well, cost functions must reflect the user need, thus they can be learned to fit a specific goal. For instance, the goal can be to reduce the gap between the ground-truth matchings and the optimal matchings. The ground-truth matching is usually given by human experts (aka Oracle) and reflects the true matching between a pair of graphs. When the ground-truth is missing, cost functions can also be hand-crafted based on domain-dependent knowledge introduced by an expert of the application. In the examples illustrated in Figure 2.10 (Page 52), actually red lines indicate wrong correspondences computed by the GED solver, and they are detected by comparing them to the ground-truth matching. In fact, there could be two reasons to why there are mismatched vertices: the first reason is that GED solver used is not an exact method and thus the computed matching is not the optimal one. A second reason is that the GED solver has computed a very good solution (the best minimum) but the cost functions defined are not learned based on the ground-truth matchings. Defining adequate cost functions is a problem in itself, known as learning cost functions for graph matching.

There are a lot of works in the literature proposing methods to learn cost functions to fit specific goals (Moreno-García et al., 2016; Cortés and Serratosa, 2016, 2015; Serratosa et al., 2011). Despite the fact that solving the GED problem and learning cost functions are two different problems, they are related. Generally, GED solvers are considerably influenced by cost functions. Choosing carefully cost functions might help GED

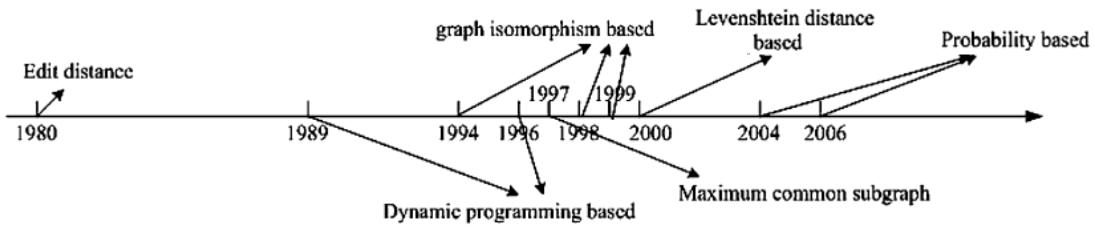


Figure 2.6: GED time line (taken from Gao et al. (2010)

solvers in converging faster towards good solutions with respect to applications. Because cost functions can help locally to differentiate vertices/edges between each other, which enables GED solvers finding the right correspondences easily. Otherwise, there will be a lot of symmetry and resemblance between vertices and edges and of course the problem of matching them becomes more complicated. In the presence of attributes on vertices and edges can help distinguishing between them, such that cost functions will use the attributes to measure a similarity or dissimilarity distance between vertices and edges. If attributes are numerical, then any representative norm can be used, e.g. norm L_2 or the Euclidean distance. In the case of nominal or symbolic attributes, a distance can be defined by either transforming them to numerical (if possible) attributes and then as before use any norm as a distance. Otherwise, a discriminative distance over the symbolic attributes has to be defined. However, as the GED is $\mathcal{NP}$-hard, then even when having good cost functions, it might be still very hard and time consuming to solve complex instances. The important two things to keep in mind are: cost functions affect the performance of GED solvers, but there is always a need to develop efficient (heuristics and exacts) algorithms to solve the minimization problem.

2.3.3 The GED problem at a glance

The GED problem was born after several attempts to refine, as much as possible, the distances or (dis)-similarity measures between graphs. The first attempt was made by Sanfeliu and Fu (1983), who had proposed a distance between graphs by counting the number of relabeling of vertices and edges, together with the number of vertices and edges deletions and insertions. Next, Messmer and Bunke (1994, 1998) extended this first attempt by computing the minimum cost for all error-correcting subgraph isomorphisms. These works made the link between the GED problem and error-tolerant subgraph isomorphism problems. During the same period, Bunke (1997) showed that there is a strong relation between the GED problem and the maximum common subgraph problem. However, there was still an ambiguity about the definition of the cost functions for edit operations on one side, and the sensitivity of the GED problem to cost functions on another side. These unclear points were addressed by Bunke (1999) by commenting on the uniqueness of the cost functions.

Figure 2.6 shows the evolution of the GED problem in time. The figure is taken from the survey by Gao et al. (2010). It points out the methods that were proposed to tackle the

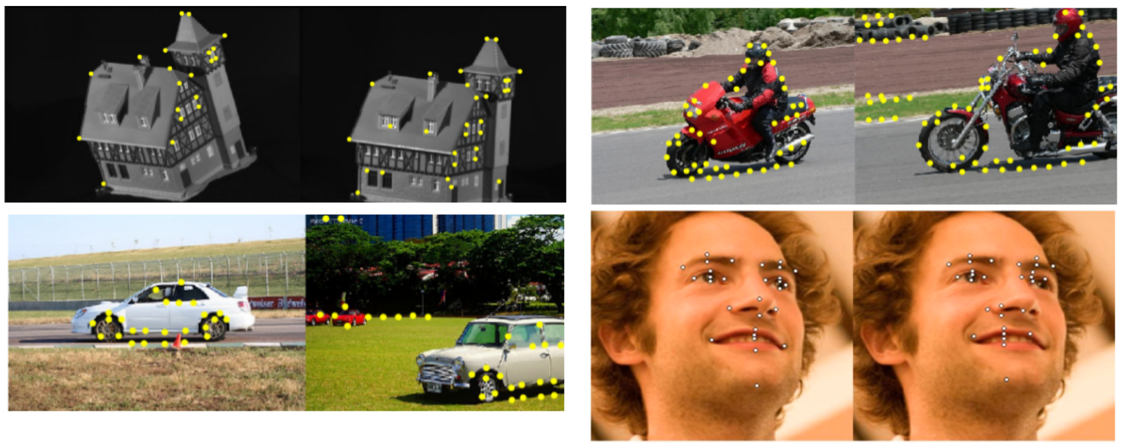


Figure 2.7: Examples of keypoints that represent an object inside an image. The keypoints become the vertices in the graph (Lê-Huu and Paragios, 2017; Zhou et al., 2013).

GED problem. As seen, in the early 90's the methods were dynamic programming-based and adapted to solve the GED problem, but originally proposed to solve the string edit distance problem. The work by Zhang and Shasha (1989) refers to a dynamic programming-based method for the GED problem, and it is based on the dynamic program by Wagner and Fischer (1974) for the string edit distance. Next, Zhang (1996) had proposed another dynamic program that only works on trees. Also inspired by string edit distance methods, there were several methods based on *Levenshtein distance* to evaluate the similarity of pair-wise strings, which are derived from graphs (Myers et al., 2000). However, such a method suffered from the problem of not fully exploiting the statistical dependencies existing in the local context. To overcome this inconvenience, methods based on *Markov random field* were designed. Wei (2004) had developed a probabilistic based method known as Markov edit distance. Again from the string edit distance, the *Hamming distance* was involved in a method to solve the GED problem. It was used to compute the hamming distance between the structural units of graphs (Wilson and Hancock, 1997). Consequently, the edit distance problem in general have played an important role into shifting to graphs and being able to measure distances between them. This have helped in refining the GED problem and switch the focus on designing and developing efficient methods to solve it.

2.3.4 Applications

The GED problem appears in many application fields and a taxonomy has been proposed by Stauffer et al. (2017). They belong to many research fields such as *Pattern Recognition*, *Chem-informatics*, *Bio-informatics*, *Internet Of Things*. In most of the applications, solving the GED problem has been done to perform either graph retrieval or classification tasks. In the following, the important applications are reviewed with some examples.

Image analysis. Graphs are used in images to represent objects and patterns. They are flexible, so they can represent objects in both 2D- or 3D-images. Vertices model the main

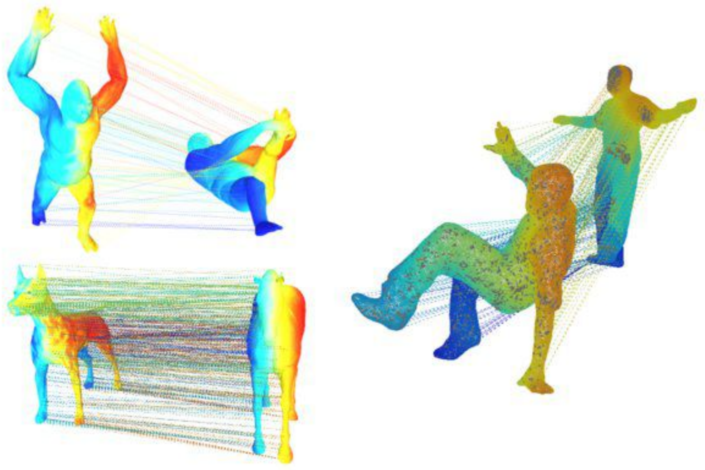


Figure 2.8: Keypoints are spotted in silhoutes and then used as vertices in graphs (Mateus et al., 2008).

components that form the object, and each vertex has a list of attributes that characterizes the component, e.g. $(x, y)-$coordinates, color intensities around, special features, etc. Then, the edges are used to link the components, with additional attributes to carry information describing those links. Figure 2.7 illustrates an example of graphs modeling objects, e.g. houses, cars, bikes and even human face features. Another usage of graphs is shown in Figure 2.8, where graphs are modeling silhouettes of objects extracted from videos. Then, the GED problem can be solved in order to compute dissimilarities between graphs. Having small distance values reflects high similarties between objects and they likely belong to the same family. By doing so, it gives the ability to compare objects and patterns and therefore to perform: object detection and recognition, image segmentation (Zhang et al., 2016; Madi et al., 2017; Hasegawa and Tabbone, 2012; Seidl et al., 2014; Robles-Kelly and Hancock, 2005). The GED problem appreas as well in classification tasks as presented by Raveaux et al. (2011); Riesen et al. (2007b); Bunke and Riesen (2012).

Handwritten document analysis. Graphs are constructed over segmented words in images, where vertices represent the keypoints or strokes, and the edges link pairs of keypoints or strokes. A graph models the documents words and their relations. Then, GED can be applied between a query graph and documents graphs to find correspondences. Such an application is called keyword spotting and there exists many works in the literature that use GED (Riesen et al., 2014; Wang et al., 2014b,a; Stauffer et al., 2016; Bui et al., 2015; Riba et al., 2015)

Biometrics. Retina vessels, fingerprints or signatures are considered as biometrical characteristics. There are many applications with the goal of identifying an individual based on the fingerprint or signature. So, graphs can be used to model a fingerprint, where vertices represents segmented core areas, and edges relates adjacent areas. Graphs can then be classified by using GED solutions as a dissimilarity measure between graphs (Choi and

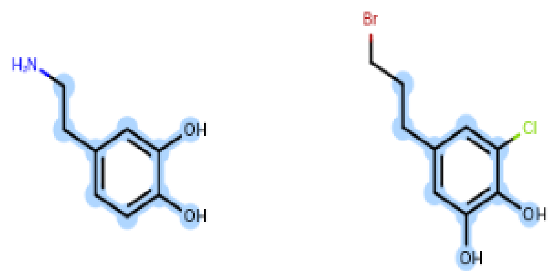


Figure 2.9: Examples of graphs modeling chemical molecules. The vertices and edges are drawn in light blue.

Kim, 2010). Similar approaches are used in identification systems based on retina vessels, where also graphs are classified by means of a GED solver (Lajevardi et al., 2013). Another application based on signatures can be found in the literature (Wang et al., 2011).

Bio- and Chem-informatics. In the field of Bio-informatics, graphs are used to model DNA, protein sequences and enzymes. This enables analyzing biological structures. A very important example is the ability of detecting cancerous tissues. Tissues are modeled by graphs and then a classifier is built by using GED distances to classify normal, low-grade and high-grade cancerous tissues (Ozdemir and Gunduz-Demir, 2013). In chemistry field and precisely when considering chemical molecules, graphs form a natural representation of the atom-bond structure of molecules. Each vertex of the graph then represents an atom, while an edge represents a molecular bond (Raymond and Willett, 2002). By using GED as a distance between graphs, it provides a way to compare molecules between each other and to detect similar activities and properties, which answers a major question in this field. Brun et al. (2010) and Gaüzère et al. (2011) have employed graphs and the GED problem in order to compare chemical molecules. Figure 2.9 illustrates an example of chemical molecules represented by graphs.

Knowledge and process management. Graphs can be useful in this context. For instance, process management is composed of various methods, which are connected and share complex relations. It aligns processes in a way to guarantee a good overcome. The processes and their relationships can be expressed by graphs and then GED is applied to extract similarities between bands processes (Niedermann, 2016). Another application is to estimate the execution time of SPARQL (Sparkle recursive acronym query language) queries (Hasan and Gandon, 2014).

Malware detection. The efficiency of using graphs to construct relational models for malicious executables of the same family, makes it suitable to employ GED in the task of malware detection in Anti-viruses. Graphs are called *call graphs* and represent a malware sample with certain variations. Then, based on GED solutions after comparing the graphs,

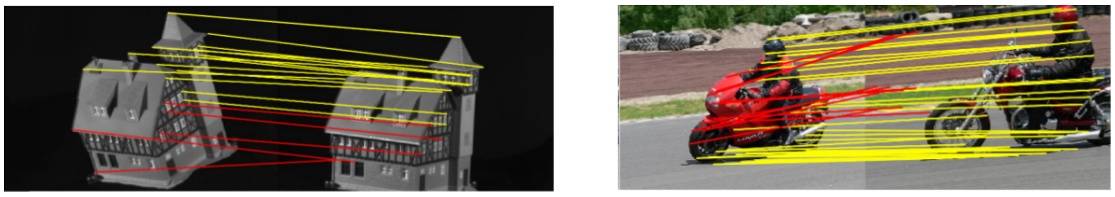


Figure 2.10: Examples of matching solution computed by a GED solver between couple of graphs. Yellow lines show the correct correspondences between vertices. Red lines are for wrong correspondences computed by the solver.

certain properties can be extracted based on the similarities found (Bourquin et al., 2013; Elhadi et al., 2012) and (Kostakis et al., 2011). In those examples, the GED problem is not the main problem, but it is used for graphs comparison and then builds up on it to achieve the objectives of detecting malicious executables.

Other applications. Among others, more are applications are: stories retrieval (Paul, 2013), sketches retrieval (Florez-Puga et al., 2013) and plagiarism detection (Kammer et al., 2011; Røkenes et al., 2012).

2.3.5 Key-points and advantages of the GED problem

Most of the applications listed in section 2.3.4, require performing graph searches among databases of graphs. For instance, an unknown graph that models an object in an image must be compared with all graphs in a database of known objects in order to find similarities. Zeng et al. (2009) classify graph searches into three categories, for a database of graphs $D = \{g_1, g_2, ..., g_n\}$, and a query graph q:

- Full search: find all graphs g_i in D that are the same as q,
- Subgraph search: find all graphs g_i in D that contain or are contained by q,
- Similarity search: find all graphs g_i in D that are similar to q, based on some defined similarity measure.

The GED problem can be involved in the three aforementioned categories. Furthermore, in most of the applications, the graph search mostly used is the similarity search. The reason is that the graph query may differ from graph models stored in the database, due to reasons such as the presence of noise inside an image. Therefore, graph and subgraph isomorphism might not be useful as much as similarity search, which is flexible and tolerates structures and attributes differences in graphs. In addition, similarity search is effective in supervised classification by k-nearest neighbors and unsupervised classification by k-medians clustering. Another important application using the similarity search is graph retrieval. GED seems to be a good fit to perform similarity search and graph retrieval.

Besides providing a distance/dissimilarity measure, GED also computes the matching between two graphs. A matching, which also called assignments of vertices, consists of the operations selected and applied on vertices. For two sets $V = \{u_1, u_2, u_3\}$ and $V' = \{v_1, v_2\}$, a matching is expressed with the matrix:

$$Matching = \begin{array}{c} \\ \\ \\ \\ \end{array}\begin{array}{cccc} u_1 & u_2 & u_3 & \epsilon \end{array} \\ Matching = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{array}{c} v_1 \\ v_2 \\ \epsilon \end{array}$$

Whenever a vertex $u \in V$ (resp. in $v \in V'$) is matched with ϵ, it is said to be deleted (resp. inserted), otherwise u is substituted with v. The same as vertices matching, edges matching can be represented by a matrix. It is of interest to end-users to evaluate and understand the matching, by looking at the matched components for a query graph with similar graphs found in the database. Actually, the matching can help interpreting the results and detecting relevant spot patterns. GED problem enables having both the distance and the matching, which is convenient to end-users in helping them observing the correspondences between graphs. Figure 2.10 shows an example of visualized matching solutions computed by a GED solver. Unlike, other similarity search approaches such as graph embedding into vector space, the matching is lost and cannot be reconstructed easily. Kernel-based methods are an example of graph embedding, and there are plenty of them in the literature, such as the work by Neuhaus and Bunke (2006b,a). The concept of kernel-based approach is to use kernel machines to map the classification problem from the pattern domain to a vector space domain. It embeds the graph into a vector using a kernel function. Then, some existing techniques can be applied for pattern analysis over vectors in order to compute a distance between two graphs. One inconvenient of such methods, only the distance can be computed without the matching, and it is not clear how to backtrack and re-construct the graphs matching from these vectors.

2.3.6 GED challenges

Taking a look at the literature, there are many heuristics that can be found to solve the GED problem. Some of them are designed to converge very fast, while others favor the quality of the solutions over the running time. In all cases, there is a compromise between the running time of the heuristic and the accuracy of the computed solution. The only way to overcome this is by checking the application requirements and factors like graph size, density, type of attributes, which are in general responsible in making the problem hard to solve. An application like object detection in images is, in most cases, considered as a real-time application. Therefore, there is a need for having a very fast algorithm, but on the other hand, the graphs being compared should be of small sizes. In other applications, such as finding similar graphs of complex chemical molecules, it is affordable to spend extra running time in order to obtain accurate results.

1. Is it worth spending time to compute accurate solutions?

2. What is the impact of a better solution on the similarity search or on the matching, with respect to the application?

These are the main questions and challenges that have to be considered when dealing with the GED problem. The literature contain several promising attempts and heuristics trying to address those questions.

2.3.7 Exact methods

This section is dedicated to review the most famous and efficient exact methods found in the literature to solve the GED problem. These methods are designed to compute optimal solutions, i.e. solutions having the minimum value of the total dissimilarity cost. Of course and since the problem is NP-hard, these methods tend to require an exponential running time as the graph sizes increase. On the hand, they guarantee finding the best solution possible to the problem.

2.3.7.1 A*-based algorithms

A* is considered as the first algorithm to solve the GED problem. It is an exact algorithm because it guarantees finding the optimal solution. A* builds a search tree by enumerating all possible combinations, until reaching the bottom (leaf), that is a feasible solution to the problem. Algorithm 1 details the steps of the method, and is taken from the version by Hart et al. (1968). It starts by initializing and ordering the set of vertices V_1 and V_2 of the two input graphs G_1 and G_2. At the root node of the tree, it selects a vertex u from the first graph, and then builds the child nodes corresponding to all possible edit operations on u. For all vertices $w \in V_2$, all substitution edit operations are $(u \to w)$, plus one delete operation $(u \to \epsilon)$. This defines the first level of the search tree (lines $1-3$). The added operations are handled in the set $OPEN$. The next step in the algorithm is to decide which child node to choose in order to continue building the next level. There are many strategies that are usually applied in this case such that:

- Depth-first: the first unselected child node at the left side is picked,
- Breadth-first: it explores the neighbor nodes first (nodes at the same level) before moving to the next level,
- Best-first: it selects the child node with the best (lowest) estimation. This search strategy requires having a function to compute heuristically an estimation of the cost of exploring a given node further.

This version of A* uses the best-first search strategy. After creating a new level, a cost is computed for each node using $g(p) + h(p)$. Where $g(p)$ represents the costs of the edit operations that are selected so far until node p. The function $h(p)$ computes an estimated cost from node p to a leaf node. So, the algorithm decides the next node to explore by selecting the one with the smallest value of $g(p) + h(p)$ (line 5). In order to guarantee the admissibility of the algorithm, which means it will eventually end up by finding the optimal solution, the following property must be met: the estimated costs $h(p)$ have to be always lower than or equal to the real costs. The algorithm carries on by building the next levels, until reaching a leaf node where it stops and return the operations that were selected all

Algorithm 1: Minimization of the graph edit distance by A* algorithm

```
Input  : Non-empty graphs $g_1 = (V_1, E_1, \mu_1, \xi_1)$ and $g_2 = (V_2, E_2, \mu_2, \xi_2)$,
         where $V_1 = \{u_1, ..., u_{|V_1|}\}$ and $V_2 = \{v_1, ..., v_{|V_2|}\}$
Output: A minimum-cost edit path from $g_1$ to $g_2$,
         e.g. $p_{min} = \{u_1 \to v_3, u_2 \to \epsilon, ..., \epsilon \to v_6\}$

1  Initialize OPEN to the empty set
2  For each vertex $w \in V_2$, insert the substitution $(u_1 \to w)$ into OPEN
3  Insert the deletion $(u_1 \to \epsilon)$ into OPEN
4  loop
5      Remove $p_{min} = argmin_{p \in OPEN}\{g(p) + h(p)\}$ from OPEN
6      if $p_{min}$ is a complete edit path then
7          Return $p_{min}$ as the solution
8      else
9          Let $p_{min} = \{u_1 \to v_{i_1}, ..., u_k \to v_{i_k}\}$
10         if $k < |V_1|$ then
11             For each $w \in V_2 \setminus \{v_{i_1}, ...v_{i_k}\}$, insert $p_{min} \cup (u_{k+1} \to w)$ into OPEN
12             Insert $p_{min} \cup (u_{k+1} \to \epsilon)$ into OPEN
13         else
14             Insert $p_{min} \cup \bigcup_{w \in V_2 \setminus \{v_{i_1}, ..., v_{i_k}\}} (\epsilon \to w)$ into OPEN
15         end
16     end
17 end loop
```

the way down from the root node (line 7). In the case where all the vertices of V_1 are processed (substituted or deleted), the remaining vertices of V_2 are inserted. Note that edit operations on edges are implied by edit operations on their adjacent vertices (based on GED property 1), and the function $g(p)$ takes into account the edges edit operations.

One way to compute the estimation $h(p)$ at a node p is as follows:

- Let n_i (resp. n_j) be the number of unprocessed vertices for graph G (resp. graph G'),
- construct the cost matrix of size $min(n_i, n_j)$ for vertices assignment,
- solve the linear assignment problem to get an minimum assignment cost $assign_s$,
- compute the deletion cost $assign_d$ by adding the costs of remaining vertices $max(0, n_i - n_j)$,
- compute the insertion cost $assign_i$ by adding the costs of remaining vertices $max(0, n_j - n_i)$,
- finally $h(p) = assign_s + assign_d + assign_i$.

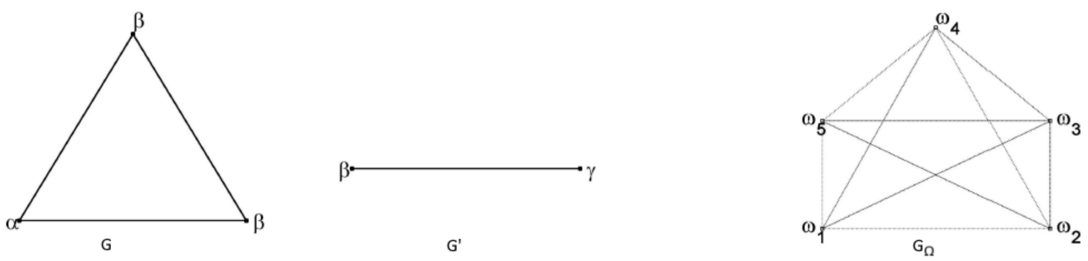


Figure 2.11: An example of the edit grid graph generated for two input graphs G and G' by JH formulation.

To make the computation faster, edges operations could be ignored, or substitution costs could be set to null. This may lead to having multiple substitutions of vertices or edges, which means that the edit path is invalid. But, the cost $h(p)$ is certainly a lower bound of the exact cost.

A* is a straightforward method and very simple, however it appeared to perform poorly in practice. The algorithm suffers from high computational complexity and it is exponential in the number of vertices of the graphs. It could be used until graphs of size 12, beyond that it explodes in terms of running time and memory. Therefore, there were few attempts to improve it, like A*-Beamsearch and A*-Pathlength versions by Neuhaus et al. (2006). They both intend to reduce the number of nodes to be explored by choosing a fixed number as in the first one, or modifying the estimation function as in the second one. Another variation to A* which works efficiently was developed by Abu-Aisheh et al. (2015b). It reduces the computation time by improving lower and upper bounds, which help in pruning unpromising child nodes.

2.3.7.2 JH MILP formulation

Justice and Hero (JH) have proposed a MILP formulation to optimally solve the GED problem (Justice and Hero, 2006). It is one of the efficient formulations, however it deals with a special case of GED. It takes into account the attributes over vertices by using a cost function to compute a cost (distance), but it requires a fixed cost for operations on edges. So, for graph instances with attributes on edges, JH does not include these attributes and instead, it assigns zero as cost for substitution operations and a unitary cost for deletion and insertion operations.

Given two graphs $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$, the main idea consists in determining the permutation matrix minimizing the L_1 norm of the difference between adjacency matrix of the input graph G and the permuted adjacency matrix of the target one G'. The authors have introduced a framework for edits, in which both graphs G and G' are embedded in bigger graphs denoted as *edit grids*. Figure 2.11 illustrates an example of two graphs and their corresponding edit grid. At first, it constructs an edit grid G_Ω of size $|V| + |V'|$, that is a complete graph with all vertices and edges having the *null* attribute. Next, G is embedded in G_Ω by relabeling the null vertices and edges with the actual attributes. The next step is to determine the series of edit operations needed to transform the version of G in the edit grid to an isomorphic graph of G' in the edit grid.

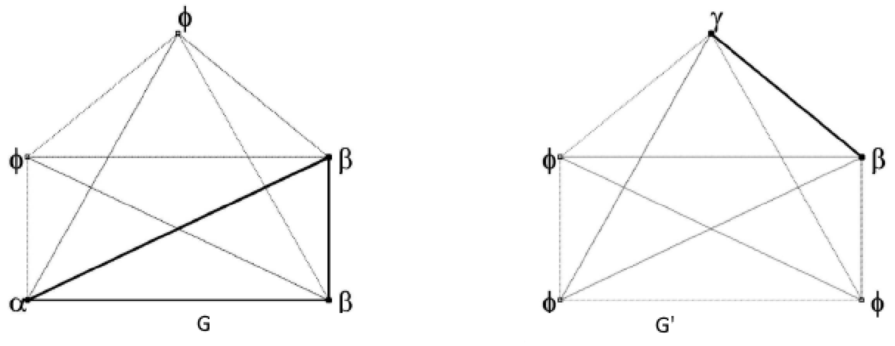


Figure 2.12: Isomorphisms of graphs G and G' on the edit grid G_Ω. The edit operations applied are: $(\omega_1, \alpha \to \phi), (\omega_2, \beta \to \phi), (\omega_4, \phi \to \gamma), ((\omega_1, \omega_2), 1 \to 0), ((\omega_1, \omega_3), 1 \to 0), ((\omega_2, \omega_3), 1 \to 0), ((\omega_3, \omega_4), 0 \to 1)$

The edit operations available on the edit grid are the following:

- $(\omega_1, \alpha \to \phi)$, corresponds to a vertex deletion operation such that the vertex ω_1 in the edit grid is labeled by α and is being now relabeled by ϕ (null), implying deleting the α vertex,
- $(\omega_1, \phi \to \alpha)$, corresponds to a vertex insertion operation. A vertex with label α is being inserted in the edit grid,
- $((\omega_1, \omega_2), 1 \to 0)$, is an edge deletion operation,
- $((\omega_1, \omega_2), 0 \to 1)$, is an edge insertion operation.

Note that the substitution operations are selected at a first step, when embedding graph G in the edit grid. Figure 2.12 depicts an example, following the input graphs in Figure 2.11, of a transformation between graphs, with the selected operations. The goal is to find the proper set of operations that will give an isomorphism of G' in the edit grid. The authors have used the adjacency matrices A and A' of size $|V| + |V'|$, to represent graphs G and G' in the edit grid. Then, the problem is written as follows:

$$d_c(G, G') = \min_{P \in B} \sum_{i=1}^{N} \sum_{k=1}^{N} c(\mu(i), \mu(k)) P_{ik} + \frac{1}{2} \cdot \kappa \cdot |A - PA'P^T|_{ik}, \tag{2.18}$$

with $N = |V| + |V'|$, $B = \{X \in \{0,1\}^{N \times N} / \sum_j X_{k,j} = \sum_i X_{i,k} = 1 \forall k\}$ the set of all permutation matrices, and $\kappa \in \mathbb{R}^+$ a constant cost for edges insertions and deletions. The second term in the equation makes it a non-linear (quadratic) function. To linearize it, the authors have followed the same strategy as in (Almohamad and Duffuaa, 1993). The details of the linearization are skipped and the linear version is given next, for further details kindly refer to the full paper Justice and Hero (2006).

Data. Since the cost functions are known and defined. Vertices cost matrix $[c_v]$ is computed as in equation 2.7 for every couple $(i, k) \in V \times V'$. The ϵ column is added to store the cost of deleting i vertices, while the ϵ row stores the costs of inserting k vertices. In this formulation, there is no need to compute $[c_e]$ since edges edit operations have a constant cost. In this case, $\kappa \in \mathbb{R}^+$ is set beforehand. Also, as part of problem data, there are the two adjacency matrices A and A' for graphs G and G'.

Variables. After the linearization, JH formulation is left with three sets of binary variables:

- $x_{i,k} \in \{0,1\}$, $\forall i \in \{1,2,...,N\}, \forall k \in \{1,2,...,N\}$: $x_{i,k} = 1$ when vertices i and k are matched, and 0 otherwise. Note that, the indices of i and k are bounded by N, which means that deletions (resp. insertions) occur when i (resp. k) has ϕ as attribute.
- $s_{i,k} \in \{0,1\}$, $\forall i \in \{1,2,...,N\}, \forall k \in \{1,2,...,N\}$. It is used to manage edges matching. When $s_{i,k} = 1$, it represents a matching between the first couple of vertices of edges including i and k.
- $t_{i,k} \in \{0,1\}$, $\forall i \in \{1,2,...,N\}, \forall k \in \{1,2,...,N\}$. It is the complementary variables for edges matching. When $t_{i,k} = 1$, it means a matching between the second couple of vertices of edges including i and k. It follows that in order to match edges $((i,j) \rightarrow (k,l)$, $s_{i,k} = t_{j,l} = 1$.

Objective function. The objective function to minimize is the following:

$$\min_{x,s,t \in \{0,1\}^{N\times N}} \left(f(x,s,t) = \sum_{i=1}^{N} \sum_{k=1}^{N} c\left(\mu(i), \mu'(k)\right) \cdot x_{i,k} + \left(\frac{1}{2} \cdot \kappa \cdot (s_{i,k} + t_{i,k})\right)\right) \tag{2.19}$$

The first part of the objective function computes the cheapest permutation x for vertices. The second part is to account for the edges operations costs.

Constraints. JH has 3 sets of constraints:

$$\sum_{j=1}^{N} A_{i,j} \cdot x_{j,k} - \sum_{c=1}^{N} x_{i,c} \cdot A'_{c,k} + s_{i,k} - t_{i,k} = 0, \ \forall i,k \in \{1,2,...,N\} \tag{2.20}$$

$$\sum_{i=1}^{N} x_{i,k} = 1, \ \forall k \in \{1,2,...,N\} \tag{2.21}$$

$$\sum_{j=1}^{N} x_{k,j} = 1, \ \forall k \in \{1,2,...,N\} \tag{2.22}$$

Constraints 2.20 make sure that when matching two couples of vertices, the edges between them have to be matched as well. Then, constraints 2.21 and 2.22 guarantee the integrity of matrix x, i.e. one vertex in G can be matched with only one vertex in G'. This model has a limitation that it does not consider the attributes on edges, so edge substitution cost is 0 while deletion and insertion have a $\kappa \in \mathbb{R}^+$ fixed cost. Another limitation is that JH supports only undirected graphs. Nevertheless, JH was proven by Lerouge et al. (2017), to be the most efficient formulation among other formulations in the exact context, in the cases of undirected and edge-free attributes graphs.

The number of variables in JH formulation are $3 \cdot (|V| + |V'|) \cdot (|V| + |V'|)$ with $(|V| + |V'|) \cdot (|V| + |V'|) + 2 \cdot (|V| + |V'|)$ constraints.

2.3.7.3 F1 MILP formulation

F1 is a formulation designed by Lerouge et al. (2017). It is a direct formulation of the GED problem for undirected graphs. F1 involves a set of binary variables for each edit operation possible and then minimizes the total costs. The formulation is defined as follows.

Data. The cost functions are assumed to be given, therefore $[c_v]$ and $[c_e]$ are computed as in equations 2.7 and 2.8.

Variables. F1 formulation has 6 sets of binary variables:

- $x_{i,k} \in \{0,1\}$, $\forall i \in V, \forall k \in V'$: $x_{i,k} = 1$ when vertices i and k are matched, and 0 otherwise.
- $y_{ij,kl} \in \{0,1\}$, $\forall (i,j) \in E, \forall (k,l) \in E'$: $y_{ij,kl} = 1$ when edge (i,j) is matched with (k,l), and 0 otherwise.
- $u_i \in \{0,1\}$, $\forall i \in V$: $u_i = 1$ when vertex i is deleted, and 0 otherwise.
- $e_{ij} \in \{0,1\}$, $\forall (i,j) \in E$: $e_{ij} = 1$ when edge ij is deleted, and 0 otherwise.
- $v_k \in \{0,1\}$, $\forall k \in V'$: $v_k = 1$ when vertex k is inserted, and 0 otherwise.
- $f_{kl} \in \{0,1\}$, $\forall (k,l) \in E'$: $f_{kl} = 1$ when edge kl is inserted, and 0 otherwise.

Objective function. The objective function to be minimized is the following:

$$\begin{gathered} \min_{x,y,u,v,e,f} \sum_{i \in V} \sum_{k \in V'} c_v(i,k) \cdot x_{i,k} + \sum_{(i,j) \in E} \sum_{(k,l) \in E'} c_e(ij,kl) \cdot y_{ij,kl} + \\ \sum_{i \in V} c_v(i,\epsilon) \cdot u_i + \sum_{k \in V'} c_v(\epsilon,k) \cdot v_k + \sum_{ij \in E} c_e(ij,\epsilon) \cdot e_{ij} + \sum_{kl \in E'} c_e(\epsilon,kl) \cdot f_{kl} \end{gathered} \tag{2.23}$$

The objective function minimizes the cost of vertices (resp. edges) substitutions, deletions and insertions.

Constraints. F1 has 6 sets of constraints:

$$u_i + \sum_{k \in V'} x_{i,k} = 1, \ \forall i \in V \tag{2.24}$$

$$v_k + \sum_{i \in V} x_{i,k} = 1, \ \forall k \in V' \tag{2.25}$$

$$e_{ij} + \sum_{(k,l) \in E'} y_{ij,kl} = 1, \ \forall (i,j) \in E \tag{2.26}$$

$$f_{kl} + \sum_{(i,j) \in E} y_{ij,kl} = 1, \ \forall (k,l) \in E' \tag{2.27}$$

$$y_{ij,kl} \leq x_{i,k} + x_{i,l}, \ \forall (i,j) \in E \ and \ \forall (k,l) \in E' \tag{2.28}$$

$$y_{ij,kl} \leq x_{j,l} + x_{j,k}, \ \forall (i,j) \in E \ and \ \forall (k,l) \in E' \tag{2.29}$$

Constraints 2.24 ensures that a vertex i is either matched with one vertex or deleted. The opposite case is handled by constraints 2.25. Constraints 2.26 and 2.27 make sure that an edge can be substituted with exactly one edge or deleted/inserted. Regarding the topological condition given by definition 15, constraints 2.28 and 2.29 guarantee satisfying it. Note that, since the graphs are undirected, constraints 2.28 (resp. 2.29 allow vertex i (resp. j) to be matched with k or l.

F1 formulation can be easily modified to cope with directed graphs, by simply replacing the last two constraints 2.28 and 2.29, with the following constraints:

$$y_{ij,kl} \leq x_{i,k}, \ \forall (i,j) \in E \ and \ \forall (k,l) \in E' \tag{2.30}$$

$$y_{ij,kl} \leq x_{j,l}, \ \forall (i,j) \in E \ and \ \forall (k,l) \in E' \tag{2.31}$$

The total number of variables in the formulation is $(|V| + |V'| + |E| + |E'| + |V| \cdot |V'| + |E| \cdot |E'|)$, with $(|V| + |V'| + |E| + |E'| + 2 \cdot |E| \cdot |E'|)$ constraints. F1 formulation is effective in general for the GED problem, but a more compact formulation, denoted by F2, was designed later by the same authors.

2.3.7.4 F2 MILP formulation

F2 is the best MILP formulation for the GED problem - in the general case - in the literature. It was proposed by Lerouge et al. (2017) and it is an evolution of F1 formulation. F2 formulation is a more compact and improved version of F1, obtained by reducing the number of variables and constraints. The compactness of F1 comes from the design of the objective function to be optimized. At first, it considers all vertices and edges of G as deleted and vertices and edges of G' as inserted. Then, it solves the problem of finding the cheapest assignments/matching between the two sets of vertices and the two sets of edges. The matching in this context is the substitution edit operations for vertices and edges. Once, the cheapest matching is computed, by excluding from the substitution cost both costs of deletions and insertions. Afterwards, the deletion and insertion operations can be deduced: all the remaining vertices in V (resp. in V') that are not matched with any vertex in V' (resp. in V), are considered as deleted (resp. inserted). The edges are treated in the same manner. Such design is helpful in reducing the number of variables and constraints in the formulation. The reduction is done by factorizing the objective function's terms.

Data. The cost functions are assumed to be given, therefore $[c_v]$ and $[c_e]$ are computed as in equations 2.7 and 2.8.

Variables. As mentioned earlier, F2 formulation focuses on finding the correspondences between the two sets of vertices and the two sets of edges. That is why the decision variables are reduced to two sets:

- $x_{i,k} \in \{0,1\}$, $\forall i \in V, \forall k \in V'$: $x_{i,k} = 1$ when vertices i and k are matched, and 0 otherwise.
- $y_{ij,kl} \in \{0,1\}$, $\forall (i,j) \in E, \forall (k,l) \in E'$: $y_{ij,kl} = 1$ when edge (i,j) is matched with (k,l), and 0 otherwise.

Objective function. The objective function to be minimized is the following:

$$\begin{aligned} \min_{x,y} \sum_{i \in V} \sum_{k \in V'} \left(c_v(i,k) - c_v(i,\epsilon) - c_v(\epsilon,k)\right) \cdot x_{i,k} + \\ \sum_{(i,j) \in E} \sum_{(k,l) \in E'} \left(c_e(ij,kl) - c_e(ij,\epsilon) - c_e(\epsilon,kl)\right) \cdot y_{ij,kl} + \gamma \end{aligned} \tag{2.32}$$

The objective function minimizes the cost of assigning vertices and edges with the cost of substitution subtracting the cost of insertion and deletion. The γ value, which is a constant and given in equation 2.33, compensates the subtracted costs of the assigned vertices and edges. This constant does not impact the optimization algorithm and it could be removed. It is there to obtain the GED value.

$$\gamma = \sum_{i \in V} c_v(i,\epsilon) + \sum_{k \in V'} c_v(\epsilon,k) + \sum_{(i,j) \in E} c_e(ij,\epsilon) + \sum_{(k,l) \in E'} c_e(\epsilon,kl) \tag{2.33}$$

Constraints. F2 formulation has 3 sets of constraints:

$$\sum_{k \in V'} x_{i,k} \leq 1 \ \forall i \in V \tag{2.34}$$

$$\sum_{i \in V} x_{i,k} \leq 1 \ \forall k \in V' \tag{2.35}$$

$$\sum_{(k,l) \in E'} y_{ij,kl} \leq x_{i,k} + x_{j,k} \ \forall k \in V', \forall (i,j) \in E \tag{2.36}$$

Constraints 2.34 and 2.35 impose that a vertex can be matched with at most one vertex. It is possible that a vertex is not assigned to another: in this case it is considered as deleted or inserted. Here is the key point of this formulation: F2 is flexible by allowing some vertices/edges not to be matched. The objective function gets to decide whether a substitution is cheaper than a deletion/insertion or not. γ value takes care of the unmatched vertices/edges and includes their deletion or insertion costs to the objective function. Finally, constraints 2.36 guarantee preserving edges matching between two couple of vertices.

In other words, to match two edges $(i,j) \to (k,l)$, their vertices must also be matched, i.e. $i \to k$ and $j \to l$ OR $i \to l$ and $j \to k$ (this is the topology condition defined in 15). The number of variables is $(|V| \cdot |V'| + |E| \cdot |E'|)$ and there are $(|V| + |V'| + |V| \cdot |E|)$ constraints.

The presented version of F2 formulation is applied to undirected graphs. For the case of directed graphs, constraints 2.36 are split into two sets of constraints as follows:

$$\sum_{(k,l)\in E'} y_{ij,kl} \leq x_{i,k},\ \forall k \in V', \forall (i,j) \in E \tag{2.37}$$

$$\sum_{(k,l)\in E'} y_{ij,kl} \leq x_{j,l},\ \forall l \in V', \forall (i,j) \in E \tag{2.38}$$

2.3.7.5 A QAP formulation

The error-tolerant subgraph isomorphism was modeled as a *Quadratic Assignment Problem* (QAP) in many works (Cho et al., 2013; Lyzinski et al., 2016; Caetano et al., 2009; Simić, 1991). A QAP formulation constitutes of a quadratic function to maximize the similarities between a pair of graphs. It has been used to perform tasks as learning graph models to improve GM as in (Cho et al., 2013) and (Caetano et al., 2009). Modeling the GED problem by a QAP formulation relates to assigning simultaneously two vertices from G to two vertices from G', instead of assigning vertices one by one. By doing so, it is then possible to figure out the edges assignments simultaneously when assigning the vertices. The general QAP formulation is well studied in the literature in other domains such as *Operation Research* (OR) and *Optimization*. It was proven to be a $\mathcal{NP}$-hard problem by Sahni and Gonzalez (1976). Therefore, many theoretical works can be found in the literature that propose convex and concave relaxations, linearization techniques, etc. It becomes a reflection when bringing up GM problem to think straight away of QAP. It is not strange then to find GED modeled by a QAP formulation, thanks to Bougleux et al. (2017).

A framework is designed first to store the costs of assigning edges and vertices in one big matrix D. The organization of cost values inside the matrix and sub-matrices is very important to keep track of the indices that represent vertices of the graphs and to consider all operations (substitution, insertion and deletion of vertices and edges). A sub-matrix $D^{i,j}$ is divided into 4 quadrants as follows.

$$D^{i,j} = \begin{array}{ccc|ccc|c} 1 & \dots & m & \epsilon_1 & \dots & \epsilon_n & \\ c_{1,1} & \dots & c_{1,m} & c_{1,\epsilon_1} & \dots & \infty & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{n,1} & \dots & c_{n,m} & \infty & \dots & c_{n,\epsilon_n} & n \\ \hline c_{\epsilon_1,1} & \dots & \infty & 0 & \dots & 0 & \epsilon_1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \infty & \dots & c_{m,\epsilon_m} & 0 & \dots & 0 & \epsilon_m \end{array} \tag{2.39}$$

with $n = |V|$ and $m = |V'|$. Line wise the matrix is extended by m lines, where each line is an ϵ_j. ϵ vertices are added to model deletion insertion operations, e.g. a vertex in V

(resp. V') is mapped to ϵ is considered as deleted (resp. inserted). The same for columns, they are extended by n columns and each one is an ϵ_i. So, $D^{i,j}$ is of size $(n+m) \times (n+m)$. The top-left quadrant is of size $(n \times m)$ and stands for the costs of substitution operations. The diagonal values of the quadrant top-right (resp. bottom-left) holds the costs of vertex deletions (resp. insertions). The non-diagonal values are set to ∞ to avoid getting selected. And the last quadrant bottom-right contains only 0 values and serves only to complete the symmetry in the matrix. $D^{i,j}$ as in Equation 2.39 stores the costs of all possible operations for two vertices i and j: it is similar to matrix $[c_v]$ given by Equation 2.7 with values organized in a different manner. Next step is to build the big matrix D, which is given in Equation 2.40.

$$D = \left[\begin{array}{ccc|ccc} D^{1,1} & \dots & D^{1,n} & D^{1,\epsilon_1} & \dots & D^{1,\epsilon_m} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ D^{n,1} & \dots & D^{n,n} & D^{n,\epsilon_1} & \dots & D^{n,\epsilon_m} \\ \hline D^{\epsilon_1,1} & \dots & D^{\epsilon_1,n} & D^{\epsilon_1,\epsilon_1} & \dots & D^{\epsilon_1,\epsilon_m} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ D^{\epsilon_m,1} & \dots & D^{\epsilon_m,n} & D^{\epsilon_m,\epsilon_1} & \dots & D^{\epsilon_m,\epsilon_m} \end{array}\right] \tag{2.40}$$

Matrix D is of size $(n+m)^2 \times (n+m)^2$, and is constituted of sub-matrices $D^{i,j}$. Each value in D represents the cost of assigning two (real or not) edges, i.e. $[D^{i,j}]_{k,l} = c_e(ij, kl)$ such that $i, j \in V \cup \{\epsilon_1, ..., \epsilon_m\}$ and $k, l \in V' \cup \{\epsilon_1, ..., \epsilon_n\}$. Calculating the costs is not a hard task, it depends on i, j, k and l, e.g. if $j = \epsilon_j$ and the i is a valid vertex, then the cost is the insertion of edge (k, l), because (i, j) is not a valid edge. The diagonal values of the matrix D contains the costs of vertices matching. The functions to compute the costs inside the matrix D are explained by Bougleux et al. (2017). As a result, D refers to the cost matrix with all possible combinations of matching two couples of vertices at same time, which is the goal of a quadratic objective function.

The QAP formulation proposed for the GED problem is a minimization quadratic objective function w.r.t. two constraints. Given $x \in \{0,1\}^{(n+m)\times(n+m)}$, the objective function is given by Equation 2.41, with respects to constraints 2.42 and 2.43 that makes sure a vertex can be matched with only one vertex.

$$\min_x x^T D x \tag{2.41}$$

such that

$$\sum_{i=1}^{n+m} x_{i,k} = 1, \ \forall k \in \{1, 2, ..., n+m\} \tag{2.42}$$

$$\sum_{k=1}^{n+m} x_{i,k} = 1, \ \forall i \in \{1, 2, ..., n+m\} \tag{2.43}$$

QAP formulations in general are known to be less efficient than MILP formulations, mainly because of the quadratic form of the objective functions. However, there are good tools and techniques that can be applied in order to compute good solutions. Bougleux et al. (2017) have developed two heuristics that operate over the QAP formulation, and based on the experiments they were able to beat good heuristics in the literature.

2.3.8 Heuristic methods

In contrast to exact methods, heuristic methods are designed to compute a solution in a reasonable amount of time, typically in polynomial time of the size of the input. Generally, the obtained solutions are sub-optimal, and even if they are optimal, they cannot be proved to be so. Such methods are used to overcome the inefficiency of exact approaches in computing good solutions in reasonable time, especially for hard instances. In most cases, the heuristics developed to solve the GED problem are based on the analysis and the extraction of problem-dependent knowledge and characteristics. There are plenty of these methods in the literature to solve the GED problem, and in this section the most efficient ones are reviewed.

2.3.8.1 The BeamSearch GM heuristic

This heuristic, also known as *A*-Beamsearch*, has been presented by Neuhaus et al. (2006). A beam-search heuristic is an algorithm that explores a truncated search tree to compute a feasible solution to the problem. This is basically the A* method explained in section 2.3.7, with one difference that is not all child nodes are explored at each level. Instead a chosen number of nodes are heuristically picked to continue building the tree. Given two graphs $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$, the heuristic first picks a vertex $u \in V$ at the root node, and builds the child nodes corresponding to all possible edit operations on u. For all $v \in V'$, all substitution edit operations are $(u \to v)$, plus one delete operation $(u \to \epsilon)$. This defines the first level of the search tree. Then, only the first α nodes, starting from the left side of the tree, are selected to continue the construction of the search tree, with α the beam size, which is an input parameter of the algorithm. For each of the selected nodes, another vertex $u \in V$ is chosen and the process for creating and selecting child nodes is repeated. Reaching the bottom of the search tree means a complete edit path is built by the way defining a solution. The best solution found is finally returned. This method is known to be very fast because, generally, the chosen beam size is small.

2.3.8.2 The bipartite GM heuristic

The bipartite graph matching, referred to as BP in the literature, heuristic is the most famous heuristic to solve the GED problem and it has been originally presented by Riesen et al. (2007a). BP transforms the problem from finding the cheapest matching for vertices and edges simultaneously, into finding only the cheapest matching for vertices by making use of a special cost matrix. The problem is then reduced to a *Linear Sum Assignment Problem* (LSAP), which can be solved by the Hungarian algorithm in polynomial time. LSAP is the problem of finding the best (cheapest in this case) matching between two sets, such that an object in one set must be matched to one object in the second set. The LSAP problem is detailed before explaining the BP heuristic.

Data. Given two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ of elements, a cost matrix $[c_{ij}]$ can be calculated for every couple of $(i, j) \in \mathcal{S}_1 \times \mathcal{S}_2$. Line wise, the matrix has all the elements of $\mathcal{S}_1$. Similarly, the elements of $\mathcal{S}_2$ are place on the columns of the matrix. The cost can be computed by using a defined function that measures a representative distance between the couple i and j.

Variables. Only one set of decision variables is needed:

- $x_{i,j} \in \{0,1\}$, $\forall (i,j) \in \mathcal{S}_1 \times \mathcal{S}_2$: $x_{i,j} = 1$ if elements i and j are matched, and 0 otherwise.

Objective function. The objective function to be minimized is the following:

$$\min_{x} \sum_{i \in \mathcal{S}_1} \sum_{j \in \mathcal{S}_2} c(i,j) \cdot x_{i,j} \tag{2.44}$$

The objective function minimizes the cost of assigning the elements in $\mathcal{S}_1$ to the elements in $\mathcal{S}_2$.

Constraints. 2 sets of constraints are required.

$$\sum_{j \in \mathcal{S}_2} x_{i,j} = 1, \ \forall i \in \mathcal{S}_1 \tag{2.45}$$

$$\sum_{i \in \mathcal{S}_1} x_{i,j} = 1, \ \forall k \in \mathcal{S}_2 \tag{2.46}$$

Constraints 2.45 and 2.46 ensure that the matching is bijective and every object in the first set is matched to exactly one object in the second set. There exists a polynomial time algorithm to compute the optimal solution to this problem, which is called the Hungarian algorithm and is designed by Munkres (1957). The complexity of the algorithm is $O\left(max(|\mathcal{S}_1|, |\mathcal{S}_2|)^3\right)$.

In BP heuristic, the problem is reduced to a LSAP instance, where the sets of vertices of the graphs replace the sets $\mathcal{S}_1$ and $\mathcal{S}_2$. Given two graphs $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$, the cost matrix is of size $N \times N$, with $N = |V| + |V'|$, and is divided into four quadrants, similar to the matrix in Equation 2.39. The first quadrant represents the substitution of vertices and is of size $|V| \times |V'|$. The second and third quadrants are respectively for vertices deletions and insertions and are of size $|V| \times |V|$ and $|V'| \times |V'|$. Only diagonal values represent valid assignments, so the rest is set to a high value (∞) to avoid being selected. Finally, the fourth quadrant is just to complete the matrix and preserve the symmetric form: it is of size $|V'| \times |V|$ and contains only 0 values. The cost matrix is then as follows:

$$C^{BP} = \left[\begin{array}{cccc|cccc} c_{1,1}+\theta_{1,1} & c_{1,2}+\theta_{1,2} & \dots & c_{1,m}+\theta_{1,m} & c_{1,\epsilon}+\theta_{1,\epsilon} & \infty & \dots & \infty \\ c_{2,1}+\theta_{2,1} & c_{2,2}+\theta_{2,2} & \vdots & c_{2,m}+\theta_{2,m} & \infty & c_{2,\epsilon}+\theta_{2,\epsilon} & \vdots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \infty \\ c_{n,1}+\theta_{n,1} & c_{n,2}+\theta_{n,2} & \dots & c_{n,m}+\theta_{n,m} & \infty & \dots & \infty & c_{n,\epsilon}+\theta_{n,\epsilon} \\ \hline c_{\epsilon,1}+\theta_{\epsilon,1} & \infty & \dots & \infty & 0 & 0 & \dots & 0 \\ \infty & c_{\epsilon,2}+\theta_{\epsilon,2} & \vdots & \vdots & 0 & 0 & \vdots & \vdots \\ \vdots & \vdots & \ddots & \infty & \vdots & \vdots & \ddots & 0 \\ \infty & \dots & \infty & c_{\epsilon,m}+\theta_{\epsilon,m} & 0 & \dots & 0 & 0 \end{array}\right] \tag{2.47}$$

with $n = |V|$ and $m = |V'|$. Each value $C^{BP}_{ik} = c_{ik} + \theta_{ik}$, where c_{ik} is the vertex edit operation cost induced by matching vertex i with vertex k, and θ_{ik} is the cost of matching the set of edges $E_i = \{(i, w) \in E\}$ to $E_k = \{(k, w) \in E'\}$. θ_{ik} represents an estimation cost for incident edges matching, in the case where i is assigned to k. It estimates the contribution of the edges matching resulted when matching i with k to the total cost. The small assignment (matching) problem of edges, which is of size $max(|E_i|, |E_k|) \times max(|E_i|, |E_k|)$, is solved by the Hungarian algorithm as well. This is the key point of BP method, it considers local structures information around vertices by computing an estimation cost and embedding it in the cost matrix (Eq. 2.47). Then, the LSAP is solved leading to the best matching between vertices, including vertices to be deleted (the ones matched with ϵ_i) and vertices to be inserted (the ones matched with ϵ_k). From the set of vertices matching, the full edit path can be reconstructed, by determining the edges operations induced by vertices matching. Finally, the total cost (distance) is computed based on the operations selected for vertices and edges, after discarding θ_{ik} values and using the base matrices $[c_v]$ and $[c_e]$ as presented in equations 2.7 and 2.8.

BP heuristic is known to be very fast and can scale up to large instances of graphs. It is very famous in the literature as a GED solver and it is involved in many GED applications such as image analysis, handwritten document analysis, biometrics, etc. However, it lacks generality and ability to compute near-optimal solutions, and this is because it considers only information about local structures around vertices rather than the global one. Therefore, there were several attempts to improve BP by using different cost estimations, trying to include more information about vertices, and neighbor vertices as well (Serratosa and Cortés, 2015) and (Riesen and Ferrer, 2016). These attempts have led to better results and accuracy.

2.3.8.3 The fast bipartite GM heuristic

Fast bipartite GM, or FBP as denoted in the literature, is an improved version of BP. Serratosa (2014) has proposed modifications to reduce the size of the LSAP matrix. The author assumed that the cost function is a valid distance function, and therefore it satisfies all cost function conditions (Eq. 2.9 to 2.16, presented in section 2.3.1). Then, the author proves the following lemma.

Lemma 1. *Given two graphs $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$, and if the cost function is a metric, then:*

- *If $|V| \geq |V'|$, then the optimal solution does not involve any vertex deletion of G'*
- *If $|V| \leq |V'|$, then the optimal solution does not involve any vertex insertion of G*

The proof to this lemma is easy, because $c(u \to v) \leq c(u \to \epsilon) + c(\epsilon \to v)$, the cost of substituting two vertices is less than deleting one and inserting the other. This property is valid since the cost function is a distance function. Then, the author introduces a new cost matrix (Eq. 2.48). As in BP's cost matrix, the first quadrant contains costs of substitution operations, whilst in FBP each substitution operation subtracts the deletion and insertion costs from the substitution cost. By doing so, the diagonal values in the second and third quadrants are set to zero. For the sake of simplicity, θ_{ij} is omitted in the matrix, but of course it has to be added to consider local structures around vertices.

$$C^{FBP} = \left[\begin{array}{ccc|cccc} c_{1,1} - (c_{1,\epsilon} + c_{\epsilon,1}) & \dots & c_{1,m} - (c_{\epsilon,1} + c_{\epsilon,m}) & 0 & \infty & \dots & \infty \\ \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ c_{n,1} - (c_{n,\epsilon} + c_{\epsilon,1}) & \dots & c_{n,m} - (c_{n,\epsilon} + c_{\epsilon,m}) & \infty & \dots & \infty & 0 \\ \hline 0 & \dots & \infty & 0 & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ \infty & \dots & 0 & 0 & \dots & 0 & 0 \end{array}\right] \quad (2.48)$$

As in BP method, the Hungarian algorithm is used to compute the cheapest substitution operations over the first quadrant only of C^{FBP}. Of course, this way the algorithm is faster since the size of the matrix is $n \times m$. Note that, the matrix might not be square $n \neq m$, but the Hungarian algorithm can still operate by enlarging the matrix and making it square (adding the appropriate lines and columns filled with 0). The result will be an approximation and not the actual cost, because it is only the substitution operations of vertices and also because of the modifications made to costs in the first quadrant. Next, the GED distance is computed by recomputing the actual cost of matching (vertices substitution), plus the cost of deletion and insertion of the rest of vertices, without forgetting the edges operations inferred from vertices operations. The pseudo-code snippet given in Algorithm 2 details FBP procedure. The complexity of FBP heuristic is $O(max(n, m)^3)$.

FBP is an improved version of the BP heuristic and it is faster because it reduces the size of the cost matrix. However, it requires special conditions to be met by the cost function definition, otherwise it does not solve the GED problem.

2.3.8.4 The square fast bipartite GM heuristic

In the same spirit, square fast bipartite GM, known by SFBP in the literature, is a heuristic that improves the performance of FBP and BP heuristics. It is presented by Serratosa (2015b). Simply, and relying always on Lemma 1, the idea is to make the cost matrix a square matrix of size $max(n, m) \times max(n, m)$ instead of $(n \times m)$ as in FBP. To this end, SFBP treats each case alone: if $n \leq m$ it computes $C^{SFBP}_{n \leq m}$ (Eq. 2.49), else it computes $C^{SFBP}_{n \geq m}$ (Eq. 2.50).

Algorithm 2: Fast bipartite GM algorithm

```
Input  : $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$
Output: $d_{min}$

1 $C^{FBP} = ComputeMatrix(G, G')$
2 $Q_1 = FirstQuadrant(C^{FBP})$        // Extract the sub-matrix of size $n \times m$
3 $P = Hungarian(Q_1)$        // Find the cheapest permutation for vertices
    matching
4 $d'_{min} = ComputeCostPermutation(P)$        // Calculate the cost based on $Q_1$
5 $d_{min} = DeduceActualCost(P, d_{min})$        // Deduce the complete edit path and
    compute its cost
```

$$C^{SFBP}_{n \leq m} = \left[\begin{array}{cccc} c_{1,1} & c_{1,2} & \cdots & c_{1,m} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,m} \\ \vdots & \vdots & & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,m} \\ \hline c_{\epsilon,1} & c_{\epsilon,2} & \cdots & c_{\epsilon,m} \\ c_{\epsilon,1} & c_{\epsilon,2} & \cdots & c_{\epsilon,m} \\ \vdots & \vdots & & \vdots \\ c_{\epsilon,1} & c_{\epsilon,2} & \cdots & c_{\epsilon,m} \end{array}\right]_{m \times m} \tag{2.49}$$

$$C^{SFBP}_{n \geq m} = \left[\begin{array}{cccc|cccc} c_{1,1} & c_{1,2} & \cdots & c_{1,m} & c_{1,\epsilon} & c_{1,\epsilon} & \cdots & c_{1,\epsilon} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,m} & c_{2,\epsilon} & c_{2,\epsilon} & \cdots & c_{2,\epsilon} \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,m} & c_{n,\epsilon} & c_{n,\epsilon} & \cdots & c_{n,\epsilon} \end{array}\right]_{n \times n} \tag{2.50}$$

SFBP heuristic is faster than FBP, and it computes a different solution to the GED problem. But as in FBP, SFBP builds the complete edit path from the permutation found by the Hungarian method over matrix $C^{SFBP}_{n \leq m}$ or $C^{SFBP}_{n \geq m}$, and then compute the actual GED value.

To sum up, BP, FBP and SFBP heuristics are GED solvers, which rely on solving LSAP instance with special cost matrices by embedding estimation costs about local structures around vertices. SFBP is the fastest heuristic, but it is limited, as well as FBP heuristic, to cost definition conditions, otherwise it does not solve the GED problem. The main advantage of these three heuristics is the speed-up and ability to scale up to large instances of graphs. Later, Serratosa has conducted an experimental study on the three methods and the results are discussed in Serratosa (2015a).

2.3.8.5 The sorted bipartite beam heuristic

The sorted bipartite beam, denoted by SBPBeam, heuristic stands for *Sorted BP-Beam*: it is a mix of two local search heuristics and is developed by Ferrer et al. (2015).

SBPBeam heuristic uses a beam-search approach as a local search to improve the initial solution obtained by the BP heuristic. A solution S computed by BP heuristic is of the form: $S = \{(u_1 \rightarrow v_3), (u_2 \rightarrow v_1), ..., (u_4 \rightarrow \epsilon), ...\}$. Then for each operation, a weight is computed based on different modes. For instance, *confident* is a mode in which weights are computed based on considering c_{ik} value, which is the cost of substituting u_i with v_k. Another mode, called *unique*, consists in computing a weight for each operation based on Eq. 2.51. For a given u_i, if this weight is negative, this means that the current assignment $u_i \rightarrow v_k$ is sub-optimal and there exists another assignment $u_i \rightarrow v_l$ with a smaller cost.

$$\max_{\forall k \in \{1,...,|V'|\} \setminus \{l\}} c_{ik} - c_{il} \tag{2.51}$$

Many other modes are presented by Ferrer et al. (2015) such as: divergent, leader, interval, etc. After having the weights computed, SBPBeam heuristic sorts the operations inside S by ascending or descending order of the weights. Next a search tree is built, by selecting two operations and swapping them as follows:

1. Assume that after sorting S the operations order becomes $S_{sorted} = \{(u_1 \rightarrow v_3), (u_2 \rightarrow v_1), ..., (u_4 \rightarrow \epsilon), ...\}$,
2. SBPBeam selects the first two assignments $(u_1 \rightarrow v_3), (u_2 \rightarrow v_1)$,
3. then it swaps them, so they become $(u_1 \rightarrow v_1), (u_2 \rightarrow v_3)$.

So, the root node of the search tree is the first operation in S_{sorted}, and the child nodes are all the possible swapping with the rest of the operations. Next, at a child node, the total cost (or distance) is re-computed and evaluated. If it is smaller than the initial/best cost then the child is marked and second level is constructed by actually applying the swapping in the solution. Else, another child is evaluated and the current is discarded. However, to keep the algorithm tractable and compute a good solution in reasonable time, there is a parameter α, the beam size, which limits the number of child nodes to be evaluated at each level of the search tree.

SBPBeam heuristic is known to be fast and provide good sub-optimal solutions. However, it is sensitive to the value of α. Increasing it by much will increase the running time of the algorithm and setting it to a very small value will degrade the quality of the computed solution. (Ferrer et al., 2015) suggest that with $\alpha = 5$ the heuristic provides good results.

2.3.8.6 The Integer Projected Fixed Point Heuristic

Integer Projected Fixed Point (IPFP) is a heuristic that solves the GED problem, which has been proposed by Bougleux et al. (2017). It is based on a heuristic originally proposed by Leordeanu et al. (2009), also known as Frank-Wolfe algorithm, to find a solution to the quadratic assignment problem (QAP). Bougleux et al. (2017) model the GED as a QAP problem, reviewed in previous section 2.3.7.5, and then propose to apply an adapted IPFP heuristic to compute a solution. QAP formulation is known to be hard to solve. This is also the case of its continuous relaxation (by making its variables continuous instead of binaries). Both are $\mathcal{NP}$-hard problems. Algorithms as IPFP could be very helpful in computing good solutions to this minimization problem.

Algorithm 3: IPFP

```
Input  : $maxIter$
Output: $x$
1 $x^0 \leftarrow BipartiteHeur(c)$ // Initialize by calling BP heuristic
2 $x \leftarrow x^0$
3 while a fixed point is not reached OR $nbIter < maxIter$ do
4     $b^* \leftarrow LinearApproximation(x)$ // Linear approximation by 1st-order
        Taylor's expansion
5     $t^* \leftarrow SolveQAPRelaxed(x, b^*)$ // Compute the objective function by
        relaxing the variables
6     $x \leftarrow x + t^*(b^* - x)$
7 end
```

Algorithm 3 explains the steps of IPFP heuristic. The idea is to try to linearly approximate the quadratic objective function by its 1st-order Taylor's expansion around an initial solution x^0 (line 3 in Algorithm 3). The quadratic function is derived to obtain a linear function. From this linear function, a new LSAP problem (because QAP hides a LSAP related to vertices matching) is solved to obtain a solution b (line 5), which gives the direction of the largest possible decrease in the quadratic function. Then, the quadratic function consists in minimizing the QAP in the continuous domain along the direction given by b (line 7). This is repeated and after some iterations the method converges to a local minimum of the relaxed problem. This is called fixed point in the algorithm code snippet. To limit the computational time, the method has a maximum number of iterations (*maxIter*) before it stops, in case it has not converged.

Generally, IPFP heuristic converges quickly but the solution quality highly depends on the initial solution x^0, which can be computed using fast heuristics such as BeamSearch or BP heuristics. IPFP is very sensitive to the quality of the initial solution because it might lead to local optimum in bad regions of the solution space. IPFP has achieved better results than other heuristics in the literature.

2.3.8.7 The Graduated NonConvexity and Concativity Procedure heuristic

Graduated NonConvexity and Concativity Procedure (GNCCP) heuristic is introduced by Bougleux et al. (2017). It is a heuristic for the GED problem that works similarly to IPFP by solving the QAP problem. GNCCP existed in different versions as in Liu and Qiao (2014) and Zaslavskiy et al. (2009). Bougleux et al. (2017) have followed the existing works with the proper modifications to make it suitable for the GED problem.

GNCCP heuristic mainly starts by reformulating the quadratic objective function to a convex-concave function by introducing a parameter ζ that controls the concavity and convexity of the function. The function is given in equation 2.52, with $\zeta \in [-1, 1]$. In order to get a fully convex function $S_\zeta(x) = x^T x$, ζ should be set to 1. The contrast, when $\zeta = -1$, the function is concave and $S_\zeta(x) = -x^T x$.

$$S_\zeta(x) = (1 - |\zeta|)S(x) + \zeta x^T x \tag{2.52}$$

The heuristic decreases iteratively the value of ζ by small positive quantities (e.g. $d = 0.1$) in order to smoothly switch between convex and concave relaxations. Each time, it minimizes the new objective function S_ζ using IPFP heuristic. GNCCP converges when ζ reaches the value of -1 or when a valid vertices assignment (valid permutation) is found at an iteration. The convergence towards a valid permutation is always guaranteed with GNCCP: this is an important feature to this method that was proven by Liu and Qiao (2014). The results of the experiments have shown better results in terms of accuracy compared to IPFP heuristic but with more running time.

2.3.8.8 The Anytime GM heuristic

This heuristic was proposed by Abu-Aisheh et al. (2016), and it is based on an A* approach mixed with new strategies to improve its performance. This heuristic first transforms A* (presented in section 2.3.7.1) into a branch-and-bound (B&B) algorithm. So, instead of keeping all the child nodes in the $OPEN$ list, as in Algorithm 1, only important and promising nodes are stored. The others are removed (aka pruned): such decision is done by evaluating $g(p) + h(p)$ at a node p and compare it to a computed upper-bound (UB). If the current cost is worse than the best known UB ($g(p) + h(p) > UB$), the child node is pruned. Other branching and selection strategies are presented as well, such as sorting the child nodes by ascending order according to a computed lower bound $lb(p)$ and following a depth-first search strategy. B&B has more advantages than A*:

1. it is not greedy,
2. it selects carefully the children to explore,
3. it keeps tightening up the bounds that helps in pruning bad children.

Of course, letting B&B runs without time limit will reach the optimal solution, but at the price of an exponential running time and memory consumption.

The concept of anytime GM heuristic is the ability to stop the B&B algorithm at any point and return the best solution computed so far. By varying the time, the quality of the solution may vary as well: allowing more time will lead to better solutions. This gives an advantage over A* methods, by overcoming memory and execution time bottlenecks. Based on the results published in (Abu-Aisheh et al., 2017), the unparallelized version of the heuristic was outperformed by the IPFP heuristic.

2.3.9 Other approaches to solve the GED problem

The list of methods and in particular heuristics presented, does not cover all the existing ones in the literature. It lists the most famous or newest methods that solve the GED problem. Nonetheless, other heuristic approaches can be found in the literature. A recent work, by Chang et al. (2017), proposes an efficient way to compute a tight lower bound for partial mappings along the search. This lower bound can be used in A* and tree-search based algorithms to improve the pruning and selection strategies.

With the same intention, Chen et al. (2017) have designed a method for generating successors of nodes in the search tree in A* algorithm. They eliminate invalid and redundant mappings along the search. Also, they improve the memory consumption of A* by storing visited nodes in beam-stack, which helps in backtracking phases. Beam-stack search is a combination of a depth-first search and beam search strategies, with the ability of storing partial solutions and backtracking technique.

Gouda and Hassaan (2016) have proposed a A* algorithm based on depth-first search strategy to solve the GED problem, but instead of carrying it out over vertices, the search tree is constructed based on edges edit operations. In fact, matching two edges underlies two substitution operations on their incident vertices. The vertices matching are deduced after determining the edges matching.

2.4 Summary and prospects for moving forward

This chapter has given an insight about GM problems and the GED problem in particular, and their importance in many research domains with numerous applications. The outline of the chapter started by presenting the graphs as powerful tools in representing structural information such as objects and patterns. Then, the chapter carried on with the emergence of graphs in many research domains and the need of designing common frameworks to perform graph comparison. This has led to the birth of graph matching problems, which have been studied carefully, resulting in two classes of matching: exact (EGM) and error-tolerant (ETGM). Moreover, many problems have been defined for each class. It is argued earlier (section 2.2.3) the importance of ETGM over EGM. Afterwards, the chapter provides a listing of ETGM problems such as substitution-tolerant graph isomorphism, error-tolerant graph isomorphism, graph edit distance, etc. Among those problems and after a fair comparison (in section 2.2.4), the GED problem was selected as the ETGM problem for which this thesis will be focusing on. Few important factors have put the GED problem before other ETGM problems, which are:

- The diversity of the applications where the GED problem is involved that touch many application fields, such as: Pattern Recognition, Chem- and Bio-informatics, etc.
- The flexibility of the problem that can be adapted to solve other GM problems such as maximum common subgraph, graph and subgraph isomorphism.
- The generality of the problem, since the GED problem has been considered widely as dissimilarity measure for graphs.

That is why there is a section dedicated to the GED problem: it starts by the problem statement and definition, applications, challenges and existing methods to solve it. When discussing GED challenges, two questions have arisen:

1. Regarding the final application, is it worth spending time to compute accurate solutions?
2. What is the impact of a better solution on the similarity search or on the matching, with respect to the application?

Table 2.2: Summary of GED (exact and heuristic) methods with two criteria speed and accuracy.

	Method	**Speed**	**Accuracy**
Exact	A*	x	
	JH-MILP	xxxx	
	F1-MILP	xx	
	F2-MILP	xxx	
	QAP	x	
Heuristic	BeamSearch	+++	++
	BP	+++	++
	FBP	++++	++
	SFBP	++++	++
	SBPBeam	+++	++
	IPFP	+++	++
	GNCCP	+	+++
	Anytime GM	++	++

The first question is in the context of the dilemma between fast GED solvers and their accuracy. There are many heuristics designed to converge fast and return good solutions. Knowing that the GED problem is $\mathcal{NP}$-hard, those heuristics will not be able to explore the solution space of the problem efficiently in order to find optimal solutions. Often, those heuristics get stuck with local optima. The second question is to engage the importance of studying the impact of the solution on the similarity search. To verify if reaching near-optimal solutions is as good as the optimal ones.

Many GED methods are reviewed and discussed, some of them are exact and others are heuristic algorithms. A summary of all those methods is found in Table 2.2, commenting on two factors the speed and the accuracy. Note that the scores given to each method are deduced after reviewing the papers that presented it and relying on the experimentation results reported. All exact methods are accurate because they always compute the optimal solutions, however the variant is the speed of the method in finding the optimal solution. Considering the exact methods, JH formulation seems to be the fastest, but it does not solve the general case of the GED problem. This leaves us with two formulations F1 and F2 that are definitely faster than A* and QAP. Talking about the exact methods, which are limited, there is clearly a room for having new and competitive MILP formulations that model the problem efficiently and solve it at least as fast as JH but for the general problem, or even faster.

Regarding the heuristic methods, the comparison shows that most of them are focused on the speed and thus having almost the same accuracy. Nevertheless, most of the heuristics are considered as good GED solvers with good accuracy (to a certain limit) for many applications. It is challenging to provide a new heuristic that can make a better trade-off between speed and accuracy. Such a contribution is always appreciated, because generally, heuristic approaches are more suitable for final GED applications.

To sum up, and based on the methods mentioned in Table 2.2, it is interesting to investigate new techniques, which have not yet been applied to the GED problem, that lead to designing new and efficient exact and heuristic methods. Providing such methods is a good contribution to the GED problem and GM problems in general. The focus of this thesis is going, then, to be:

- Develop an exact method that solves the general GED problem, which is efficient in terms of speed and can solve harder instances to optimality than existing ones. It, then, can be considered as a basis method for heuristics evaluation, by comparing the performance and the solutions computed by the heuristics to the optimal and best known solutions.
- Design a heuristic method that solves the GED problem and that offers a good compromise between speed and accuracy and outperform the existing heuristic algorithms.

Chapter 3

Optimization and complexity

Contents

3.1 Operations Research (OR)

Based on the Operational Research Society of Great Britain, OR is defined by (Operational Research Society, 1962):

"Operational research is the application of the methods of science to complex problems arising in the direction and management of large systems of men, machines, materials and money in industry, business, government, and defense. The distinctive approach is to develop a scientific model of the system, incorporating measurements of factors such as chance and risk, with which to predict and compare the outcomes of alternative decisions, strategies or controls. The purpose is to help management determine its policy and actions scientifically."

OR is an essential scientific research field that touches numerous disciplines such as applied mathematics, combinatorial optimization, computer science and economics. It is basically concerned by problems that involve making a decision, and to be more precise, making the best decision among a possibly large set of decisions. Usually, such problems are called *complex decision-making problems*. OR problems are crucial because they answer to practical problems in important disciplines, notably industrial engineering and operations management, and computer science among others. Often, the goal of a decision problem is to, for example, determine the most effective decision regarding an objective (increasing a profit, decreasing a loss, etc.), that is going to be based on given complex data, along with satisfying some defined constraints. This decision is, then, called a solution to the problem and more, it is said to be optimal if it is the best decision taking into account the present circumstances. Then, it is not possible to come up with a better one. Depending

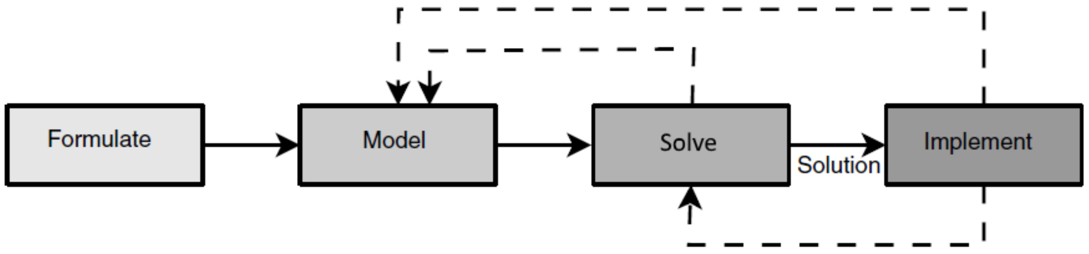


Figure 3.1: The classical process in decision making: formulate, model, solve, and implement Talbi (2009)

on the nature of the problem and the application field, seeking such high quality solutions leads to important operational improvements, e.g. greater productivity, lesser risk, better performance and efficiency. A real-world example where OR based-decisions are really effective, is when considering the *Travelling Salesman Problem* (TSP) (Hoffman et al., 2013), that has a wide range of real applications. Given a list of cities and distances between them, the TSP is the problem of finding the shortest route, such that each city is visited once and then return to the starting city. A real-world problem, relates to the TSP, is companies that offer delivery products to customers, where they have to answer the demands of all customers at the minimum cost to increase their profits. So, finding the best (optimal) route, hereby the best decision, is necessary and of interest to those companies, because it allows them reducing the expenses and increasing the profits. However, the TSP is a pure combinatorial optimization problem that is known to be very hard to solve to optimality.

OR offers a collection of mathematical techniques and tools to model practical decision problems. The idea is to come up with a rigorous system model that takes, as an input, the quantitative data available to the problem, and returns a decision as an output. Figure 3.1 depicts the possible steps of a decision making process. So, after filtering the important factors and objectives of the problem by degrees of importance, a system can be designed incorporating those factors. This step is called problem formulation. The next step is the modeling, which provides a scientific model to the problem based on the formulation. The model becomes, then, a way for analyzing the behavior of the system. The art of modeling problems, despite its importance, is very complex and there are plenty of modeling classes, such as mathematical models, probabilistic models, prediction models, heuristic models, etc. The online book by Mishra (2009) gives more insights and details about modeling classes and techniques. Few important rules must be taken into accounts, which are:

- the model must be as simple as possible,
- the model must cope with all importance aspects of the problem,
- the model must be validated through an evaluation procedure,
- the model's solution must fit the needs and the purpose of the problem.

Those rules, among others not fully detailed here, are compulsory to design coherent and representative models.

Mathematical models, which are also called symbolic models, are very important and well established in OR field. They represent the decision variables of the system using numbers or letters. The variables are, then, employed in mathematical equations that describe properties and constraints of the system. The main advantage of these models is that there are numerous solution techniques to manipulate and extract solutions from mathematical equations and functions, especially in the case of linear functions. Furthermore, the ability to design a model with a function relating the decision variables, plus mathematical equations to force conditions on decision variables, turns out to be very suitable to optimization problems. Optimizing such a function aims at minimizing or maximizing a cost, depending on the system, while satisfying a certain number of constraints. Mathematical optimization models have proven efficiency dealing with a wide range of practical combinatorial problems in many sectors such as transportation, health-care, economics, finance, etc.

Besides the modeling techniques, and throughout the years, OR has developed many solution techniques and methods to solve mathematical models. The solution techniques are usually expressed by algorithms, referred to as implementation step in Fig. 3.1, that are composed of strict iterative instructions to find a solution to the problem. Notice that, if the process is repeated to improve the model and the solution, it is, then, called the optimization model. Basically, the algorithm starts with either finding partial solutions and then keep iterating until reaching the optimal solutions, or computing an initial solution and keep trying to improve it by getting closer to the optimum. Of course, one may wonder if an algorithm can solve the problem regardless of the size of the instance or the input data. To answer this question, OR gives the ability of measuring the difficulty of the problem, which reflects the complexity of the algorithm to solve it. This is known as complexity theory. A difficult problem is most likely to have a huge number of decisions, therefore an algorithm to solve it will have, typically, to lookup all the decisions and select the best one. For certain problems, this might require exponential running time and/or space. In such case, the problem is distinguished from other problems, called polynomial problems, where it is possible to design an algorithm to find the optimal decision in polynomial running time and space. It is imperative in most of the cases, if possible, to know the difficulty of the problem, which will help in designing efficient algorithms to solve it. OR takes care of this by providing means to study the complexity of algorithms and problems, and classify them. In the next section, complexity theory of algorithms and some important notions are discussed, followed by an overview of the most used solution techniques in solving combinatorial optimization problems.

3.2 Combinatorial optimization and notion of complexity

It is quite common and natural to ask the question about the complexity of a decision or an optimization problem. Analyzing the complexity enables determining whether the problem can have a polynomial time algorithm to solve it, or the problem is very hard and thus, finding the optimal solution is intractable. This is crucial because it notably impacts the decision of which kind of algorithms to design to solve the problem. Of course, there is more to know about complexity, but the main idea is to study and evaluate the difficulty of a given problem. The journey all started with the works of Cook (1971) and

Karp (1972) about problem reduction and that the class $\mathcal{NP}$-complete is not empty (under certain assumptions). Then, the book by Garey and Johnson (1979), entitled "Computers and intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness", summarized all previous works and was exclusively the first on the theory of $\mathcal{NP}$-completeness. The authors, in the book, have changed the following sentence "I can't find an efficient algorithm. I guess I am dumb" to "I can't find an efficient algorithm, because there is no such algorithm possible!". Also, they discuss the intractability of difficult decision problems and they class them in the $\mathcal{NP}$-complete class of problems. Those are the problems for which finding the optimal solutions is not possible in polynomial time and space. In contrast, problems that can be solved in polynomial time and space are in class $\mathcal{P}$ (polynomial). This section is organized as follows: it gives a general definition of optimization models, then discusses the complexity of problems and finally introduces some classic optimization algorithms to solve them.

3.2.1 Optimization models

An optimization problem is defined by 4-tuples (I, S, m, g), where:

- I is the set of instances;
- $S(i)$ is the set of feasible solutions, for an instance $i \in I$. It is called also the search space or solution space;
- $m(i, s)$ is the measure of solution $s \in S(i)$, for an instance $i \in I$;
- $g \in \{min, max\}$ is the objective function.

For a given instance $i \in I$, the optimization model can be expressed as follows:

$$O(i) = g_{s\in S(i)}\{m(i, s)\} \tag{3.1}$$

Definition 19 (Global optimum)**.** *$s^*(i)$ is said to be a global optimum if and only if it has the smallest (resp. largest) objective function in the case g is min (resp. max), among all the other solutions $s(i) \in S(i)$.*

Therefore, the main objective when solving an optimization problem is to find the optimal solution $s^*(i)$ for an instance $i \in I$. The difference between a decision problem and an optimization problem is: the former's goal is to find a feasible solution $s(i) \in S(i)$, while the latter aims at finding the optimal solution $s^*(i) \in S(i)$. In addition, there exist many ways to formulate optimization models, but a very common one is by using mathematical programming, and in particular linear programming models. It is not the case of all optimization models, especially the case of complex models, where it is not easy to linearly express the problem. Linear programs are composed of: a set of decision variables, a linear objective function and a set of linear constraints. A Linear Program

(LP) can be formulated as follows:

$$\begin{aligned} &\min_x\{c^T x\} \\ &Ax \leq b \\ &x \in \mathbb{R}^n \\ &x \geq 0 \end{aligned} \tag{3.2}$$

where $c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ are vectors of coefficients, $A \in \mathbb{R}^{m\times n}$ is a matrix of coefficients, and x is a vector of continuous variables to be computed. Vectors c, b and matrix A are known, and solving the LP will result in computing the optimal solution $s^*(x^*)$, with x^* the vector of variables associated to an optimal solution. The reason behind the popularity of LP models is that there exists an exact algorithm to solve them to optimality. The algorithm is the well known *Simplex* algorithm, that was designed by Dantzig (1951). Both, the objective function and the feasible region of LP problems, are convex, which makes it easy to find any local optima that is also a global optimum. Consequently, being able of modeling an optimization problem with a LP model, means solving the problem to optimality is guaranteed, which is very convenient to the decision maker. Note that, without lose of generality, Eq. 3.2 uses *min*, because if the function is being maximized, it is equivalent to minimizing its negative, i.e. $max\{f\} \iff min\{-f\}$. LP is an example of mathematical programming techniques for problems modeling and, of course other models exist under this category. The important ones are discussed later in this section.

3.2.2 Complexity theory

An algorithm to solve an optimization problem is characterized by two factors: the time and space it requires. The complexity of the algorithm relates to how much time it needs in order to compute an optimal solution. Given an instance of size n to a problem, the time complexity is the number of steps required during the solution process of the problem. Since the complexity is based on the worst-case analysis, then it is not mandatory to obtain the exact number of steps, but instead an asymptotic bound of the total number of steps. This is represented by the notation O, which is defined as follows.

Definition 20 (O notation)**.** *The complexity of an algorithm is $f(n) = O(g(n))$ if there exist positive constants n_0 and c such that, $\forall n > n_0$, $f(n) \leq c.g(n)$.*

Definition 20 introduces the notion of upper bound on the complexity and Definition 21 introduces the notion of lower bound.

Definition 21 (Ω notation)**.** *The complexity of an algorithm is $f(n) = \Omega(g(n))$ if there exist positive constants n_0 and c such that, $\forall n > n_0$, $f(n) \geq c.g(n)$.*

Definition 22 (Θ notation)**.** *The complexity of an algorithm is $f(n) = \Omega(g(n))$ if there exist positive constants n_0, c_1 and c_2 such that, $\forall n > n_0$, $c_1.g(n) \leq f(n) \geq c_2.g(n)$.*

Based Definition 22, the complexity $f(n)$ can be bounded by the function $g(n)$.

Generally, Ω and Θ are not easy to find, and can be derived after defining the O complexity of the algorithm. Finding these complexities enables comparing different algorithms

Table 3.1: Comparison of polynomial and exponential time complexities depending on the size of the problem (Garey and Johnson, 1979)

	Complexity	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
Polynomial	$O(n)$	0.00001 s	0.00002 s	0.00003 s	0.00004 s	0.00005 s
	$O(n^2)$	0.0001 s	0.0004 s	0.0009 s	0.0016 s	0.0025 s
	$O(n^5)$	0.1 s	0.32 s	24.3 s	1.7 min	5.2 min
Exponential	$O(2^n)$	0.001 s	1.0 s	17.9 min	12.7 days	35.7 years
	$O(3^n)$	0.059 s	58.0 min	6.5 years	3855 centuries	2×10^8 centuries

in terms of the worst-case complexity. This is achievable because the complexity gives an estimation of the growth rate of the algorithm running time as a function of the problem size. It follows next the definitions of polynomial and exponential algorithms, which are strongly related to the complexity of the algorithm.

Definition 23 (Polynomial algorithm)**.** *A polynomial time algorithm is of complexity $O(p(n))$, where $p(n)$ is a polynomial function of n.*

Definition 24 (Exponential algorithm)**.** *An exponential time algorithm is of complexity $O(c^n)$, with $c > 1$.*

An algorithm with an $O(n^k)$ complexity is said polynomial, with k the degree of the polynomial function. An algorithm with an $O(3^n)$ complexity is said exponential. To give a clearer picture about the big difference between exponential and polynomial complexities, Table 3.1 shows different complexities running times for algorithms and how they grow with the size of the problem. The exponential complexity is very sensitive and explodes with the growth of the problem size to reach centuries, which makes the algorithms inefficient in solving the problems.

Generally, a problem can have several algorithms to solve it, so the complexity of the problem is determined by selecting the fastest algorithm to solve it in the worst-case. A problem is said to be polynomial, if there exists an algorithm to solve it in polynomial time. Otherwise, the problem is said intractable. In complexity theory, there are different complexity classes for problems. Basically, the classification is done over decision problems, but there are also classes for optimization problem derived from the decision problem classes. In the following, a short review of the main complexity classes is given.

Definition 25 ($\mathcal{P}$ class)**.** *A decision problem belongs to class $\mathcal{P}$, if there exists a deterministic algorithm to solve it that has a polynomial complexity.*

The decision problems belonging to this class are known to be relatively easy to solve. Problems like finding the shortest path or the minimum spanning tree in a graph, are in the class $\mathcal{P}$ and there exist polynomial time algorithms to solve them to optimality. The next class that comes after $\mathcal{P}$ class is called *Non-Deterministic Polynomial* ($\mathcal{NP}$).

Definition 26 ($\mathcal{NP}$ class)**.** *A decision problem belongs to class $\mathcal{NP}$, if it can be solved by a non-deterministic algorithm in polynomial time.*

The above definition relates to the notion of *Turing machine*, which can be seen as an algorithm. An algorithm has usually a series of inputs, and it reads each input and

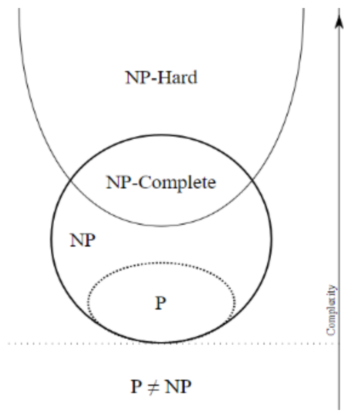


Figure 3.2: Classes $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-complete and $\mathcal{NP}$-hard relations in case where $\mathcal{P} \neq \mathcal{NP}$ (Commons, 2018)

infers the next step. In a deterministic algorithm, at each new input, only one new state is possible based on the current state and the read input. This is not the case for non-deterministic algorithms, in which it can occur that when reading a new input, multiple alternative states are possible. In this case, it assumes that there is an "oracle", which can tell which new state to consider in order to reach the final solution. The ability of guessing the next correct step is the particularity of a non-deterministic algorithm. If the algorithm is capable of finding a positive answer at the end, by using some oracle, then the computing complexity is polynomial and such a problem is then classified in the class $\mathcal{NP}$. The graph isomorphism problem explained in the first chapter (section 2.2.2) belongs to class $\mathcal{NP}$. Obviously, every problem in class $\mathcal{P}$ can have a non-deterministic algorithm to solve it, but is the opposite valid? Is it possible to find a deterministic algorithm to solve $\mathcal{NP}$ problems? This is a fundamental question in complexity theory, and it is admitted that if the answer is yes, then $\mathcal{P} = \mathcal{NP}$. It is still an open question and it is listed on the millennium prize problems by the Clay Mathematics Institute for a prize of 1 million USD for a first theoretical proof. So far, it is assumed that $\mathcal{P} \subseteq \mathcal{NP}$ and other classes are derived based on this assumption. A more specific class of problems that is included in $\mathcal{NP}$ but not in $\mathcal{P}$, is the class $\mathcal{NP}$-complete ($\mathcal{NPC}$).

Definition 27 ($\mathcal{NPC}$ class)**.** *A decision problem A is $\mathcal{NP}$-complete, if A is in $\mathcal{NP}$ and every problem in $\mathcal{NP}$ is reducible to A in polynomial time.*

If the decision problem can have all its solutions verified in polynomial time ($\mathcal{NP}$), and if there exists a polynomial time algorithm to solve it, then this algorithm can be used to solve all problems in $\mathcal{NP}$. This is the concept of reduction. So, $\mathcal{NPC}$ problems are the hardest ones in $\mathcal{NP}$. The notion of reduction and in particular polynomial time reduction is defined as follows.

Definition 28 (Polynomial reduction)**.** *Given two decision problems A and B, A is reduced polynomially to the problem B if:*

1. *for all I_A instance of A, there exists a function that transforms I_A into an instance I_B of B in polynomial time to the size of I_A,*

2. the answer to instance I_A is yes if and only if the answer is yes to instance I_B.

Another important class is the $\mathcal{NP}$-hard class, which handles optimization problems.

Definition 29 ($\mathcal{NP}$-hard class)**.** *An optimization problem is $\mathcal{NP}$-hard, if its associated decision problem is $\mathcal{NP}$-complete, or if another $\mathcal{NP}$-hard optimization problem is reducible to it, and it does not belong to $\mathcal{NP}$.*

Figure 3.2 shows the relationships between the four presented classes, under the general assumption that $\mathcal{P} \neq \mathcal{NP}$. In fact, complexity theory has been studied further and other classes and sub-classes are introduced, especially the ones that consider the space complexity beside the time. For example, the class $\mathcal{NL} \subseteq \mathcal{P}$, which contains decision problems that can be solved in polynomial time using a logarithmic amount of space. Other classes exist such as PSPACE (polynomial-space), EXPTIME (exponential-time), EXPSPACE (exponential-space), etc.

This section covers the basic notions of algorithms and problems complexities. It is a major factor to consider when dealing with optimization problem, because knowing the complexity of the problem helps in the design of the best algorithm to solve it. If, for instance, an optimization problem is $\mathcal{NP}$-hard, then it is clear that designing a polynomial time optimal algorithm is not possible (unless $\mathcal{P} = \mathcal{NP}$). In such case, it is preferable to consider heuristic and approximation approaches, which can provide sub-optimal solutions in polynomial time. Later in this chapter, most common solution techniques that OR provides to tackle optimization problems, will be discussed.

3.2.3 Mathematical programming

Mathematical programming is often considered when dealing with optimization problems. It comprises a set of modeling techniques that can be used to express decision and optimization problems mathematically. The definition of an optimization model is given in Eq. 3.1, which consists of an objective function to be minimized or maximized depending on the problem, and a solution space that is defined based on the constraints and properties of the problem. Then, mathematical programming gives a mathematical representation by formulating optimization models by the means of linear or non-linear functions and equations. Mathematical programming can result in multiple mathematical models depending on the problem, which are classified under different categories. This section is dedicated to review the most common optimization models in OR. Of course the list is not exclusive and other modeling techniques exist in the literature.

3.2.3.1 Linear programming

This form of mathematical programs is the most common one. They are used to model any problem, whose objective function and constraints can be modeled using linear equations. An example of a linear model is given in Section 3.2.1. The model has a set of decision variables, an objective function, and a set of constraints. They are all put together in Eq. 3.2. The main characteristics of this model are:

- decision variables are continuous,

- the objective function is linear,
- all model's constraints are linear as well.

The importance of such models comes from the fact that they can be solved to optimality in polynomial time by using, for example, the Simplex method designed by Dantzig (1951). These models are very easy to solve and considered as the foundation of mathematical programming, because they are heavily used when solving other mathematical programs.

3.2.3.2 Integer linear programming

Integer linear programming, denoted as ILP, is another form of mathematical programming. One difference between linear and integer linear programs is that the decision variables in the model are integers instead of continuous. The objective function and constraints are still linear.

$$\begin{array}{c} \min_x\{c^T x\} \\ Ax \leq b \\ x \in \mathbb{N}^n \end{array} \tag{3.3}$$

where $c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ are vectors of coefficients, $A \in \mathbb{R}^{m \times n}$ is a matrix of coefficients, and x is a vector of integer variables to be determined. ILP models are very useful in modeling a wide range of optimization problems in many operation fields. However, the fact that the model's variables are integer, which reflects the physical indivisibility of the objects represented by the variables, makes the model very hard to solve. Solving an ILP model by an exact method, which enumerate all the feasible solutions (e.g. branch-and-bound algorithm), is very costly and requires an exponential amount of time with the growth of the number of variables. In its general form, an ILP model is a $\mathcal{NP}$-hard problem (Floudas and Pardalos, 2001). ILP models are, instead, solved by techniques that aim at reducing the solution space, such as: cutting planes, dynamic programming, relaxation-based method, etc. Another particular form of ILP models are 0-1 integer linear programming (0-1ILP) or binary linear programming (BLP). Those models have only binary decision variables, and they are also very hard to solve.

$$\begin{array}{c} \min_x\{c^T x\} \\ Ax \leq b \\ x \in \{0,1\}^n \end{array} \tag{3.4}$$

3.2.3.3 Mixed integer linear programming

Mixed Integer Linear Programming, denoted by MILP, is quite similar to ILP models, except a partial set of the variables are continuous. So, it is a mix of LP and ILP models.

$$\begin{gathered}\min_x\{c^T x\}\\ Ax \leq b\\ x_i \in \{0,1\}^{|B|}, \forall i \in B\\ x_j \in \mathbb{N}^{|I|}, \forall j \in I\\ x_k \in \mathbb{R}^{|C|}, \forall k \in C\end{gathered} \tag{3.5}$$

where $c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ are vectors of coefficients, $A \in \mathbb{R}^{m\times n}$ is a matrix of coefficients, and x is a vector of variables to be determined. The variable index set is split into three sets (B, I, C), respectively, stand for binary, integer and continuous variables. These models are known to be hard to solve and several works have tried to analyze the source of complexity of MILP models such as the work by Till et al. (2003). MILP (as well ILP, BIP) models can be solved efficiently with existing black-box solvers such as CPLEX, Gurobi or Xpress.

3.2.3.4 Discrete optimization

Discrete optimization models are a more general form of, without loss of generality, MILP models. In discrete optimization models, some or all the variables are defined over a set of discrete sets. For instance, the integer variables in an ILP model belong to a subset of integers. In contrast, LP models have variables defined over continuous intervals, and then can have any value within a range of values. The discrete set can be composed of a set of objects, assignments, combinations, schedules, etc.

3.2.3.5 Optimization Under Uncertainty

All the aforementioned models are considered as deterministic: the input data (vectors c, b and matrix A) are known and fixed beforehand for a given instance. However, this assumption of data availability is not valid for many real-life problems and for many reasons. Sometimes, the data is subject to measurement errors or noise, which is the case, for instance, of data collected from sensors. Other times, the data is not all/partially known at the present time and may represent information about the future (e.g. product demand). The above models will fail in providing realistic and applicable solutions, even if they were able to find the optimal solutions. An alternative and efficient way to model such problems is by using optimization under uncertainty. This kind of optimization alters the objective function by introducing an additive noise, which is represented by a probability distribution. The noise, for instance, can be assumed to be normally distributed $N(0, \sigma)$, with mean equal to 0 and a variance equal to σ (Jin and Branke, 2005). Another distribution can be used, such as Non-Gaussian Cauchy. The model will have an objective function of this form:

$$f_{noise} = \int_{-\infty}^{+\infty} [f(x) + z]p(z)dz \tag{3.6}$$

with $p(z)$ the probability distribution of the noise. Each solution may have multiple values of f_{noise}, and a simple solution to overcome this is to take the mean of the possible solutions.

3.2.3.6 Dynamic Optimization

Another variant of optimization problems, which is also non-deterministic, involves input data that may be affected over time. Dynamic optimization takes good care of such a case, by introducing a time factor in the objective function. Unlike optimization under uncertainty, the objective function is deterministic at a time t, but may varies over time, i.e. multiple evaluations at time t always return the same solution.

$$f_{dynamic}(x) = f_t(x) \tag{3.7}$$

with t the time at which the objective function is evaluated. The main concern in dynamic optimization is how to detect the changes in the system and the ability to adapt the current state and adjust it to the change, without resetting the solution of the problem. Doing so may require involving forecasting and prediction strategies to accommodate for possible future changes while staying near optimal solutions.

3.2.3.7 Robust Optimization

Robust optimization can be considered relatively as another way of modeling uncertainty in an optimization problem. In this modeling approach, the decision variables are subject to perturbations after a final solution has been computed. The robustness, then, is the ability of computing solutions that are acceptable with respect to minor changes in some decision variables. Similarly as in optimization under uncertainty, a probabilistic distribution of the variations is injected into the objective function.

$$f_{robust}(x) = \int_{-\infty}^{+\infty} f(x+\delta)p(\delta)d\delta \tag{3.8}$$

with $p(\delta)$ the probability distribution of the variables perturbations. The objective function in robust optimization is considered as deterministic. In addition, it seeks the best compromise between the quality of the solutions and their robustness with regards to the perturbations in the decision variables. Jin and Sendhoff (2003) have shown that such model can be reformulated as multi-objective optimization problem.

3.3 Optimization methods

OR researches have developed throughout the years many solution methods to solve optimization problems. As discussed earlier, the solution method choice is directly related to the complexity of the problem. A $\mathcal{NP}$-hard optimization problem cannot be solved to optimality, or in other words, it is not possible to develop an exact algorithm to solve it in polynomial time. The notion of exact algorithm stands for algorithms that seek optimal solutions for a decision/optimization problem. There are some exact algorithms

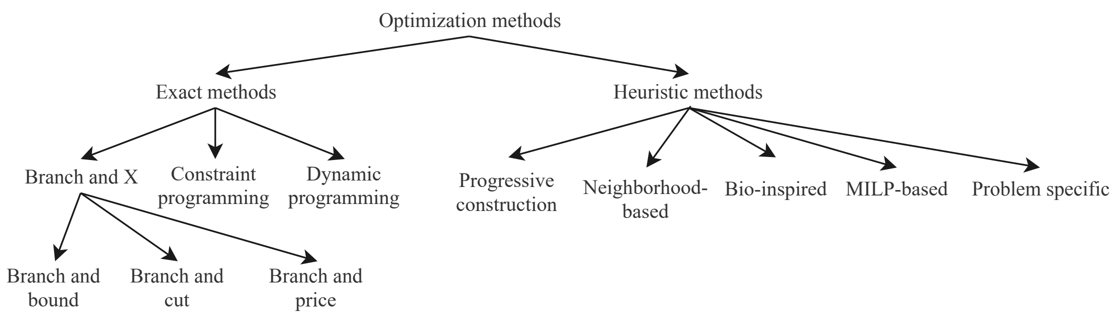


Figure 3.3: Hierarchy of optimization methods

that are general enough to be used for solving any optimization problem, even the $\mathcal{NP}$-hard ones. However, there is the risk of facing exponential running time before finding the optimal solution. In this case, researchers tend to design heuristic methods that aim at computing sub-optimal solutions in polynomial time. Then, the art of heuristics design relies on their capabilities in finding solutions that are close as much as possible to the optimal ones. Therefore, they are suitable for hard optimization problems. The hierarchy of optimization methods is depicted in Fig. 3.3, with more sub-levels details about different methods belonging to exact and heuristic families of methods. In the rest of this section, a general review of the important methods is given.

3.3.1 Exact methods

For problems in class $\mathcal{P}$, exact polynomial algorithms are usually specific for these problems. However for $\mathcal{NP}$-hard problems, three main classes of exact exponential algorithms, can be found in the literature: Branch and X, Constraint programming and Dynamic programming.

3.3.1.1 Branch and X

Branch-and-bound. The most common method to solve optimization problems, especially those modeled by ILP and MILP, is branch-and-bound (B&B). Later, it was modified and improved resulting in methods, like branch-and-cut and branch-and-price. A B&B method proceeds by exploring all the solution space of the problem, by means of a search tree. The root node of the tree represents the problem being solved, and then children nodes are created by selecting a variable to branch on and assigning to it all possible values: a child node represents an assignment. Next, it picks a node according to a search strategy: this one controls the way the search tree is explored. Many search strategies exist like depth-first, breadth-first or best-first. Building the tree is continued until reaching a leaf node, which in this case represents a feasible solution to the problem. The children nodes represent partial solutions, which are not feasible. In the basic form of B&B, the tree is fully constructed and among all the feasible solutions found, the optimal (best) one is retained. However, it turns out that employing techniques, such as pruning, help in discarding some partial solutions that will not lead to the optimal one. This is a very important technique in order to reduce the size of the tree in memory and the construction time. To perform pruning, a lower bound can be computed at each node that is an estimation of the solution value in case the exploration continues that way. This lower bound can be compared to an upper bound, which is the best solution known so far. If this lower bound is greater (in the case of a minimization problem) than the upper bound, then continuing down that road is useless and the node can be ignored. The process in itself is simple, but the time and memory complexities can quickly grow and become exponential, when building the search tree. The performance of the method is very sensitive to the variable selection when branching (also called branching scheme), search strategy and lower bound computation for child nodes. There have been many works to improve those strategies, and lately it is shifting towards techniques based on prediction and forecasting to guarantee good decisions. A survey was conducted by Lodi and Zarpellon (2017) about the existing strategies for variable selections and search strategies in the context of Machine Learning.

Branch-and-cut. It is based on cutting plane approaches, which were proposed by Gomory et al. (1958). The idea is to generate cuts that are constraints determined by analyzing the structure of the problem. The constraints are added to the LP relaxation (all variables become continuous) of the MILP model of the problem. The intent is to improve and tighten the lower bound. Note, that these constraints must be valid and must not chop regions of the solution space containing optimal solutions. To fulfill their purpose, the cuts must be efficient in improving the lower bound and reducing the gap with the upper bound. The cuts have been shown to be very effective in solving some problems that have particular structures, however they perform poorly on some other problems. A survey about cuts generation techniques and embedding them in B&B, which results in branch-and-cut, can be found in (Jiinger et al., 1995) and (Nemhauser and Wolsey, 1988).

Branch-and-price. This method is based on column generation techniques. Column generation was proposed by Gilmore and Gomory (1961), and it intends at decomposing the main (master) problem into smaller sub-problems, to reduce the complexity and memory requirement to solve it. The decomposition produces problem formulations easier to solve, which gives better bounds than solving the LP relaxation. The decomposition is based on selecting a group of variables among the whole set of decision variables in the master problem, such that these variables are most likely to be part of optimal solutions (being set to 1 in the case of binary variables). The belief is that, in large problems with many columns (variables), the majority of the columns are useless for finding the optimal solution. The new problem (with the selected variables) is called the restricted master problem. To guarantee the optimality, another sub-problem is solved to determine the new columns that can be added at the next iteration, (pricing problem) to improve the solution of the master problem. In case where the pricing problem is hard to solve, heuristic approaches can be employed to find those variables. Next, reoptimizing the relaxed version of the restricted master problem with the newly added variables, normally, improves the lower bound. The process is repeated until no more variables can be added or an optimal solution is found. This is the branch-and-price method, which may be problem dependent because it requires the problem to be formulated in a way such that branching rules can be extracted and so the pricing problem is not very hard to solve. Finally, it is possible to implement cutting planes to tighten the LP relaxation of the problem, in this case the method is called branch-price-and-cut. The paper by Barnhart et al. (1998) presents a detailed survey about column generation techniques.

Note that while presenting these methods, the optimization problem assumed to be solved is a minimization one, without loss of generality (because a maximization problem can be rewritten as a minimization one). It is worth mentioning that branch-and-bound method is the main algorithm implemented in MILP solvers.

3.3.1.2 Constraint programming

Constraint programming (CP) introduces a unique and general method for solving optimization problems, as well as, modeling them. It models the problem by means of variables and constraints to link the variables. In addition, CP is based on logic programming. For instance, to force variables in a given set to be different/distinct, a global constraint can be added, e.g. $all\_different(x_1, x_2, ..., x_n)$. CP is known to be efficient in solving feasibility problems (finding a feasible solution) rather than optimization problems (finding an optimal solution). The goal is to narrow down a vary large set of possible solutions to a subset by, incrementally, adding constraints to the problem (this is similar to cuts in cutting plane technique). There are two approaches to solve a CP model (Mayoh et al., 2013). The first one is called refinement, and it assumes that all variables in the model are free and can have any value between their bounds. Then, the solution procedure performs analysis over the model's constraints to infer some properties that help in reducing the variables bounds until reaching the point where it is possible to assign a specific value to some variables. The inference is continued over the rest of the variables as much as possible. The second approach is solving by perturbation. Unlike refinement, this approach presumes initial values to certain variables, then each constraint is checked for feasibility. When a constraint is not satisfied, a step back is taken to change the initial value of a variable and this change is propagated to test constraint feasibility and to try to fix new variables. The perturbation carries on until no more possible variables can be fixed without affecting the feasibility of the partial solution. Note that an advantage of the refinement approach is that the variables are not assigned and therefore multiple solutions can be found at the end. However, both approaches may stop without finding all the values of the variables, if this is the case, a search algorithm is executed. The search algorithm can be a tree based approach, such as B&B, that will compute the rest of the partial solutions to get finally feasible ones. However, the branching here might be slightly different, by either assigning a feasible value to a variable or by adding a new constraint on that variable.

The interest in CP models has been growing lately, because CP has succeeded in proving efficiency in solving famous planning and scheduling optimization problems. The book by Apt (2003) discusses in details the principles of constraint programming.

3.3.1.3 Dynamic programming

The fundamental idea of dynamic programming relies on decomposing recursively the problem. Basically, the general optimization problem is broken down to easier sub-problems in a recursive fashion. Then, by solving apart each sub-problem and finding their optimal solutions, the optimal solution to the main problem can be constructed from those solutions. To decompose the problem, it should be possible to define N stages and for each stage a finite number of states. The initial solution for the first stage is computed over its states. Next, a function should be defined to express the recursive relation between a stage k and earlier stages. Consequently, at stage $k + 1$, the function can be used to compute the solution up to stage $(k + 1)$ based on the solution of the previous stage k. The final solution is obtained when reaching the final stage N. Dynamic programming has been used to solve problems such as, knapsack (Martello et al., 1999), planning (Sutton, 1990), routing (Novoa and Storer, 2009), and the results obtained were interesting. However, if the problem cannot be decomposed recursively into relatively easy sub-problems, dynamic programming cannot be applicable.

3.3.2 Heuristic methods

Unlike exact methods, the optimality requirement is dropped in heuristic methods. These methods are focused on solving $\mathcal{NP}$-hard problems, where exact methods fail, in a very fast manner and by computing good quality solutions. In general, heuristics do not provide theoretical guarantee on the quality of solutions, but experimentally it is possible to evaluate the quality of the solutions. Since, most of the practical optimization problems are $\mathcal{NP}$-hard, heuristics are well established in OR field. Heuristic methods are numerous, therefore a classification of the heuristics based on their solution mechanisms is given in Fig. 3.3. The classification is inspired from the one proposed in (Gotha, 1993). In the rest of this section, the most important heuristics, under each class, are reviewed providing a general overview about the most common techniques involved in heuristic methods. Of course, the list is not exclusive and other heuristics can be found in the literature.

3.3.2.1 Progressive construction-based heuristics

The main objective of progressive construction-based heuristics is to compute a feasible solution to the optimization problem in polynomial time. They construct a solution by making random or predefined decisions, in a sequential or parallel scheme, until reaching a feasible solution. Generally, the decisions are very intuitive and simple, so the method converges fast towards a solution. A first example of such heuristics is the heuristics based on brute-force strategies, where the solution construction is based on priority rules. Iteratively, priority rules are consulted to make the next decision, until building a feasible solution. Another example is the heuristics based on search trees, like BeamSearch GM heuristic explained in chapter 2, section 2.3.8, which constructs a search tree and selects α number of nodes to explore, until reaching a leaf (feasible solution). The fact that multiple (α) nodes are selected in BeamSearch, makes it a parallel progressive construction-based heuristic. A* heuristic also fits into this class of heuristics. An important advantage for this heuristics is the speed up, however the quality of the solutions is questionable, but expected not to be very good because they are too simple and naive.

3.3.2.2 Neighborhood-based heuristics

The main idea of neighborhood-based heuristics is to improve a given solution by exploring its neighborhood. From an initial solution, they proceed by exploring a neighborhood defined around the initial solution. A neighborhood referred to all the solutions that are attainable from the current solution by performing simple *moves*. The best solution found in the neighborhood is retained and forms the basis of the next neighborhood. These methods tend to improve the solutions iteratively by checking neighbor solutions. Many simple and complex local search heuristics have been proposed to solve optimization problems.

Local search. Local search (LS) is a heuristic that appeared in the late 1950s in algorithms to solve the TSP problem (Bock, 1958; Croes, 1958). The idea is that starting from an initial solution S^0, LS, iteratively, performs a *move* step, which consists in selecting a close solution (neighbor) S^1. The objective function value of S^1 has to be better than S^0 in order to be selected for the next iteration. The move step is generally not complicated, e.g. if S^0 is composed of binary variables, it may be just changing the value of one variable from 1 to 0 or the way around. Moves can be performed as long as new improved neighboring solutions are being found. When no new better solution is found, then a local optimum is reached and the algorithm stops. LS suffers from two problems: it gets stuck in local optima and it is sensitive to the quality of the initial solution. However, LS has one important advantage that is the speed, because it does not involve sophisticated mechanisms. Some improvements were suggested later, mainly concerning the definition of neighboring solutions (deterministic/stochastic, small/large neighborhoods) and the selection strategy of a neighbor solution. To overcome the problem of getting stuck in local optima, some strategies were proposed, such as:

- Initiating the search from multiple initial solutions. This strategy is implemented in iterated local search, greedy randomized adaptive search procedure (GRASP), and many other heuristics.
- Selecting neighboring solutions with worse objective function value. This strategy is adopted from simulated annealing heuristic.

LS remains one of the simplest and most common heuristic and it is incorporated in many complex heuristics. The book by Aarts et al. (2003) gives more insights and details about local search-based heuristics.

Simulated annealing. This heuristic is inspired from the field of statistical physics, it is based on an observation of a transformation reaction of a metal (Kirkpatrick et al., 1983). It imitates the annealing process that requires fast heating a substance and then slowly cooling it to obtain a crystalline structure. The process is very sensitive, because insufficient high temperature or fast cooling may result in a deformed structure. To control the output, a condition is simulated over the temperature changes (high to low) that must be met to guarantee a good output. This is called equilibrium state, which is the point of convergence of the reaction. Back to optimization models, the objective function can be considered as the energy state of the system. The global optimum corresponds to the best stable state of the system, whereas local optima correspond to metastable states.

Simulated annealing underlies an iterative local search mechanism, but with the main objective to escape local optima. Every iteration, simulated annealing generates randomly a neighbor. If the neighbor is better than the current solution, the move is accepted and then it carries on with the next iteration. Otherwise, the neighbor may be selected, even if it is worse, depending on a probability computed over two factors: the current temperature and ΔE. ΔE is the amount of energy degradation that occurs to the objective function in the case of selecting the neighbor, i.e. it is the difference between the current and the new solutions. The probability will decrease over time: at the beginning the system tolerates degradation in the energy (objective function), until reaching a point where it becomes more strict. The probability is computed using this equation:

$$P(\Delta E, T) = e^{-\frac{f(S')-f(S)}{T}} \tag{3.9}$$

with T the temperature that controls the tolerance and strictness in accepting non-improving solutions, S (resp. S') the current (resp. new) solution, and $f(.)$ the value of the objective function. T will be decreased gradually over time and when an equilibrium state is reached based on a cooling schedule (time intervals to reduce T), which makes the system more strict. Simulated annealing stops when temperature parameter reaches a predefined value, or if there is a maximum number of iterations imposed. Finally, the best solution found throughout the process is returned.

Simulated annealing has achieved very interesting results in solving optimization problems. However, few problematics appear in this heuristic that require attention:

- as in other heuristic, the sensitivity to initial solutions,
- the tolerance probability of non-improving solutions,
- and the definition of the cooling schedule, i.e. how to decrease the temperature?

Yet, this heuristic is widely used, so taking a look to the literature, there are many good (to a certain point) ways to solve those problematics. For more details, please refer to the book of Talbi (2009).

Tabu search. Tabu search is a local search based heuristic, originally proposed by Glover (1989), that is considered an improvement to LS heuristic. It performs some moves to define a neighborhood around an initial solution. Then, it explores the neighborhood looking for better solutions. The process is repeated until reaching a local optimum. Tabu search deals with local optima, by selecting a worse solution than the current and carries on the local search around it. Of course, visiting a previous solution may occur, which is referred to as cycles. Tabu search enhances the local search by avoiding cycles, i.e. by memorizing the search trajectory. It stores information about the solutions visited in a short-term memory, called the tabu list. So, every time a new solution is found, the algorithm will verify the tabu list to make sure this solution is new before selecting it (it updates the list if the new solution is selected). Note that, the information stored in the list represent solution features or moves features. This causes discarding non-generated solutions, by relying on the quality of the features and not the whole solution, even though they may be attractive. This drawback can be corrected by introducing an aspiration criteria that (if satisfied) enables selecting tabu solutions, even if they were not generated.

In addition to problems raised in LS method, Tabu search may become slow, because the tabu list increases with time and at each iteration the list must be consulted. The definition of the aspiration criteria affects the performance of the heuristic; e.g. one good criteria is to select, under some conditions, a tabu move if it leads to a better solution. The interested readers are kindly referred to the book of Glover and Laguna (1998) for more details.

Iterated local search. In the spirit of improving LS heuristic, and in particularly dealing with the problem of bad initial solutions, Iterated Local Search (ILS) proposes a simple strategy to solve it (Lourenço et al., 2003). ILS goes like this:

1. A local search is performed around an initial solution, until getting a local optimum.
2. The solution is perturbed to generate a new solution (may be worse).
3. Local search is applied on the perturbed solution.
4. The best solution is accepted based on some defined criteria.

This process is repeated until reaching a stopping criteria. Two key points need to be chosen carefully, because they affect the performance of the algorithm, that are: the perturbation mechanism and the acceptance criteria. Talbi (2009) discusses different implementations of perturbation and acceptance criterion.

Variable neighborhood search. It is denoted by VNS, and was proposed by Mladenović and Hansen (1997). Unlike other heuristics, VNS tends to take a better care of the neighborhood structure. Therefore it defines N_k different neighborhoods, with $k = \{1, 2, ..., n\}$. It starts from an initial solution x, and the first step is called *shaking* that refers to selecting randomly a new solution x' from the first neighborhood N_1. The next step is to perform a local search around x', until reaching a local optimum x''. Then, it distinguishes between two cases:

1. If the new solution x'' is better than x, then x'' is chosen to be the current solution for the next iteration, and k is reset to 1.
2. Else, it goes to the shake phase with $k = k + 1$, i.e. the next shake step will compute a new x' in a different neighborhood.

The process is iterated until a stopping criterion is met. Of course, VNS in this implementation is stochastic, because of picking randomly x' solutions in the neighborhoods. There is deterministic version, which is called *Variable Neighborhood Descent.*

The intuition behind VNS is that a LS heuristic is known to generate local optimum in a neighborhood, and it is also known that the global optimum is also a local optimum in some neighborhood. So, VNS kicks off the solution space exploration in different neighborhoods that may be of different sizes and structures, hoping at getting the neighborhood where a global optimum resides. By the way, defining small neighborhoods in VNS results in a simple local search algorithm. However, defining very large neighborhoods imitates a multi-start local search.

3.3.2.3 Bio-inspired heuristics

These heuristics are based on natural or biological observations. Evolution theory and species mutation have inspired computer scientist researches into imitating some interesting natural processes to come up with heuristics and solve optimization problems. Another inspiration observation is swarm intelligence, which stands for the study of species behaviors such as ants, bees, etc. Researchers have tried to replicate the collective behavior of very small particles, like the communication between particles, movements and decisions. These are interesting ideas to develop heuristics that has turned out to be good in solving many optimization problems.

Genetic algorithms. Genetic algorithms, also denoted by GA, are population-based algorithms and belong to *Evolutionary Algorithms* (EA) school (Goldberg and Holland, 1988). They are inspired from the theories of creation of new species and their evolution. The main idea is to simulate the concept of evolution. It starts by selecting randomly a population of individuals, which are initial solutions to the optimization problem. The value of the objective function for a given individual is called fitness, and represents its relevance to the problem. GA algorithm carries on as follows:

1. Individuals are selected, to form the parents, from the population based on a selection strategy taking into account the fitness as factor.

2. Operators (e.g. crossover, mutation) are applied to selected individuals to reproduce new offsprings.

3. Based on a so called *replacement scheme*, some individuals will be picked to survive from the offsprings and parents.

4. The process is repeated until a stopping criteria is met.

So, GA generates solutions and employs crossover and mutation operations to mix them and generate new solutions. This requires a coding paradigm for the solutions. The new and old solutions are then evaluated, and only important ones are kept for the next iteration. The "important" criteria considers many factors like the value of objective function and a probabilistic selection, which makes GA a stochastic method. This is different from local search approaches, because the solutions might not be close in the solution space and the notion of neighbors is not used.

Genetic algorithms are suitable for solving complex problems such as multimodal, multiobjective and highly constrained problems. Kindly refer to the book of Man et al. (2012) for more details about genetic algorithms.

Ant colony optimization. This is a stochastic method that belongs to the family of swarm intelligence methods (Dorigo and Stützle, 2003). It is inspired from the behavior of real ants. Ants are known to be smart in performing hard tasks and, especially, finding the shortest paths. They count on the collectivity, so once an ant has discovered the shortest path, it leaves a trail (pheromone), so the other ants can follow. When other ants uses the same path, the trail becomes stronger indicating that the path is very important.

Replicating this behavior on optimization problem gives these iterative steps:

1. Constructing solutions based probabilistic state transition rule. In somehow, a final feasible solution is constructed from partial solutions, generated using stochastic greedy procedures (but they follow a probabilistic distribution).

2. Pheromone trail is modeled in the memory by storing the characteristics that have led to generating good solutions.

3. Evaporate pheromone trail, which is very important and automatically occurs to all values. They will disappear if they never gets reinforced, which also means that they are not interesting.

4. Reinforcement pheromone is related to important trails detected when generating the solutions. It increases the values of pheromones stored in the memory to keep them active.

However, implementing this heuristic may require including problem-dependent information to improve its performance, which was done for several optimization problems, e.g. scheduling, routing or assignment problems. For more details about ant colony optimization, please refer to the book of Dorigo et al. (2004).

Other heuristics exist in the literature, which are not detailed here. Basically, they share certain characteristics with the above ones. For instance, heuristics that are based on swarm intelligence are *Scatter search* and *Particle swarm optimization*. For an exhaustive list of these heuristics, and more details about the ones reviewed in this section, please refer to the book of (Talbi, 2009).

3.3.2.4 MILP-based heuristics

This kind of heuristics, called also matheuristics, aims at making use of the power of MILP solvers in solving MILP formulations. And at the same time including heuristic techniques (e.g. intensification, diversification, etc.) to guide the solvers into computing efficient sub-optimal solutions. By default, the solvers are intended to solve MILP formulations and find optimal solutions. Nowadays, these solvers are very good at doing so, but still not enough to cope up with real-life instances. The principle of matheuristics relies on exploring the neighborhood of a current solution by solving MILP formulations. Heuristics based on MILP formulations are gaining more reputation after reaching good results in solving optimization problems.

Local branching. Local branching is a heuristic that was proposed by Fischetti and Lodi (2003). Typically, a local branching heuristic is a local search algorithm, which improves an initial solution by exploring a series of defined neighborhoods via the solution of restricted MILP formulations. Local branching consists of three main ingredients:

1. Neighborhood definition: given a solution s to the problem, the neighborhood $N(s)$ is defined by adding a *local branching constraint* to the MILP model. The neighborhood size is controlled by a parameter π, that is the distance between the solutions in the neighborhood from s. For instance, the Hamming distance can be used in the neighborhood definition.

2. Intensification: once the neighborhood is defined, the restricted MILP formulation (with the local branching constraint) is given to the solver to find the best solution in that neighborhood. The new solution is accepted and the heuristic will define the neighborhood around this one and again intensify the search.

3. Diversification: what would happen if the intensification is not able to find a new solution? As in other heuristics, this means a local optimum is hit and to escape it, a diversification step must be done to change the explored region of the solution space. The diversification is also done by adding a branching constraint to the MILP formulation that tells the solver to look for solutions that are far from the current one by some defined parameter value.

There is more to local branching on how to employ the three ingredients together, how to skip previously visited solutions and how to guarantee good diversification. This heuristic has many parameters as well. Further details can be found in the paper of Fischetti and Lodi (2003).

Variable partitioning local search. Denoted shortly by VPLS, it is also a heuristic based on local search (Della Croce et al., 2013). As in local branching, the neighborhood definition is based on adding new constraints to the original MILP formulation. In VPLS, and given a solution s with its corresponding x vector for integer variables, the neighborhood is defined by splitting x into two sets x_1 and x_2 such that, variables in x_1 will be released and variables in x_2 will remain to their values in x. Therefore, the new MILP formulation has variables already fixed (a partial solution), and it will be solved by the solver, which will assign better values to variables in x_1, so compute a new solution $s' < s$. The process is iterated until a stopping criteria is met. The question to be asked is how to define the set of variables to be freed in the neighborhood definition? Well, interesting ideas comes up to answer this question, such as selecting the variables based on problem-dependent information. For example, determining a set of important variables (because they have high costs), which freeing them leads the solver to improve the current solution.

Other MILP-based matheuristics can be found in the literature that strongly rely on the continuous relaxation of the MILP formulations of the optimization problem, e.g. feasibility pump for general MILPs. Note that, for this heuristic to perform good, the continuous relaxation of the problem must provide tight bounds, otherwise it might be inefficient. For more details about matheuristics, please refer to the paper of Della Croce et al. (2013).

3.3.2.5 Problem-specific heuristics

From the name of this class, one can evinced that those heuristics are related to the problem at hand. They only solve a particular optimization problem and most of the time they are hard to be generalized. Problem-specific heuristics make use of the problem data, structure and particularity and they come up with techniques that help exploring the solution space in reasonable time and compute good solutions. They tend to be greedy in most of the cases, but they are still good at solving problems. Examples of problem-specific heuristics can be found in Section 2.3.8, where heuristics such as bipartite graph matching and square bipartite graph matching can be seen as problem-specific heuristics. Because they are designed to solve the GED problem in particular and cannot be applied to other problems in the way they are defined.

3.3.2.6 Heuristics and more

In the literature, there is a distinction between heuristics, which is based on the purpose and portability of the heuristic. Basically, there is:

- Metaheuristics: they are the general form of heuristic approaches that are by definition applicable to any optimization problem. The general paradigm of a metaheuristic consists of two main criteria: exploration of the solution space (known as diversification), exploitation of the best solutions (known as intensification). So, a metaheuristic, first, defines some interesting regions in the solution space, which then are explored by an intensification technique looking for good feasible solutions. Any heuristic following this paradigm, and does not include problem-dependent information, can be considered as a metaheuristic. For example and from the aforementioned heuristics, all local search-based and MILP-based heuristics are considered as metaheuristics.
- Matheuristics: they are alsoe metaheuristics, but they embed MILP solvers into heuristic algorithms, e.g. local branching and VPLS are considered as matheuristics.

Approximation heuristic. Some particular heuristic algorithms are called approximation algorithms. The major difference between them is that the latter have to provide theoretically a guarantee on the bound of the computed solution from the optimal one (Hochbaum, 1996). They are denoted by $\epsilon-$approximation algorithm, where ϵ is an approximation factor to represent the gap factor between the found solution and the optimal one. Of course, the smaller the ϵ, the better the solution quality. The property to be proven in an $\epsilon-$approximation algorithm is the following:

$$(s^* - \epsilon) \leq s \leq (s^* + \epsilon) \tag{3.10}$$

with s^* the optimal solution and s the solution computed by the algorithm. If this property is assured, the algorithm generates an *absolute performance guarantee*. In practical, the absolute guarantee is unlikely to be obtained, therefore a less rigid property can be proved, which is:

$$\frac{S}{S^*} < \epsilon \tag{3.11}$$

The algorithms, then, generates a *relative performance guarantee.* Please note, for the equation is given for a minimization problem. If it is a maximization problem, the terms in the division are reversed. Then, $\epsilon-$approximation provides tight worst-case bounds and they are problem dependent. There two well-known classes for approximation algorithms:

- *Polynomial-time approximation scheme* (PTAS): A minimization (resp. maximization) problem belongs to this class if it has a polynomial-time $(1+\epsilon)$-approximation (resp. $(1-\epsilon)$-approximation) algorithm for any fixed ϵ.

- *Fully polynomial-time approximation scheme* (FPTAS): A minimization (resp. maximization) problem belongs to this class if it has a polynomial-time $(1+\epsilon)$-approximation (resp. $(1-\epsilon)$-approximation) algorithm for any fixed ϵ, such that the running time is polynomial in the size of the instance and $1/\epsilon$.

For many complex optimization problems, it is not easy to design an $\epsilon-$approximation algorithm, which limits their applicability.

3.4 Summary and prospects for moving forward

This chapter has introduced briefly OR basics. The review is not totally fair because it is too short and does not cover all the aspects of OR field. Often, decision problems that come from real-life problems are quite hard, and in some cases taking a good decision is required to optimize the productivity and efficiency. OR offers a process to deal with such decision problems, by formulating and modeling them, i.e. translating them into quantitative mathematical models. These models can be solved using algorithms to make good decisions.

A very important and widely used modeling technique is mathematical programming, that deals with decision and optimization problems. Mathematical programming comes with different forms of modeling that cope with problems needs and requirements: linear programs, integer linear programs, mixed integer linear programs, optimization under uncertainty, etc. Each one has its own characteristics and level of flexibility in representing the problem, its objective and constraints. Another aspect to consider is the problem complexity, which tells whether a problem is tractable and thus can be solved in polynomial time or simply intractable and it will require exponential time to find the optimal solution.

Not only modeling and determining complexities of problems, but also OR intervenes in solving those models by offering a vast panel of methods. They are divided into two main families: exact and heuristic methods. The exact family contains methods that seek the optimal solution of a problem, e.g. branch-and-bound, dynamic programming. However solving problems to optimality has a major drawback, especially when dealing with $\mathcal{NP}$-hard problems, because of the exponential time required. While, heuristic methods are more suitable for this kind of problems. They aim at finding sub-optimal solutions in polynomial time. Heuristic methods are classified based on their solution mechanisms, resulting in:

1. Progressive construction-based heuristics: a solution is constructed incrementally based on random or defined steps.

2. Local search-based heuristics: they perform local searches in neighborhoods around solutions, hoping to find improving solutions.
3. Bio-inspired heuristics: these heuristics are inspired from natural observation and they imitate natural behaviors. Most of those heuristics are stochastic.
4. MILP-based heuristics: they embeds MILP solvers into solving MILP formulations by applying heuristic techniques (e.g. intensification, diversification)
5. Problem-specific heuristics: they are designed to solve particular problems, and most of the time they rely on specific information extracted from the problem.

The evaluation of a heuristic accuracy, in computing very good solutions, is done experimentally. While for approximation heuristics, it requires an additional theoretical proof on the quality, that guarantee worst-case bounds on the computed solution from the optimal one.

Going back to GM problems and in particular to the GED problem, which was discussed in Chapter 2. It is a $\mathcal{NP}$-hard optimization problem. The link between the GED problem and OR techniques has not yet been clearly realized. The main focus of this thesis is on solving the GED problem, and more precisely by employing OR techniques to model and solve it efficiently. The GED problem, which comes from PR field, and optimization techniques, which come from OR field, can be brought together bridging the two fields and making a good contribution to both fields. Based on GED methods reviewed in Chapter 2, there are certain methods (exact and heuristics) inspired from OR field, but yet not very common and they are very specific. To realize the objectives of this thesis, new and interesting techniques may be adopted and adapted from OR field to solve the GED problem. The chosen techniques must remain unique, novel and promising. The choice of the methods is based on two main factors that are invoked in the conclusion of Chapter 2:

- Solving the GED problem in the exact context: reviewing the state-of-the-art exact methods for the GED problem has revealed a lack in such methods. The argument is based on the fact that only two MILP formulations (JH and F2) have proven to be the most efficient methods in solving the problem to optimality. They perform better than other mathematical formulations (F1 and QAP) and even better than a B&B method (Lerouge et al., 2017). Yet, JH formulation tackles a sub-problem of GED and cannot be used to solve the general problem. This leaves only one formulation (F2) to solve the GED problem. An interesting contribution, yet not easy, may be developing new MILP models to solve the GED problem. Why MILP formulations? The answer is simply because MILP solvers capability in solving those formulations is increasing quickly. The solvers nowadays are able to solve bigger and harder instances. Hence, if designing a good MILP formulation and solving it today with MILP solvers, yield good results, then better results are expected in the next coming years without the need to modify the formulation because the solvers are going to be improved.
- Solving the GED problem heuristically: this may seem very tricky and risky, but at the same time interesting. Many good and fast heuristics already exist in the

literature (e.g. BP, SBPBeam, IPFP, etc.). In addition, some of these heuristics are based on metaheuristic approaches, such as beam-search based methods (BeamSearch and SBPBeam). However, one thing which has not yet been investigated is the application of MILP-based heuristics (matheuristics) to solve the GED problem. This kind of heuristic approaches is discussed earlier and has shown great capability in solving hard optimization problems. Since the first part is to design MILP formulations to solve the GED problem to optimality, it might be interesting to use these formulations in heuristic approaches. This is at the same time promising and has not yet been tested on the GED problem. So, questions to answer at the end are: how matheuristics methods will perform when solving the GED problem? Could they be efficient and live up to the performance of existing heuristics?

To sum up, based on the reviews of GED methods and the best OR methods in solving optimization problems, the lines that this thesis will follow to achieve its objectives, are based on:

1. The use of modeling techniques to design efficient MILP formulations to solve the GED problem.
2. The use of matheuristic methods (e.g. Local branching, VPLS) on the basis of existing or newly designed MILP formulations, to solve heuristically the GED problem.

The contributions of this thesis, if good results are achieved, may be very important because they will benefit two research communities: PR community in introducing new solution methods like matheuristics, which has not yet been done. As well, OR community in bringing attention of researchers to GM and GED problems that have a wide range of applications in machine learning and patter recognition fields.

Part II

Optimization techniques for solving the GED problem

Chapter 4

A matheuristic to solve the GED problem without attributes on the edges

Contents

4.1 Introduction to the problem

In Chapter 2, when presenting the GED problem, it was pointed out that JH MILP formulation, proposed by Justice and Hero (2006), solves the special case where the edges do not carry attributes. JH formulation is designed to operate over unitary costs for edges edit operations, which means it does not include the attributes associated with edges. This differs from other exact methods and MILP formulations, which are designed to solve the general GED problem. Due to this difference, and to conduct a study over exact methods, this chapter will deal with this special case and in Chapter 5 the GED problem is considered in its more general definition. To this end, a sub-problem of GED, denoted by GED^{EnA} (Edges no Attributes), is introduced in the next sub-sections.

4.1.1 Definition of the GED^{EnA} problem

The Graph Edit Distance - Edges no Attributes (GED^{EnA}) problem shares the same definition as the general GED problem (Definition 18, Section 2.3.1), with one difference, that is the edges of the graphs do not carry attributes. As well, even if the graphs do have attributes, those attributes will be ignored. Therefore, the function $\xi : E \rightarrow L_E$, which assigns attributes to edges, has $L_E = \{\phi\}$ in the GED^{EnA} problem. This implies that the costs of edges edit operations become:

- $c(e \rightarrow f) = 0, \ \forall e \in E, \forall f \in E'$,
- $c(e \rightarrow \epsilon) = \kappa, \ \forall e \in E$,
- $c(\epsilon \rightarrow f) = \kappa, \ \forall f \in E'$,

with $\kappa \in \mathbb{R}^+$. This is similar to the costs defined in the case of unattributed graphs in Section 2.3, but without modifying the vertices costs. Having the edges costs defined like this, is also called unitary costs. In terms of costs matrices, $[c_v]$ remains the same, however $[c_e]$ values change and become as shown in Eq. 4.1.

$$c_e = \begin{array}{c c} \begin{array}{ccccc} e_1 & e_2 & \dots & e_{|E|} & \epsilon \end{array} & \\ \begin{bmatrix} 0 & 0 & \dots & 0 & \kappa \\ 0 & 0 & \dots & 0 & \kappa \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 \\ \kappa & \kappa & \dots & \kappa & \kappa \end{bmatrix} & \begin{array}{c} f_1 \\ f_2 \\ \vdots \\ f_{|E'|} \\ \epsilon \end{array} \end{array} \tag{4.1}$$

In the literature, the GED^{EnA} problem is treated as a special case of the GED problem, but no clear definition is given. This differentiation is important because a method that solves the special case cannot be used to solve the general problem. As an example, JH formulation, presented in Section 2.3.7.2, is designed to solve instances of GED^{EnA}, but not GED. It is better and more precise to call JH formulation an exact method to solve the GED^{EnA} problem. The sub-problem can be considered as a lighter version, since edges substitutions cost nothing and only deletion/insertion edit operations have fixed costs. This somehow reduces the difficulty of determining edges operations, but does not make the problem any easier. A reason why JH formulation is the best exact method in the literature to solve the GED^{EnA} problem, is possibly related to the fact that other exact methods are designed to solve the GED problem, which is more general. With respect to the complexity theory, the GED^{EnA} is as well a $\mathcal{NP}$-hard problem. It can be demonstrated by following the same proof done for the GED problem, by reducing an induced subgraph isomorphism instance to a GED^{EnA} instance. This is possible and valid because the induced subgraph isomorphism is independent from the attributes of the graphs.

In the rest of this chapter, the focus is on the solution of the GED^{EnA} problem, while the next chapter is dedicated to the general GED problem.

4.1.2 Considerations on the GED^{EnA} problem

For two given graphs $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$, solving the GED^{EnA} problem consists in finding the complete edit path with the least cost to transform G into G'. The edit path is composed of multiple edit operations, which can be: substitution,

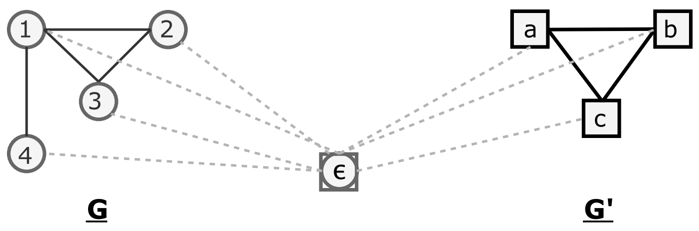


Figure 4.1: Two graphs G and G' with the ϵ node and dummy edges.

deletion and insertion. Deletion (resp. insertion) operations are represented by matching a vertex or edge from G (resp. G') to the ϵ (null) node. Therefore, applying the operations of a complete edit path to G will result in a graph isomorphic to G'

Proposition 1. *The insertion operations of vertices (resp. edges) into G can be replaced by deletion operations from G'.*

Proof. The aim is to obtain isomorphic graphs G and G', which is also possible to be obtained by modifying G' and not only G. Let $\lambda(G, G') = \{o_1, ..., o_k\}$ a complete edit path. The operations can be split into three sets as follows: R contains deletion operations, S is the set of substitution operations and I is the set of insertion operations ($\lambda(G, G') = \{R, S, I\}$). Applying first the operations in R to G results in a new graph $\hat{G}$. Next, applying operations in S to $\hat{G}$ does not affect the topology of the graph but only modifies the label functions, since the operations are only substitutions. Finally, applying operations in I to $\hat{G}$ gives G' (by definition). Trivially, $\hat{G}$ is a subgraph of G', because $\hat{V} \subset V'$ and $\hat{E} \subset E' \cap \hat{V} \times \hat{V}$. So, instead of inserting vertices and edges that are part of I, they can be deleted from G', which will result in the same graph $\hat{G}$. $\square$

In the rest of this thesis, the edit operations considered are reduced to only substitutions and deletions. The latter could be from G or G' and the same notations, as defined in Section 2.3, are used: operation $i \to \epsilon$ refers to deleting i from G, and $\epsilon \to k$ is deleting k from G'. The same for edges: deleting edge (i, j) from G is denoted by $(i, j) \to \epsilon$, and instead of inserting edge (k, l) into G, it is deleted from G' ($\epsilon \to (k, l)$).

In addition, for the sake of generality and clarity, ϵ node is considered and, actually, added to the sets of vertices. Therefore, the new sets of vertices are denoted by:

- $\overline{V} = V \cup \{\epsilon\}$
- $\overline{V}' = V' \cup \{\epsilon\}$

In order to represent the deletion operations of edges, a dummy edge is created linking every vertex to ϵ node. Hence, the edges sets are extended as follows:

- $\overline{E} = E \cup \{(i, \epsilon), \forall i \in V\}$

- $\overline{E}' = E' \cup \{(k, \epsilon), \forall k \in V'\}$

Finally, G and G' become $\overline{G} = (\overline{V}, \overline{E}, \mu, \xi)$ and $\overline{G}' = (\overline{V}', \overline{E}', \mu', \xi')$. Figure 4.1 shows an example of graphs $\overline{G}$ and $\overline{G'}$.

These considerations can be applied to the GED problem as well.

4.2 Graph databases

Here is a review of the most famous graph databases frequently used by researchers working with GM problems. The databases are collected from different sources and represent different objects and patterns. Each graph database has special settings (sets of attributes, density, etc.) and it may/may not have the cost functions of edit operations defined. The most common databases are used in the experiments conducted on the proposed methods in this thesis. Such review was helpful in carefully selecting the appropriate and more representative graph databases to run the evaluation experiments on the methods.

Mutagenicity (MUTA). This graph database is very common and contains 4337 graphs modeling chemical molecules. The graphs are undirected and attributed (both vertices and edges). To reduce the number of graph comparisons to be done, few sample graphs are selected and grouped into 8 subsets (Abu-Aisheh et al., 2015a). The first 7 subsets contain 10 graphs each of the same size (same number of vertices), starting from 10 until 70 vertices. The last subset has also 10 graphs but of mixed sizes. This database is interesting because it has large graphs (sizes 50; 60; 70), which they are known to be difficult for matching algorithms. The total number of instances is 800 (100 instances per subset). The cost functions are defined as in Eq. 4.2.

$$\begin{gathered}
Sub_v(\mu(i), \mu'(k)) = \begin{cases} 0 \; if \mu(i) = \mu(k) \\ 5500 \; otherwise \end{cases}, \\
Del_v(\mu(i)) = 5500, \\
Ins_v(\mu'(k)) = 5500, \\
\forall i \in V, \forall k \in V'. \\
Sub_e(\xi(e), \xi'(f)) = 0, \\
Del_e(\xi(e)) = 825, \\
Ins_e(\xi'(f)) = 825, \\
\forall e \in E, \forall f \in E'.
\end{gathered} \tag{4.2}$$

PAH. This database contains 94 graphs modeling chemical molecules. The graphs are undirected and unattributed. Each pair of graphs is considered as an instance, which gives a total of 8846 instances. The cost functions are given in Eq. 4.3, as published by

Abu-Aisheh et al. (2015a).

$$\begin{gathered}
Sub_v(\mu(i),\mu'(k)) = 0,\\
Del_v(\mu(i)) = 3,\\
Ins_v(\mu'(k)) = 3,\\
\forall i \in V, \forall k \in V'.\\
Sub_e(\xi(e),\xi'(f)) = 0,\\
Del_e(\xi(e)) = 3,\\
Ins_e(\xi'(f)) = 3,\\
\forall e \in E, \forall f \in E'.
\end{gathered} \tag{4.3}$$

CMU-HOUSE. This database contains 111 graphs corresponding to $3D$-images of houses, and each graph consists of 30 vertices with attributes described using *Shape Context* feature vector. Every edege has one attribute that is the distance between its incident vertices. So, the graphs are undirected and attributed. The particularity of this database is that graphs are extracted from $3D$-images of houses, where the houses are rotated with different angles. This is interesting because it enables testing and comparing graphs representing the same house but positioned differently inside the images. The database also comes with the ground-truth matchings for all pairs of graphs and was published by Moreno-García et al. (2016). Due to the importance of this database in GM field, three versions are considered, and each one with different cost functions.

1. HOUSE-NA: it is the version where attributes are ignored in the cost functions. The values are given in Eq. 4.4.

$$\begin{gathered}
Sub_v(\mu(i),\mu'(k)) = 0,\\
Del_v(\mu(i)) = \infty,\\
Ins_v(\mu'(k)) = \infty,\\
\forall i \in V, \forall k \in V'.\\
Sub_e(\xi(e),\xi'(f)) = 0,\\
Del_e(\xi(e)) = 0.5,\\
Ins_e(\xi'(f)) = 0.5,\\
\forall e \in E, \forall f \in E'.
\end{gathered} \tag{4.4}$$

2. HOUSE-A: the version where attributes are included in the cost functions by computing the differences between the values using a L_1-norm distance. Eq. 4.5 defines the cost functions.

$$\begin{gathered}
Sub_v(\mu(i),\mu'(k)) = ||\mu(i) - \mu'(k)||_1,\\
Del_v(\mu(i)) = \infty,\\
Ins_v(\mu'(k)) = \infty,\\
\forall i \in V, \forall k \in V'.\\
Sub_e(\xi(e),\xi'(f)) = 0,\\
Del_e(\xi(e)) = 0.5,\\
Ins_e(\xi'(f)) = 0.5,\\
\forall e \in E, \forall f \in E'.
\end{gathered} \tag{4.5}$$

3. HOUSE-REF: this version defines the cost functions as in (Zhou and De la Torre, 2012). For vertices, the substitution has a null cost, while the deletion costs some very high value (∞). This leads to favoring substitutions over deletions. However, for edges, all costs are based on the attributes as shown in Eq. 4.6.

$$\begin{gathered}
Sub_v(\mu(i), \mu'(k)) = 0, \\
Del_v(\mu(i)) = \infty, \\
Ins_v(\mu'(k)) = \infty, \\
\forall i \in V, \forall k \in V'. \\
Sub_e(\xi(e), \xi'(f)) = 1 - exp(\frac{-(\xi(e)-\xi'(f))^2}{2500}), \\
Del_e(\xi(e)) = 1 - exp(\frac{-(\xi(e))^2}{2500}), \\
Ins_e(\xi'(f)) = 1 - exp(\frac{-(\xi'(f))^2}{2500}), \\
\forall e \in E, \forall f \in E'.
\end{gathered} \tag{4.6}$$

CMU-HOTEL. This graph database is similar to CMU-HOUSE, but the graphs are modeling hotels inside $3D$-images. It contains 101 graphs and the hotels are also rotated inside the images with different angles. The same cost functions as in CMU-HOUSE (Eq. 4.4, 4.4, 4.6) can be used for this database.

PALMPRINT. It is a database of 160 graphs modeling the palm of human hands, with high quality images (Han et al., 2007). The graphs are considerably big, with more than 800 vertices for some graphs, and around 2000 edges. They are undirected and attributed. The attributes assigned to a vertex i are: $[x_i, y_i, angle_i, type_i, \ell_i^1, \ell_i^2, \ell_i^3, \ell_i^4]$, storing features like the coordinates of i in the image, the angle, type, etc.

$$\begin{gathered}
Sub_v(\mu(i), \mu'(k)) = \begin{cases} 0.85 \times min(angle_i - \ell_k^3, 360 - angle_i - \ell_k^3) + \\ 0.15 \times (||\ell_k^1 - x_i||_2 + ||\ell_k^2 - y_i||_2), \\ \quad if\ type_i = type_k \\ \infty,\ otherwise \end{cases}, \\
Del_v(\mu(i)) = 210, \\
Ins_v(\mu'(k)) = 210, \\
\forall i \in V, \forall k \in V'. \\
Sub_e(\xi(e), \xi'(f)) = 0, \\
Del_e(\xi(e)) = 5, \\
Ins_e(\xi'(f)) = 5, \\
\forall e \in E, \forall f \in E'.
\end{gathered} \tag{4.7}$$

VOC-CAR. Graphs are representing cars inside images and they are undirected and attributed (Zhou and De la Torre, 2012). The attributes over edges consist of two values: $[d_e, \theta_f]$, respectively, the distance and the angle between the two vertices. The cost functions are defined in Eq. 4.8.

$$
\begin{aligned}
Sub_v(\mu(i),\mu'(k)) &= 1 - exp(-|\mu(i)-\mu'(k)|),\\
Del_v(\mu(i)) &= \infty,\\
Ins_v(\mu'(k)) &= \infty,\\
&\forall i \in V, \forall k \in V'.\\
Sub_e(\xi(e),\xi'(f)) &= 1 - exp(-\tfrac{1}{2}|d_e - d_f| - \tfrac{1}{2}|\theta_e - \theta_f|),\\
Del_e(\xi(e)) &= \infty,\\
Ins_e(\xi'(f)) &= \infty,\\
&\forall e \in E, \forall f \in E'.
\end{aligned}
\tag{4.8}
$$

VOC-BIKE. There are 440 undirected and attributed graphs modeling bikes inside images. The cost functions are the same as those given in Eq. 4.8.

PROTEIN. The database contains 600 undirected and attributed graphs of proteins (Riesen and Bunke, 2008). Three attributes are assigned to each vertex: type, sequence (seq) and the length of the sequence. The edges attributes are: frequency, one or more type and distance depending on the edge, e.g. $[freq : 1, type0 : 1, distance0 : 17.1, type1 : 1, distance1 : 4.2, ...]$. Therefore, the cost functions are given in Eq. 4.9.

$$
\begin{aligned}
Sub_v(\mu(i),\mu'(k)) &= \begin{cases} string\_distance(seq_i, seq_k), \ if \ type_i = type_k \\ 8.25, \ otherwise \end{cases},\\
Del_v(\mu(i)) &= 8.25,\\
Ins_v(\mu'(k)) &= 8.25,\\
&\forall i \in V, \forall k \in V'.\\
Sub_e(\xi(e),\xi'(f)) &= Hungarian([type, distance]_e, [type, distnace]_f),\\
Del_e(\xi(e)) &= 0.25 * freq_e,\\
Ins_e(\xi'(f)) &= 0.25 * freq_f,\\
&\forall e \in E, \forall f \in E'.
\end{aligned}
\tag{4.9}
$$

SYNTETHIC. This is a manually generated graph database. A library available in Python and published by Csardi and Nepusz (2006), enables generating various types of graphs (directed or undirected, attributed or not). The library is, then, used to generate undirected and attributed graphs. One arbitrary attribute based on a uniform distribution, of decimal type from the range between 0 and 1, is assigned to each vertex and edge in the graph. A nice feature of this library is the possibility of controlling the density of the generate graphs. The density is the percentage of actual number of edges in the graph, out of the maximum number of edges that could be added (the case of a complete graph). As an example, generating an undirected graph of size 30, the maximum number of edges is $\frac{30\times(30-1)}{2} = 435$. Then, a graph with a density of 10% will have approximately 44 edges. So, based on this, multiple subsets are created with different densities. And, two versions of synthetic databases are generated.

Table 4.1: Summary of graph databases suitable for the GED^{EnA} problem and/or the GED problem

	Nb.	Size	Edges has attri	Edge cost fct	GED^{EnA}
MUTA	4437	Small - Medium	yes	constant	yes
PAH	94	Small	no	constant	yes
Protein	600	Medium	yes	not constant	no
COIL-DEL	7200	Medium	yes	constant	yes
CMU-HOUSE	111	Medium	yes	constant	yes
CMU-HOTEL	101	Medium	yes	constant	yes
VOC-car	660	Medium	yes	not constant	yes
VOC-bike	440	Medium	yes	not constant	yes
PalmPrint	160	Large	yes	constant	yes
Web	2344	Large	yes	not constant	no
California-road-map	31	Large	yes	not constant	no
MNIST	711	Large	yes	not constant	no
GREC	1100	Small	yes	not constant	no
Letter	2250	Small	no	constant	yes
SYNTHETIC	2000	Medium - Large	yes	not constant	no

1. SYNTHETIC-30: this database contains 10 subsets. Each one has 10 undirected and attributed graphs, with different density values, starting from 10%, 20% till 100% (fully connected graph). The total number of instances is 1000 (100 per subset).

2. SYNTHETIC-100: the same type and number of instances as in the first database, but with graphs of size 100.

The goal of generating these graphs is to have medium and large instances and to be able to test GED methods by considering different sizes and different densities. It might help determining whether the difficulty of an instance is due to its size or its density or both. The cost values for all edit operations are given in Eq. 4.10.

$$\begin{gathered}
Sub_v(\mu(i), \mu'(k)) = ||\mu(i) - \mu'(k)||_2, \\
Del_v(\mu(i)) = ||\mu(i) - 0||_2, \\
Ins_v(\mu'(k)) = ||0 - \mu'(k)||_2, \\
\forall i \in V, \forall k \in V'. \\
Sub_e(\xi(e), \xi'(f)) = ||\xi(e) - \xi'(f)||_2, \\
Del_e(\xi(e)) = ||\xi(e) - 0||_2, \\
Ins_e(\xi'(f)) = ||\xi'(f) - 0||_2, \\
\forall e \in E, \forall f \in E'.
\end{gathered} \tag{4.10}$$

Many other graph databases can be found in the literature, and the above list enumerates the most interesting and representative ones of medium and large sizes, taken from different domains such as chemistry, biology and pattern recognition. It gives an overview on the databases and makes the selection for the experiments of GED methods

Table 4.2: Density (% of connectivity) of graph databases

Database	**Density %**
MUTA	9.13
PAH	12.22
CMU	18.00
VOC	13.47
PALMPRINT	0.77
PROTEIN	16.00

easier. However, some databases are only valid for the GED^{EnA} problem, because their edges cost functions do not include the attributes. To make it more clear and separate those databases, Table 4.1 shows a summary of databases, covering the listed ones, plus other databases existing in the literature. It indicates whether a database is suitable for GED^{EnA} and/or for the general GED problem. Note that, a valid graph database for the GED^{EnA} problem is also valid for the GED problem, but not the opposite. Graph databases that have no attributes on edges, have certainly constant costs for edges operations, e.g. PAH and Letter databases. For instance, graph databases such as Protein and Web have functions based on edges attributes to compute the costs, which makes them irrelevant for the GED^{EnA} problem. The notation small, medium and large stands, respectively, for graph sizes: less than 30, between 30 and 70, greater than 70. Databases GREC, Letter, Web, COIL-DEL listed in Table 4.1, are published by Riesen and Bunke (2008). The California-road-map database was found in the work by Leskovec et al. (2009). MNIST database was used in the experiments in the work by Cecotti (2016). Finally, Table 4.2 shows the densities of the databases that were detailed earlier. In general and based on the experiments, databases with densities less than 10% are considered as non-dense graphs. Others, such as CMU, PROTEIN and SYNTHETIC databases are considered as dense graphs.

4.3 Comparison of existing MILP formulations

As a first step in this thesis and after reviewing the available MILP formulations for the GED^{EnA} problem, a comparison was conducted in order to study their performances. The goal is to determine the best MILP formulation existing in the literature. This can help exploiting the formulations and check their limits, in terms of scaling up to big instances and form a basis of an interesting contribution. Also, it enables extracting useful properties about the efficiency of the best MILP, which can help in the choices that have to be made later in this thesis. The three MILP formulations are: JH, F1 and F2, and they are explained in details in Section 2.3.7. In Table 4.3, a comparison in terms of number of variables and constraints for each formulation, is given. The following are the remarks extracted based on this comparison:

- JH has a number of variables and constraints independent from the number of edges in the graphs.

- In the case of dense graphs, F1 has a lot more variables and constraints than F2 and JH.
- In the case of non-dense graphs, F2 has less number of variables and constraints than F1 and possibly JH.

Table 4.3: MILP formulations comparison (nb. of variables and constraints)

	Variables	**Constraints**
JH	$3 \cdot (\lvert V\rvert + \lvert V'\rvert) \cdot (\lvert V\rvert + \lvert V'\rvert)$	$(\lvert V\rvert + \lvert V'\rvert) \cdot (\lvert V\rvert + \lvert V'\rvert) + 2 \cdot (\lvert V\rvert + \lvert V'\rvert)$
F1	$\lvert V\rvert + \lvert V'\rvert + \lvert E\rvert + \lvert E'\rvert + \lvert V'\rvert \cdot \lvert V'\rvert + \lvert E'\rvert \cdot \lvert E'\rvert$	$\lvert V\rvert + \lvert V'\rvert + \lvert E\rvert + \lvert E'\rvert + 2 \cdot \lvert E\rvert \cdot \lvert E'\rvert$
F2	$\lvert V\rvert \cdot \lvert V'\rvert + \lvert E\rvert \cdot \lvert E'\rvert$	$\lvert V\rvert + \lvert V\rvert + \lvert V'\rvert \cdot \lvert E\rvert$

It is hard to tell by just looking at the size of the formulations, which one is more efficient. Therefore, they are compared on real graph instances. Two experiments are executed for this purpose: standalone MILP formulations, and MILP formulations with pre-processing. The details of each experiment and the obtained results are given in the next sections.

Instances and experimentation settings. MUTA graph database is selected to test the formulations, because it is compatible with the GED^{EnA} problem. All the details (subsets, cost functions) of MUTA are discussed in Section 4.2. In this experiment, 7 subsets (10 to 70) are involved, which gives a total of 700 instances. JH, F1 and F2 are implemented in C language. The solver CPLEX 12.6.0, in single thread mode, is used to solve the formulations. A maximum running time limit, of 900 seconds per instance, is imposed on CPLEX. If the optimal solution has not been found/proven before the 900 seconds, then CPLEX halts and returns the best solution found. Experiments are ran on a machine with Windows 7 $x64$, Intel Xeon $E5$ 2.30 GHz, 4 cores and 8 GB of RAM.

Evaluation indicators. The graph instances are solved for each formulation with the settings explained above and the following indicators are recorded:

- t_{min}: the minimum CPU time in seconds over all instances in a subset,
- t_{avg}: the average CPU time in seconds over all instances in a subset,
- t_{max}: the maximum CPU time in seconds over all instances in a subset,
- d_{min}: the minimum deviation, in percentage, between the solutions obtained by a formulation against the best solutions found by the three formulations,
- d_{avg}: the average deviation, in percentage, between the solutions obtained by a formulation against the best solutions found by the three formulations,
- d_{max}: the maximum deviation, in percentage, between the solutions obtained by a formulation against the best solutions found by the three formulations,
- η: the number of optimal solutions computed,

- η': the number of solutions (whether optimal or not) computed by a formulation, which are the best/minimum among those computed by all formulations.

For an instance I and a formulation F, the deviation is computed by Eq. 4.11.

$$deviation_I^F = \begin{cases} 0, \text{ if } bestSol_I = 0 \text{ and } solution_I^F = 0 \\ 100, \text{ if } bestSol_I = 0 \text{ and } solution_I^F > 0 \\ \frac{solution_I^F \times bestSol_I}{bestSol_I} \times 100, otherwise \end{cases}, \tag{4.11}$$

$$with\ bestSol_I = min\{solution_I^{JH}, solution_I^{F1}, solution_I^{F2}\}.$$

4.3.1 Standalone MILP formulations comparison

The results obtained by each formulation and for each subset are given in Table 4.4. Starting by η values, it seems that for subsets 10 and 20 (small graph instances), all formulations were able to compute the optimal solutions for all 100 instances in each subset. For subsets 30, 40 and 50, JH has the highest η values (93, 87, 67) comparing to F1 and F2. However, the numbers drop for subsets 60 and 70 to 25 and 18, which contain large instances, yet still better than η values of F1 and F2. JH also scores the highest η' values for all subsets. This, in turn, explains why the average deviation (d_{avg}) of JH is the lowest for all subsets. It is also the fastest in terms of CPU time (t_{avg}) with the smallest values for all subsets, except for subset 10 where F2 is a bit faster with $0.12s$ against $0.17s$. Next, and as expected, F2 performs better than F1 in all terms (number of optimal solutions, deviations and CPU time). F2 is designed over F1 by reducing the number of variables and constraints, which has worked in its favor. Moreover, and to be more precise about the running time of the formulations, Table 4.5 shows the running time obtained only for instances where optimal solutions were found by all formulations. What changes from the previous results, is that on subsets $50, 60$ and 70, F1 has scored the lowest average CPU time. The optimal solutions obtained for these subsets, are only for the 10 instances where $G = G'$. To conclude the analysis of the results shown in both tables, JH is the fastest in solving MUTA instances, except for subset 10. In addition, JH solves a lot more instances to optimality than F1 and F2.

Conclusion. The conclusion of this experiment is that JH is the best formulation to solve to optimality the GED^{EnA} problem.

4.3.2 Comparison of MILP formulations with pre-processing

In this experiment a pre-processing step is involved, which is a procedure that can help fixing some binary variables to optimality, tightening bounds on variables, or generating new constraints to reduce the size of the instance. The variable fixing technique considered is a pre-processing step that performs analysis on the LP solution and fixes binary variables to optimality in the MILP formulation. This is intended to help the solver by reducing the number of binary variables to compute. This pre-processing technique has achieved good results when applied to $\mathcal{NP}$-hard optimization problems like flowshop scheduling problems

Table 4.4: Comparison of MILP formulations

	S	10	20	30	40	50	60	70
JH	t_{min}	0.06	0.14	0.28	0.49	0.77	1.18	1.70
	t_{avg}	0.13	**1.02**	**141.07**	**247.80**	**451.40**	**723.68**	**745.91**
	t_{max}	0.49	3.52	900.20	900.42	900.46	900.71	900.92
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	**0.00**	**0.00**	**0.06**	**0.04**	**0.00**
	d_{max}	0.00	0.00	0.00	0.00	3.11	2.21	0.00
	η	**100**	**100**	**93**	**87**	**67**	**25**	**18**
	η'	**100**	**100**	**100**	**100**	**98**	**98**	**100**
F1	t_{min}	0.01	0.02	0.03	0.05	0.06	0.08	0.11
	t_{avg}	0.18	26.73	741.12	786.99	810.08	810.11	810.14
	t_{max}	0.90	486.18	900.23	900.17	900.12	900.35	901.07
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	2.99	6.40	14.93	19.26	23.63
	d_{max}	0.00	0.00	21.43	29.58	47.85	49.15	91.07
	η	**100**	**100**	22	14	10	10	10
	η'	**100**	**100**	51	25	13	10	10
F2	t_{min}	0.03	0.07	0.10	0.15	0.36	0.44	0.76
	t_{avg}	**0.12**	1.17	367.54	631.04	792.96	803.84	809.64
	t_{max}	0.38	7.79	900.20	900.19	900.48	900.38	900.17
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	0.17	1.28	4.14	5.95	11.91
	d_{max}	0.00	0.00	5.69	11.69	18.58	24.00	29.72
	η	**100**	**100**	77	36	14	11	10
	η'	**100**	**100**	95	67	34	19	13

Table 4.5: Comparison of MILP formulations - optimal solutions

	S	10	20	30	40	50	60	70
JH	t_{min}	0.06	0.14	0.28	0.49	0.77	1.18	1.70
	t_{avg}	0.13	**1.02**	**2.04**	**18.83**	0.79	1.23	1.76
	t_{max}	0.49	3.52	13.89	206.20	0.82	1.29	1.85
	η	100	100	22	13	10	10	10
F1	t_{min}	0.01	0.02	0.03	0.05	0.06	0.08	0.11
	t_{avg}	0.18	26.73	180.28	71.14	**0.07**	**0.09**	**0.11**
	t_{max}	0.90	486.18	873.12	400.59	0.08	0.10	0.12
	η	100	100	22	13	10	10	10
F2	t_{min}	0.03	0.07	0.10	0.15	0.36	0.44	0.76
	t_{avg}	**0.12**	1.17	71.79	21.38	0.49	1.20	1.35
	t_{max}	0.38	7.79	658.01	244.74	0.81	2.72	1.99
	η	100	100	22	13	10	10	10

(T'kindt et al., 2007) and (Baptiste et al., 2010), map sectorization problem (Tang et al., 2014), and multidimensional knapsack problem (Osorio et al., 2002). Variable fixing uses information such as reduced costs to handle non-basic variables and penalties to handle basic variables. The notion of basic/non-basic variables and costs are explained in the next section.

Algorithm 4: Pre-processing for variable fixing procedure.

```
/* Fix non-basic variables                                        */
$\forall x_i \in \overline{B} \cap \{x_i : i \in I\}$
if $x_i = 0$ then
    if $Z^{LP} + r_i > UB$ then
        $x_i = 0$ in the MILP formulation
    end
end
if $x_i = 1$ then
    if $Z^{LP} - r_i > UB$ then
        $x_i = 1$ in the MILP formulation
    end
end
/* Fix basic variables                                            */
$\forall x_j \in B \cap \{x_i : i \in I\}$
if $x_j \cdot l_j + Z^{LP} > UB$ then
    $x_j = 1$ in the MILP formulation
end
if $u_j \cdot (1 - x_j) + Z^{LP} > UB$ then
    $x_j = 0$ in the MILP formulation
end
```

4.3.2.1 Prep-processing procedure

A mixed integer linear program with n variables and m inequality constraints can be rewritten as follows:

$$\begin{array}{c} \min c^T x \\ Ax \leq b \iff \sum_{i=1}^{n} a_{ij} \cdot x_i + s_j = b_j,\ \forall j \in \{1, ..., m\} \\ s_j \geq 0 \\ x_i \in \{0, 1\},\ \forall i \in I \\ x_i \in \mathbb{R},\ \forall i \in C \\ s_j \in \mathbb{R},\ \forall j \in \{1, ..., m\} \end{array} \tag{4.12}$$

where $c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ are vectors of coefficients, $A \in \mathbb{R}^{m \times n}$ is a matrix of coefficients, and x is a vector of variables to be computed. The variable index set is split into two sets (I, C), which respectively stand for integer (boolean) and continuous. Variables s are known as slack variables and are added to transform inequality constraints to equality constraints. By relaxing the problem and modifying the integer variables whose indexes are in I to become continuous, the new relaxed problem (LP) can, then, be solved efficiently by the *Simplex* method in polynomial time. The objective function's value of the LP optimal solution is called Z^{LP}. And, it is considered as a *lower bound* (LB) to the optimal solutions of the initial MILP formulation. Moreover, by looking at the optimal solution of LP, the following two sets can be defined:

- $B = \{x_i \neq 0, s_j \neq 0\}$, the set of basic variables that may have fractional values in the solution of the relaxed problem. For all $x_i \in B \cap \{x_i : i \in I\}$, $x_i \in [0, 1]$.
- $\overline{B} = (\{x_i : i \in I\} \cup \{x_i : i \in C\} \cup \{s_j : j \in \{1, ..., m\}\}) \setminus B$, is the set of non-basic variables. For all $x_i \in \overline{B} \cap \{x_i : i \in I\}$, $x_i \in \{0, 1\}$ in the solution of the relaxed problem.

Other useful values can be obtained from the *Simplex* method's results: $[r_i] \cup [r_j]$, with $i \in I \cup C$ and $j \in \{1, ..., m\}$, are the reduced costs of variables x and s. Reduced costs are under-estimation of the cost that will be added/reduced from the value of the objective function, in the case of changing the value of a non-basic variable from 0 to 1 (or the opposite). Mathematical programming theory provides the following basic statements:

- $\forall x_i \in \overline{B}$, if $x_i = 0 \Rightarrow r_i > 0$
- $\forall s_j \in \overline{B}$, if $s_j = 0 \Rightarrow r_j > 0$
- $\forall x_i \in \overline{B}$, if $x_i = 1 \Rightarrow r_i < 0$
- $\forall s_j \in \overline{B}$, if $s_j = 1 \Rightarrow r_j < 0$

The pre-processing procedure uses the reduced costs and the value Z^{LP} to try figuring out the values of the binary variables in an optimal solution of the original MILP formulation. Algorithm 4 shows the core of the pre-processing procedure for fixing non-basic variables (lines 1-12) and basic variables (lines 13-20). It, simply, selects the non-basic binary variables and two cases are tested. If the variable is equal to 0, it checks if its reduced cost plus Z^{LP} exceeds UB (a computed upper bound): if so, that variable must remain 0 in an optimal solution of the MILP formulation. The second case is the opposite, and the variable is equal to 1 with a negative reduced cost: so if subtracting the reduced cost from Z^{LP} exceeds UB, this means the variable must remain 1. In the next phase, pre-processing attempts to fix basic variables. To do so, lower (l_j) and upper (u_j) bounds are computed for each variable. They are also called penalties and they were introduced by Driebeek (1966). Penalty l_j (resp. u_j) represents an estimation to the increase of Z^{LP} when $x_j = 0$ (resp. $x_j = 1$). So, the algorithm tests if l_j estimation increases Z^{LP} to exceed the UB, then x_j can be set to 1 in the MILP formulation. On the other hand, if the complement value $(1 - x_j)$ multiplied by u_j increases Z^{LP} to exceed the UB, then x_j is set to 0. Of course, an important factor is to have very tight (close) bounds (UB and Z^{LP}), so the use of reduced costs will lead to fixing a good amount of variables. If the gap is large, then pre-processing will not be able to fix many variables.

Table 4.6: Comparison of MILP formulations with pre-processing

	S	10	20	30	40	50	60	70
JH	t_{min}	0.08	0.27	1.13	2.30	4.32	7.43	12.14
	t_{avg}	0.15	1.21	**108.80**	**240.66**	**453.09**	**751.01**	**790.87**
	t_{max}	0.86	6.08	902.77	907.41	919.38	942.83	986.34
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	**0.00**	**0.00**	**0.03**	**0.02**	**0.00**
	d_{max}	0.00	0.00	0.00	0.00	2.86	1.81	0.00
	η	**100**	**100**	**96**	**84**	**67**	**23**	**18**
	η'	**100**	**100**	**100**	**100**	**99**	**90**	**100**
	% varFix	**62.54**	**32.62**	**25.19**	**22.80**	**20.87**	**16.92**	**14.78**
F1	t_{min}	0.03	0.09	0.43	0.76	3.96	7.19	17.97
	t_{avg}	0.15	17.86	735.95	790.22	816.69	825.17	847.59
	t_{max}	0.66	185.43	902.39	908.14	925.72	939.60	1054.34
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	2.80	5.71	14.93	19.05	23.66
	d_{max}	0.00	0.00	20.69	26.32	47.85	49.15	91.07
	η	**100**	**100**	22	14	10	10	10
	η'	**100**	**100**	54	26	13	10	10
	% varFix	43.08	19.33	14.33	11.18	10.13	10.00	10.00
F2	t_{min}	0.04	0.13	0.35	0.54	1.55	2.11	3.71
	t_{avg}	**0.12**	**1.16**	373.41	614.69	778.26	808.90	812.29
	t_{max}	0.35	5.90	901.39	902.97	906.91	910.18	925.94
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	0.29	1.14	3.87	5.32	11.90
	d_{max}	0.00	0.00	5.69	9.89	18.58	24.00	34.04
	η	**100**	**100**	75	39	18	11	11
	η'	**100**	**100**	91	71	40	20	14
	% varFix	48.77	24.94	17.30	13.77	11.49	10.43	10.39

4.3.2.2 Results with pre-processing

Before presenting the results, one evaluation indicator is added to each formulation, that is the average percentage of variables fixed ($\%varFix$) during the pre-processing phase, for all instances in a subset. It shows how good is pre-processing in fixing variables.

The results of the experiment are shown in Table 4.6. Analyzing based on the deviation and η' indicators leads to the same order as in the first experiment: JH is the best and it has computed the best solutions, followed by F2 and at last F1. Regarding, the percentage of fixed variables, the highest values are scored by JH. However, based on the recorded t_{avg}, and comparing with the values of the first experiment (Table 4.3), the variable fixing procedure did not have an impact on the running time. It is maybe because the percentage of fixed variables is rather low ($< 25\%$) on subsets between 30 and 70. As in the previous experiment, Table 4.7 reports the average times, and the average percentages of fixed variables on instances where all MILP formulations have found the optimal solutions. Based on the results, F2 has the best t_{avg} for all subsets. Except for subset 30 where JH is very fast comparing to F2 with $t_{avg} = 3.23s$ against $64s$. This means that applying pre-processing and fixing variables have helped F2 formulation in solving those instances faster than F1 and JH formulations.

Table 4.7: Comparison of MILP formulations with pre-processing - optimal solutions

	S	10	20	30	40	50	60	70
JH	t_{min}	0.08	0.27	0.71	1.39	2.55	4.30	6.92
	t_{avg}	0.15	1.21	**3.23**	41.57	2.59	4.49	7.19
	t_{max}	0.86	6.08	16.94	520.83	2.67	4.69	7.72
	η	100	100	22	13	10	10	10
	% varFix	62.54	32.62	54.27	89.17	100.00	100.00	100.00
F1	t_{min}	0.03	0.09	0.23	0.40	2.01	3.63	9.04
	t_{avg}	0.15	17.86	150.11	73.32	3.14	6.32	13.63
	t_{max}	0.66	185.43	872.84	385.91	4.59	9.39	22.10
	η	100	100	22	13	10	10	10
	% varFix	43.08	19.33	50.60	78.80	100.00	100.00	100.00
F2	t_{min}	0.04	0.13	0.22	0.34	0.96	1.27	2.23
	t_{avg}	**0.12**	**1.16**	64.01	**20.20**	**1.35**	**3.10**	**3.88**
	t_{max}	0.35	5.90	658.87	242.10	2.00	6.40	5.89
	η	100	100	22	13	10	10	10
	% varFix	48.77	24.94	52.88	80.77	100.00	100.00	100.00

Conclusion. The second experiment confirms that JH is the best MILP formulation in the literature in terms of computing the best and optimal solutions. However, F2 turns to be faster with pre-processing for instances where the optimal solutions have been found. Finally, the pre-processing algorithm was not very helpful in improving the performances of the formulations.

These results are published in the conference ROADEF2017:
Darwiche, M., Conte, D., Raveaux, R., & T'Kindt, V. (2017, February). Evaluation de modèles mathématiques pour le problème de la distance d'édition entre graphes. In *ROADEF2017*.

4.4 An adapted local branching heuristic to solve the GED^{EnA} problem

The efficiency of JH formulation was shown, in Section 4.3, in solving to optimality the GED^{EnA} problem, even though it is limited to small-size instances. In general, MILP formulations are known to be very powerful tools in modeling combinatorial optimization problems. Usually, such formulations are solved using black-box solvers such as CPLEX, Gurobi, etc. These solvers are equipped with an arsenal of effective algorithms to solve MILP formulations. However, they are not capable all the time of finding the optimal solutions. Especially in the case of large and complex instances, due to high computational time and memory size requirements. Lately, a new family of metaheuristics, namely matheuristics, has been introduced in Operation Research community. They involve both MILP formulations and solvers in a defined scheme with one goal, that is to explore the solution space efficiently and compute very good quality solutions. One well-known matheuristic, called *local branching*, was introduced by Fischetti and Lodi (2003). Its main idea is to perform a series of local searches based on a MILP formulation, and to focus the search on defined regions looking for good quality solutions. Starting from an initial solution,

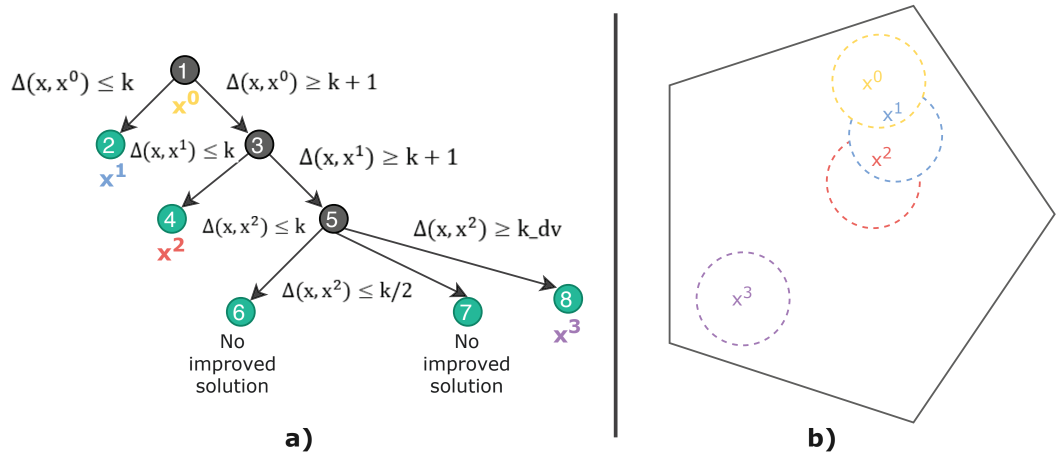


Figure 4.2: Local branching flow. a) depicts the left and right branching. b) shows the neighborhoods in the solution space

it defines the neighborhood around it and performs an intensification step looking form better solutions. Local branching knows well how to escape from local optima, by employing a diversification mechanism. So, this heuristic combines several heuristic techniques (neighborhood definition, intensification and diversification) in a defined branching scheme. Local branching was used to solve many optimization problems and has obtained very good results, such as the capacitated ring tree problem (Hill and Voß, 2018), the capacitated fixed-charge network design problem (Rodríguez-Martín and Salazar-González, 2010), and the open pit mine production scheduling problem (Samavati et al., 2017). A first attempt to design a good heuristic for the GED^{EnA} problem is then, to use JH formulation with CPLEX solver and to embed them in a local branching procedure. Such a heuristic has not yet been tested on the GED problem in the literature.

4.4.1 Main features of the Local Branching heuristic

This section covers the functionalities and the main features of the local branching heuristic specifically implemented to solve the GED^{EnA} problem. This heuristic version follows the original version introduced by Fischetti and Lodi (2003), with improvements outlined when appropriate.

This method is a local search heuristic, which embeds the truncated solution of JH formulation into a search tree. It is based on 4 main ingredients:

- Neighborhood definition: giving a solution $x^p \in \{0,1\}^{N\times N}$ to the problem, with $N = |V|+|V'|$, $\mathcal{N}(x^p, \pi)$ is the π-*opt neighborhood* around x^p, with π a given positive integer. $\mathcal{N}(x^p, \pi)$ is defined by adding the following *local branching constraint* to JH:

$$\Delta(x, x^p) = \sum_{(i,k)\in S^p} (1 - x_{i,k}) + \sum_{(i,k)\notin S^p} x_{i,k} \leq \pi \qquad (4.13)$$

  with, $S^p = \{(i,k) : x^p_{i,k} = 1\}$. The neighborhood set contains the solutions that are within a distance no more than π from x^p (in the sense of the *Hamming distance*).

- Intensification: after defining $\mathcal{N}(x^p, \pi)$ by adding the local branching constraint, LocBra solves the corresponding MILP formulation by CPLEX. This intensifies the search by exploring the neighborhood around x^p looking for a new and a better solution. This step implies focusing the search into a small region of the solution space instead of exploring the whole space. Note that, for large instances and even after defining a neighborhood, the modified formulation might still be hard to solve in reasonable time. For this reason, a time limit, called *node_ time_ limit*, is imposed during the intensification.

- Complementary intensification: it may happen that, when exploring a neighborhood $\mathcal{N}(x^p, \pi)$, CPLEX fails to compute a feasible solution because the node time limit is reached. This corresponds to the case where the neighborhood is too large. Then, a complementary intensification phase is performed in the restricted neighborhood $\mathcal{N}(x^p, \pi/2)$.

- Diversification: this step is introduced when the complementary intensification step fails to find an improved solution, which basically means that the current solution x^p is a local optimum. The main goal of this step is to skip local minima and switches the exploration to new regions in the solution space. In an attempt to improve the original diversification mechanism proposed by Fischetti and Lodi (Eq.4.14), a more complex and problem-dependent one is proposed and described in section 4.4.2.

$$\Delta(x, x^p) = \sum_{(i,k)\in S^p} (1 - x_{i,k}) + \sum_{(i,k)\notin S^p} x_{i,k} \geq 1 \tag{4.14}$$

The basis of LocBra is illustrated in Fig. 4.2 that shows how the 4 ingredients are put together. First, LocBra starts with an initial solution x^0, and defines its π*-opt neighborhood* $\mathcal{N}(x^0, \pi)$. This is translated by adding the following *local branching constraint* to JH formulation:

$$\Delta(x, x^0) = \sum_{(i,k)\in S^0} (1 - x_{i,k}) + \sum_{(i,k)\notin S^0} x_{i,k} \leq \pi \tag{4.15}$$

with, $S^0 = \{(i,k) : x^0_{i,k} = 1\}$. This new formulation is then solved leading to the search of the best solution in $\mathcal{N}(x^0, \pi)$. This step, which corresponds to node 2 in Fig. 4.2-a, is referred to as the intensification phase. If a new solution x^1 is found, the constraint (Eq. 4.15) is replaced by $\Delta(x, x^0) \geq \pi + 1$, and the right branch emanating from node 1 is explored. This guarantees that an already visited neighborhood will not be visited again. Next, a left branch is created but now using the solution x^1. And the neighborhood $\mathcal{N}(x^1, \pi)$ is explored by solving the JH formulation with the constraint $\Delta(x, x^1) \leq \pi$ (node 4 in Fig. 4.2-a). Then, the process is repeated until a stopping criterion is met, e.g. a *total time limit* is reached. The stopping criteria are discussed in the following subsections. For instance, assuming at node 6 in Fig. 4.2-a, the solution of JH formulation plus equation $\Delta(x, x^2) \leq \pi$ does not lead to a feasible solution in the given time limit. Then, a complementary intensification step is applied, by replacing the last constraint on x^2 by $\Delta(x, x^2) \leq \pi/2$ and solving the new sub-problem: this results in the exploration of a reduced neighborhood around x^2. If again no feasible solution is found (node 7 in Fig.

4.2-a), then a diversification step is applied to jump to another point in the solution space. Figure 4.2-b shows the evolution of the solution search and the neighborhoods.

4.4.2 Problem-dependent features of the Local Branching heuristic

The original version of local branching is designed to solve any optimization problem with existing MILP formulations. Of course, integrating GED properties and information about the instance in the heuristic will help in improving its performance. The improvements are integrated by adapting certain mechanisms of the method.

The first particularization relates to the choice of the variables when defining the neighborhoods. Traditionally, in a local branching heuristic all boolean variables are considered to in the local branching constraint $\Delta(x, x^p) \leq \pi$. However, for the GED^{EnA} problem it turns out that the crucial variables are the $x_{i,k}$'s, which model vertices matchings. Other sets of variables ($s_{i,k}$ and $t_{i,k}$) in JH, which correspond to edges matchings, can be easily fixed by the solver as soon as the vertices are matched. This realization is concluded based on GED Property 1 explained in Chapter 2. Letting LocBra explores the solution space only on the basis of $x_{i,k}$ variables, leads to the consideration of a smaller number of variables in the local branching constraint. By the way, this strengthens the local search by avoiding fast convergence towards local optima. Consequently, the local branching constraint as defined in Eq. 4.15 involves only $x_{i,k}$ variables.

Another important improvement is proposed for the diversification mechanism: again not all binary variables are included but a smaller set of *important* variables is used instead. Note that, only $x_{i,k}$ variables that represent vertices matchings are considered. The diversification constraint is then

$$\Delta'(x, x^p) = \sum_{(i,k)\in S^p_{imp}} (1 - x_{i,k}) + \sum_{(i,k)\notin S^p_{imp}} x_{i,k} \geq \min(\pi\_dv, \psi) \tag{4.16}$$

with $S^p_{imp} = \{(i,k) \in B_{imp} : x^p_{i,k} = 1\}$ and B_{imp} the index set of important binary variables. This constraint replaces the original diversification constraint (Eq. 4.14). The notion of important variable is based on the idea that when changing its value from $1 \rightarrow 0$ (or the opposite), it highly impacts the objective function value. Forcing the solver to modify such variables enables escaping from local optima and changing the matching. And it does not matter if the new solution is worse than the current solution x^p. The pseudo-code snippet given in Algorithm 5 describes how to determine those important variables. Accordingly, B_{imp} is obtained as follows:

(i) First, compute a special cost matrix $[M_{i,k}]$ for each possible assignment of a vertex $i \in V \cup \{\epsilon\}$, to a vertex $k \in V' \cup \{\epsilon\}$.

$$M = \begin{bmatrix} c_{1,1} + \theta_{1,1} & c_{1,2} + \theta_{1,2} & \dots & c_{1,|V'|} + \theta_{1,|V'|} & c_{1,\epsilon} + \theta_{1,\epsilon} \\ c_{2,1} + \theta_{2,1} & c_{2,2} + \theta_{2,2} & \vdots & c_{2,|V'|} + \theta_{2,|V'|} & c_{2,\epsilon} + \theta_{2,\epsilon} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ c_{|V|,1} + \theta_{|V|,1} & c_{|V|,2} + \theta_{|V|,2} & \dots & c_{|V|,|V'|} + \theta_{|V|,|V'|} & c_{|V|,\epsilon} + \theta_{|V|,\epsilon} \\ c_{\epsilon,1} + \theta_{\epsilon,1} & c_{\epsilon,2} + \theta_{\epsilon,2} & \dots & c_{\epsilon,|V'|} + \theta_{\epsilon,|V'|} & 0 \end{bmatrix} \tag{4.17}$$

Each value $M_{i,k} = c_{i,k} + \theta_{i,k}$, where $c_{i,k}$ is the vertex edit operation cost induced by assigning vertex i to vertex k, and $\theta_{i,k}$ is the cost of assigning the set of edges $E_i = \{(i, w) \in E\}$ to $E_k = \{(k, w') \in E'\}$. E_i (resp. E_k) is the set of edges connected to vertex i (resp. k). A cost matrix can be built over the two sets E_i and E_k, since all edges edit operations costs are known. This assignment problem, of size $max(|E_i|, |E_k|) \times max(|E_i|, |E_k|)$, is solved by the Hungarian algorithm (Munkres (1957)), which requires $O(max(|E_i|, |E_k|)^3)$ time, and this gives the value of $\theta_{i,k}$.

(ii) Next, the standard deviation is computed at each row of the matrix $[M_{i,k}]$, resulting in a vector $\sigma = [\sigma_1, ..., \sigma_{|V|}]$. Typically, a high value of σ_i means that the contribution to the objective function of the matching of vertex i with a vertex k strongly varies depending on k. Such variables are considered as important.

(iii) To isolate the ones with the highest σ_i values, a simple clustering algorithm is applied. Two clusters C_{min} and C_{max} are built by starting with the minimum σ_{min} and maximum σ_{max} values as the centers of the clusters. For all $i \in V \cup \{\epsilon\}$, if $|\sigma_i - avg_{C_{min}}| < |\sigma_i - avg_{C_{max}}|$ then $\sigma_i \rightarrow C_{min}$, otherwise $\sigma_i \rightarrow C_{max}$, with $avg_{C_{min}}$ and $avg_{C_{max}}$ the averages of the selected values in the clusters C_{min} and C_{max}, respectively. Every time a value σ_i is added to C_{min} or C_{max}, its average value $avg_{C_{min}}$ or $avg_{C_{max}}$ is updated.

(iv) At last, for every σ_i belonging to C_{max} cluster, the indexes of all binary variables $x_{i,k}$ that correspond to the assignment of vertex i are added to B_{imp}.

Finally, note that the right hand side of the diversification constraint (Eq. 4.16) is equal to $\min(\pi\_dv, \psi)$. Variable π_dv is a positive integer parameter, which basically should be higher than the π variable used in the local branching constraint (Eq. 4.15). This guarantees better diversification by imposing a minimal number of variables $x_{i,k}$ to be changed. However, π_dv should not exceed ψ, which is the number of i vertices selected in C_{max} ($\psi = |C_{max}|$). Otherwise, solving JH with the diversification constraint will lead to infeasibility and violating the formulation constraints. Here is an example to make the picture more clear:

- Let be two graphs of sizes $|V| = 100$ and $|V'| = 100$, and $\pi\_dv = 50$.
- Assuming that $\psi = |C_{max}| = 30$, this means that 30 vertices are selected from V. Therefore, $|B_{imp}| = 30 \times 100 = 3000$ variable indexes.
- How many variables at most can be changed in B_{imp}?

The answer is 30, because changing more than 30 $x_{i,k}$ variables will lead to matching a vertex i with multiple k vertices in V', which violates the constraints 2.21 and 2.22 in JH formulation.

Eventually, ψ might be very big for large graphs and if the diversification only relies on it, this might lead to a very difficult sub-problem, for which the solver might not be able to find a solution. To avoid crippling the diversification, which is an important phase in LocBra, the right hand side of the constraint is bounded by π_dv (a positive integer parameter to the method).

Algorithm 5: Algorithm to compute important variables for diversification

```
Input  : $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$
Output: Vector $B_{imp}$

/* Compute a special cost matrix                                    */
Let $[M_{i,k}]$ be a matrix of size $|\overline{V}| \times |\overline{V}'|$
foreach $i \in V$ do
    foreach $k \in V'$ do
        Let $E_i = \{(i, w) \in E\}$
        Let $E_k = \{(k, w') \in E'\}$
        Let $\theta_{i,k} = Hungarian(E_i, E_k)$
        $M_{i,k} = c_{i,k} + \theta_{i,k}$ // compute the minimum cost to assign the edges
    end
end
/* Fix non-basic variables                                          */
Let $[\sigma_i]$ be a vector of size $|V|$
foreach $i \in V$ do
    $\sigma_i = StandardDeviation(M_i)$ // compute the standard deviation at line
      $i$
end
/* Compute clusters $C_{min}$ and $C_{max}$                              */
Let $C_{min}$ and $C_{max}$ be two vectors of size $|V|$
Let $\sigma_{min}$ and $\sigma_{max}$ be, respectively, the minimum and maximum values in $[\sigma_i]$
Add $\sigma_{min}$ to $C_{min}$
Add $\sigma_{max}$ to $C_{max}$
Let $avg_{C_{min}} = Average(C_{min})$ // average of values in $C_{min}$
Let $avg_{C_{max}} = Average(C_{max})$ // average of values in $C_{max}$
foreach $i \in V$ do
    if $|\sigma_i - avg_{C_{min}}| < |\sigma_i - avg_{C_{max}}|$ then
        Insert $\sigma_i$ to $C_{min}$
    else
        Insert $\sigma_i$ to $C_{max}$
    end
    $avg_{C_{min}} = Average(C_{min})$
    $avg_{C_{max}} = Average(C_{max})$
end
/* Collect important variables indexes                              */
foreach $\sigma_i \in C_{max}$ do
    foreach $k \in V'$ do
        Add $(i, k)$ to $B_{imp}$
    end
end
Return $B_{imp}$
```

This version of diversification significantly improves the local branching heuristic, better than the original one introduced by Fischetti and Lodi (2003), which was quite inefficient for escaping local optima. Both versions are evaluated and the results are reported in Section 4.4.5. Regardless of the new solution's quality - whether it is better or worse than the current best solution -, what is important is that the diversification mechanism succeeds in diversifying the search and escaping local optima. Because, the intensification steps afterwards will lead to a deep exploration of the solution space around the new solution.

4.4.3 The local Branching algorithm

A detailed algorithmic presentation of LocBra heuristic is provided in Algorithm 6, and the details of the functions used, are given in Algorithm 7. The core function of the heuristic takes the following parameters as input:

1. π, is the neighborhood size.
2. π_dv, is for diversification, to guarantee that the next solution to be found is far enough from the current one by at least π_dv changes of binary variables.
3. $total\_time\_limit$, is the total running time allowed for LocBra before stopping.
4. $node\_time\_limit$, is the maximum running time given to the solver at any node to solve the JH formulation.
5. UB_time_limit, is the running time allowed to the solver to compute an initial solution.
6. ℓ_max, is used to force a diversification step after a sequence of ℓ_max intensification steps returning solutions with the same objective function value. This parameter avoids spending a lot of time searching in a region where no improving solutions are found.
7. dv_max, is the maximum number of diversification steps allowed during the execution of LocBra. The rationale behind such a parameter comes from preliminary experiments, which have shown that first diversification steps are useful to reach very good solutions. Then, this parameter enables to decrease the global execution time without losing in the quality of the solutions returned by LocBra.
8. dv_cons_max, serves as a stopping criterion. The heuristic stops after consecutive diversification steps returning solutions with the same value of the objective function. When this situation occurs, then the diversification mechanisms is inefficient in escaping from the current local optimum.

From the above list of parameters, the stopping criteria of LocBra heuristic do not only rely on the total time spent. The algorithm stops whenever one of these three conditions is met:

(i) the total execution time exceeds the $total\_time\_limit$, or

Algorithm 6: *LocBra* algorithm

```
1  bestUB := UB := $\infty$; $x^* := \bar{x} := \tilde{x} := undefined$
2  tl := elapsed_time := dv := $\ell$ := dv_cons := 0
3  mode_dv := false; opt := false; first_loop := true
1  Function LocBra($\pi, \pi\_dv, total\_time\_limit, node\_time\_limit,$
          $UB\_time\_limit, dv\_max, \ell\_max, dv\_cons\_max$)
       Output: $x^*$, opt
2      tl = $UB\_time\_limit$ // Set the time to compute the initial solution
3
4      InitLocBra()
5      ImprovedSolution()
6      elapsed_time := tl
7      tl = $node\_time\_limit$ // Set the time for branching
8
9      while $elapsed\_time < total\_time\_limit$ and $dv < dv\_max$ and $dv\_cons < dv\_cons\_max$ do
10         tl := min{tl, total_time_limit − elapsed_time}
11         status := MIP_SOLVER(tl, UB, $\tilde{x}$)
12         tl := node_time_limit
13         if $f(\tilde{x}) = f(\bar{x})$ and $mode\_dv = true$ then $\ell := \ell + 1$ else $\ell := 0$
14         if $\ell \geq \ell\_max$ then Diversification(); continue
15         if $status$ = "$opt\_sol\_found$" then
16             if $\tilde{x} \neq \bar{x}$ then ImprovedSolution() else Diversification()
17         end
18         if $status$ = "$infeasible$" then Diversification()
19         if $status$ = "$feasible\_sol\_found$" then
20             if $f(\tilde{x}) < UB$ then
21                 ImprovedSolution()
22             else
23                 if $mode\_dv = false$ then Intensification() else
                    Diversification()
24             end
25         end
26         elapsed_time := elapsed_time + tl
27     end
28 End
```

Algorithm 7: *LocBra* helper functions

```
1 Function InitLocBra()
2     status := MIP_SOLVER(tl, UB, $\tilde{x}$)
3     if status = "opt_sol_found" then opt := true; $x^*$ := $\tilde{x}$; exit
4     if status = "infeasible" then opt := false; exit
5 End
1 Function ImprovedSolution()
2     if mode_dv = false and $\bar{x} \neq undefined$ then
3         replace last added constraint $\Delta(x,\bar{x}) \leq \pi$ by $\Delta(x,\bar{x}) \geq \pi + 1$
4     end
5     $\bar{x}$ := $\tilde{x}$; UB := $f(\tilde{x})$; mode_dv := false; dv_cons := 0
6     add new constraint $\Delta(x,\bar{x}) \leq \pi$
7     if UB < bestUB then $x^*$ := $\tilde{x}$; bestUB := $f(\tilde{x})$
8 End
1 Function Diversification()
2     replace last constraint $\Delta(x,\bar{x}) \leq \pi$ with $\Delta'(x,\bar{x}) \geq \min(\pi\_dv, \psi)$
3     UB := $\infty$; dv := dv + 1; mode_dv := true; dv_cons := dv_cons + 1
4 End
1 Function Intensification()
2     replace last added constraint $\Delta(x,\bar{x}) \leq \pi$ by $\Delta(x,\bar{x}) \leq \frac{\pi}{2}$
3     mode_dv := false; dv_cons := 0
4 End
```

(ii) the number of diversification steps done during the search exceeds dv_max, or

(iii) the number of consecutive diversification steps done exceeds dv_cons_max.

The output of the algorithm is the best solution found (x^*) along the search, and a flag to indicate whether it has been proved to be optimal or not (*opt*). The initial solution x^0 used by LocBra is obtained by solving JH formulation within a time limitation of UB_time_limit seconds. First, it calls *InitLocBra* function that initializes the heuristic by computing an initial solution $\tilde{x}$ (it is the first solution x^0 as introduced in Section 4.4.1). If at this point, JH is solved to optimality or no feasible solution has been found, the heuristic halts and returns the available solution and/or the status. Otherwise, the current solution $\bar{x}$ is set to the solution found and the exploration begins. Lines 2 to 23 present the core of the heuristic as previously described. At each iteration and after a left/right branching constraint is added, the solver is called through *MIP_Solver(tl,UB,$\tilde{x}$)* function, and the returned status is considered to make the next decision. Note that, *tl* variable corresponds to the time limit imposed when solving JH. $\tilde{x}$ and UB are, respectively, the solution computed by the solver (new solution) and its objective function value. Three possible statuses may occur:

(i) **Optimal solution** is found at a branch, and then two cases must be distinguished (line 11). If the new solution $\tilde{x}$ is better than the current solution $\bar{x}$, then *Improved-Solution* is called to update the current and best solutions (if needed), and to define a new neighborhood by adding the constraint Eq. 4.15 using the new solution $\tilde{x}$. If the new solution $\tilde{x}$ and the last solution $\bar{x}$ are equal, i.e. $\tilde{x} = \bar{x}$, then *Diversification* is called to skip the current neighborhood and search in a different region in the search space. *Diversification* function ensures that the current solution is skipped with a distance $min(\pi\_dv, \psi)$, and the upper bound UB is reset to ∞ to allow finding a new solution even if it is worse than the best known one.

(ii) The formulation is **infeasible** (line 14). Therefore *Diversification* is triggered to switch the last local branching constraint and look into a new neighborhood in the search space.

(iii) A **feasible solution** is returned (line 15). This is very similar to the first case, except when a worse solution is found, i.e. $f(\tilde{x}) > UB$. An additional *Intensification* step is done but within a neighborhood limited to $\pi/2$ variable changes from $\bar{x}$ in order to improve it. However, if the solver fails again, then a *Diversification* step is performed.

In addition, there is the condition (at line 10) that forces the diversification step, in the case where ℓ_max consecutive intensification iterations have returned solutions with the same objective function value. This, in turn, guarantees the exploration of many neighborhoods in different regions of the solution space, rather than remaining stuck in one region.

4.4.4 Parameters tuning

LocBra has multiple parameters to control the exploration of the search space. They are split into two groups: pre-fixed parameters and tuned parameters. The first group contains parameters that are fixed for all the experiments and databases. Their values were chosen based on existing similar works, or suggestions by the authors of the original version of local branching. The second group of parameters are the ones that need to be tuned for every database of graphs. The two groups are as follows:

- Pre-fixed: the parameters values are set after selecting instances from different databases and testing them with different values. The best combination that led to the best results, are retained.

  1. π, its value is set to 20 based on the experiments done by Fischetti and Lodi (2003). They have suggested to set π to a value between $[10, 20]$, where good performances are obtained. As well, it makes sense to have such a value, because choosing a bigger value means that the neighborhood $\mathcal{N}(x, \pi)$ is rather large and the sub-problem might be more complicated to solve. A second reason not to increase this value, is that local branching is a metaheuristic based on small neighborhoods, and by having large neighborhoods it becomes more like a large neighborhood search metaheuristic.
  2. π_dv, the value assigned to this variable is 30.
  3. l_max, it is set to 3.
  4. dv_max, it is set to 5.
  5. dv_cons_max, it is set to 2.

- Tuned: the parameters values are set for each database.

  1. *total_time_limit*, the value of this variable is set after preliminary experiments over few instances from the database. The value that gives the best results is, then, set to run the method over all instances.
  2. *node_time_limit*, it is regulated based on preliminary tests by selecting instances from the database.
  3. *UB_time_limit*, since computing a good initial solution is crucial in LocBra, this parameter is also tuned by running tests on some instances. Then, selecting the value that yielded the best results and set it for the rest of the instances.

Basically, all the parameters are set based on preliminary tests. However, they can be tuned in an adaptive fashion, which could be a complex problem by itself. Performing a study on each parameter, its influence on the method and the relations between the parameters is not an easy task. It was convenient to obtain fair results and performance by regulating the values of the parameters empirically. Henceforth, each time an experiment is conducted over a database, preliminary tests are conducted with different values of parameters. The combination that gives the best results, is the one used to execute the heuristic over all the instances in the database. The parameters values are indicated for each database in each experiment.

4.4.5 Original local branching vs improved local branching

The proposed version of local branching is designed and adapted to solve the GED^{EnA} problem. As stated in Sections 4.4.2 and 4.4.3, the local branching constraints are changed and replaced by Eq. 4.13. The new constraints consider only binary variables that represent vertices matchings. Another modification is the diversification constraint (Eq. 4.16) that replaces the original one (Eq. 4.14). The new version with the modifications has been tested empirically on two hard instances, but this is not enough to generally asses the improvements. Therefore, to wrap-up the work and prove experimentally the importance of those modifications in solving better the problem, a comparison with the original local branching scheme of Fischetti and Lodi, is conducted.

Methods. There are two of them:

1. LocBra-ori: original version of local branching as proposed by Fischetti and Lodi (2003).
2. LocBra-imp: improved version of local branching as introduced in this work.

Database. The subset 70 of graph's database MUTA, presented in Section 4.2, is selected in the experiment. The graphs in this subset are large and they are known to be very hard for GED heuristics.

Instances and experimentation settings. Both local branching versions are implemented in C language. The solver CPLEX 12.6.0, in single thread mode, is used in this experiment. The machine configuration: Windows 7 $x64$, Intel Xeon $E5$ 2.30 GHz, 4 cores and 8 GB of RAM. Finally, the parameters values set to both methods are:

$LocBra-ori$	$\pi = 20$, $total\_time\_limit = 900s$, $node\_time\_limit = UB\_time\_limit = 180s$
$LocBra-imp$	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 900s$, $node\_time\_limit = UB\_time\_limit = 180s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$

Comparison indicators. The following indicators are computed for each method: t_{min}, t_{avg}, and t_{max}, which are respectively the minimum, average and maximum CPU times in seconds over all instances. Correspondingly, d_{min}, d_{avg}, and d_{max} are the deviations (in percentage) of the solutions obtained by LocBra, from the best solutions computed by both methods. The deviation is computed by using Eq. 4.11. η_I is the number of instances for which a given heuristic has found the best solutions.

Results. The results, reported in Table 4.8, prove the superiority of LocBra-imp over LocBr-ori. The former has a better (smaller) $d_{avg} = 0.44\%$ against 0.83%. It has computed better solutions as well, than the original version with $\eta_I = 84$ against 75 over 100 instances. These differences might seem small, but as the instances used in the experiment are very hard to solve, they are significant and very important.

Table 4.8: LocBra-ori vs. LocBra-imp

	LocBra-ori	*LocBra-imp*
t_{min}	1.75	1.36
t_{avg}	759.26	**751.44**
t_{max}	901.11	900.36
d_{min}	0.00	0.00
d_{avg}	0.83	**0.44**
d_{max}	9.50	5.88
η_I	75	**84**

Conclusion. Experimentally, the improved local branching designed to solve the GED^{EnA} problem is more efficient than the original version, and solves better the problem instances. This is all thanks to the analysis and the problem-dependent diversification mechanism, that considers information and local structures around vertices for the instance at hand.

4.4.6 Evaluation of local branching

This section presents the experimentation results to the different tests conducted on LocBra and the heuristics selected from the literature that are known to be the best at solving the problem. The goal of the experiments is to study the effectiveness of LocBra heuristic and its contribution to GED applications. The experiments are divided into three categories:

1. **Effectiveness of LocBra w.r.t. competitor heuristics**. LocBra is tested against the most competitive heuristics picked from the literature, designed to solve the GED problem. And of course they can be applied to the GED^{EnA} problem.

2. **Effectiveness of LocBra w.r.t. an exact method**. The goal of this experiment is to measure the accuracy and closeness of LocBra solutions from the optimal or best known ones.

3. **Accuracy of LocBra from an application point of view**. This experiment aims at evaluating the heuristic from an application point of view. It is a serious attempt to answer the questions raised in *GED challenges* (Section 2.3.6). To the best of our knowledge, this kind of experiment has not yet been considered when evaluating GED methods. It consists of two tests:

   (a) The first one is in the context of similarity search, which is relevant when seeking the nearest neighbor graphs. A query graph is compared to each graph in a database, thanks to a given GED^{EnA} heuristic, which returns each time the proposed matching as well as the distance between the two graphs. Then, distances are sorted in ascending order to obtain a ranking. The target of this experiment is to compare the rankings given by all heuristics against the optimal ranking given by optimal methods. Though, this kind of evaluation has not been done in the literature when evaluating GED heuristics.

(b) The second test studies the relation between the ground-truth matchings given by human experts and the matchings computed by the heuristics. This evaluation is also not common in the context of heuristics comparison, in spite of its importance. Visualizing the computed matching and comparing it with the ground-truth matching enables assessing the impact of mismatched vertices.

Common configuration. LocBra algorithm is implemented in C language. The solver CPLEX 12.6.0 is used to solve the MILP formulations. Experiments are ran on a machine with Windows 7 $x64$, Intel Xeon $E5$ 2.30 GHz, 4 cores and 8 GB of RAM. When solving a MILP formulation within LocBra heuristic, CPLEX solver is parametrized to use a single thread (unless indicated otherwise) even if 4 cores are available. The aim of this, in the experiments, is to evaluate the efficiency of the inner mechanism of LocBra. It can be expected that its efficiency is going to be improved when enabling more threads. Although the machine configuration is $x64$ based, the code was compiled in $x86$ mode, which means LocBra process can reach a maximum of $2GB$ in terms of memory occupancy.

4.4.6.1 Effectiveness of LocBra w.r.t. competitor heuristics

These experiments answer the following question: which heuristic is the best minimizer? It is about comparing the distances computed by each heuristic and finds out which heuristic returns the smallest distances.

Methods. LocBra is being tested against the following heuristics:

i- *CPLEX-t* is the solver CPLEX ran on JH formulation with t seconds as a time limit.

ii- *CPLEX-LocBra-t* refers to enabling local branching heuristic implemented in CPLEX solver. Note that in the default settings, this heuristic is disabled. So, *CPLEX-LocBra-t* is the local branching heuristic as implemented in CPLEX with a time limit of t seconds. The time limit is imposed in order to compute an initial solution, which will be given to CPLEX to apply local branching on.

iii- *BeamSearch-α*, the BeamSearch heuristic with α the beam size.

iv- *SBPBeam-α*, the SBPBeam heuristic with α the beam size.

v- *IPFP-it*, the IPFP heuristic with it the maximum number of iterations.

vi- *GNCCP-d*, the GNCCP heuristic with d the quantity to be subtracted from the ζ variable at each iteration. ζ is the variable that controls the concavity and convexity of the objective function of the QAP model solved by GNCCP heuristic.

The first two heuristics are also based on solving the JH formulation by CPLEX, and considering them as a part of the experiment will show if the branching scheme of LocBra is capable of performing better than the solver CPLEX and its embedded heuristics. The other heuristics are picked from the literature after reviewing the most important and competitive ones. Their descriptions and details can be found in Chapter 2, Section 2.3.8.

Comparison indicators. All heuristics are executed on different databases and for all of them, the following indicators are computed: t_{min}, t_{avg}, and t_{max} are the minimum, average and maximum CPU times in seconds over all instances. Correspondingly, d_{min}, d_{avg}, and d_{max} are the deviations of the solutions obtained by one heuristic, from the best solutions found by all heuristics. The deviations are computed based on Eq. 4.11 and are expressed in percentage. Lastly, η_I is the number of instances for which a given heuristic has found the best solutions.

Evaluations on MUTA database. This is the first database of graphs selected in this experiment. All the details about it are given in Section 4.2. MUTA is a database that is divided into 8 subsets of different sizes of graphs and it contains easy and hard instances.

All the heuristics have one or more parameters to control their performances. For the heuristics taken from the literature, each method has default parameters that are suggested by the authors. Some of them are designed to converge very fast. It is not the case with LocBra, it is based on solving the MILP formulation multiple times. Therefore, it may require more time than the others. To harmonize all the tests, two versions of the heuristics are considered:

- Default versions: heuristics with their default parameters as suggested by their authors.
- Extended versions: heuristics with large parameter values that bring up their running time to almost the same time required by LocBra. Those values are determined based on preliminary tests to achieve this goal. Note that, *CPLEX-t* is left out of this experiment because its running time is already equal to the running time given to *LocBra* in the default versions.

The parameters are provided for each heuristic before the analysis of the results.

Default versions. The following are the values of the parameters set for each method. The values are set for LocBra, *CPLEX-t* and *CPLEX-LocBra-t* based on preliminary tests. *CPLEX-LocBra-t* is given t seconds equal to the time given to CPLEX to compute the initial solution in LocBra heuristic.

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 900s$, $node\_time\_limit = UB\_time\_limit = 180s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
CPLEX-t	$t = 900$
CPLEX-LocBra-t	$t = 180$
BeamSearch-α	$\alpha = 5$
SBPBeam-α	$\alpha = 5$
IPFP-it	$it = 10$
GNCCP-d	$d = 0.1$

Based on the results shown in Table 4.9, the heuristics *LocBra*, *CPLEX-900*, *CPLEX-LocBra-180*, which are MILP-based, have the highest η_I for all the subsets. And they strongly outperform the two beam search-based heuristics, *IPFP-10* and *GNCCP-0.1*. On

Table 4.9: LocBra vs. heuristics on MUTA instances

	S	10	20	30	40	50	60	70	Mixed
LocBra	t_{min}	0.06	0.13	0.28	0.45	0.69	0.95	1.36	0.14
	t_{avg}	0.17	1.12	212.36	364.86	580.04	753.48	751.44	332.32
	t_{max}	2.92	3.63	900.13	900.12	900.17	900.27	900.36	902.25
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	**0.00**	0.06	**0.02**	**0.17**	**0.59**	**0.04**
	d_{max}	0.00	0.00	0.00	3.90	2.03	3.35	5.57	1.77
	η_I	**100**	**100**	**100**	98	**99**	**93**	**79**	**95**
CPLEX-900	t_{min}	0.06	0.14	0.28	0.49	0.77	1.18	1.70	0.09
	t_{avg}	0.13	1.02	141.07	247.80	451.40	723.68	745.91	305.72
	t_{max}	0.49	3.52	900.20	900.42	900.46	900.71	900.92	900.70
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	**0.00**	**0.00**	0.30	0.55	1.05	0.12
	d_{max}	0.00	0.00	0.00	0.00	6.42	5.04	8.57	5.49
	η_I	**100**	**100**	**100**	**100**	90	81	68	**95**
CPLEX-LocBra-180	t_{min}	0.09	0.22	0.41	0.73	1.03	1.45	1.98	0.14
	t_{avg}	0.21	1.51	60.36	104.19	141.43	167.59	181.18	86.32
	t_{max}	0.74	5.77	182.86	194.08	195.43	217.38	263.60	223.53
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	**0.00**	0.16	1.16	1.41	4.24	0.36
	d_{max}	0.00	0.00	0.00	3.90	7.19	6.70	27.20	6.86
	η_I	**100**	**100**	**100**	94	72	57	41	82
BeamSearch-5	t_{min}	0.00	0.00	0.01	0.01	0.02	0.04	0.06	0.01
	t_{avg}	**0.00**	**0.00**	**0.01**	**0.03**	**0.07**	**0.11**	**0.18**	**0.09**
	t_{max}	0.07	0.02	0.04	0.11	0.09	0.13	0.22	0.21
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	15.17	36.60	47.21	58.69	72.13	62.96	68.71	21.20
	d_{max}	110.00	124.59	147.37	186.67	200.00	146.37	210.71	112.71
	η_I	35	10	10	10	10	10	10	12
SBPBeam-5	t_{min}	0.01	0.08	0.31	1.11	2.69	4.87	9.02	0.05
	t_{avg}	0.01	0.10	0.45	1.37	3.19	5.56	10.72	3.38
	t_{max}	0.05	0.14	0.54	1.60	3.71	6.85	12.79	12.05
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	20.43	44.90	76.45	82.54	98.90	95.02	94.62	27.16
	d_{max}	90.00	127.87	206.90	204.71	314.29	198.50	280.36	135.91
	η_I	15	10	10	10	10	10	10	10
IPFP-10	t_{min}	0.00	0.01	0.02	0.03	0.06	0.10	0.15	0.01
	t_{avg}	0.01	0.06	0.20	0.30	0.39	0.66	1.05	0.46
	t_{max}	0.08	0.20	0.35	0.59	0.56	1.01	1.49	1.39
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	3.44	10.84	18.31	21.34	22.59	25.9	27.63	7.45
	d_{max}	30.00	80.77	90.41	93.33	66.67	66.67	99.08	49.72
	η_I	69	28	14	11	10	10	10	19
GNCCP-0.1	t_{min}	0.02	0.12	0.38	0.89	1.68	2.88	4.59	0.15
	t_{avg}	0.16	1.30	4.77	11.78	22.08	72.29	111.30	28.53
	t_{max}	0.29	2.52	10.86	31.58	73.46	145.53	255.88	218.99
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	13.29	22.00	26.93	21.69	31.46	21.99	27.61	10.66
	d_{max}	411.43	400.00	188.79	119.12	205.36	205.77	101.79	125.16
	η_I	57	35	6	6	5	9	4	17

easy instances (subsets 10 to 40 vertices), the MILP-based heuristics have yielded the best solutions for almost all instances (except 2 instances for subset 40). However, a major difference starts to appear on hard instances (subsets $50, 60, 70$), where *LocBra* scores the highest values, with $\eta_I = 99$ for subset 50, $\eta_I = 93$ for subset 60 and $\eta_I = 79$ for subset 70 of best solutions. *CPLEX-900* comes at the second place, followed by *CPLEX-LocBra-180*, regarding the number of best solutions obtained. Considering the average deviations, *LocBra*, on hard instances, has the smallest value ($d_{avg} < 0.6\%$). And again *CPLEX-900* scores the second lowest deviations $0\% \leq d_{avg} \leq 1.05\%$. *IPFP-10* and *GNCCP-0.1* have a maximum deviation (d_{avg}) of 28%, which means they perform better than the beam-search based heuristics. *BeamSearch-5* and *SBPBeam-5* are very poor in terms of solutions quality with a very high d_{avg} (about at most 98.9%). Looking at the solution time, *BeamSearch-5* is the fastest with a running time between 0 and 0.18 seconds. Heuristics *IPFP-10*, *SBPBeam-5* and *GNCCP-0.1* come after *BeamSearch-5* in terms of CPU time. Note that, for the instances of mixed sizes, the above conclusions regarding the quality and time still hold.

Table 4.10: LocBra vs. heuristics with extended running time on MUTA instances

	S	10	20	30	40	50	60	70	Mixed
LocBra	t_{min}	0.06	0.13	0.28	0.45	0.69	0.95	1.36	0.14
	t_{avg}	**0.17**	**1.12**	212.36	364.86	580.04	753.48	751.44	332.32
	t_{max}	2.92	3.63	900.13	900.12	900.17	900.27	900.36	902.25
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	**0.00**	0.02	**0.02**	**0.13**	**0.54**	**0.04**
	d_{max}	0.00	0.00	0.00	1.69	2.03	2.94	5.57	1.77
	η_I	**100**	**100**	**100**	99	**99**	**94**	**80**	**97**
CPLEX-LocBra-800	t_{min}	0.08	0.21	0.38	0.67	1.01	1.40	1.94	0.20
	t_{avg}	0.20	1.34	130.26	230.68	424.70	662.58	**688.13**	291.64
	t_{max}	0.71	3.90	802.16	806.16	821.39	839.69	869.65	849.58
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	**0.00**	**0.00**	0.38	0.57	1.02	0.13
	d_{max}	0.00	0.00	0.00	0.00	6.42	5.04	11.27	5.49
	η_I	**100**	**100**	**100**	**100**	89	81	71	95
BeamSearch-15000	t_{min}	0.00	0.00	0.03	0.10	0.55	0.24	2.28	0.03
	t_{avg}	8.57	80.65	167.48	279.11	439.68	640.29	938.66	828.52
	t_{max}	31.52	118.71	230.63	419.73	771.90	878.89	1385.11	1800.00
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-
	d_{avg}	1.35	26.66	47.45	52.29	63.98	62.51	63.71	-
	d_{max}	30.00	142.31	165.52	180.00	150.00	157.63	226.79	-
	η_I	88	12	10	10	10	10	10	-
SBPBeam-400	t_{min}	0.76	9.02	39.85	116.11	288.38	548.04	1019	1.98
	t_{avg}	0.84	10.02	47.65	139.75	322.43	590.86	1155	326.64
	t_{max}	0.96	11.27	54.11	152.34	360.47	657.26	1310	1225.92
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	20.43	44.90	76.45	82.45	98.90	94.94	94.54	26.95
	d_{max}	90.00	127.87	206.90	204.71	314.29	198.50	280.36	135.91
	η_I	15	10	10	10	10	10	10	10
IPFP-20000	t_{min}	0.00	0.01	0.02	0.3	0.11	0.10	0.18	0.01
	t_{avg}	1.20	9.62	48.90	115.14	240.54	528.82	903	303.21
	t_{max}	8.52	53.83	165.44	456.93	771.64	1620	2839	1827
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	3.44	10.18	16.45	17.17	19.00	18.99	20.70	6.03
	d_{max}	30.00	80.77	90.41	56.47	47.62	50.53	85.71	38.03
	η_I	69	29	14	11	10	10	10	21
GNCCP-0.03	t_{min}	0.03	0.18	0.58	1.26	2.44	4.33	6.65	0.25
	t_{avg}	0.55	6.41	**29.80**	**81.24**	**195.89**	**396.37**	946.25	**185.55**
	t_{max}	1.13	16.81	71.94	167.06	450.41	797.39	2330	1398.72
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	3.23	6.67	17.20	15.74	18.38	16.12	18.17	5.13
	d_{max}	90.00	30.7 7	82.76	57.35	95.24	52.29	77.06	25.35
	η_I	81	34	4	6	5	9	5	17

Extended versions. The parameter values, given here, are set empirically to extend the running time of the heuristics picked from the literature to reach approximately the 900s given to LocBra.

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 900s$, $node\_time\_limit = UB\_time\_limit = 180s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
CPLEX-LocBra-t	$t = 800$
BeamSearch-α	$\alpha = 15000$
SBPBeam-α	$\alpha = 400$
IPFP-it	$it = 20000$
GNCCP-d	$d = 0.03$

Table 4.10 shows the results of the heuristics with extended running times. *LocBra* and *CPLEX-LocBra-800* seem to have d_{avg} very close for small instances. The difference starts to grow on hard and mixed instances where *LocBra* scores the lowest values (less than 0.6%). The average deviation remains very high for beam-search based heuristics: increasing the beam size did not actually improve the results obtained by BeamSearch and SBPBeam. *BeamSearch-15000* did not return feasible solutions for the set of mixed graphs, therefore the deviations and η_I are not computed. *IPFP-20000* and *GNCCP-0.03* perform better and get smaller deviation comparing to the original versions (Table 4.9). However, they remain far from LocBra heuristic. Regarding the running time, LocBra is the fastest for subsets 10 and 20, then *GNCCP-0.03* becomes the fastest method for the rest of the subsets. Note that, GNCCP and IPFP are heuristics that have converging conditions, which means that they stop if these conditions are satisfied, regardless of the number of iterations left. For this reason, GNCCP is the fastest because it does not always reach its maximum number of iterations.

Concolusion. With respect to the solution quality, the results show that LocBra strongly outperforms all the literature heuristics, in the case where the default parameters as in their original references, are used. Besides, LocBra also outperforms *CPLEX-900* and *CPLEX-LocBra-180.* To this end, the proposed local branching heuristic is more effective than the solver and its generic implementation of local branching. However, LocBra is slower than the existing heuristics.

The same experiments are done on other databases: PAH, HOUSE-NA and HOUSE-A. The results and the analyses can be found in Appendix A, Section A.1.1. The conclusions of those experiments are:

- **PAH database:** *LocBra*, in both default and extended versions, outperforms the existing heuristics by computing better solutions for PAH instances. However, *CPLEX-t* heuristic is also good at solving PAH instances and performs better in the average case. This due to the presence of very small instances in PAH database, which are very easy to solve by CPLEX to optimality.

- **HOUSE-NA database:** The results are very interesting and shows that *LocBra* is better than all existing heuristics in terms of solutions quality. Moreover, it is also faster than *SBPBeam* and *GNCCP* heuristics.

- **HOUSE-A database:** For HOUSE-A database, *LocBra* was able to solve efficiently all the instances and with reasonable running times. It is even faster than existing heuristics such as *SBPBeam* and *GNCCP*. It is very suitable for this graph database.

General conclusions based on the evaluations. Based on all the experiments reported earlier and the summary in Table 4.11, the proposed local branching heuristic significantly outperforms the heuristics available in the literature. It is shown, as well, the capacity of LocBra in improving the solutions quality over the default behavior of CPLEX and the embedded local branching version in CPLEX. It is not the fastest in terms of running time, but it is the most effective and thus it is a very good minimizer to the GED^{EnA} problem. It is worth mentioning that on HOUSE-NA and HOUSE-A, LocBra was faster than SBPBeam and GNCCP and it has computed the best solutions.

Table 4.11: Summary of LocBra comparison w.r.t. competitor heuristics

	Database	Solutions quality	Speed
Default versions	MUTA	LocBra	BeamSearch
	PAH	CPLEX	BeamSearch
	HOUSE-NA	LocBra	IPFP
	HOUSE-A	LocBra/CPLEX	IPFP
Extended versions	MUTA	LocBra	GNCCP/LocBra
	PAH	CPLEX	IPFP
	HOUSE-NA	LocBra	IPFP
	HOUSE-A	-	-

4.4.6.2 Effectiveness of LocBra w.r.t. an exact method

This experiment will answer the following question: how close the LocBra solutions are from the optimal solutions?

Methods. The exact approach consists in solving the JH formulation using CPLEX limitations, in order to get the optimal solutions. Note that, here the default settings of the solver are used, this means the solver uses 4 threads. However, even without imposing restrictions and limitations on the resources of CPLEX, it can occur that CPLEX may not be able, on the largest instances, to compute the optimal solutions after several hours (time limit was set to 10 hours). In this case and when CPLEX is stopped, the best solution found is returned. The exact approach is denoted by CPLEX-∞.

Comparison indicators. The following indicators are computed for each database: t_{min}, t_{avg}, and t_{max}, which are respectively the minimum, average and maximum CPU times in seconds over all instances. Correspondingly, d_{min}, d_{avg}, and d_{max} are the deviations (in percentage) of the solutions obtained by LocBra, from the optimal or best solutions found. The deviation is computed by using Eq. 4.11. In addition, η_I is the number of optimal solutions found, and η'_I is the number of solutions found by LocBra that are equal to the optimal or best known ones. At last, η''_I is the number of solutions computed by LocBra that are better than the best known solutions, when CPLEX-∞ was not able to find optimal solutions.

Table 4.12: LocBra vs. Exact solution on MUTA instances

	CPLEX-∞ (4 threads)				*LocBra (1 thread)*									*LocBra (4 threads)*								
S	t_{min}	t_{avg}	t_{max}	η_I	t_{min}	t_{avg}	t_{max}	d_{min}	d_{avg}	d_{max}	η_I	η'_I	η''_I	t_{min}	t_{avg}	t_{max}	d_{min}	d_{avg}	d_{max}	η_I	η'_I	η''_I
10	0.07	**0.12**	0.32	100	0.06	0.17	2.92	0.00	**0.00**	0.00	100	**100**	0	0.07	0.16	0.48	0.00	**0.00**	0.00	**100**	**100**	0
20	0.15	**0.95**	19.74	100	0.13	1.12	3.63	0.00	**0.00**	0.00	100	**100**	0	0.14	1.00	21.80	0.00	**0.00**	0.00	**100**	**100**	0
30	0.31	101.53	2865.24	100	0.28	212.36	900.13	0.00	**0.00**	0.00	78	**100**	0	0.32	**101.33**	900.10	0.00	**0.00**	0.00	**91**	**100**	0
40	0.52	266.00	9243.72	99	0.45	364.86	900.12	0.00	0.06	3.90	63	98	0	0.49	**179.45**	900.13	0.00	**0.00**	0.00	**84**	**100**	0
50	0.83	682.71	4212.68	92	0.69	580.04	900.17	-1.79	0.04	4.14	37	97	**1**	0.73	**435.16**	900.32	-1.79	**0.00**	2.07	**54**	**98**	**1**
60	1.24	2419.33	14732.35	71	0.95	753.48	900.27	-2.68	0.36	3.57	16	82	2	1.09	**718.25**	901.66	-3.31	**-0.03**	3.21	**21**	**90**	**6**
70	1.80	3740.34	24185.25	35	1.36	751.44	900.36	-2.67	0.78	8.85	17	52	14	1.48	**741.52**	901.35	-3.90	**0.22**	3.65	**18**	**60**	**16**
Mixed	0.09	1613.41	17084.43	91	0.14	332.92	902.25	-2.67	**0.03**	3.43	64	88	**5**	0.09	**324.17**	900.27	-1.35	0.05	1.87	**66**	**92**	2

Evaluations on MUTA database. It is said earlier that MUTA instances, especially subsets $40, 50, 60$ and 70, are harder than other instances of graphs. To this end, two versions of LocBra are included in this experiment: the first one is with one thread used by CPLEX to solve the MILP formulations, and the second one with 4 threads. The aim is to evaluate the gain for LocBra heuristic when increasing the calculation capacity of CPLEX solver. LocBra parameters are set to the following values: $\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 900s$, $node\_time\_limit = UB\_time\_limit = 180s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$.

The results on MUTA instances are reported in Table 4.12. All optimal solutions are found by CPLEX-∞ (4 threads) for easy instances (subsets 10 to 40), except one. And both LocBra with 1 and 4 threads have $d_{avg} = 0\%$ (except for 2 instances in subset 40). Clearly both versions have returned the same optimal or best solutions. For hard instances (subsets 50 to 70), d_{avg} is always less than 1% and even less than 0% (-0.03%) for the version with 4 threads on subset 60. Note that, a negative deviation implies that for some instances CPLEX-∞ was not able to find the optimal solutions and that LocBra provided better solutions. The values of η_I reveal that this case occurred. η''_I for hard instances reveals that the heuristics have outperformed CPLEX-∞ and found improved solutions (better than the best ones obtained) for 17 instances with 1 thread and 23 instances with 4 threads. Considering the running time, it drastically increases for CPLEX-∞ and reaches thousands ($t_{avg} = 3740s$), while the proposed heuristic has a maximum of $t_{avg} = 751s$. It is the same conclusion for subset Mixed: both LocBra versions are faster in terms of average CPU time, and also they have very low average deviations.

Conclusion. The first conclusion from the experiments on MUTA instances is that local branching mechanism as implemented is very efficient in reaching optimal or near-optimal solutions. The second conclusion relies on the number of threads used within LocBra heuristic: even if, as expected, increasing the number of used threads in CPLEX leads to an improvement of the heuristic, this one is reduced enough to render this improvement marginal.

More results and discussions can be found in Appendix A, Section A.1.2 for other databases (PAH, HOUSE-NA and HOUSE-A). The conclusions are:

- **PAH database:** This experiment shows that PAH instances are easy ones for the JH formulation, so CPLEX is very fast in computing the optimal solutions. In addition, they are easy for LocBra, which has computed solutions with a deviation of 0.35% from the optimal ones.

- **HOUSE-NA database:** On HOUSE-NA instances, LocBra is 5 times faster than CPLEX on the average. Also, LocBra reaches a maximum time of $10s$, while CPLEX reaches more than $2000s$ in the worst case. The solutions computed by LocBra are close from the optimal solutions ($d_{avg} = 11\%$).
- **HOUSE-A database:** The instance of HOUSE-A are very easy to solve by CPLEX and LocBra. The average deviation is 0%, which means that LocBra has computed solutions equal to the optimal ones. The CPU times of CPLEX and LocBra are very close, where CPLEX is a bit faster than LocBra (by $0.2s$).

General conclusions based on the evaluations. In the case of easy instances (small graphs) as in PAH database, HOUSE-A and subsets 10 to 40 in MUTA database, CPLEX is very efficient in solving JH formulation and obtaining the optimal solutions. On the other hand, LocBra results prove also its efficiency in obtaining near-optimal solutions. One advantage of LocBra is that it is always faster in the worst case because of the time limit imposed. Regarding the hard instances, as in MUTA (subsets $50, 60, 70$) and HOUSE-NA, LocBra was able to overcome CPLEX and achieved better results on instances where *CPLEX* failed to compute the optimal solutions. All those results have answered the question asked at the beginning of this experiment. LocBra is capable of computing solutions that are very close to the optimal/best ones.

The results of LocBra comparisons against the competitive heuristics and the exact method are published in Computers & Operations Research journal:
Darwiche, M., Conte, D., Raveaux, R., & T'Kindt, V. (2018). A local branching heuristic for solving a graph edit distance problem. *Computers & Operations Research.*

4.4.6.3 Accuracy of LocBra from an application point of view

To evaluate LocBra from an application point of view, it should be put in tests involving real-life cases of using graphs comparison. Two tests are conducted: the first one is in similarity search context, and the second one is about the correlation with ground-truth.

Similarity search application
An important question is brought up: what is the impact of GED heuristics on the (dis)similarity search? This experiment intends to answer this question by evaluating the ranking of graphs, which is considered as an important task in graph retrieval.

Methods. The heuristics involved in this experiments are:

i- *LocBra.*

ii- *SBPBeam-α*, the BeamSearch heuristic with α the beam size.

iii- *IPFP-it*, the IPFP heuristic with *it* the maximum number of iterations.

iv- *GNCCP-d*, the GNCCP heuristic with d the quantity to be subtracted from the ζ variable at each iteration. ζ is the variable that controls the concavity and convexity of the objective function of the QAP model.

LocBra has shown superiority over *CPLEX-t* and *CPLEX_LocBra-t*, therefore those heuristics are not considered in this experiment. *BeamSearch-α* is also dropped out, because it has shown poor results compared to the rest of the heuristics. Note, that only extended versions of the methods are considered in this experiment. The reason is because *LocBra* was more accurate in comparison to both original and extended versions of the heuristics, in the previous experiments. In addition, the extended versions of the heuristics performed better than the original versions.

Comparison indicators The current experiment is about evaluating the ranking of the graphs based on the distances computed by each heuristic. For a target graph, the distance is computed against the rest of the graphs in the database. Then, ordering the obtained solutions by ascending order (of the computed distance) will show first the graphs with high resemblance w.r.t. the target one. Consequently, the experiment will compare the orders provided by the heuristics and the order provided by an exact method. It proceeds as follows: assuming a graph database with 5 graphs $D_G = [g_1, g_2, ..., g_5]$. Starting with graph g_1, the optimal and heuristics solutions (distances) are computed for all possible pairs of graphs e.g. $(g_1, g_1); (g_1, g_2); ...(g_1, g_5)$. Then, graphs are ordered by ascending order based on the distances. For instance, assuming that the optimal order for g_1 is $g_1^{opt} = [g_1, g_2, g_4, g_3, g_5]$, and a heuristic H order is $g_1^H = [g_1, g_5, g_4, g_3, g_2]$. Then, the metric used is Kendall rank correlation coefficient τ_b: it is a statistic used to study the correlation between two ranked/sorted ordinal variables (Agresti, 2010). Computing τ_b consists in measuring the degree of concordance between the two ranked variables. An ordinal variable is a categorical variable in which the possible values are ordered. The correlation τ_b is computed between g_1^{opt} and g_1^H. Then, to analyze τ_b values, a *p-value* test is applied, which is a statistical test based on the *null hypothesis* that assumes two variables (vectors) are uncorrelated and $\tau_b = 0$. The *alternative hypothesis* is that the variables are correlated, and τ_b is non-zero. P-value represents the probability of obtaining results similar or better to what was observed. If the p-value is less than a threshold (referred to as significance level), then the null hypothesis is rejected. Otherwise, the null hypothesis cannot be rejected (but it does not accept the alternative hypothesis in all cases). τ_b and p-value are computed for the rest of the heuristics after ordering the solutions for all pairs of graphs. So far, this is done for g_1, the same process is repeated for the rest of the graphs in the database. Finally, the p-values obtained for each graph and for each heuristic are grouped by heuristics, and the average of the values smaller than the threshold are calculated for each heuristic. The heuristic with the highest percentage, means that the null hypothesis is rejected and eventually the solutions returned by it, are strongly correlated with the optimal ones.

The MUTA subset with graph size 30 is picked in this experiment, because all optimal solutions are known for these instances (100 instances) and 30 is the average graph sizes. All PAH instances (8836) are selected to be part of this experiment as well.

Table 4.13: Average p-value for each heuristic on PAH instances

	Average p-value
LocBra	100
SBPBeam	14
IPFP	21
GNCCP	23

Table 4.14: Average p-value for each heuristic on MUTA-30 instances

	Average p-value
LocBra	100
SBPBeam	50
IPFP	50
GNCCP	70

Evaluations on PAH database. The parameters values are set to:

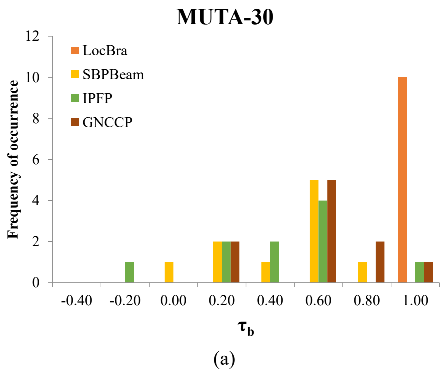
LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 12.25s$, $node\_time\_limit = UB\_time\_limit = 1.75s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
SBPBeam-α	$\alpha = 140$
IPFP-it	$it = 2000$
GNCCP-d	$d = 0.09$

Table 4.13 presents the p-values computed on PAH instances. *LocBra* is at 100%, so the null hypothesis is always rejected for all instances. *LocBra* has a very strong correlation with the optimal ranking. The other heuristics have lower percentages and are far from *LocBra*, the highest (23%) being obtained by *GNCCP-0.09*. In Fig. 4.3; chart(b) shows τ_b distribution for PAH instances. *LocBra* has correlation values between $[0.6, 1]$ and all the other heuristics are below those values. This proves that the ranking obtained by *LocBra* is very similar to the optimal ranking. In the second place comes *GNCCP-0.09*, followed by *IPFP-2000* and then *SBPBeam-140* at last.

Evaluation on MUTA database. The parameters of each heuristic are the same as before:

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 900s$, $node\_time\_limit = UB\_time\_limit = 180s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
SBPBeam-α	$\alpha = 400$
IPFP-it	$it = 20000$
GNCCP-d	$d = 0.03$

Similar results as on PAH instances are noted in Table 4.14 for MUTA-30 instances. The average p-value is 100% for *LocBra*. Hence, there is a strong correlation between the ranking of *LocBra* and the optimal ranking. Moreover, *GNCCP-0.03* has scored 70%,



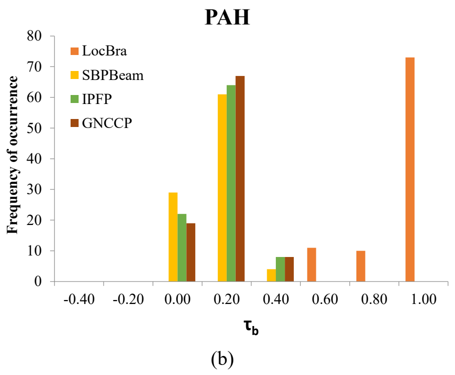


Figure 4.3: Histograms showing τ_b distribution for each heuristic for MUTA-30 (a) and PAH (b)

higher than *SBPBeam-400* and *IPFP-20000* (both 50%). *GNCCP-0.03* should reject the null hypothesis in 70% of the cases. Regarding the correlation distribution shown in Fig. 4.3; chart (a), all the values obtained by *LocBra* are uniformly in bin 1. This means that *LocBra* ranking is perfectly correlated with the optimal ranking. *GNCCP-0.03* comes in the second place but the correlation values are distributed in a wide range between $[0.2, 1]$. *IPFP-20000* shows poor correlation with the optimal and has negative value (-0.2) for one instance.

Conclusion. These experiments have proved indeed that *LocBra* ranking is strongly correlated with the optimal ranking. Therefore, *LocBra* is suitable for GED applications, especially in the context of similarity search and graph retrieval. In other words, the use of inaccurate heuristics may lead to really wrong results in terms of nearest neighbors of a graph query.

Correlation with the ground-truth

The main question to be answered by this experiment is: does the best minimizer guarantees finding the ground-truth matching? As discussed earlier, there are certain databases that come with ground-truth matchings, which are given by human experts. They represent the true matching expected to be obtained between each pair of graphs. Therefore, this experiment consists in studying the closeness of the matchings computed by the heuristics to the ground-truth matchings.

Methods. The same as in the above experiment:

i- *LocBra*.

ii- *SBPBeam-α*, the SBPBeam heuristic with α the beam size.

iii- *IPFP-it*, the IPFP heuristic with it the maximum number of iterations.

iv- *GNCCP-d*, the GNCCP heuristic with d the quantity to be subtracted from the ζ variable at each iteration. ζ is the variable that controls the concavity and convexity of the objective function of the QAP model.

v- *CPLEX*-∞, to compute the optimal solutions.

As earlier, only extended versions are considered. Note that *CPLEX*-∞ is included in order to compute the optimal solutions. This enables evaluating the optimal and the ground-truth solutions and detect if they are conformed. This proves the relevance of the edit operations cost values defined for the databases.

Comparison indicators. The indicator used here is the *Hamming Distance* between the ground-truth and heuristics matchings. A matching is represented through a binary matrix, in which each value refers to an assignment of two vertices, e.g. $[x_{ik}]$ is a matching matrix, where i and k are the indexes of two vertices $u_i \in V \cup \{\epsilon\}$ and $v_k \in V' \cup \{\epsilon\}$. The Hamming distance simply counts the number of positions at which the corresponding values are different. For two matching matrices $[x_{ik}]$ and $[\hat{x}_{ik}]$, with $i \in \{1, 2, ..., N\}$, $k \in \{1, 2, ..., M\}$ and $N = |V| + 1$, $M = |V'| + 1$, the Hamming distance $HD(x, \hat{x})$ is defined by:

$$\mathcal{HD}(x, \hat{x}) = \sum_{i=1}^{N} \sum_{k=1}^{M} (1 - \delta(x_{ik}, \hat{x}_{ik})), \tag{4.18}$$

where $\delta : \mathbb{R}^2 \rightarrow \{0, 1\}$ is the Kronecker delta function, which returns 1 when $x_{ik} = \hat{x}_{ik}$ and 0 otherwise. Note that the value obtained is normalized by dividing by $(N + M)$, in order to return representative quantities between $[0, 1]$. This normalized distance is denoted by $\widehat{\mathcal{HD}}$. Therefore, what is of interest is when $\widehat{\mathcal{HD}} = 0$, because it means that the heuristic matchings is equivalent to the ground-truth matchings.

Evaluation on HOUSE-NA database. For this database, the settings of the heuristics are as follows:

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 10s$, $node\_time\_limit = 2s$, $UB\_time\_limit = 4s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
SBPBeam-α	$\alpha = 8$
IPFP-it	$it = 400$
GNCCP-d	$d = 0.09$

HOUSE-NA (also HOUSE-A) database contains graphs modeling rotated houses inside images with different angles. In this experiment, the graphs are grouped by rotation angles, to show more consistent and representative results, which are reported in Fig. 4.4-(a). Before analyzing heuristics behaviors, the *optimal* matching is considered in order to confirm the relation and the closeness between the ground-truth and the *optimal* matchings. A strong relation is noted when looking at the *optimal* line, since it is close to 0 for all rotation angles. Next, the heuristic with the smallest values is *LocBra* for all the rotation angles. The line is constantly close to 0 and almost linear, which means that *LocBra* has computed matchings very close to the ground-truth ones. *GNCCP-0.09* and *IPFP-400* are at the second and third positions after *LocBra*, except for rotated images at 90° where *IPFP-400* outperforms *GNCCP-0.09* and the $\widehat{\mathcal{HD}}_{avg}$ drops to 0.2. *SBPBeam-8* comes last with the highest $\widehat{\mathcal{HD}}_{avg}$ for all rotation angles. Regarding, the average CPU time charts,

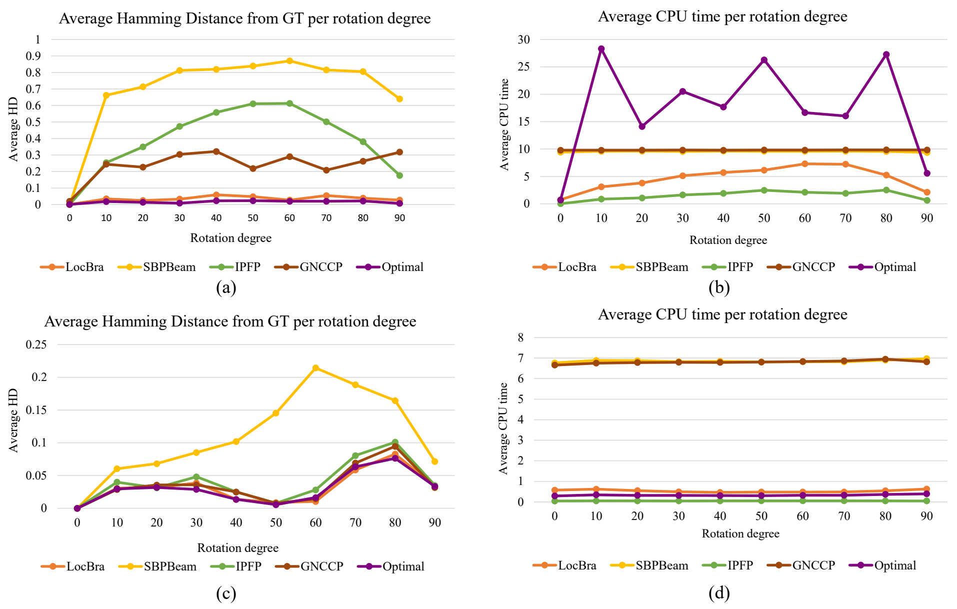


Figure 4.4: Hamming distance and time averages for heuristics matchings vs. ground-truth matchings. Charts (a) and (b) are the results for HOUSE-NA database, and charts (c) and (d) are the results for HOUSE-A database

as shown in Fig. 4.4-(b), *IPFP-400* and *LocBra* are the fastest, while *SBPBeam-8* and *GNCCP-0.09* are very slow and close to each other. This means that *IPFP-400* and *LocBra* are able to find solutions and converge faster than the others. For the *optimal* method, as expected, it is not the fastest because CPLEX spends more time proving the optimality of the solution found.

Evaluation on HOUSE-A database. For this database, the heuristics parameters are set to:

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 2s$, $node\_time\_limit = 0.5s$, $UB\_time\_limit = 1s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
SBPBeam-α	$\alpha = 5$
IPFP-it	$it = 10$
GNCCP-d	$d = 0.01$

The same conclusion, as for HOUSE-NA database, can be deduced when looking at Fig. 4.4-(c). However, an important remark is that the gap between *LocBra*, *IPFP-10* and *GNCCP-0.01* is reduced. Their lines are very close to each other and below 0.1 for all rotation angles. This is due to the fact that HOUSE-A instances are easier to solve, and therefore all the heuristics (except *SBPBeam-5*) are able to compute accurate matchings. Remarkably, the Shape Context features are meaningful and the objective function guides well the exploration of the solution space. Another important point is that the *optimal* method has scored always the smallest values except at 60 and 70 degrees, where it is slightly outperformed by *LocBra*: this is due to the fact that on some of the instances CPLEX was not able to find the optimal matchings. Figure 4.4-(d) shows the average running time for all methods on HOUSE-A database. As earlier, *IPFP-10* is the fastest, followed by the *optimal* and *LocBra*. An important gap is seen between *GNCCP-0.01*, *SBPBeam-5* and the rest of the methods, which are actually slow.

Conclusion. Among the set of compared heuristics, *LocBra* has shown to be very competitive and gave the best accuracy and closeness to the ground-truth matchings.

General conclusions based on the evaluations.
These two experiments have shown the efficiency of LocBra in computing good quality solutions that are very close to the optimal and the ground-truth solutions. They have shown also that LocBra is very accurate, when used as a method to compute distances between graphs and to determine a ranking. Therefore, it is suitable to be used in graph retrieval and determining nearest neighbor graphs tasks.

The results of the experiments were published in Pattern Recognition Letters journal: Darwiche, M., Conte, D., Raveaux, R., & T'Kindt, V. (2018). Graph Edit Distance: Accuracy of Local Branching from an application point of view. *Pattern Recognition Letters*.

4.5 Summary and contributions

This chapter is dedicated to the GED^{EnA} problem, which is defined and explained in Section 4.1.1. It shed the light and emphasizes on the importance of distinguishing between the general GED problem and the sub-problem GED^{EnA}. The main reason is because a method (exact or heuristic) that solve the sub-problem cannot be applied to the GED problem. This clearly appears in the literature, and especially when reviewing JH formulation, which is an exact model of the sub-problem and not the general one. In addition, it shows the influence of the cost functions on graph databases. If edges operations cost functions do not consider attributes on edges, then the graph database is relevant to both the general GED and the GED^{EnA} problems. Whilst, the graph database is only applicable to the GED problem if edges attributes are considered. Making this distinction enables deriving more effective algorithms to solve the problem.

First, a comparison is conducted on three MILP formulations that exist in the literature and can solve the GED^{EnA} problem. It shows that JH formulation, which is dedicated to solve the GED^{EnA} problem, is the best one in computing optimal solutions. These positive results have led to choosing JH formulation in the implementation of local branching heuristic to solve the GED^{EnA} problem. This heuristic, denoted shortly by LocBra, is a first attempt of bringing a matheuristic from OR field and apply it the GED problem. LocBra's main idea is to perform a series of local searches and focuses the search in defined regions looking for good quality solutions of a MILP formulation. It combines several heuristic techniques (neighborhood definition, intensification and diversification) in a defined branching scheme by solving small sub-problems using a MILP black-box solver. A dedicated version of local branching is designed to solve the GED^{EnA} problem. The key points of LocBra are:

- Neighborhood definition: it is done by considering only variables modeling vertices matching, which leads to a better decomposition of the problem.

- Problem-dependent diversification: to improve the diversification, which is considered as an important step to escape local minima, LocBra adapts a special diversification based on analyses done over the data of the instance at hand. It determines the important variables that guarantee very good diversification. It was shown that this diversification is more efficient than the original one.

- LocBra is very flexible and its performance can be controlled by a set of input parameters.

Next, this new heuristic is intensively evaluated on reference databases against the best heuristics available in the literature. The experiments try to capture different points of view on GM field. They are categorized as follows:

Experiment type	Main comparison indicator	Application
Distance minimization	Deviation	Near-optimal quality
Ranking	Kendall correlation	Similarity search/graph retrieval
Ground-truth matching	Hamming Distance	Result interpretation

The results of all the experiments have shown that LocBra is a very effective heuristic. It is capable of computing better solutions than the other heuristics. Also, those solutions are pretty close to the optimal solutions. From an application point of view, the two experiments, based on ranking and ground-truth correlations, have shown very high correlation with the optimal and the ground-truth solutions. This, in turn, proves that LocBra is very suitable to be applied in GED applications to perform full (sub-)graph and similarity searches. LocBra is efficient when dealing with complex graphs where neighborhoods and attributes do not allow to easily differentiate between vertices. Therefore, it is suitable for chemical graphs and graphs extracted from images. However, it cannot be generalized unconditionally on all graph types, since some exceptions or untested scenarios may be encountered in cases where graphs are a bit different (very sparse, unconnected vertices, ...). In addition, the results obtained on HOUSE-NA and HOUSE-A, have confirmed the importance of having good and representative cost functions in helping the algorithm spotting the dissimilarities faster. The CPU time was much more lower for HOUSE-A instances, but very expensive for HOUSE-NA.

It is important to note that the proposed LocBra heuristic is a significant contribution to the Pattern Recognition research field: this heuristic solves very efficiently some classes of Graph Matching problems, which could improve the solution of other application domains that use such problems like graph classification and clustering.

Additionally to the results reported in this chapter, more results and videos are published on a dedicated website [1].

The works presented in this chapter were communicated to the following conferences and scientific journals:

[1]https://sites.google.com/site/gedlocbra/

- Darwiche, M., Conte, D., Raveaux, R., & T'Kindt, V. (2017, February). Evaluation de modèles mathématiques pour le problème de la distance d'édition entre graphes. In *ROADEF2017*.
- Darwiche, M., Conte, D., Raveaux, R., & T'Kindt, V. (2017, July). The Graph Edit Distance Problem treated by the Local Branching Heuristic. In *MIC17 12th Metaheuristics International Conference*.
- Darwiche, M., Raveaux, R., Conte, D., & T'Kindt, V. (2017, November). A Local Branching Heuristic for the Graph Edit Distance Problem. In *Iberoamerican Congress on Pattern Recognition* (pp. 194-202). Springer, Cham.
- Darwiche, M., Conte, D., Raveaux, R., & T'Kindt, V. (2018). A local branching heuristic for solving a graph edit distance problem. *Computers & Operations Research*.
- Darwiche, M., Conte, D., Raveaux, R., & T'Kindt, V. (2018). Graph Edit Distance: Accuracy of Local Branching from an application point of view. *Pattern Recognition Letters*.
- Darwiche, M., Raveaux, R., Conte, D., & T'kindt, V. (2018, June). Solving a special case of the Graph Edit Distance Problem with Local Branching. In *Matheuristics 2018*.

LocBra has achieved potentially good results in solving the GED^{EnA} problem, therefore it will be interesting to apply it to the general GED problem but at the cost of changing JH formulation. In the next chapter, the focus will be on developing a new MILP formulation, then applying LocBra over it, and investigate other matheuristics.

Chapter 5

MILP formulations and matheuristics to solve the GED problem

Contents

5.1 Introduction

This chapter is dedicated for the GED problem, which is the main objective of this thesis. All kinds of graph databases and cost functions are accepted, even if there are attributes on edges. This chapter presents propositions of exact and heuristic methods, based on mathematical programming techniques, to solve the GED problem. The methods are organized by type and each one is evaluated against the appropriate methods existing in the literature.

5.2 Proposed MILP formulations

Multiple MILP formulations have been designed for solving the GED problem. The formulations are developed based on theoretical analysis of the problem's properties, and by adapting known modeling techniques in OR field. Each formulation is presented, followed by experimentation results to study its performance and effectiveness in computing good solutions and improving the existing methods.

5.2.1 Vertex-based Model (VbM)

VbM is a formulation inspired by existing ones, in particular F1 and F2 formulations explained in Section 2.3.7. Its fundamental idea is based on Property 1 (Page 44) , which states that edges matching are very dependent from vertices matching. Also, it is known that binary decision variables, in most cases, are the ones responsible of increasing the complexity of a MILP formulation. So, VbM follows the same way to model vertices and edges matchings as in F1 and F2 formulations, by introducing $x_{i,k}$ and $y_{ij,kl}$ variables. However, only $x_{i,k}$ variables are binary, and $y_{ij,kl}$ variables are continuous. In addition, $y_{ij,kl}$ variables will store the costs of edges operations. The belief is that by doing so, it may reduce the complexity of the formulation. But, this requires rewriting the constraints to satisfy the GED definition.

5.2.1.1 VbM formulation

This formulation works on graphs $\overline{G}$ and $\overline{G}'$, so deletions are allowed in both graphs and assigning a vertex to ϵ means that vertex is deleted. For more details, please see Section 4.1.2.

Data. The cost functions are assumed to be given, therefore $[c_v]$ and $[c_e]$ are computed as in equations 2.7 and 2.8.

Variables. One set of binary variables and another set of continuous variables are needed:

- $x_{i,k} \in \{0,1\}$, $\forall i \in \overline{V}, \forall k \in \overline{V}'$: $x_{i,k} = 1$ when vertices i and k are matched, and 0 otherwise.
- $y_{ij,kl} \in \mathbb{R}$, $\forall (i,j) \in \overline{E}, \forall (k,l) \in \overline{E}'$: $y_{ij,kl} = c_e(ij,kl)$ when edge (i,j) is matched with (k,l), and 0 otherwise. The variable, when selected, holds the cost of the underlying edge operation. To be more precise, every $y_{ij,kl} \in \{0, c_e(ij,kl)\}$.

Objective function. The objective function to be minimized is the following:

$$\min_{x,y} \sum_{i\in\overline{V}} \sum_{k\in\overline{V}'} c_v(i,k) \cdot x_{i,k} + \sum_{(i,j)\in\overline{E}} \sum_{(k,l)\in\overline{E}'} y_{ij,kl} \tag{5.1}$$

The objective function minimizes the cost of vertices (resp. edges) matching.

Constraints. First, all constraints are given and then, the role of each constraint is given next.

$$\sum_{k\in\overline{V}'} x_{i,k} = 1, \ \forall i \in V \tag{5.2}$$

$$\sum_{i\in\overline{V}} x_{i,k} = 1, \ \forall k \in V' \tag{5.3}$$

$$y_{ij,kl} \geq c_e(ij,kl) \cdot (x_{i,k} + x_{j,l} - 1),\ \forall (i,j) \in E, \forall (k,l) \in E' \tag{5.4}$$

$$y_{ij,kl} \geq c_e(ij,kl) \cdot (x_{i,l} + x_{j,k} - 1),\ \forall (i,j) \in E, \forall (k,l) \in E' \tag{5.5}$$

$$y_{ij,\epsilon\epsilon} \geq c_e(ij,\epsilon\epsilon) \cdot (x_{i,k} + x_{j,l} - 1),\ \forall (i,j) \in E, \forall (k,l) \notin E' \tag{5.6}$$

$$y_{\epsilon\epsilon,kl} \geq c_e(\epsilon\epsilon,kl) \cdot (x_{i,k} + x_{j,l} - 1),\ \forall (i,j) \notin E, \forall (k,l) \in E' \tag{5.7}$$

$$y_{ij,\epsilon\epsilon} \geq c_e(ij,\epsilon\epsilon) \cdot x_{i,\epsilon},\ \forall (i,j) \in \overline{E} \tag{5.8}$$

$$y_{ij,\epsilon\epsilon} \geq c_e(ij,\epsilon\epsilon) \cdot x_{j,\epsilon},\ \forall (i,j) \in \overline{E} \tag{5.9}$$

$$y_{\epsilon\epsilon,kl} \geq c_e(\epsilon\epsilon,kl) \cdot x_{\epsilon,k},\ \forall (k,l) \in \overline{E}' \tag{5.10}$$

$$y_{\epsilon\epsilon,kl} \geq c_e(\epsilon\epsilon,kl) \cdot x_{\epsilon,l},\ \forall (k,l) \in \overline{E}' \tag{5.11}$$

$$y_{ij,kl} \geq 0,\ \forall (i,j) \in \overline{E}, \forall (k,l) \in \overline{E}' \tag{5.12}$$

Constraints 5.2 (resp. 5.3) ensures that a vertex $i \in V$ (resp. $k \in V'$) can be matched with one vertex in $\overline{V}$ (resp. $\overline{V}'$). ϵ is in both $\overline{V}$ and $\overline{V}'$, and a vertex can be matched with it only one time. Constrains 5.4 and 5.5 make sure that when matching two couple of vertices (e.g. $i \rightarrow k$ and $j \rightarrow l$ OR $i \rightarrow l$ and $j \rightarrow k$), the two edges, if they exist, (i,j) and (k,l) have to be matched. If the matching of the two couples of vertices is selected, and one of the edges does not exist, then the existing edge must be deleted. This case is handled by constraints 5.6 and 5.7. Then, there is the case where a vertex i is chosen to be deleted, then all edges connected to i have to be deleted (by GED definition). Constraints 5.8, 5.9, 5.10 and 5.11 take care of this case for both graphs. Finally, constraints 5.12 forces the $y_{ij,kl}$ variables to be positive.

This formulation works perfectly with undirected graphs and it copes with the symmetry case, i.e. $(i,j) = (j,i)$. It can be applied on directed graphs, as well, by removing constraints 5.5 and 5.7.

The number of variables in this formulation is: $(|\overline{V}| \cdot |\overline{V}'|)$ binary variables, plus $(|\overline{E}| \cdot |\overline{E}'|)$ continuous variables. And the number of constraints is: $(|V| + |V'| + 2|E| \cdot |E'| + |E| \cdot |\widehat{E}'| + |\widehat{E}| \cdot |E'| + 2|\overline{E}| + 2|\overline{E}'| + |\overline{E}| \cdot |\overline{E}'|)$, with $\widehat{E}$ (resp. $\widehat{E}'$) the complement set of edges of E (resp. E').

5.2.1.2 Evaluation of VbM formulation

VbM is a formulation designed to solve the GED problem, so it will be evaluated w.r.t. the best known formulations in the same context, which are F1 and F2. Note that, JH formulation has performed the best among the existing ones, but it only solves the GED^{EnA} problem. Therefore it is not included in the experiments.

Methods. The two formulations F1 and F2 are involved in this experiments. The details of these formulations are presented in Section 2.3.7.

Instances and experimentation settings. MUTA graph database is selected in this experiment. The details of this database can be found in Section 4.2. 7 subsets (10 to 70) are considered, resulting in a total of 700 instances. All formulations are implemented in C language. The solver CPLEX 12.6.0, in single thread mode, is used to solve the formulations. A maximum running time limit, of 900 seconds per instance, is imposed on CPLEX. Experiments are ran on a machine with Windows 7 $x64$, Intel Xeon $E5$ 2.30 GHz, 4 cores and 8 GB of RAM.

Evaluation indicators. The following indicators are computed for each subset:

- t_{min}: the minimum CPU time in seconds,
- t_{avg}: the average CPU time in seconds,
- t_{max}: the maximum CPU time in seconds,
- d_{min}: the minimum deviation, in percentage, between the solutions obtained by a formulation against the best solutions found by the three formulations,
- d_{avg}: the average deviation, in percentage, between the solutions obtained by a formulation against the best solutions found by the three formulations,
- d_{max}: the maximum deviation, in percentage, between the solutions obtained by a formulation against the best solutions found by the three formulations,
- η: the number of optimal solutions computed,
- η': the number of solutions (whether optimal or not) computed by a formulation, which are the best/minimum among those computed by all formulations.

The deviations are computed based on Eq. 4.11.

Evaluation analysis. The results of the experiment are shown in Table 5.1. In terms of average deviation (d_{avg}), F2 has scored the lowest values (always $\leq 0.55\%$) for all subsets. F1 comes in the second place, and it was able to solve all instances in subsets 10 and 20, with $d_{avg} = 0\%$. As the size of the graph increases, the difference starts to grow, from 2.80% on subset 30 to reach 12% on subset 60. VbM formulation comes in the last position, with average deviations very high compared to F1 deviations, except for subset 10 where the $d_{avg} = 0\%$. The average deviation for VbM reaches 87.20% on subset 70, which means that solutions computed by VbM are very far from the best solutions obtained by F2. Moreover, F2 has computed the highest number of optimal solutions (η) for all subsets, except for subsets 10, 20 and 70 where η values of F2 are equal to η values of F1. F1 formulation has computed better solutions than VbM for subset 20 with 100 instances against 25, and subset 30 with 22 instances against 10. Looking at η' values, again F2 has scored the highest values for all subsets, with values always between 90 and 100. Regarding the average CPU time (t_{avg}), F2 was the fastest among the others, but the gap gets smaller as the graph sizes increase, where the instances become harder and all formulations require high computational time.

Table 5.1: Comparison of VbM, F1 and F2 formulations

	S	10	20	30	40	50	60	70
VbM	t_{min}	0.02	0.08	0.28	0.58	1.25	2.22	3.67
	t_{avg}	0.22	694.04	621.06	763.68	810.78	799.82	813.99
	t_{max}	0.72	901.55	900.60	900.88	901.52	941.70	904.59
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	0.52	22.29	42.43	59.77	76.04	87.20
	d_{max}	0.00	12.50	113.79	487.14	142.86	193.22	655.81
	η	**100**	25	10	10	10	10	**10**
	η'	**100**	91	12	10	10	10	10
F1	t_{min}	0.01	0.02	0.03	0.05	0.06	0.08	0.11
	t_{avg}	0.18	26.73	741.12	786.99	810.08	810.11	810.14
	t_{max}	0.90	486.18	900.23	900.17	900.12	900.35	901.07
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	2.88	5.05	10.71	12.62	11.01
	d_{max}	0.00	0.00	21.43	26.32	47.85	41.94	60.00
	η	**100**	**100**	22	14	10	10	**10**
	η'	**100**	**100**	54	30	18	14	22
F2	t_{min}	0.03	0.07	0.10	0.15	0.36	0.44	0.76
	t_{avg}	**0.12**	**1.17**	**367.54**	**631.04**	**792.96**	**803.84**	**809.64**
	t_{max}	0.38	7.79	900.20	900.19	900.48	900.38	900.17
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	**0.05**	**0.00**	**0.25**	**0.02**	**0.55**
	d_{max}	0.00	0.00	2.76	0.00	12.59	1.55	11.88
	η	**100**	**100**	**77**	**36**	**14**	**11**	**10**
	η'	**100**	**100**	**98**	**100**	**97**	**99**	**90**

Table 5.2: Comparison of VbM, F1 and F2 formulations - optimal solutions

	S	10	20	30	40	50	60	70
VbM	t_{min}	0.02	0.08	0.28	0.58	1.25	2.22	3.67
	t_{avg}	0.22	114.42	0.30	0.69	1.32	2.31	3.79
	t_{max}	0.72	749.55	0.32	0.74	1.38	2.44	3.99
	η	100	25	10	10	10	10	10
F1	t_{min}	0.01	0.02	0.03	0.05	0.06	0.08	0.11
	t_{avg}	0.18	0.81	**0.04**	**0.05**	**0.07**	**0.09**	**0.11**
	t_{max}	0.90	2.58	0.05	0.05	0.08	0.10	0.12
	η	100	25	10	10	10	10	10
F2	t_{min}	0.03	0.07	0.10	0.15	0.36	0.44	0.76
	t_{avg}	**0.12**	**0.28**	0.26	0.30	0.49	1.20	1.35
	t_{max}	0.38	0.94	0.64	0.58	0.81	2.72	1.99
	η	100	25	10	10	10	10	10

To get a better idea about the running time of each formulation, the evaluation indicators are recomputed for instances where optimal solutions were found by all formulations. The results are reported in Table 5.2. For subsets 10 and 20, F2 formulation is faster than the others. However, F1 becomes faster on average for the rest of the subsets. And, VbM is the slowest comparing to F1 and F2 formulations.

Conclusion. Based on the results obtained, F2 formulation is the best in solving MUTA instances to optimality. The proposed formulation did not succeed in performing better than the existing ones.

An additional experiment is done to compare the three formulations on the same database, with the pre-processing procedure. The results of this experiment are reported in Appendix B, Section B.1. The results have shown that F2 is the best formulation compared to F1 and VbM. F2 was able to compute better solutions than the other formulations. This is the same conclusion as in the experiment without pre-processing. So, pre-processing procedure did not help improving the performances of the formulations.

5.2.1.3 General conclusions

Despite the theoretical analysis when designing VbM formulation, by reducing the number of binary variables, VbM did not succeed in performing better than the existing formulations. Reducing the number of binary variables have resulted in increasing the number of constraints, which clearly made the formulation more complex and hard to solve. Therefore, VbM is less effective than the existing formulations. Yet, it is an interesting one, because it has investigated the idea of reducing the number of binary variables and focusing on vertices assignment variables. In addition, it can be added to the list of exact methods for solving the GED problem.

5.2.2 Object-based Model (ObM)

The main idea of this formulation is based on the notion of objects, where an object represents the matching of some vertices and edges. On the contrary to other formulations where vertices and edges matching are expressed separately (as such, one variable to model the matching of two vertices and another variable to model the matching of two edges). ObM formulation is an attempt to create high level and complex objects that model multiple matchings of vertices and edges at once. An object o_p is, then, defined by 4-tuples (u_i, u_j, v_k, v_l) with $u_i, u_j \in \overline{V}$ and $v_k, v_l \in \overline{V}'$. The object underlies three edit operations on vertices and edges depending on the selected vertices. For an object $o_p = (u_i, u_j, v_k, v_l)$, the operations are as follows.

- First edit operation is one of the following:
$$\begin{cases} \text{Substitution: } u_i \rightarrow v_k \text{ if } u_i \in V \text{ and } v_k \in V' \\ \text{Deletion from G: } u_i \rightarrow \epsilon \text{ if } u_i \in V \text{ and } v_k = \epsilon \\ \text{Deletion from G': } \epsilon \rightarrow v_k \text{ if } u_i = \epsilon \text{ and } v_k \in V' \end{cases}$$

- Second edit operation is one of the following:

$$\begin{cases} \text{Substitution: } u_j \rightarrow v_l \text{ if } u_j \in V \text{ and } v_l \in V' \\ \text{Deletion from G: } u_j \rightarrow \epsilon \text{ if } u_j \in V \text{ and } v_l = \epsilon \\ \text{Deletion from G': } \epsilon \rightarrow v_l \text{ if } u_j = \epsilon \text{ and } v_l \in V' \end{cases}$$

- Third edit operation is one of the following:
$$\begin{cases} \text{Substitution: } (u_i, u_j) \rightarrow (v_k, v_l) \text{ if } (u_i, u_j) \in E \text{ and } (v_k, v_l) \in E' \\ \text{Deletion from G: } (u_i, u_j) \rightarrow \epsilon \text{ if } (u_i, u_j) \in E \text{ and } (v_k, v_l) \notin E' \\ \text{Deletion from G': } \epsilon \rightarrow (v_k, v_l) \text{ if } (u_i, u_j) \notin E \text{ and } (v_k, v_l) \in E' \end{cases}$$

If $u_i = \epsilon$, then v_k is deleted from G', and if the edge $(v_k, v_l) \in E'$, it is removed from G' as well. The presence of ϵ in an object indicates deletion operations of corresponding vertices and the incident edge (if it exists).

The following list shows few examples of objects and their underlying vertices and edges matchings. These objects are constructed based on the graphs in Fig.4.1.

- $o_1 = (1, 2, a, b)$, the operations are: substitution $1 \rightarrow a$, substitution $2 \rightarrow b$ and substitution $(1, 2) \rightarrow (a, b)$
- $o_2 = (1, 4, c, \epsilon)$, the operations are: substitution $1 \rightarrow c$, deletion $4 \rightarrow \epsilon$ and deletion $(1, 4) \rightarrow \epsilon$
- $o_3 = (3, 4, a, c)$, the operations are: substitution $3 \rightarrow a$, substitution $4 \rightarrow c$ and deletion $\epsilon \rightarrow (a, c)$ since $(3, 4) \notin \overline{E}$

Unlike F2 which is considered as a compact formulation, ObM is an expanded one. Every object implies multiple matchings represented by selecting four vertices at a time. Generating the objects will, then, requires multiplying the input data, in particular the sets of vertices $\overline{V}$ and $\overline{V'}$. This leads to increasing the number of variables in the formulation and hopefully making the problem easier to solve. This kind of expanded formulations has been tested on other optimization problems such as scheduling problems with time-indexed formulations, where good results are achieved.

5.2.2.1 Objects generation

The trivial case is to create all objects by considering all permutations of $u_i, u_j \in \overline{V}$ and $v_k, v_l \in \overline{V}'$, which results in $(\overline{V} \cdot \overline{V}')^2$ objects. In fact, this will introduce a lot of redundancies and increase the number of binary variables in the formulation and thus its complexity. Therefore, objects generation is considered as a very important step, and only necessary objects are generated, covering all possible matchings that could occur between two graphs. The procedure to generate the objects goes as follows.

1. Let L_G be the list of vertices sorted by increasing order after assigning integer *id*s to each vertex in V and placing ϵ at the end of the list.
2. Let $L_{G'}$ be the list of vertices sorted by increasing order after assigning integer *id*s to each vertex in V' and placing ϵ at the end of the list.

3. Let $\mathcal{O}$ be the list of objects:

$$o_p = (u_i, u_j, v_k, v_l) \begin{cases} \forall u_i, u_j \in L_G \text{ such that } i < j \text{ and } (i \neq j \text{ or } u_i = u_j = \epsilon) \\ \forall v_k, v_l \in L_{G'} \text{ such that } k \neq l \text{ or } v_k = v_l = \epsilon \end{cases}$$

The number of objects in $\mathcal{O}$ is: $|\mathcal{O}| = \left(\frac{|\overline{V}|.(|\overline{V}|-1)}{2} + 1\right) \times \left(|\overline{V}'|.|\overline{V}'| - V'\right)$. Algorithm 8 gives the details of the objects generation procedure.

Algorithm 8: Algorithm for generating objects

```
Let $L_G$ the order set of vertices $V$
$L_G := L_G \cup \{\epsilon\}$
Let $L_{G'}$ the order set of vertices $V'$
$L_{G'} := L_{G'} \cup \{\epsilon\}$
$\mathcal{O} = \{\}$
Function GenerateObjects()
    for $i \leftarrow 1$ to $|\overline{V}|$ by $1$ do
        for $j \leftarrow i+1$ to $|\overline{V}|$ by $1$ do
            for $k \leftarrow 1$ to $|\overline{V}|'$ by $1$ do
                for $l \leftarrow 1$ to $|\overline{V}|'$ by $1$ do
                    if  $k = l$ and $v_k \neq \epsilon$ and $v_k \neq \epsilon$ then
                        continue
                    end
                    $o_p = (u_i, u_j, v_k, v_l)$
                    $\mathcal{O} = \mathcal{O} \cup \{o_p\}$
                end
            end
        end
    end
    /* Next add objects where $u_i$ and $u_j$ are $\epsilon$        */
    $u_i := u_j := \epsilon$
    for $k \leftarrow 1$ to $|\overline{V}|'$ by $1$ do
        for $l \leftarrow 1$ to $|\overline{V}|'$ by $1$ do
            if  $k = l$ and $v_k \neq \epsilon$ and $v_k \neq \epsilon$ then
                continue
            end
            $o_p = (u_i, u_j, v_k, v_l)$
            $\mathcal{O} = \mathcal{O} \cup \{o_p\}$
        end
    end
End
```

Only attributed and undirected graphs are considered so far. The procedure can be replicated and altered so it can deal with the directed case.

5.2.2.2 ObM formulation

Given two undirected graphs $\overline{G}$ and $\overline{G}'$, ObM formulation is as follows.

Data. Besides the costs matrices $[c_v]$ and $[c_e]$ (computed as in equations 2.7 and 2.8), the following sets are needed:

- Let $\mathcal{O}$ be the set of all generated objects of the form (u_i, u_j, v_k, v_l).
- The cost associated to objects $c_o(p)$: $\forall o_p = (u_i, u_j, v_k, v_l) \in \mathcal{O}$ is equal $c_e(ij, kl)$.
- Let $\mathcal{L}_i = \{o_p = (u_i, -, -, -) \text{ } or \text{ } (-, u_i, -, -), u_i \in V\}$ be the set of all objects o_p where vertex u_i appears.
- Let $\mathcal{L}'_k = \{o_p = (-, -, v_k, -) \text{ } or \text{ } (-, -, -, v_k), v_k \in V'\}$ be the set of all objects o_p where vertex v_k appears.
- The sets intersection $\mathcal{L}_i \cap \mathcal{L}_j = \{o_p = (u_i, u_j, -, -) \text{ } or \text{ } (u_j, u_i, -, -), u_i, u_j \in V\}$ returns objects where vertices u_i and u_j appear.
- The sets intersection $\mathcal{L}'_k \cap \mathcal{L}'_l = \{o_p = (-, -, v_k, v_l) \text{ } or \text{ } (-, -, v_l, v_k), v_k, v_l \in V'\}$ returns objects where vertices v_k and v_l appear.
- Let $\mathcal{L}_{i,k} = \{o_p = (u_i, -, v_k, -) \text{ } or \text{ } (-, u_i, -, v_k), u_i \in V, v_k \in V'\}$ be the set of objects where u_i is matched with v_k.
- Let $\mathcal{T}_{i,k} = \{o_p = (u_i, -, v_l, -) \text{ } or \text{ } (-, u_i, -, v_l), u_i \in V, v_k \in V', v_l \in V', v_k \neq v_l\}$ be the set of objects where u_i is not matched with v_k but matched instead with v_l, such that $v_k \neq v_l$.
- Let $\mathcal{T}'_{i,k} = \{o_p = (u_j, -, v_k, -) \text{ } or \text{ } (-, u_j, -, v_k), u_i \in V, u_j \in V, v_k \in V', u_i \neq u_j\}$ be the set of objects where v_k is not matched with u_i but matched instead with u_j, such that $u_i \neq u_j$.

Variables. Two sets of binary variables are used:

- $x_p \in \{0, 1\}, \forall o_p = (u_i, u_j, v_k, v_l) \in \mathcal{O}$. if $x_p = 1$, then the object is selected, otherwise $x_p = 0$.
- $y_{i,k} \in \{0, 1\}, \forall u_i \in \overline{V}, \forall v_k \in \overline{V}'$. $y_{i,k}$ is set to 1 when vertex u_i is matched with vertex v_k.

Objective function. The objective function to be minimized is the following:

$$\min_{x,y} \sum_{o_p \in \mathcal{O}} c_o(p) \cdot x_p + \sum_{\forall i \in \overline{V}, \forall k \in \overline{V}'} c_v(i, k) \cdot y_{i,k} \tag{5.13}$$

x variables represent the objects in the formulation, and each object has an associated cost, which is the cost of matching the underlying edges. In order to compute the cost

Table 5.3: Number of variables and constraints in ObM formulation

Variables	x	$\left(\frac{\lvert\overline{V}\rvert\cdot(\lvert\overline{V}\rvert-1)}{2}+1\right)\times\left(\lvert\overline{V}'\rvert\cdot\lvert\overline{V}'\rvert-\lvert V'\rvert\right)$
	y	$\lvert\overline{V}\rvert\cdot\lvert\overline{V}'\rvert$
Constraints	5.14	$\lvert E\rvert$
	5.15	$\lvert E'\rvert$
	5.16	$\lvert\overline{V}\rvert\cdot\lvert\overline{V'}\rvert.\lvert\mathcal{L}_{i,k}\rvert\cdot\lvert\mathcal{T}_{i,k}\rvert$
	5.17	$\lvert\overline{V}'\rvert\cdot\lvert\overline{V}\rvert.\lvert\mathcal{L}_{i,k}\rvert\cdot\lvert\mathcal{T}'_{i,k}\rvert$
	5.18	$\lvert\overline{V}\rvert\cdot\lvert\overline{V'}\rvert$

of vertices matchings, y variables are introduced, such that their associated costs are the cost of vertices matchings. Therefore, the objective function, Eq. 5.13, minimizes both the costs induced by x and y variables. It may seem sufficient to use only x variables, but in fact y variables are needed to correct the value of the objective function and compute the right distance between the two graphs. Because of the possibility of selecting multiple objects in the final solution that share the same vertices matchings as in this example: $o_p = (1, 2, a, b)$ and $o'_p = (1, 3, a, c)$. So, including the cost of vertices matchings in the cost of the objects may lead to counting multiple times the cost of one operation. Therefore, the use of y variables is essential.

Constraints. The required constraints are:

$$\sum_{o_p \in \mathcal{L}_i \cap \mathcal{L}_j} x_p = 1,\ \forall (i,j) \in E \tag{5.14}$$

$$\sum_{o_p \in \mathcal{L}'_k \cap \mathcal{L}'_l} x_p = 1,\ \forall (k,l) \in E' \tag{5.15}$$

$$x_p + x_{p'} \leq 1,\ \forall i \in \overline{V}, \forall k \in \overline{V}', \forall o_p \in \mathcal{L}_{i,k}, \forall o'_p \in \mathcal{T}_{i,k} \tag{5.16}$$

$$x_p + x_{p'} \leq 1,\ \forall k \in \overline{V}', \forall i \in \overline{V}, \forall o_p \in \mathcal{L}_{i,k}, \forall o'_p \in \mathcal{T}'_{i,k} \tag{5.17}$$

$$y_{i,k} \geq \frac{\sum_{o_p \in \mathcal{L}_{i,k}} x_p}{|\mathcal{O}|}, \forall i \in \overline{V}, \forall k \in \overline{V}' \tag{5.18}$$

Constraints 5.14 and 5.15 ensures that each edge can be assigned to at most one edge. Next, constraints 5.16 and 5.17 make sure that every vertex is matched with no more than one vertex. Theses constraints are called *disjunctive constraints*, because they select each pair of objects that are in disjunction and forbid them from being selected at the same time in the final solution. Lastly, constraints 5.18 maintain the relation between x_p and $y_{i,k}$ variables, by forcing to one $y_{i,k}$ variables that correspond to vertices assignments appearing in a selected o_p. And, this guarantees the correctness of the objective function value and avoid counting more than once the matching cost of two vertices (if they happen to appear in multiple selected objects). Table 5.3 shows the number of variables and constraints in ObM formulation.

5.2.2.3 Evaluation of ObM formulation

First, to verify the validity of this formulation, F2 is selected as the base formulation for comparison, which also can be justified since F2 is the best existing one. To have a glance at the formulations sizes, the numbers of variables and constraints in each formulation are compared. Since it is not easy to compare based on the equations given earlier, because ObM constraints require computing the sets $\mathcal{L}$ and $\mathcal{T}$. So, the comparison is done over real instances after generating the actual variables and constraints. Two instances are selected: $I_{4,3}$ is the instance of graphs shown in Fig. 4.1 ($|G| = 4, |G'| = 3$), and $I_{10,10}$ is an instance selected from the subset 10 in MUTA database ($|G| = 10, |G'| = 10$). Table 5.4 sums up the numbers obtained for instance $I_{4,3}$. Clearly, F2 has generated way less binary variables 24 against 163 variables in ObM. The constraints in F2 are 24 against 6669 constraints in ObM. The numbers obtained for instance $I_{10,10}$ are shown in Table 5.5. The number of constraints has drastically increased and reached millions ($\approx$ 10 *millions* against 110 in F2).

Table 5.4: Number of variables and constraints in F2 and ObM for instance $I_{4,3}$

		F2	ObM
Variables	x	12	143
	y	12	20
	Total	**24**	**163**
Constraints	Total	**19**	**6669**

Table 5.5: Number of variables and constraints in F2 and ObM for instance $I_{10,10}$

		F2	ObM
Variables	x	100	6216
	y	90	100
	Total	**190**	**6316**
Constraints	Total	**110**	**9520515**

Despite the high number of constraints in ObM, which are mostly disjunctive constraints (Eq. 5.16 and 5.17), ObM might still have a chance in solving efficiently the instances. These constraints might be easy to satisfy and the solver may be able to solve the instance quickly. This is an assumption that can be validated experimentally. Therefore, the two instances $I_{4,3}$ and $I_{10,10}$ are solved by CPLEX for both formulations, and the results are reported in Table 5.6. Both formulations were able to solve the instances to optimality. However, in terms of CPU time, F2 is much faster than ObM. The difference is small for instance $I_{4,3}$, but it becomes drastically big for instance $I_{10,10}$, with $0.27s$ needed by F2 against $208s$ by ObM. This proves wrong the assumption made earlier. There a lot of disjunctive constraints and the formulation is not easy to solve at all. Nevertheless, the hand-made test instance $I_{4,3}$ and the real instance $I_{10,10}$ are notably small instances, and basically the solver is capable of solving them in a matter of milliseconds for all existing formulations (see the results in Section 4.3). Consequently, ObM formulation, as it is, is not efficient in solving GED instances, especially the hard ones.

Table 5.6: Results (CPU time in seconds and objective value) of $I_{4,3}$ and $I_{10,10}$

		CPU time	Objective value
ObM	$I_{4,3}$	0.19	31
	$I_{10,10}$	208.31	22.275
F2	$I_{4,3}$	**0.02**	31
	$I_{10,10}$	**0.27**	22.275

5.2.2.4 Improving ObM formulation

The reason behind the inefficiency of ObM formulation, is the number of generated disjunctive constraints. After carefully reviewing and analyzing the constraints and the objects, it seemed possible to aggregate multiple disjunctive constraints into one constraint, without violating the GED definition. Here is an example of such a case taken from instance $I_{4,3}$ (graphs in Fig. 4.1):

- Considering the following objects: $o_1 = (1, 2, a, b)$, $o_2 = (1, 2, a, c)$, $o_3 = (1, 2, a, \epsilon)$ and $o_4 = (2, 3, b, c)$.
- A disjunctive constraint must be created for objects o_1 and o_2, i.e. objects o_1 cannot be selected at the same time with o_2, because: on one hand, o_1 is selected which means $1 \to a$, $2 \to b$ and $(1, 2) \to (a, b)$. On the other hand, selecting o_2 means that $1 \to a$, $2 \to c$ and $(1, 2) \to (a, c)$. Obviously, vertex 2 is matched with two vertices a and c, which contradicts the GED problem definition. These two objects must be in disjunction.
- It is the same case for pairs o_1 and o_3, o_2 and o_3.
- ObM formulation will generate three disjunctive constraints for each pair: $x_1+x_2 \leq 1$, $x_1 + x_3 \leq 1$ and $x_2 + x_3 \leq 1$.
- It will remain valid to replace the three disjunctive constraints with the following constraint: $x_1 + x_2 + x_3 \leq 1$. Because only one of them can be selected in the final solution.
- This is an aggregated constraint, which contains multiple objects in disjunction.
- However, to include more objects to this aggregated form, the new object must be in disjonction with all existing objects. In this example, o_4 is in disjunction with o_2 and o_3 but not with o_1, therefore it cannot be added to the aggregated constraint.

This seems to be an important realization, which may help in reducing the number of constraints in ObM formulation. But as mentioned in the above example, generating the aggregated constraints requires vigilance, so not to violate the definition of the GED problem by preventing correct matchings. The question is then, how to generate the aggregated disjunctive constraints?

Method-1: Aggregation based on cliques in a graph. To objects are in disjonction if their matched vertices are overlapping. Then, it is possible to create a graph G_o, where vertices are the objects, and edges are the disjunction relation between two objects. The presence of an edge between two vertices means that the objects are in disjunction. And there will be a disjunctive constraint for them in ObM. One way to aggregate the constraints is to compute all maximal cliques in the graph. A maximal clique in a graph is a clique such that, all pairs of vertices in the clique are connected with only one edge (complete subgraph), and it is not possible to add an additional vertex to the clique while preserving its complete connectivity. This is exactly what is needed in order to aggregate all the disjunctive constraints. In fact, a maximal clique will contain all objects that are in disjunction between each other, and no more objects can be added to them without breaking the rule. Therefore finding all maximal cliques guarantees finding the best list of aggregate constraints in ObM formulation.

Algorithm 9: Bron & Kerbosch Algorithm for generating all maximal cliques in a graph

```
1  Let $P$ be the set of vertices of the graph
2  Let $R = \{\phi\}$
3  Let $X = \{\phi\}$
4  Let $N = \{\phi\}$
1  Function BronKerbosch($R$, $P$, $X$)
2      if $P$ and $X$ are both empty then
3          Return $R$ as a maximal clique
4      end
5      foreach vertex $v$ in $P$ do
6          Add all $v$ neighbors into $N(v)$
7          BronKerbosch($R \cup \{v\}$, $P \cap N(v)$, $X \cap N(v)$) // recursive call
8          $P = P \setminus \{v\}$
9          $X = X \cup \{v\}$
10     end
11 End
```

A famous algorithm, developed by Bron and Kerbosch (1973), can be found in the literature which computes all maximal cliques in a graph. It is a simple algorithm that runs recursively on every vertex in the graph to detect the cliques. The steps of the algorithm are given in Algorithm 9. Many improvements to the algorithm are suggested later by considering pivoting and vertices set ordering based on the degree in ascending order, to reduce the number of recursive calls (Johnston, 1976). It is argued in the literature that the algorithm tends to be time consuming because the number of cliques can grow exponentially with every new added vertex. Furthermore, there is a study regarding the worst-case running time required by the algorithm (the version with pivoting), done by Tomita et al. (2006), that gives a bound of $O(3^{n/3})$ on the running time (with n the number of vertices in the graph). So, the algorithm computes the maximal cliques in linear time relative to the number of cliques, but becomes exponential with the growth of the number of the cliques.

Back to the test instance $I_{10,10}$, it was shown in Table 5.5 that ObM has created 6316 objects. The worst-case running time is a huge number, but again it depends on the cliques number in the graph. After building the disjunction graph, which has 6316 vertices and 9520515 edges, and executing the algorithm during $5min$, it only generated 767 aggregated constraints. Those constraints were covering only 10% out of the 6316 objects. Clearly, the number of maximal cliques is very big and it is not possible to generate all of them. In the end, the 767 constraints were not enough to compute a feasible solution to the problem.

Method-2: Generate disjunctive constraints when needed. ObM formulation is created without generating disjunctive constraints Eq. 5.16 and 5.17. Then, it is solved by the solver, and the solution returned may not be feasible to the problem (a vertex could be matched with multiple vertices). By looking at the solution and detecting the wrong assignments, their vertices are selected and the idea is to add the disjunctive constraints for those vertices and re-solve the formulation. Repeating this procedure will avoid generating all disjunctive constraints from the beginning and hoping that after few iterations only needed constraints will be added to get to the optimal solution. The first found feasible solution will be an optimal one as well, and the iterations can be stopped.

Testing this method on instance $I_{10,10}$, took only $9.2s$ to find the optimal solution with only 3 iterations. The procedure had added 14140 disjunctive constraints. This is, actually, interesting and the solution time was reduced from $208s$ to $9.2s$, plus the number of constraints dropped from 9.5 *millions* to 14140. Yet, $9.2s$ running time to solve an easy instance where F2 formulation takes $0.27s$ to solve, is still relatively high.

Method-3: Hybridization of method-1 and method-2. This time, methods 1 and 2 and are put together in motion. The idea is to give few seconds to generate as much as possible aggregated constraints based on Bron & Kerbosch algorithm. Next, the formulation is solved and a first solution is computed. If it is not feasible for the GED problem, then as in method-2 the needed disjunctive constraints to prevent wrong assignments are generated and the formulation is solved. The process is repeated until a feasible solution is found, which will be the optimal as well.

This method was tested on instance $I_{10,10}$, and it spent $27s$ to obtain the optimal solution with 7 iterations and 19400 disjunctive constraints. The hybridization has obtained better results than method-1 but not method-2.

5.2.2.5 General conclusions

ObM is a formulation based on the notion of objects and multiple vertices and edges matchings at once. Theoretically, this concept sounds interesting, especially because all existing formulations are based on the notion of single vertice/edge operation per decision variable and such an idea was not investigated before. It turned out that ObM requires a very big number of disjunctive constraints to prevent matching a vertex with multiple vertices. This has led to build very big formulations, even for small instance of graphs, compared to F2 formulation. To solve this problem and to reduce the complexity of the formulation, three methods were tested to avoid generating all the disjunctive constraints.

However, none of these methods has succeeded in reducing the running time when solving the formulation on a test instance. The best that was done is $9.2s$ instead of $208s$ with the original formulation. But, the $9.2s$ is still very big compared to $0.27s$ required by F2 to solve the same instance.

The conclusion is that ObM formulation will not be able to compete with the best existing formulation (F2). Nevertheless, it is a new concept that is studied and it will serve as reference for future works. Of course, it is an add-on to the list of exact methods for solving the GED problem.

This work was published in the conference ROADEF2018:
Darwiche, M., Conte, D., Raveaux, R., & T'kindt, V. (2018, February). Formulation linéaire en nombres entiers pour le problème de la distance d'édition entre graphes. In *ROADEF18*.

5.2.3 F3 formulation - an improvement of F2

The ObM formulation can be seen an expansion of the compact F2 formulation, which did not perform as good as F2. This work has led to think of a different approach to improve F2. The result is a F3 that is a new and an improved MILP formulation, inspired by $F2$, to solve the GED problem. It shares some parts with $F2$ formulation but rewrite the constraints in a different fashion. The main improvements are removing the symmetry case from the constraints to become in the objective function, which has led to new way of writing the constraints. The number of the new constraints is totally independent from the number of edges in the graphs.

5.2.3.1 F3 formulation

Data. Same as in F2 formulation, F3 uses the cost matrices $[c_v]$ and $[c_e]$, defined in equations 2.7 and 2.8.

Variables. F3 introduces two sets of decision variables $x_{i,k}$ and $y_{ij,kl}$ as in F2. However, it includes more y variables, by creating two variables: $y_{ij,kl}$ and $y_{ij,lk}$ for every $((i,j),(k,l)) \in E \times E'$. Let $\widetilde{E'} = \{(l,k) : \forall (k,l) \in E'\}$. The variables of the formulation are as follows.

- $x_{i,k} \in \{0,1\}$ $\forall i \in V, \forall k \in V'$; $x_{i,k} = 1$ when vertices i and k are matched, and 0 otherwise.
- $y_{ij,kl} \in \{0,1\}$ $\forall (i,j) \in E, \forall (k,l) \in E' \cup \widetilde{E'}$; $y_{ij,kl} = 1$ when edge (i,j) is matched with (k,l), and 0 otherwise.

Objective function. It is basically the same function as in F2 formulation, except for the sum of costs over the y variables to include all of them.

$$\min_{x,y} \sum_{i \in V} \sum_{k \in V'} (c_v(i,k) - c_v(i,\epsilon) - c_v(\epsilon,k)) \cdot x_{i,k} + \sum_{(i,j) \in E} \sum_{(k,l) \in E' \cup \widetilde{E'}} (c_e(ij,kl) - c_e(ij,\epsilon) - c_e(\epsilon,kl)) \cdot y_{ij,kl} + \gamma \tag{5.19}$$

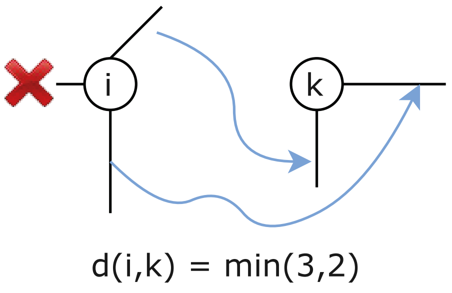


Figure 5.1: Example of edges assignment when assigning two vertices

Constraints. F3 formulation has the following constraints:

$$\sum_{k \in V'} x_{i,k} \leq 1 \ \forall i \in V \tag{5.20}$$

$$\sum_{i \in V} x_{i,k} \leq 1 \ \forall k \in V' \tag{5.21}$$

These two constraints are the same as in F2, and they guarantee that a vertex can be only matched with one vertex at most. However, the constraints 2.36 is replaced with the following constraint:

$$\sum_{(i,j) \in E} \ \sum_{(k,l) \in E' \cup \widetilde{E}'} y_{ij,kl} \leq d_{i,k} \cdot x_{i,k} \ \forall i \in V, \forall k \in V', \tag{5.22}$$

with $d_{i,k} = min(degree(i), degree(k))$. The degree of a vertex is the number of edges incident to the vertex. The constraints stands for: whenever two vertices are matched, e.g. $(i \rightarrow k)$, the maximum number of edges substitution that can be done is equal to the minimum degree of the two vertices. Figure 5.1 shows an example of the case. Two edges at most can be substituted and the third of i has to be deleted. Of course, the deletion of all edges is possible, if it costs less than the substitutions. These constraints force matching the edges and respecting the topological constraint defined in the GED problem.

The given formulation handles the case of undirected graphs. Though, it can be adapted to deal with the directed case, by setting $\widetilde{E}' = \{\phi\}$ (because edges (i,j) are different from (j,i) and they are already included in E), and replacing the objective function Eq. 5.19 by the objective function of F2 Eq. 2.32.

5.2.3.2 Comparison of F2 and F3 formulations

The most important improvement in the proposed formulation is that F3 has a number of constraints independent of the number of edges in the graphs. Constraints 5.20 and 5.21 are shared by both formulations and they do not include edges. However, constraints 2.36 rely on the edges of G, which is not the case of the constraints 5.22 in F3. Table 5.7 shows the number of variables and constraints in both formulations. Clearly, F3 has (2 times) more y variables than F2. The reason behind creating two y variables for each couple of edges, is to accommodate to the symmetry case that appears when dealing with undirected

Table 5.7: Nb. of variables and constraints in F2 and F3

	Nb. of variables	Nb. of Constraints
F2	$\|V\| \cdot \|V'\| + \|E\| \cdot \|E'\|$	$\|V\| + \|V'\| + \|V\| \cdot \|E\|$
F3	$\|V\| \cdot \|V'\| + \|E\| \cdot \|E'\| \cdot 2$	$\|V\| + \|V'\| + \|V\| \cdot \|V'\|$

graphs, i.e. $(i, j) = (j, i)$. By doing so, constraints 2.36 can be re-written differently by relying only on the vertices in the graphs (constraints 5.22). Note that, this comparison is done for undirected graphs. In the other case, the symmetry is discarded, and both formulations have the same number of variables.

In the GED problem, edge operations are driven by vertex-vertex matching. On this basis, the difficulty in F2 and F3 comes from the x decision variables, rather than the y variables. Moreover, F2 formulation is more sensitive to the density of the graphs, because its constraints depend on the edges, which is not the case in F3. This reasoning led to making the following two hypotheses, by distinguishing between two cases:

1. Non-dense graphs: even if F3 has more y variables than F2, its performance will not be degraded compared to F2.

2. Dense graphs: F3 will have less constraints than F2, since F3 has a number of constraints independent from the number of edges. Consequently, F3 tends to perform better than F2.

To validate those hypotheses, both formulations are tested over graph databases and real instances. The results are discussed in the next section.

5.2.3.3 Evaluation of F3 formulation

F3 formulation is designed to solve the GED problem. So, the graph databases must have instances of GED and not only GED^{EnA} problem.

Methods. F3 is evaluated against F2 formulation, which is the best one in the literature.

Experimentation settings. Both formulations are implemented in C language. The solver CPLEX 12.7.1, in single thread mode, is used to solve the formulations, with a maximum time limit of $900s$. Experiments are ran on a machine with Windows 7 $x64$, Intel Xeon $E5$ 2.30 GHz, 4 cores and 8 GB of RAM.

Evaluation indicators. The following indicators are computed for each subset:

- t_{min}: the minimum CPU time in seconds,
- t_{avg}: the average CPU time in seconds,
- t_{max}: the maximum CPU time in seconds,

- d_{min}: the minimum deviation, in percentage, between the solutions obtained by a formulation against the best solutions found by the three formulations,
- d_{avg}: the average deviation, in percentage, between the solutions obtained by a formulation against the best solutions found by the three formulations,
- d_{max}: the maximum deviation, in percentage, between the solutions obtained by a formulation against the best solutions found by the three formulations,
- η: the number of optimal solutions computed,
- η': the number of solutions (whether optimal or not) computed by a formulation, which are the best/minimum among those computed by all formulations.

The deviations are computed based on Eq. 4.11.

Table 5.8: Evaluation of F3 on MUTA database

	S	10	20	30	40	50	60	70	mixed
F3	t_{min}	0.02	0.05	0.16	0.47	0.83	2.96	6.94	0.06
	t_{avg}	0.10	3.07	365.44	575.65	770.61	810.51	811.10	410.08
	t_{max}	0.27	24.60	900.11	900.17	900.50	900.31	900.64	901.03
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	0.74	0.54	1.78	3.60	**2.55**	0.80
	d_{max}	0.00	0.00	20.69	12.86	13.04	25.42	44.63	9.33
	η	**100**	**100**	**81**	**76**	**31**	10	10	**62**
	η'	**100**	**100**	91	**90**	68	53	**61**	78
F2	t_{min}	0.00	0.00	0.06	0.11	0.22	0.41	0.61	0.02
	t_{avg}	**0.05**	**0.99**	**320.35**	**571.65**	**766.63**	**802.94**	**802.69**	**370.36**
	t_{max}	0.30	14.52	900.05	900.02	900.16	900.08	900.00	900.14
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	**0.21**	**0.51**	**1.52**	**1.46**	2.76	**0.15**
	d_{max}	0.00	0.00	4.20	7.06	9.63	11.69	46.15	3.54
	η	**100**	**100**	79	48	19	**11**	**11**	61
	η'	**100**	**100**	**93**	84	**69**	**69**	60	**91**

Table 5.9: Evaluation of F3 on MUTA database - optimal solutions

	S	10	20	30	40	50	60	70	mixed
F3	t_{min}	0.02	0.05	0.16	0.47	0.83	2.96	6.94	0.06
	t_{avg}	0.10	3.07	217.27	358.44	282.05	5.41	11.45	79.13
	t_{max}	0.27	24.60	836.98	814.03	869.97	6.69	16.63	729.77
	η	100	100	66	45	16	10	10	59
F2	t_{min}	0.00	0.00	0.06	0.11	0.22	0.41	0.61	0.02
	t_{avg}	**0.05**	**0.99**	**152.67**	**185.27**	**168.24**	**0.95**	**1.16**	**27.59**
	t_{max}	0.30	14.52	797.35	784.28	883.22	3.63	2.06	684.20
	η	100	100	66	45	16	10	10	59

Evaluations on MUTA database. The obtained results are shown in Table 5.8. For easy instances (subsets 10 and 20), both formulations were able to solve all of them to optimality, with 0% as average deviation. A difference in average deviations starts to appear for the rest of the subsets. F2 was able to score better average deviations than F3, except for subset 70 which contains the hardest instances. Though, the average deviations are very close (always less than 1%), excluding subset 60 where F2's average deviation is better by 2% from F3. Next, F3 was able to solve more instances to optimality (η) for subsets 30, 40, 50 and *mixed*. Regarding the best solutions obtained (η'), F2 has slightly higher values than F3 for subsets 30, 50, 60 and *mixed*. In addition, F2 was the fastest on the average running time for all subsets, but the difference is very marginal e.g. on subsets 30 and 40, F2 is faster by $4s$ than F3. Note that for hard instances, both formulations have reached $800s$, so they are not far from the maximum time limit given to the solver. In terms of solution quality, it can be called a tight between the two formulations. There is not a formulation that have performed better than the other.

In Table 5.9, the running time are filtered by considering only instances solved to optimality by both formulations. The average running times scored recorded for F2 are better than the ones for F3, which means F2 is faster than F3.

Conclusion. In fact, these results are very expected, and they confirm the first hypothesis mentioned earlier. MUTA database is considered as non-dense graphs, and its density is 9.13%. The first hypothesis states that for non-dense graphs, F3 formulation is capable of performing as good as F2. Because F2 and F3 will have almost the same number of constraints, but F3 will have twice y variables than F2. The claim is that those variables will not increase the complexity of the formulation. This has turned out to be true experimentally, because F3 and F2 formulations were very close after analyzing the solutions quality indicators (d_{avg}, η and η'). However, F2 formulation was faster in computing the optimal solutions, which is expected since F3 has more variables and will require a bit more running time to converge. Consequently, the first hypothesis is validated and F3 is as good as F2 on non-dense graphs.

Evaluations on CMU-HOUSE database. The three versions (HOUSE-NA, HOUSE-A and HOUSE-REF) of this database, as detailed in Section 4.2, are considered in this experiment. The results are given in Table 5.10. Both formulations were able to solve to optimality all instances of HOUSE-A, with average deviations 0%. For HOUSE-NA, which is harder than the other two versions, F3 has scored $d_{avg} = 0.70\%$ against a huge difference of 600% by F2. F3 has solved to optimality more than 50% of the instances, against 25 instances by F2. Also, F3 has found the best solutions for 644 instances, while F2 has found the best solutions only for 54 instances. For the last database HOUSE-REF, F3 has the smallest average deviation of 0.22%, the highest η and η' values w.r.t. F2 formulation. On the other hand, F2 is faster on the average running time for HOUSE-A and HOUSE-REF, while F3 is faster for HOUSE-NA.

Table 5.11 sums up the running times by keeping only instances solved to optimality by both formulations. F3 is 20 times faster than F2 for HOUSE-NA. But, F2 becomes faster on HOUSE-REF instances with more than $250s$ gap.

Conclusion. CMU-HOUSE has a density of 18%, and it is considered as a dense graph

Table 5.10: Evaluation of F3 on CMU-HOUSE database

		A	NA	REF
F3	t_{min}	0.19	1.68	66.07
	t_{avg}	2.95	**497.07**	416.75
	t_{max}	236.08	901.25	900.67
	d_{min}	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.70**	**0.22**
	d_{max}	0.00	250.00	88.34
	η	**660**	**365**	**633**
	η'	**660**	**644**	**652**
F2	t_{min}	0.11	93.07	1.26
	t_{avg}	**0.61**	880.74	**278.78**
	t_{max}	34.21	900.02	900.02
	d_{min}	0.00	0.00	0.00
	d_{avg}	**0.00**	604.11	4.68
	d_{max}	0.00	3600.00	163.86
	η	**660**	25	505
	η'	**660**	54	548

Table 5.11: Evaluation of F3 on CMU-HOUSE database - optimal solutions

		NA	REF
F3	t_{min}	2.54	66.07
	t_{avg}	**20.26**	374.51
	t_{max}	42.18	869.39
	η	25	482
F2	t_{min}	93.07	1.26
	t_{avg}	395.33	**89.89**
	t_{max}	828.52	891.76
	η	25	482

database. The results have proven that F3 formulation is better than F2 formulation in solving instances of HOUSE-NA and HOUSE-REF. This is, probably, because F3 has way less constraints than F2, which makes it easier for the solver to converge and to find the optimal solution. This proves the second hypothesis, that for dense graphs F3 has less constraints and therefore it is better than F2 formulation.

Table 5.12: Evaluation of F3 on SYNTHETIC-30 database

	D	40	60	80	100
F3	t_{min}	899.75	888.29	900.09	900.25
	t_{avg}	899.92	899.92	901.16	900.93
	t_{max}	900.03	900.14	902.45	901.48
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	9.34	10.25	10.70	**10.40**
	d_{max}	100.00	100.00	100.00	100.00
	η	0	0	0	0
	η'	**72**	**78**	**66**	**77**
F2	t_{min}	18.69	54.10	96.38	395.62
	t_{avg}	**812.42**	**818.37**	**829.83**	**862.04**
	t_{max}	900.11	901.20	903.12	903.73
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	**5.15**	**3.95**	**2.60**	55.65
	d_{max}	18.70	15.30	11.49	227.96
	η	**10**	**10**	**10**	**10**
	η'	31	22	34	23

Evaluations on SYNTHETIC-30 database. The details of this database are given in Section 4.2. It is split into different subsets based on the density of the graphs. The above experiments were done over databases with densities less than 20%. Therefore in this experiment, subsets with densities 40%, 60%, 80% and 100% are selected to study the performance of the formulations with high dense graphs. The results of this experiment are reported in Table 5.12. The best average deviations are obtained by F2 for all subsets, except subset with 100% density. The differences between the deviations varies from 4% to 8% for subset 80. And a big difference of 45% is noted for subset 100 in favor of F3 formulation. In terms of optimal solutions, F3 did not solve any instance to optimality for all subsets, which is not the case for F2 that succeeded in finding optimal solutions for 10 instances per subset. Actually, those instances are the pair of identical graphs. On the other hand, F3 has computed best solutions (η') for two times more instances than F2 on all subsets. Clearly, there is something suspicious, because F2 has smaller average deviations, yet it is not better in computing best/smaller solutions. After carefully analyzing the results per instance, it turned out that F3 had hard time in solving instances where graphs are identical, while F2 was more efficient. To give a more objective analysis, the instances are split into two groups: first group with all instances of different graphs, and the second group with instances of identical graphs.

Table 5.13: Evaluation of F3 on SYNTHETIC-30 database - without identical graphs

	D	40	60	80	100
F3	t_{min}	899.75	899.80	900.09	900.39
	t_{avg}	899.92	900.04	901.16	900.93
	t_{max}	900.03	900.13	902.45	901.48
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	**1.48**	**0.27**	**0.78**	**0.44**
	d_{max}	15.21	4.87	11.49	7.28
	η	0	0	0	0
	η'	**70**	**78**	**66**	**77**
F2	t_{min}	896.94	898.50	899.94	900.11
	t_{avg}	899.42	900.07	900.20	900.52
	t_{max}	900.11	901.20	903.12	903.73
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	5.73	4.39	2.89	61.84
	d_{max}	18.70	15.30	11.49	227.96
	η	0	0	0	0
	η'	21	12	24	13

The results of the first group are shown in Table 5.13. This time, all good average deviations are scored by F3. For instance, on subset 100 the d_{avg} scored by F3 is 0.44% against 61% by F2. η' values for F3 formulation are 3 to 4 times bigger than the values obtained by F2 formulation. Evidently, F3 performs much better than F2 in solving SYNTHETIC-30 instances. Note that both formulations have reached the maximum time limit when solving all instances, that is why t_{avg} values are always $\approx 900s$.

The results of the second group where only identical graphs are considered are reported in Table 5.14. F2 has solved all these instances to optimality (i.e. $d_{avg} = 0.00\%$ for all subsets), while F3 has failed to do so (all $d_{avg} = 100\%$ for all subsets). In addition, F2 is much more faster than F3, which reaches the $900s$ all the time against half the time by F2 in the worst case (instances of subset 100).

Conclusion. Excluding instances with identical graphs, F3 is very effective in solving SYNTHETIC-30 database instances, and it outperforms F2 formulation by far. This conclusion, without any doubts, confirms the second hypothesis that F3 performs better with dense and high dense graphs. However, there seems to be a problem when solving F3 with identical graphs instances where it suffers without converging towards the optimal solutions. F2 did not have this problem with those particular instances, but it was not effective in solving the others. Further digging and troubleshooting by looking at CPLEX output when solving those instances, showed that CPLEX spent all the time generating cuts before it starts the branching with F3 formulation. While it behaves differently with F2 formulation and it starts the branching earlier. Then, the discussion is moving towards CPLEX

Table 5.14: Evaluation of F3 on SYNTHETIC-30 database - identical graphs

	D	40	60	80	100
F3	t_{min}	899.83	888.29	900.13	900.25
	t_{avg}	899.92	898.88	901.13	900.84
	t_{max}	899.99	900.14	902.08	901.14
	d_{min}	100.00	100.00	100.00	100.00
	d_{avg}	100.00	100.00	100.00	100.00
	d_{max}	100.00	100.00	100.00	100.00
	η	0	0	0	0
	η'	2	0	0	0
F2	t_{min}	18.69	54.10	96.38	395.62
	t_{avg}	**29.46**	**83.06**	**196.53**	**515.77**
	t_{max}	41.23	124.97	258.15	559.33
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	**0.00**	**0.00**
	d_{max}	0.00	0.00	0.00	0.00
	η	**10**	**10**	**10**	**10**
	η'	**10**	**10**	**10**	**10**

behavior during the solution phase with its default parameters. It is highly probable that modifying CPLEX parameters, especially the ones related to cuts generation, might help finding the optimal solutions faster for F3. Though, doing so is not easy, because CPLEX has hundreds of parameters, which may require a lot of time to tune and find the best combinations. Additionally, tuning CPLEX parameters can be a research topic on its own, which is out of the scope of this thesis. Finally, this case appeared only on those particular instances synthetically generated, and they might not be very representative and close to real-life instances.

F3 formulation is evaluated against two other databases: PROTEIN and SYNTHETIC-100. The results can be found in Appendix B, Section B.2. The conclusions of those experiments are:

- **PROTEIN database:** PROTEIN is a dense graph database ($D = 16\%$) and again the second hypothesis is confirmed. F3 formulation is more effective than F2 in solving PROTEIN instances. Mainly, because it has less constraints than F2 formulation.

- **SYNTHETIC-100 database:** F3 has performed better than F2 on very big and high-dense graphs. Both formulations were not able to find any optimal solution. However, F3 has computed feasible solutions for all the instances. While F2 did not succeed in computing any feasible solution. The instances are very hard for F2 formulation.

5.2.3.4 General conclusions

F3 is a new and improved formulation inspired by the existing F2 formulation. F3 introduces more variables in the formulation to handle edges matching, but it has constraints independent from the number of edges in the graphs. Two hypotheses are made regarding two cases: first one is that F3 is as good as F2 when dealing with non-dense graphs, and the second is that F3 is better dealing with dense graphs because its constraints are independent from the number of edges. Experimentally, the first hypothesis is confirmed when testing F3 on MUTA database, which contains non-dense graphs. The second hypothesis is validated by selecting multiple databases and varying the densities from 16% until reaching 100% (complete graphs). The results have shown F3 to be more efficient than F2 formulation in solving the instances to optimality.

These results were published in the following conferences:

- Darwiche, M., Raveaux, R., Conte, D., & T'Kindt, V. (2018, April). A New Mixed Integer Linear Program for the Graph Edit Distance Problem. In *ISCO18*.
- Darwiche, M., Raveaux, R., Conte, D., & T'Kindt, V. (2018, August). Graph Edit Distance in the Exact Context. *In Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)* (pp. 304-314). Springer, Cham.

5.3 Proposed matheuristics

This section covers the second part of this thesis, which is about designing matheuristics to solve the GED problem. Matheuristics require having a MILP formulation modeling the problem and a MILP solver. So, they can employ the solver to explore the solution space of the MILP formulation looking for good solutions. The choice of the MILP formulation can be made based on the results of the previous section. The MILP formulation F3, proposed earlier and proved to be effective in solving the problem, will form the basis of the proposed matheuristics in this section.

5.3.1 An adapted local branching to solve the GED problem

In Chapter 4, an adapted local branching matheuristic is presented to solve the subproblem GED^{EnA}, where good results were obtained w.r.t. existing heuristics. To fulfill one of the main objective of this thesis, which is to develop a good heuristic to solve the general problem, local branching will be modified so it can be applied to the general GED problem. The limitation of the local branching version presented in Chapter 4 comes from the use of JH MILP formulation in the implementation of the heuristic. Changing the MILP formulation, and using instead a formulation modeling the GED problem, will result in a heuristic that solves the general problem. Basically, all LocBra features are kept the same, as explained in Section 4.4. Only JH formulation is replaced by F3 formulation (Section 5.2.3). Therefore, local branching constraints for intensification and diversification are defined over the set of $x_{i,k}$ variables in F3.

To evaluate the performance of LocBra, it is compared with the best known heuristics.

This version will be evaluated on GED instances. Unlike experiments done in Section 4.4, where only GED^{EnA} instances were involved, the following experiments will include both, GED^{EnA} instances and more importantly GED instances. The experiments are divided into two categories:

1. **Effectiveness of LocBra w.r.t. competitor heuristics**. LocBra is tested against the most competitive heuristics picked from the literature, designed to solve the GED problem.
2. **Effectiveness of LocBra w.r.t. an exact method**. The goal of this experiment is to measure the accuracy and closeness of LocBra solutions from the optimal or best known ones.

Common configuration. LocBra algorithm is implemented in C language. The solver CPLEX 12.7.1 is used to solve the MILP formulations. Experiments are ran on a machine with Windows $7x64$, Intel Xeon $E5$ 2.30 GHz, 4 cores and 8 GB of RAM. CPLEX solver is configured to use single thread, and the rest of the parameters are set to default. The aim of this, in the experiments, is to evaluate the efficiency of the inner mechanism of LocBra. It can be then expected that its efficiency is going to be improved by enabling the use of more threads.

5.3.1.1 Effectiveness of LocBra w.r.t. competitor heuristics

These experiments answer the following question: which heuristic is the best minimizer? It is about comparing the distances computed by each heuristic and finds out which heuristic returns the smallest ones.

Methods. The heuristics chosen in the evaluation are:

- **i-** *CPLEX-t* is the solver CPLEX ran on F3 formulation with t seconds as a time limit.
- **ii-** *CPLEX-LocBra-t* refers to enabling local branching heuristic implemented in CPLEX solver. The time limit is imposed in order to compute an initial solution, before running local branching on that solution.
- **iii-** *BeamSearch-α*, the BeamSearch heuristic with α the beam size.
- **iv-** *SBPBeam-α*, the SBPBeam heuristic with α the beam size.
- **v-** *IPFP-it*, the IPFP heuristic with *it* the maximum number of iterations.
- **vi-** *GNCCP-d*, the GNCCP heuristic with d the quantity to be subtracted from the ζ variable at each iteration. ζ is the variable that controls the concavity and convexity of the objective function of the QAP model solved by GNCCP heuristic.

The first two heuristics are also based on solving F3 formulation by CPLEX. Considering them as a part of the experiment will show if the branching scheme of LocBra is capable of performing better than the solver CPLEX and its embedded heuristics. The other heuristics are picked from the literature after reviewing the most important and competitive ones. Their descriptions and details can be found in Chapter 2 (Section 2.3.8).

Comparison indicators. All heuristics are executed on different databases and for all of them, the following indicators are computed: t_{min}, t_{avg}, and t_{max} are the minimum, average and maximum CPU times in seconds over all instances. Correspondingly, d_{min}, d_{avg}, and d_{max} are the deviations of the solutions obtained by one heuristic, from the best solutions found by all heuristics. The deviations are computed based on Eq. 4.11 and are expressed in percentage. Lastly, η_I is the number of instances for which a given heuristic has found the best solutions.

Evaluations on PROTEIN database. This database contains GED graph instances. It has 4 subsets of 10 graphs each, which gives 400 instances in total. The details of the database can be found in Section 4.2.

As in previous experiments, two versions of each heuristic are considered: Default and Extended. The default version is to evaluate the default behavior of the heuristic, and the extended version is to see if it is capable of improving the results by spending as much time as LocBra. The parameters of each heuristic are set based on preliminary experiments not reported here.

Default versions. The parameters are set to the following values:

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 10s$, $node\_time\_limit = 2s$, $UB\_time\_limit = 4s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
CPLEX-t	$t = 10$
CPLEX-LocBra-t	$t = 9$
BeamSearch-α	$\alpha = 5$
SBPBeam-α	$\alpha = 5$
IPFP-it	$it = 10$
GNCCP-d	$d = 0.1$

The results reported in Table 5.15 reveal that *LocBra* has scored the best/smallest average deviations for all subsets, along with the highest η_I values. In the second place, it is a tight between *CPLEX-10* and *CPLEX-LocBra-9*. The former has the best d_{avg} values for subsets 30 and *mixed* with 0.12% and 0.16% respectively, against 0.13% and 0.17% scored by the latter. Though, the difference is very small. *IPFP-10* and *GNCCP-0.1* performances are also very close, because *IPFP-10* is better on subsets 20 and 30, but not on subsets 40 and *mixed*. At last, *BeamSearch-5* and *SBPBeam-5* are close to each other, but far from the rest of the heuristics. Regarding the average running time, *BeamSearch-5* is the fastest heuristic with a big difference compared to the running times of *LocBra*. The $10s$ given to *LocBra*, however, sounds reasonable when looking at the running time values of *GNCCP-0.1* (e.g. $t_{avg} = 8.82s$ against $23.17s$ on subset 40).

Table 5.15: LocBra vs. heuristics on PROTEIN instances

	S	20	30	40	mixed
LocBra	t_{min}	0.06	0.09	0.20	0.09
	t_{avg}	6.54	8.68	8.82	8.59
	t_{max}	10.08	10.05	10.09	10.19
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	**0.07**	**0.06**	**0.39**	**0.11**
	d_{max}	0.84	0.56	1.53	2.51
	η_I	**80**	**74**	**37**	**68**
CPLEX-10	t_{min}	0.05	0.06	0.16	0.05
	t_{avg}	5.86	8.54	8.84	8.24
	t_{max}	10.03	10.05	10.06	10.03
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	0.09	0.12	0.57	0.16
	d_{max}	0.84	0.75	2.47	3.07
	η_I	77	53	29	58
CPLEX-LocBra-9	t_{min}	0.08	0.16	0.27	0.11
	t_{avg}	5.76	8.01	11.38	7.92
	t_{max}	11.23	9.48	27.25	11.56
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	0.09	0.13	0.50	0.17
	d_{max}	0.84	0.75	1.92	3.26
	η_I	75	54	31	52
BeamSearch-5	t_{min}	0.00	0.01	0.01	0.00
	t_{avg}	**0.02**	**0.07**	**0.10**	**0.06**
	t_{max}	0.04	0.13	0.22	0.13
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	5.62	3.33	2.88	5.57
	d_{max}	123.86	155.56	7.29	26.54
	η_I	10	11	10	10
SBPBeam-5	t_{min}	0.26	1.03	2.66	0.36
	t_{avg}	0.37	1.54	3.76	1.75
	t_{max}	0.54	2.26	5.05	4.39
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	3.19	5.58	3.43	2.74
	d_{max}	5.36	155.56	5.58	4.78
	η_I	12	10	10	10
IPFP-10	t_{min}	0.02	0.05	0.04	0.03
	t_{avg}	0.09	0.27	0.59	0.31
	t_{max}	0.13	0.38	0.88	0.81
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	1.05	0.97	1.13	1.02
	d_{max}	3.25	3.38	2.65	2.80
	η_I	23	13	11	13
GNCCP-0.1	t_{min}	1.32	4.50	13.93	1.45
	t_{avg}	2.05	7.21	23.17	9.36
	t_{max}	2.47	10.45	30.51	23.26
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	0.22	0.19	1.67	1.35
	d_{max}	1.08	0.80	156.60	117.42
	η_I	53	47	69	50

Table 5.16: LocBra vs. heuristics with extended running time on PROTEIN instances

	S	20	30	40	mixed
LocBra	t_{min}	0.08	0.12	0.22	0.12
	t_{avg}	14.28	24.77	26.38	23.57
	t_{max}	30.08	30.11	30.31	30.37
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	**0.02**	**0.04**	0.19	**0.07**
	d_{max}	0.28	0.51	1.68	2.90
	η_I	**91**	**81**	**58**	**80**
CPLEX-30	t_{min}	0.03	0.11	0.17	0.05
	t_{avg}	11.56	24.09	26.18	22.33
	t_{max}	30.03	30.01	30.05	30.03
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	0.05	0.07	0.36	0.10
	d_{max}	1.12	0.51	1.68	3.08
	η_I	90	67	32	69
CPLEX-LocBra-25	t_{min}	0.08	0.16	0.27	0.08
	t_{avg}	9.78	20.40	24.70	18.97
	t_{max}	27.13	25.44	42.57	27.38
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	0.04	0.07	0.35	0.10
	d_{max}	1.12	0.56	1.68	3.08
	η_I	**91**	67	33	68
BeamSearch-1500	t_{min}	0.00	0.01	0.01	0.00
	t_{avg}	3.94	13.72	23.72	13.42
	t_{max}	5.85	20.93	34.15	25.98
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	1.65	7.33	1.86	4.44
	d_{max}	21.88	311.11	6.66	26.01
	η_I	12	10	10	10
SBPBeam-30	t_{min}	1.57	6.16	15.28	1.63
	t_{avg}	2.15	8.95	23.05	**8.79**
	t_{max}	3.06	13.33	31.71	21.32
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	3.23	5.65	3.48	2.79
	d_{max}	5.66	155.56	5.89	4.78
	η_I	12	10	10	10
IPFP-500	t_{min}	0.01	0.04	0.13	0.02
	t_{avg}	**2.14**	**7.36**	**22.26**	9.01
	t_{max}	5.03	12.99	31.93	27.64
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	0.80	0.81	0.75	0.77
	d_{max}	2.71	3.01	2.10	2.22
	η_I	24	16	13	12
GNCCP-0.09	t_{min}	1.47	4.58	15.84	1.52
	t_{avg}	2.19	7.69	24.34	9.75
	t_{max}	3.32	10.96	29.79	28.58
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	0.23	0.24	**0.14**	0.24
	d_{max}	1.12	0.94	0.95	1.13
	η_I	46	27	57	30

Extended versions. LocBra is configured with a maximum running time of 30s. The rest of the heuristics are configured, as well, to reach almost the same running time.

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 30s$, $node\_time\_limit = 6s$, $UB\_time\_limit = 12s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
CPLEX-t	$t = 30$
CPLEX-LocBra-t	$t = 800$
BeamSearch-α	$\alpha = 1500$
SBPBeam-α	$\alpha = 30$
IPFP-it	$it = 500$
GNCCP-d	$d = 0.09$

The results of this version are presented in Table 5.16. Extending the running time of the heuristics has really helped *IPFP-500* and *GNCCP-0.09* in computing better solutions and therefore improving their d_{avg} values. In fact, *GNCCP-0.09* was even able to perform better than *LocBra* on the hardest instances with $d_{avg} = 0.14\%$ against 0.19%. But, *LocBra* has performed better on the other subsets in terms of average deviations and η_I values. *GNCCP-0.09* was able also to beat *CPLEX-30* and *CPLEX-LocBra-25* on subset 40. However, it is not the case for the other subsets where *CPLEX-30* and *CPLEX-LocBra-25* are still in the second position after *LocBra*. Despite, the improvement in the d_{avg} values obtained by *BeamSearch-1500* and *SBPBeam-30*, it is still not enough to compete with the other heuristics. *IPFP-500* is the fastest when looking at the average running time, except for subset *mixed* where *SBPBeam-30* is faster.

Conclusion. The results of both experiments show, on one hand, that LocBra improves the default behavior of the solver, and it is a better version than the basic embedded one in CPLEX. On the other hand, LocBra outperforms existing heuristics in their default behaviors and also when increasing their running time.

The same experiments are repeated on other databases (MUTA and HOUSE-REF), and the obtained results are reported in Appendix B, Section B.3.1. The conclusions of those experiments are:

- **MUTA database:** LocBra heuristic was able to compute better solutions for MUTA instances than the other heuristics. In both the default and the extended versions, LocBra has outperformed CPLEX-based methods and the four heuristics selected from the literature. Yet, LocBra is slower than other existing heuristics, such as BeamSearch and IPFP.
- **HOUSE-REF database:** In the default version, LocBra has succeeded in solving efficiently HOUSE-REF instances. However and remarkably, GNCCP was able to perform better than LocBra in the extended version. Yet, the difference is pretty much small with 2.7% on average deviation.

General conclusions based on the evaluations. Based on all the results of these experiments and the summary given in Table 5.17, LocBra significantly outperforms the default behavior of CPLEX, the embedded local branching version in CPLEX, and the heuristics available in the literature. Especially in terms of solutions quality, it is the most effective. However, it is not the fastest where other heuristics such as IPFP and BeamSearch are faster. There has been an exception in the extended version on HOUSE-REF instances, where GNCCP has performed better than LocBra. Since, it was not the case in the rest of the experiments, LocBra can still be considered the best minimizer to the GED problem in the general case.

Table 5.17: Summary of LocBra comparison w.r.t. competitor heuristics

	Database	Solutions quality	Speed
Default versions	PROTEIN	LocBra	BeamSearch
	MUTA	LocBra	BeamSearch
	HOUSE-REF	LocBra	BeamSearch
Extended versions	PROTEIN	LocBra	IPFP
	MUTA	LocBra	LocBra/GNCCP
	HOUSE-REF	GNCCP	IPFP

5.3.1.2 Effectiveness of LocBra w.r.t. an exact method

This experiment will answer the following question: how close the LocBra solutions are from the optimal solutions?

Methods. The exact approach in these experiments consists in solving a GED MILP formulation using CPLEX. In a similar experiment presented in Section 4.4.6.2, the exact method was running CPLEX to solve the MILP formulation until finding all the optimal solutions. In the case of MUTA database and because some of its instances are very hard, CPLEX was set to run during 10 hours. Now, the current experiment involves new databases that contain GED instances. So, the exact method must be executed to find the optimal solutions. But, due to the hard time constraint of the thesis, it was not possible to afford running CPLEX with a time limit of 10 hours. Instead, CPLEX was given only $900s$ to compute a solution per instance. Of course, this is not an exact method, however it will give an idea about the solutions quality computed by LocBra. Note that for GED instances, the formulation used is F3 with CPLEX.

Comparison indicators. The following indicators are computed for each database: t_{min}, t_{avg}, and t_{max}, which are respectively the minimum, average and maximum CPU times in seconds over all instances. Correspondingly, d_{min}, d_{avg}, and d_{max} are the deviations (in percentage) of the solutions obtained by LocBra, from the optimal or best solutions found. The deviation is computed by using Eq. 4.11. In addition, η_I is the number of optimal solutions found, and η'_I is the number of solutions found by LocBra that are equal to the optimal or best known ones. At last, η''_I is the number of solutions computed by LocBra that are better than the best known solutions, when *CPLEX* was not able to find the optimal solutions.

Table 5.18: LocBra vs. Exact solution on PROTEIN instances

	CPLEX-900				LocBra								
S	t_{min}	t_{avg}	t_{max}	η_I	t_{min}	t_{avg}	t_{max}	d_{min}	d_{avg}	d_{max}	η_I	η'_I	η''_I
20	0.03	26.83	215.59	100	0.08	**14.28**	**30.08**	0.00	0.07	0.84	60	**81**	0
30	0.09	132.00	856.79	100	0.12	**24.77**	**30.11**	0.00	0.14	0.66	19	**49**	0
40	0.14	575.70	900.52	63	0.22	**26.38**	**30.31**	**-0.14**	0.35	1.26	12	**16**	**3**
mixed	0.03	180.36	900.66	96	0.12	**23.57**	**30.37**	0.00	0.16	0.71	24	**46**	0

Evaluations on PROTEIN database. LocBra parameters are set to the following values:
$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 30s$, $node\_time\_limit = 6s$, $UB\_time\_limit = 12s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$. The exact method for this database is CPLEX with a time limit of $900s$. It turned out that the $900s$ were enough to compute the optimal solutions for 322 instances out of 400.

The results are reported in Table 5.18. The average deviations of LocBra are less than 1% for all subsets, except subset 40 because the instances are the hardest. But, on that subset LocBra has a minimum deviation of -0.14%, and $\eta''_I = 3$. Which means that LocBra has found in $30s$ solutions better than CPLEX in $900s$. Another important remark is that LocBra average and maximum running times are very small compared to *CPLEX-900* running times. LocBra, in the worst case, stops at $30s$, while *CPLEX-900* reaches $575s$ on average and $900s$ in the worst case. Such difference is remarkable, especially because LocBra in that short amount of time has found good solutions that are very close to the best or optimal ones. This proves the high quality of solutions computed by LocBra.

Conclusion. LocBra has computed very good solutions in a maximum time of $30s$ compared to CPLEX with $900s$. The deviation on average does not exceed 0.35%, which considerably small. In addition, LocBra was able to compute better solutions than CPLEX for 3 instances.

The same experiments are repeated on other databases (MUTA and HOUSE-REF), and the obtained results are reported in Appendix B, Section B.3.2. The conclusions of those experiments are:

- **MUTA database:** Those results show that LocBra in $900s$ can compute solutions at 10.78% far from the optimal/best solutions computed by solving JH formulation during 10 hours. This number is, of course, in the worst case and it is considerably good.
- **HOUSE-REF database:** The results have shown that LocBra was able to compute very good quality solutions that are far by 2.5% from the optimal/best solutions. LocBra has achieved these results with a maximum running time of $100s$, while CPLEX needed more than $400s$ on average. Moreover, LocBra in $100s$ was able to find better solutions than CPLEX in $900s$ for 6 instances.

General conclusions based on the evaluations. LocBra has shown closeness between its solutions and optimal/best solutions with MUTA instances. It, even, succeeded in computing better solutions in very short amount of time, like shown in the experiment on PROTEIN instances. In a maximum of 30s, LocBra was able to converge to better solutions than CPLEX in 900s. Same behavior and results are recorded on HOUSE-REF database as well. All those results have answered the question asked at the beginning of this experiment. They also proves that LocBra is capable of computing solutions very close to the optimal/best ones.

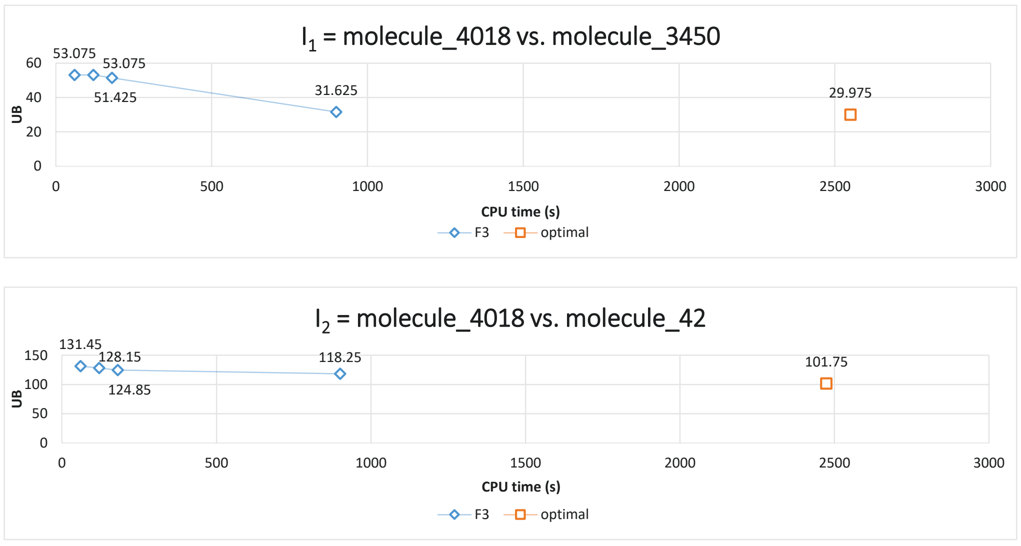


Figure 5.2: F3 formulation solved for two MUTA instances with different time limits.

5.3.1.3 LocBra parameters analysis

It is discussed in Chapter 4 (Section 4.4.4) how the parameter values are chosen for every experiment. This section gives a better insight about the parameters and their influences in LocBra, and in particular the version with F3 formulation.

Upper Bound (UB) computation. To determine a good time limit to compute an UB in LocBra, few instances are picked from each database and they are ran with different time limits. Figure 5.2 shows the UB solutions trend with different time limits when solving F3 formulation. The optimal solution for each instance is also shown in the figures. As can be seen in I_1 instance, the objective function value starts at 53.075 with 60s as a time limit. It decreases to 51.425 with 180s, and reaches 31.625 with 900s. Knowing that the optimal solution is 29.975 that requires more than 2500s to be found, the UB value obtained within 180s seems to be a good solution to start up LocBra algorithm. Similarly, in I_2 instance, the UB 124.85 obtained at 180s seems a good one to start the branching iterations. The difference between 124.85 and the optimal solution with an objective value of 101.75 is smaller than other UB values obtained at 60s and 120s. The results of other instances, not shown here, have led to the same conclusions. Therefore, 180s is chosen as the value to UB_time_limit parameter in LocBra heuristic.

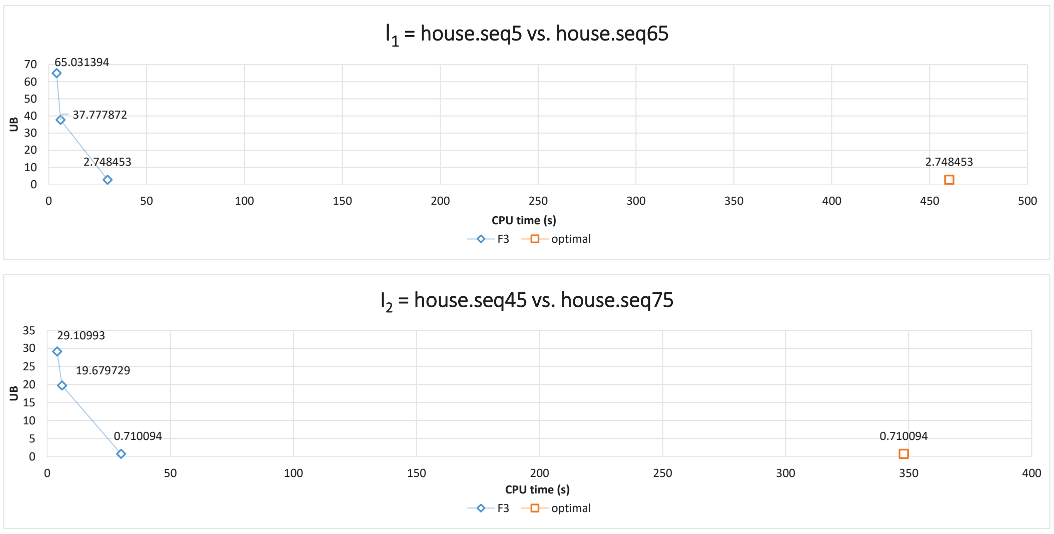
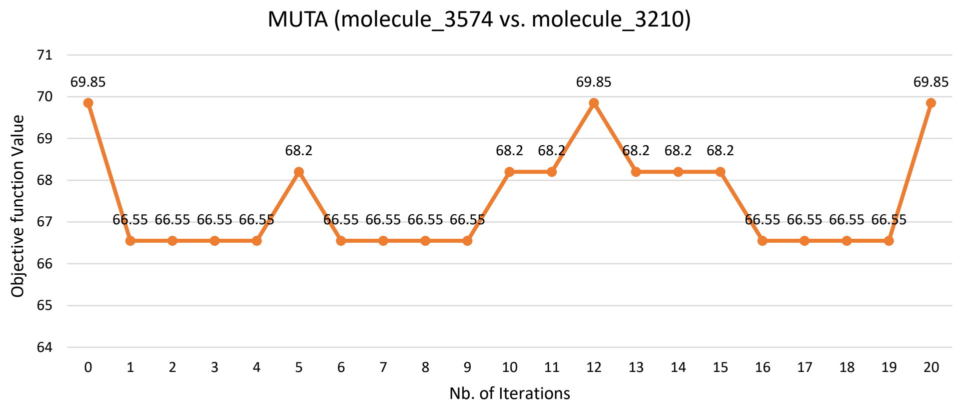


Figure 5.3: F3 formulation solved for two HOUSE-REF instances with different time limits.

Figure 5.3 shows the UB behavior with time on two HOUSE-REF instances. For both instances, the UB obtained within 30s is, actually, the objective function values of the optimal solutions. These optimal solutions were found when solving F3 formulations with time reaching hundreds of seconds. All the time spent after the first 30s, was consumed by the solver trying to prove that this is the optimal solution. So, 30s seems to be reasonable as an UB time limit, in the extended version of LocBra. And in the default version, the 30s is set as the total time limit to the heuristic. This will show what the heuristic has to offer compared to the solver CPLEX.

The evolution of solutions per iteration. LocBra is an iterative process and in each iteration it explores a neighborhood in the solution space. Figure 5.4 shows the evolution of the objective function values of the solutions found in each iteration. This gives an idea about the behavior of LocBra and shows whether an iteration is an intensification or a diversification step. For example, on MUTA instance, LocBra has reached 20 iterations. It starts with an UB of 69.85, and the next solution found had an objective function value of 66.55. Up until forth iteration, the intensification had always found solutions with the same objective function value. This case had caused a diversification step to be fired, which explains the curve jump to 68.2. The parameter ℓ_max is responsible for this diversification step. Next, it continued with an intensification step in iteration 6. The curve keeps going up and down, which means that diversification followed by intensification steps are occurring. Finally, the best solution is the one with the objective function value of 66.55.

On HOUSE-REF instance, LocBra starts off with an UB of 35.88. The first iteration improved the solution and reached a value of 7.53. In the second iteration, which is an intensification step, the solver did not find a feasible solution (represented by -1). The reason is whether the time limit given is reached, or there is no feasible solution in that neighborhood. This case has forced a diversification step, followed by intensification steps.



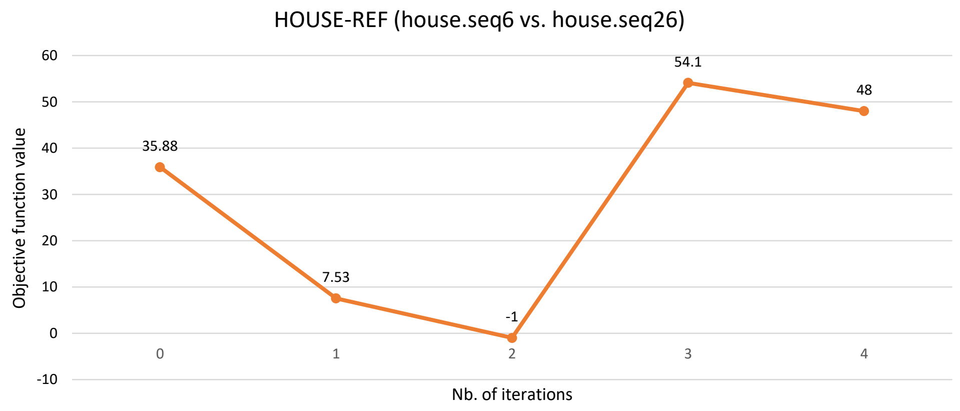


Figure 5.4: The evolution of the objective function values of computed solutions with the number of iterations for one MUTA instance and HOUSE-REF instance.

These examples confirm the usefulness of LocBra parameters and they show the importance of diversification when getting blocked in a region in the solution space.

Iterations vs. best solutions. Figure 5.5 shows at which iteration the best solutions were found for MUTA and HOUSE-REF instances. Iteration 0 indicates the phase of UB computation, where at this point it is possible to find the optimal solution. The best solutions are found in iteration 0 for 368 instances in MUTA database. Then, for 98 instances an intensification was needed to find the best solutions. The number grows for 2 intensification iterations, and then starts to drop until reaching 4 instances with 7 iterations. On HOUSE-REF database, the best solutions where found in the phase of UB computation for 282 instances. For the rest of the instances (378), 1 to 3 intensification steps were needed to improve the solutions and to converge towards the best ones. Note that the maximum number of iterations in the charts is not necessarily the actual maximum number. There could be for some instances more iterations but without finding better solutions.

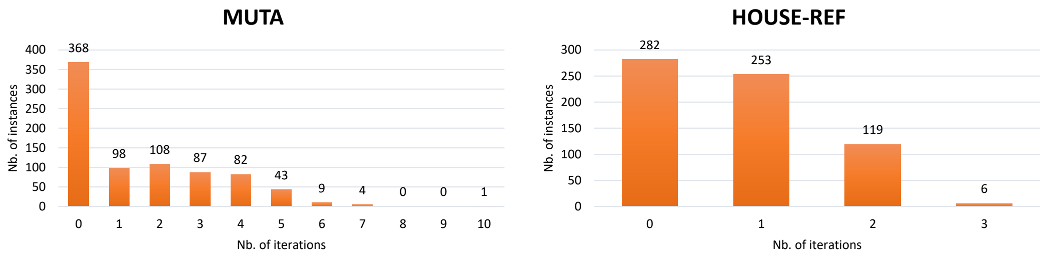


Figure 5.5: Nb. iterations vs. the number of instance where best solutions are found.

Stopping criteria statistics. Figure 5.6 shows the number of instances per stopping criterion. Based on LocBra algorithm presented in Section 4.4.3, the conditions that cause LocBra to stop are:

1. *Optimal:* if the optimal solution is found when computing the UB.
2. *TLReached:* if the total time limit imposed is reached. This is controlled by the parameter *total_time_limit*.
3. *Cons_dv:* if the number of maximum consecutive diversification steps is reached. This is controlled by the parameter *dv_cons_max*.
4. *DV_max:* if the number of maximum diversification steps allowed is reached. This is controlled by the parameter *dv_max*.

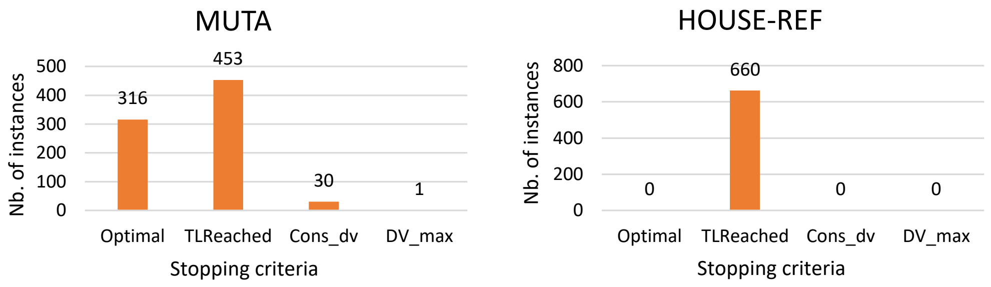


Figure 5.6: Stopping criteria vs. number of instances.

On MUTA instances, LocBra stopped after finding the optimal solution for 316 instances. It stopped because of reaching the maximum time limit for 453 instances. The parameters controlling the diversification were useful for only 31 instances. However, on HOUSE-REF instances, the only used stopping criterion is the maximum time limit for all the 660 instances.

All the above analyses and statistics are interesting and show how LocBra behaves when solving GED instances. Even, if it is not shown here, the same was done for other databases such as PROTEIN, PAH, etc.

5.3.2 An adapted VPLS to solve the GED problem

Variable Partitioning Local Search (VPLS) is matheuristic, proposed by Della Croce et al. (2013), that aims at solving optimization problems by embedding a MILP solver into heuristic algorithms. VPLS framework can be seen as a local search approach. As in local branching, VPLS is based on defining neighborhoods around feasible solutions by modifying the MILP formulation. Then, the modified formulation is handed out to the solver to explore the neighborhood. Della Croce et al. (2013) have applied VPLS to problems such as two-machine total completion time flow shop, and nurse rostering, where good and satisfying results were achieved. For this reason, and after obtaining satisfactory results with local branching matheuristic, it is interesting to test a special VPLS version to solve the GED problem. VPLS allows integrating information and characteristics into the neighborhood definition, which increases most likely the performance of the heuristic. This section will cover the details of a VPLS heuristic designed for solving the GED problem.

5.3.2.1 Main features of VPLS

Considering the general form of a MILP formulation, as given in Eq. 3.5, VPLS focuses on the list of binary variables, which are the main source of difficulty in such formulations. Let $X_B = \{x_i \mid \forall i \in B\}$ be the set of binary variables, and $\bar{X}_B = \{\bar{x}_i \mid \forall i \in B\}$ be the values assigned to binary variables for a given $\bar{X}$ feasible solution. Assuming that there exists a partition $S \subseteq B$ of "special" binary variables. The variables in S are selected based on some defined rules, where these rules underlies some analyses and observations related to the problem. As an example, those special variables can be determined based on problem-dependent information and characteristics of an instance. After determining the set S, a neighborhood $N(\bar{X}, S)$ can be defined as follows:

$$N(\bar{X}, S) = \{X_B \mid x_j = \bar{x}_j, \forall j \notin S\} \tag{5.23}$$

The neighborhood of $\bar{X}$, then, contains all solutions of the MILP such that, they share the same values of binary variables not belonging to subset S, as in the current solution $\bar{X}_B$. Meanwhile, the variables belonging to subset S remain free. An example of variables partitioning is depicted in Fig. 5.7. So, the resulting restricted MILP formulation has a part of its binary variables with default values (as in the solution $\bar{X}$. At this point, the solver can be called to solve the restricted formulation looking for the optimal/best solution in the neighborhood $N(\bar{X}, S)$. The new solution is the optimal in that neighborhood, if the prove is optimality is returned by the solver. In the case where the restricted formulation is difficult, then the solver can be forced to stop and return the best solutions found so far. This step stands for the search intensification in VPLS. Finally, the current solution $\bar{X}$ is updated with the new solution. To sum up, VPLS consists of three main steps:

1. Neighborhood definition around a current solution $\bar{X}$.
2. Intensifying the search in the neighborhood.
3. Update the current solution with the new one.

The process can be repeated until a defined stopping criterion is met.

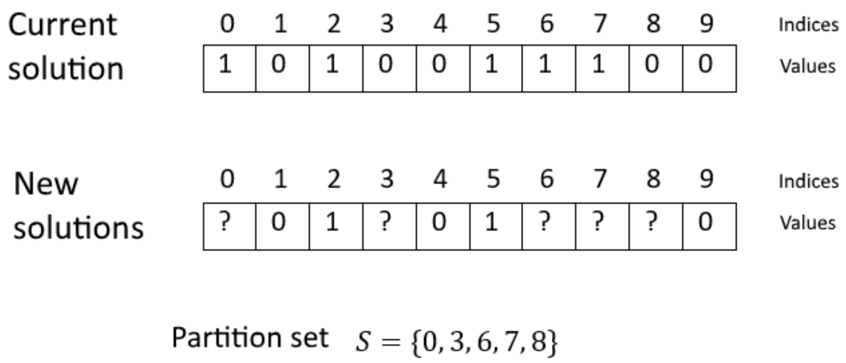


Figure 5.7: Example of VPLS partitioning.

5.3.2.2 VPLS for the GED problem

The first ingredient needed in VPLS is the MILP formulation, which will be the formulation F3. A fundamental question arises when implementing VPLS is how to define the set S? Earlier, the variables in S were referred to as special variables, and this is to indicate that they should be chosen carefully. Choosing them randomly is a possibility, but there is no guarantee that the neighborhoods will contain good and diversified solutions. Therefore, the set S will be defined based on the GED problem and the knowledge and experience picked up when designing the local branching heuristic. It is argued earlier the importance of choosing which variables to include in intensification and diversification mechanisms in LocBra. Particularly, selecting only $x_{i,k}$ variables, representing vertices matchings, is more important than selecting all the variables including the ones representing edges matchings. Moreover, the diversification is more efficient when selecting only the, so-called, *important* variables. The logic behind doing this, is based on GED property 1 that says edges matching can be deduced when determining vertices matching. Regarding the *important* variables, the procedure followed to pick them is based on determining which variables have big impact on the objective function value. The analysis was done over the cost matrices and a local estimation cost computed for each vertices assignment. Theoretically, such ideas have meaning and sounds rational, and experimentally they have shown good results and have increased the efficiency of the method. Similarly, VPLS may achieve good results by following the same directions.

So, back to defining the set S, it is essential to select variables that affect the most the matching (and at the same time the objective function). Basically, only $x_{i,k}$ variables are going to be considered when defining the set S. And next, a procedure based on the notion of spheres is followed to determine S. This procedure needs two input graphs G and G' and an initial solution x^0, and it proceeds as follows:

(i) First, define the list of spheres on graph G of radius δ. For each vertex i in G, the sphere $\mathscr{S}_i$ contains all vertices j that are distant from i with at most δ edges, e.g. if $\delta = 1$, $\mathscr{S}_i$ contains all vertices connected to i with an edge. To compute how many edges are needed to go from one vertex to another, the well-known algorithm Dijkstra is used. This is a polynomial algorithm, designed by Edsger Wybe Dijkstra in 1965 and presented in the book of Cormen (2001). It computes the shortest path

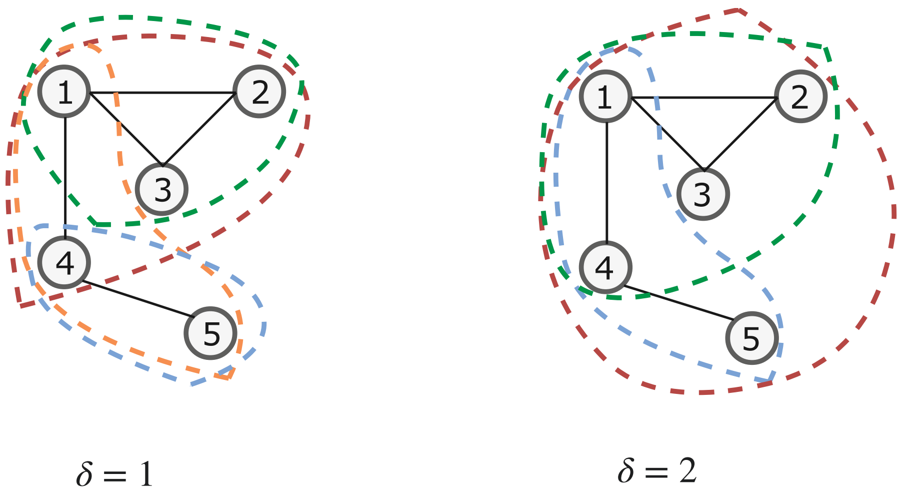


Figure 5.8: Example of generating spheres for a graph. When $\delta = 1$, in red is the sphere for vertex 1, in green is the sphere for vertices 2 and 3, in orange is the sphere for vertex 4 and in blue is the sphere for vertex 5. When $\delta = 2$, in red is the sphere for vertices 1 and 4, in green is the sphere for vertices 2, 3, and in blue is the sphere for vertex 5.

between two vertices in a graph. In fact, each sphere is a subgraph of G, containing all vertices accessible by at most δ edges, plus the edges connecting any two vertices in the sphere. Figure 5.8 shows an example of spheres with different δ values.

(ii) Next, compute a cost for each sphere $\mathscr{S}_i$ based on the assignments in the initial solution x^0. For example, if $\mathscr{S}_1$ for vertex u_1 contains vertices $\{u_1, u_2, u_3, u_4\}$. From the solution x^0, see which vertex k in G' the vertex u_1 is assigned to, and include the cost of this operation $c(u_1 \rightarrow v)$ to the cost of the sphere. The same is done for the rest of the vertices ($\{u_2, u_3, u_4\}$). As well, check the edges that are part of sphere $\mathscr{S}_1$ and find their assignments so their costs are added to the sphere's cost, e.g. if there exists an edge (u_1, u_3) in G, get the cost of the operation $(u_1, u_3) \rightarrow (k, l)$ where $(k, l) \in E'$.

$$c_{\mathscr{S}} = \sum_{\forall i \in \mathscr{S}} c(i \rightarrow assign(i)) + \sum_{\forall (i,j) \in (\mathscr{S} \times \mathscr{S}) \cap E} c((i,j) \rightarrow assign((i,j))), \tag{5.24}$$

with *assign* a function determining vertices/edges assignments based on x^p solution. The result of this step is an array $[c_{\mathscr{S}}]$ storing the costs of all spheres.

(iii) Finally, find the sphere with the highest cost in $[c_{\mathscr{S}}]$ array. Then, for every vertex i in this sphere, add all $x_{i,k}$ variables to the set S.

This procedure is called each time a new feasible solution is found to select the next sphere with the highest cost. An already selected sphere is excluded in the next iteration. This avoids selecting a sphere multiple times, and searching in the same neighborhood several times consecutively.

Once the set S is determined, the next step is to set all variables not in S to their values in the solution x^0 and the rest of the variables are left free in the MILP formulation.

Algorithm 10: VPLS algorithm

```
1  $x^* := \bar{x} := \tilde{x} := undefined$
2  tl := elapsed_time := cons_sol_eq := 0
3  opt := false; first_iter := true
4  $L_{\mathscr{S}} := \emptyset$
1  **Function** VPLS($\delta$, $total\_time\_limit$, $node\_time\_limit$, $UB\_time\_limit$,
2          $cons\_sol\_max$)
       **Output:** $x^*$, opt
3      tl = $UB\_time\_limit$ // Set the time to compute initial solution
4      /* Compute a first solution                                        */
5      status := MIP_SOLVER(tl, $\emptyset$, $\emptyset$, $\tilde{x}$)
6      **if** $status$ = "$opt\_sol\_found$" **then** opt := true; $x^* := \tilde{x}$; **exit**
7      **if** $status$ = "$infeasible$" **then** opt := false; **exit**
8      ImprovedSolution()
9      elapsed_time := tl
10     tl = $node\_time\_limit$ // Set the time for an iteration
11     ComputeSpheres($\delta$)
12     **while** $elapsed\_time < total\_time\_limit$ **and** $cons\_sol\_eq < cons\_sol\_max$ **do**
13         $\mathscr{S}_{max}$ := SelectHighCostSphere()
14         $S$ := ComputePartitionS($\mathscr{S}_{max}$)
15         tl := min{tl, total_time_limit − elapsed_time}
16         /* $S$ contains free variables, and $\bar{x}$ is current solution to fix
              variables not in $S$                                        */
17         status := MIP_SOLVER(tl, $S$, $\bar{x}$, $\tilde{x}$)
18         tl := node_time_limit
19         ImprovedSolution()
20         /* Count consecutive equivalent solutions                      */
21         **if** $first\_iter \neq true$ **and** $f(\tilde{x}) = f(\bar{x})$ **then**
22             $cons\_sol\_eq := cons\_sol\_eq + 1$
23         **else**
24             $cons\_sol\_eq := 0$
25         **end**
26         elapsed_time := elapsed_time + tl
27         first_iter := false
28     **end**
29 **End**
```

Algorithm 11: VPLS helper functions

```
Function ImprovedSolution()
    $\bar{x} := \tilde{x}$
    if $f(\tilde{x}) < f(x^*)$ then $x^* := \tilde{x}$;
End
Function ComputeSpheres($\delta$)
    foreach vertex $i$ in $V$ do
        /* Compute shortest paths towards all other vertices */
        distances[i] := Dijkstra($i$,$G$)
        foreach vertex $j$ in $V$ do
            if distances[i,j] $\leq \delta$ then
                Add vertex $j$ to sphere $\mathscr{S}_i$
            end
        end
        Add $\mathscr{S}_i$ to $L_{\mathscr{S}}$
    end
End
Function SelectHighCostSphere()
    max_cost := 0
    high_cost_sphere := $undefined$
    foreach sphere $s$ in $L_{\mathscr{S}}$ do
        cost := 0
        foreach vertex $i$ in $s$ do
            cost := cost + $c(i, assign(i))$            // get vertex cost assignment
        end
        foreach edge $(i, j)$ in $s$ do
            cost := cost + $c((i, j), assign((i, j)))$    // get edge cost assignment
        end
        if $max\_cost > cost$ then
            high_cost_sphere := $s$
            max_cost := cost
        end
    end
    return high_cost_sphere
End
Function ComputePartitionS($\mathscr{S}_{max}$)
    $S := \emptyset$
    foreach vertex $i$ in $\mathscr{S}_{max}$ do
        foreach vertex $k$ in $V'$ do
            Add $x_{i,k}$ to $S$
        end
    end
    return $S$
End
```

The solver will solve the restricted MILP formulation trying to find the best solution by finding the right values for variables in S. This is the intensification phase, that will result in a new solution x^1. Again, the spheres costs are recomputed based on x^1, and the one with the highest cost will be selected for the next iteration. An iteration, then, consists of three steps: computing and selecting the sphere to define S, defining the neighborhood based on S, intensifying the search in the neighborhood. This process is repeated until reaching some defined stopping criterion.

A detailed algorithmic presentation of VPLS heuristic is provided in Algorithm 10. Functions for spheres and set S computations are given in Algorithm 11. The input parameters for VPLS are:

1. δ, is the radius of spheres.
2. *total_time_limit*, is the total running time allowed for VPLS before stopping.
3. *node_time_limit*, is the maximum running time given to the solver to solve the restricted MILP formulation.
4. *UB_time_limit*, is the running time allowed to the solver to compute an initial solution.
5. *cons_sol_max*, serves as a stopping criterion: VPLS stops when the number of consecutive intensification steps finding solutions with the same objective function values is equal to this parameter.

There are two parameters serving as stopping criteria, so VPLS heuristic halts when at least one of them is met:

(i) the total execution time exceeds the *total_time_limit*, or

(ii) the number of consecutive intensification steps done with solutions with the same objective function values exceeds *cons_sol_max*.

The output of the algorithm is the best solution found (x^*) along the search, and a flag to indicate whether it has been proved to be optimal or not (*opt*). The initial solution x^0 used by VPLS is obtained by solving F3 formulation within a time limitation of *UB_time_limit* seconds. If at this point, F3 is solved to optimality or no feasible solution has been found, the heuristic halts and returns the available solution and/or the status. Otherwise, the current solution $\bar{x}$ is set to the solution found and the exploration begins. The spheres are computed by calling the function *ComputeSpheres* (line 11), before starting the loop. Each iteration starts by calling the function *SelectHighCostSphere* (line 13), to compute the spheres costs based on $\bar{x}$ solution, and to find the sphere with the highest cost $\mathscr{S}_{max}$. Next, the set S is defined based on $\mathscr{S}_{max}$ by calling the function *ComputeParitionS* at line 14. Then, the solver is called to solve the new formulation after setting free the variables in S and fixing the other variables to their values as in $\bar{x}$. A new solution $\tilde{x}$ is returned by the solver, and $\bar{x}$ is updated to the new solution. Also, x^* is updated if the objective function value of $\tilde{x}$ is better than the best one. The last step

Algorithm 12: Compute δ in VPLS heuristic

```
1  Let $distanceMat$ be a matrix // to store distances between all vertices
1  Function ComputeDelta($R$, $P$, $X$)
       Output: $\delta$
2      foreach $vertex\ i\ in\ V$ do
3          $distanceMat[i] = $ Dijkstra($i$,$G$) // compute all distances rooted
            from $i$ towards other vertices
4      end
5      Let $maxDistance = Max(distanceMat)$ // compute max value in the
        matrix
6      Let $H = ComputeHistogram(distanceMat)$ // compute histogram of
        values
7      /* cover quarter of the values in the matrix                    */
8      Let $sum = 0$
9      for $i \leftarrow 1$ to $maxDistance$ by 1 do
10         if $(sum + H[i]) < \frac{|V| \times |V'|}{4}$ then
11             $sum = sum + H[i]$
12         else
13             break
14         end
15     end
16     $\delta = i$
17 End
```

is to count the consecutive solutions with equal objective function values and update the elapsed time (lines 21 to 26).

To evaluate the performance of VPLS, the same experimentation protocol is used as when evaluating LocBra heuristic:

1. Effectiveness of VPLS w.r.t. competitor heuristics.

2. Effectiveness of VPLS w.r.t. an exact method.

As in LocBra, tuning VPLS parameters is based on the same concept explained in Section 4.4.4. Parameters *total_time_limit*, *UB_time_limit* and *node_time_limit* are considered as *tuned* parameters. There values are set based on preliminary tests for each database. The parameter *cons_sol_max* is considered as a *pre-fixed* parameter, and is set to 5 in all the experiments and for all databases. The value 5 is chosen based on the results of several tests on different instances selected from different databases. This value seemed to be reasonable, because for most instances getting solutions with the same objective function value more than 5 times, was enough to stop VPLS. Moreover, this solution has turned out to be the optimal one. At last, the δ parameter is set in an adaptive fashion depending on the current instance and the graph structures. The snippet of code in Algorithm 12 presents the procedure to compute δ. The main idea is to select δ that leads to generating spheres covering a quarter of the vertices in the graph. Based on the experiments done, the values of δ were varying between 2 and 4, and such values have returned good results.

Common configuration in all experiments. VPLS algorithm is implemented in C language. The solver CPLEX 12.7.1 is used to solve the MILP formulations. Experiments are ran on a machine with Windows $7x64$, Intel Xeon $E5$ 2.30 GHz, 4 cores and 8 GB of RAM. CPLEX solver is configured to use single thread, and the rest of the parameters are set to default.

5.3.2.3 Effectiveness of VPLS w.r.t. competitor heuristics

These experiments answer the following question: which heuristic is the best minimizer?

Methods. The heuristics chosen in the evaluation are:

- **i-** *CPLEX-t* is the solver CPLEX ran on F3 formulation with t seconds as a time limit.
- **ii-** *BeamSearch-α*, the BeamSearch heuristic with α the beam size.
- **iii-** *SBPBeam-α*, the SBPBeam heuristic with α the beam size.
- **iv-** *IPFP-it*, the IPFP heuristic with it the maximum number of iterations.
- **v-** *GNCCP-d*, the GNCCP heuristic with d the quantity to be subtracted from the ζ variable at each iteration. ζ is the variable that controls the concavity and convexity of the objective function of the QAP model solved by GNCCP heuristic.

Comparison indicators. All heuristics are executed on different databases and for all of them, the following indicators are computed: t_{min}, t_{avg}, and t_{max} are the minimum, average and maximum CPU times in seconds over all instances. Correspondingly, d_{min}, d_{avg}, and d_{max} are the deviations of the solutions obtained by one heuristic, from the best solutions found by all heuristics. The deviations are computed based on Eq. 4.11 and are expressed in percentage. Lastly, η_I is the number of instances for which a given heuristic has found the best solutions.

Evaluations on PROTEIN database. This database is selected to evaluate VPLS heuristic on GED instances.

Default versions. The parameters are set to the following values:

VPLS	$cons\_sol\_max = 5$, $total\_time\_limit = 10s$, $node\_time\_limit = 2s$, $UB\_time\_limit = 4s$,
CPLEX-t	$t = 10$
CPLEX-LocBra-t	$t = 9$
BeamSearch-α	$\alpha = 5$
SBPBeam-α	$\alpha = 5$
IPFP-it	$it = 10$
GNCCP-d	$d = 0.1$

Table 5.19: VPLS vs. heuristics on PROTEIN instances

	S	20	30	40	mixed
VPLS	t_{min}	0.05	0.09	0.16	0.05
	t_{avg}	3.30	5.47	6.59	5.31
	t_{max}	5.79	9.98	9.98	9.98
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	0.12	**0.10**	**0.37**	**0.11**
	d_{max}	1.08	1.31	1.25	0.84
	η_I	73	**65**	30	**64**
CPLEX-10	t_{min}	0.05	0.06	0.16	0.05
	t_{avg}	5.86	8.54	8.84	8.24
	t_{max}	10.03	10.05	10.06	10.03
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	**0.08**	0.11	0.55	0.16
	d_{max}	0.84	0.75	2.47	3.46
	η_I	**79**	60	30	60
BeamSearch-5	t_{min}	0.00	0.01	0.01	0.00
	t_{avg}	**0.02**	**0.07**	**0.10**	**0.06**
	t_{max}	0.04	0.13	0.22	0.13
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	5.61	3.33	2.86	5.57
	d_{max}	123.86	155.56	7.29	26.54
	η_I	10	11	10	10
SBPBeam-5	t_{min}	0.26	1.03	2.66	0.36
	t_{avg}	0.37	1.54	3.76	1.75
	t_{max}	0.54	2.26	5.05	4.39
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	3.18	5.57	3.41	2.74
	d_{max}	5.36	155.56	5.58	4.78
	η_I	12	10	10	10
IPFP-10	t_{min}	0.02	0.05	0.04	0.03
	t_{avg}	0.09	0.27	0.59	0.31
	t_{max}	0.13	0.38	0.88	0.81
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	1.04	0.97	1.12	1.02
	d_{max}	3.25	3.38	2.65	2.51
	η_I	23	13	11	11
GNCCP-0.1	t_{min}	1.32	4.50	13.93	1.45
	t_{avg}	2.05	7.21	23.17	9.36
	t_{max}	2.47	10.45	30.51	23.26
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	0.21	0.19	1.65	1.36
	d_{max}	1.08	0.81	156.60	117.73
	η_I	55	48	**72**	50

Table 5.19 shows the results of this experiment. *VPLS* has the smallest average deviations for all subsets, except subset 20 where it is outperformed by *CPLEX-10*, with a difference of 0.04%. On subset 40, despite the average deviation difference between *VPLS* and *GNCCP-0.1*, the latter has found best solutions two times more than *VPLS* (30 against 72). Nevertheless, *VPLS* wins w.r.t. deviations, and *GNCCP-0.1* in the worst case reaches 156% while *VPLS* reaches a max of 1.25%. The beam-search based methods are the worst in terms of solutions quality. Yet, *BeamSearch-5* is the fastest heuristic. An important remark here is the difference between t_{avg} values of *VPLS* and *CPLEX-10*, where the former is always faster.

Extended versions. Another version of LocBra is considered in this version with a maximum running time of $30s$. The rest of the heuristics are configured, as well, to reach almost the same running time.

$VPLS$	$cons\_sol\_max = 5$, $total\_time\_limit = 30s$, $node\_time\_limit = 6s$, $UB\_time\_limit = 12s$,
CPLEX-t	$t = 30$
BeamSearch-α	$\alpha = 1500$
SBPBeam-α	$\alpha = 30$
IPFP-it	$it = 500$
GNCCP-d	$d = 0.09$

The results of this experiment are reported in Table 5.20. There is not a clear winner here. On easy instances (subsets 20 and 30), *CPLEX-30* seems to have computed the best solutions, which can be seen by looking at the d_{avg} and η_I values. On hard instances (subset 40), *GNCCP-0.9* has the best average deviation at 0.12%, with the best η_I at 63, followed by *VPLS* with $d_{avg} = 0.34\%$ and $\eta_I = 40$. On medium instances (subset *mixed*), *VPLS* is the best in terms of average deviation and number of best solutions. Regarding the average running time, *IPFP-500* is the fastest on easy instances, while *VPLS* is the fastest on hard instances, and on medium instances *SBPBeam-30* is the fastest. Consequently, *VPLS* outperforms the existing heuristics on these instances, except on hard instances where *GNCCP-0.09* is better in terms of solutions quality. Moreover, *VPLS* is never the slowest heuristic and it is faster than the others on medium and hard instances.

Conclusion. On PROTEIN instances, VPLS has shown very good performance, beating BeamSearch, SBPBeam and IPFP heuristics. However, GNCCP was able on hard instances to perform slightly better in default and extended versions. One more thing, VPLS results are very close to the default behavior of CPLEX. But, VPLS heuristic is faster than CPLEX in converging towards good solutions.

More results on other databases can be found in Appendix B, Section B.4.1. The conclusions of those experiments are:

- **MUTA database:** VPLS heuristic has achieved better results than existing heuristics in terms of solutions quality, but it does not outperform the default behavior of the solver when solving F3 formulation.
- **HOUSE-REF database:** VPLS heuristic, in the default version with a maximum running time of $10s$, has performed better than all existing heuristics with their

Table 5.20: VPLS vs. heuristics with extended running time on PROTEIN instances

	S	20	30	40	mixed
VPLS	t_{min}	0.05	0.11	0.19	0.06
	t_{avg}	7.17	12.54	**15.03**	12.04
	t_{max}	13.77	29.73	29.97	29.95
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	0.06	0.08	0.27	**0.07**
	d_{max}	0.84	0.75	1.25	0.70
	η_I	82	71	41	**74**
CPLEX-30	t_{min}	0.03	0.11	0.17	0.05
	t_{avg}	11.56	24.09	26.18	22.33
	t_{max}	30.03	30.01	30.05	30.03
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	**0.03**	**0.05**	0.34	0.09
	d_{max}	0.84	0.48	1.68	3.08
	η_I	**92**	**80**	35	70
BeamSearch-1500	t_{min}	0.00	0.01	0.01	0.00
	t_{avg}	3.94	13.72	23.72	13.42
	t_{max}	5.85	20.93	34.15	25.98
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	1.64	7.30	1.85	4.44
	d_{max}	21.88	311.11	6.79	26.01
	η_I	12	10	10	10
SBPBeam-30	t_{min}	1.57	6.16	15.28	1.63
	t_{avg}	2.15	8.95	23.05	**8.79**
	t_{max}	3.06	13.33	31.71	21.32
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	3.21	5.62	3.46	2.79
	d_{max}	5.66	155.56	5.89	4.78
	η_I	12	10	10	10
IPFP-500	t_{min}	0.01	0.04	0.13	0.02
	t_{avg}	**2.14**	**7.36**	22.26	9.01
	t_{max}	5.03	12.99	31.93	27.64
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	0.79	0.79	0.73	0.76
	d_{max}	2.71	2.82	2.10	2.22
	η_I	27	15	13	12
GNCCP-0.09	t_{min}	1.47	4.58	15.84	1.52
	t_{avg}	2.19	7.69	24.34	9.75
	t_{max}	3.32	10.96	29.79	28.58
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	0.21	0.21	**0.12**	0.24
	d_{max}	1.12	0.81	0.95	1.13
	η_I	50	32	**63**	31

default parameters. However, in the extended version, VPLS came second, right after GNCCP heuristic, which surprisingly was able to solve HOUSE-REF instances efficiently.

General conclusions based on the evaluations. All the above experiments have shown the efficiency of VPLS heuristic in solving the GED problem. It outperforms the existing heuristics in general, except in few cases such as subset 40 in PROTEIN database and HOUSE-REF extended version, where GNCCP was able to perform slightly better. VPLS has also performed better than CPLEX solver in some cases such as on HOUSE-REF instances. Moreover, an important remark concerning the running time of VPLS that was better than the running time of other heuristics and CPLEX solver. Indeed, not only in terms of solution quality, VPLS is also able to compete with other heuristics in terms of running time. Table 5.21 summarizes the experiments conclusions.

Table 5.21: Summary of VPLS comparison w.r.t. competitor heuristics

	Database	Solutions quality	Speed
Default versions	PROTEIN	VPLS	BeamSearch
	MUTA	CPLEX	BeamSearch
	HOUSE-REF	VPLS	BeamSearch
Extended versions	PROTEIN	CPLEX/VPLS	IPFP/VPLS
	MUTA	VPLS	VPLS/GNCCP
	HOUSE-REF	GNCCP	IPFP

5.3.2.4 Effectiveness of VPLS w.r.t. an exact method

This experiment will answer the following question: how close the VPLS solutions are from the optimal solutions?

Methods. The exact approach in these experiments consists in solving a GED MILP formulation using CPLEX. This exact method was already executed on several databases when testing LocBra heuristic. So, the results obtained in Section 5.3.1.2 are used in this experiment.

Comparison indicators. The following indicators are computed for each database: t_{min}, t_{avg}, and t_{max}, which are respectively the minimum, average and maximum CPU times in seconds over all instances. Correspondingly, d_{min}, d_{avg}, and d_{max} are the deviations (in percentage) of the solutions obtained by VPLS, from the optimal or best solutions found. The deviation is computed by using Eq. 4.11. In addition, η_I is the number of optimal solutions found, and η'_I is the number of solutions found by VPLS that are equal to the optimal or best known ones. At last, η''_I is the number of solutions computed by VPLS that are better than the best known solutions, when *CPLEX* was not able to find optimal solutions.

Table 5.22: VPLS vs. Exact solution on PROTEIN instances

	CPLEX-900				VPLS								
S	t_{min}	t_{avg}	t_{max}	η_I	t_{min}	t_{avg}	t_{max}	d_{min}	d_{avg}	d_{max}	η_I	η'_I	η''_I
20	0.03	26.83	215.59	100	0.05	**7.17**	**13.77**	0.00	0.12	1.12	62	**75**	0
30	0.09	132.00	856.79	100	0.11	**12.54**	**29.73**	0.00	0.20	0.94	19	**42**	0
40	0.14	575.70	900.52	63	0.19	**15.03**	**29.97**	**-0.28**	0.45	1.53	12	**17**	**2**
mixed	0.03	180.36	900.66	96	0.06	**12.04**	**29.95**	**-2.64**	0.16	0.99	23	**40**	**2**

Evaluations on PROTEIN database. VPLS parameters values are: $cons\_sol\_max = 5$, $total\_time\_limit = 30s$, $node\_time\_limit = 6s$, $UB\_time\_limit = 12s$.

Table 5.22 shows the results of this experiment. There is a huge difference between the average running time of VPLS and *CPLEX-900*, that reaches more than $500s$ on subset 40. Yet, the average deviation of VPLS on that subset is 0.45%, which is very small. And the minimum deviation is -0.28% indicating that VPLS has found better solutions than *CPLEX-900*. It is the same case on the other subsets, where the average deviations are very small showing closeness between the solutions of both methods. However, the difference is that VPLS was able to compute almost good solutions in a maximum time of $30s$, where *CPLEX-900* reaches $900s$ in most cases.

Conclusion. On PROTEIN database, VPLS was able to compute in $30s$ the same and sometimes better solutions than CPLEX in $900s$. This proves the closeness of VPLS solutions from the optimal/best ones.

Additional experiments were conducted on MUTA and HOUSE-REF database, and the results are discussed in Appendix B, Section B.4.2. Here are the conclusions of those experiments:

- **MUTA database:** The solutions computed by VPLS are relatively good compared to the optimal/best ones found by CPLEX. The average deviations vary between 0% and 12% as the graph size increases. Even though the running time of VPLS is set to $900s$, the heuristic does not consume all the time. On hard instances the average time is at $504s$.

- **HOUSE-REF database:** VPLS has computed in $100s$ better solutions for 3 instances than CPLEX in $900s$. The average deviation is relatively small at 4%. In addition, VPLS is very fast compared to CPLEX, where the $t_{avg} = 68s$ for VPLS against $417s$ for CPLEX.

General conclusions based on the evaluations. To sum up and based on the analysis and the interpretation of the results, VPLS is an effective heuristic that provides good quality solutions in a short amount of time. The results have revealed the closeness of the solutions computed by VPLS from the solutions computed by CPLEX in $900s$ on PROTEIN and HOUSE-REF databases. However, on MUTA instances where the solutions were computed during 10 hours, VPLS has an average deviation of 6% from the optimal and best known ones, which is considerably not high. The results prove that VPLS is capable of computing solutions very close to the optimal/best ones.

5.3.2.5 VPLS vs. LocBra

VPLS and LocBra are matheuristics that share features, such as neighborhood definitions and intensification. However, each one uses a different approach to define the neighborhood. In this section, the results obtained in Section 5.3.1.1 for LocBra and Section 5.3.2.3 for VPLS are compared together to study the performances of the heuristics.

Evaluations on PROTEIN database. The results in Table 5.23 show that the best solutions were found by LocBra for all subsets. Yet, the solutions computed by VPLS are very close with a difference less than 0.1% on average. Moreover, VPLS has the same average deviation (0.06%) as LocBra on subset *mixed*, and also it has a better maximum deviation of 0.56% against 2.90% by LocBra. VPLS is two times faster than LocBra on the average running time.

Table 5.23: VPLS vs. LocBra on PROTEIN instances

	VPLS							LocBra						
S	t_{min}	t_{avg}	t_{max}	d_{min}	d_{avg}	d_{max}	η_I	t_{min}	t_{avg}	t_{max}	d_{min}	d_{avg}	d_{max}	η_I
20	0.05	**7.17**	**13.77**	0.00	0.05	1.12	88	0.08	14.28	30.08	0.00	**0.00**	0.00	**100**
30	0.11	**12.54**	29.73	0.00	0.08	0.75	68	0.12	24.77	30.11	0.00	**0.01**	0.30	**91**
40	0.19	**15.03**	29.97	0.00	0.14	0.84	55	0.22	26.38	30.31	0.00	**0.05**	0.55	**80**
mixed	0.06	**12.04**	29.95	0.00	**0.06**	**0.56**	75	0.12	23.57	30.37	0.00	**0.06**	2.90	**86**

Concolusion. VPLS is two times more faster than LocBra, with a deviation gap of 0.1% on average regarding the solutions quality.

The same comparison is done for MUTA and HOUSE-REF databases (see Appendix B, Section B.4.3). The conclusions are the following:

- **MUTA database:** LocBra is better than VPLS in solving MUTA instances. The difference, however, is not that important with a maximum of 5% on the average. In terms of running time, VPLS is way faster than LocBra with a difference reaching the 500s on hard instances (subset 60).

- **HOUSE-REF database:** VPLS was able to compute better solutions than LocBra for 33 instances. The average running time of VPLS is also better than the average running time of LocBra.

General conclusions based on the evaluations. Both heuristics are good in solving GED instances and finding good solutions. They both perform better than the heuristics available in the literature. However, there is a slight difference in terms of solutions quality in favor of LocBra over VPLS. This difference comes at a high price paid by LocBra, because VPLS is two times - and sometimes more - faster than LocBra. On the contrary to LocBra, VPLS does not consume all the given time when solving the problem. VPLS tends to stop because of the stopping criterion *cons_sol_max*. This is, actually, a key advantage of VPLS over LocBra.

5.3.2.6 Stress test on heuristics

This experiment aims at testing the limits of the heuristics, by studying the evolution of their solutions with time. Each heuristic is executed on the same instance with different range of time values, starting from few seconds to reach thousands of seconds. The results will show the behavior of each heuristic when pushed to the edge. Which heuristic is the fastest and yet the most accurate? Which heuristic is the best with high running time?

Methods. The following heuristics are involved in this experiment:

- **i-** *VPLS*, it has a parameter to control its running time, i.e. *total_time_limit*.
- **ii-** *LocBra*, it has a parameter to control its running time, i.e. *total_time_limit*.
- **iii-** *F3*, this is the solver CPLEX ran on F3 formulation with a maximum running time.
- **iv-** *F2*, this is the solver CPLEX ran on F2 formulation with a maximum running time.
- **v-** *IPFP*, this heuristic does not have a running time parameter, instead it has a parameter for the maximum number of iterations.
- **vi-** *GNCCP*, this heuristic has a parameter d that changes ζ the variable to control the concavity and convexity of the objective function of the QAP model. When ζ reaches a maximum the heuristic halts. So d parameter controls the speed of the method.

Databases and instances. The instances of this experiment are selected from MUTA database, because it contains many subsets of graphs with different sizes. The instances are split into three groups: hard, medium and easy. Table 5.24 presents the details of instances in each group. There are 15 instances in total, 5 instances per group. The objective function value of the optimal/best solution is given for each instance. These values are computed by CPLEX solving JH formulation with a time limit of 10 hours. Note that, the CPU time reaches thousands of seconds on hard instances. This serves better the purpose of this experiment and enables reaching thousands of seconds when executing the heuristics. That is why the instances were picked from MUTA database and not from other databases.

Comparison indicators and settings. The following indicators are computed for each heuristic: t_{avg} is the average CPU times in seconds over all instances in a group. d_{avg} is the average deviation of the solutions obtained by one heuristic, from the optimal/best solutions. The deviations are computed based on Eq. 4.11 and are expressed in percentage. All heuristics with a parameter to control their execution times are launched with the following values: $1, 2, 5, 10, 30, 60, 120, 180, 300, 900$ and 3600 seconds. IPFP is executed with the following values of iterations: $10, 50, 100, 500, 1000, 5000, 10000$ and 20000. Finally, GNCCP is executed with the following parameter values: $0.1, 0.09, 0.08, 0.07, 0.06, 0.05, 0.04$ and 0.03.

Table 5.24: Stress test instances details.

	G	G'	Size	Opt/Best	Optimal	CPU time (s)
Hard	molecule_3450	molecule_4214	70	86.625	FALSE	13499.93
	molecule_4018	molecule_4224	70	99	FALSE	24185.26
	molecule_3214	molecule_2702	70	68.75	TRUE	10125.29
	molecule_1731	molecule_3214	70	38.775	TRUE	17042.60
	molecule_4214	molecule_1731	70	105.05	FALSE	10499.79
Medium	molecule_2944	molecule_3210	30	58.025	TRUE	44.27
	molecule_3574	molecule_3168	30	45.375	TRUE	1119.13
	molecule_3763	molecule_3291	40	42.35	TRUE	2635.34
	molecule_3584	molecule_3817	40	57.75	TRUE	2025.58
	molecule_4281	molecule_3291	40	91.85	TRUE	1000.18
Easy	molecule_3486	molecule_3601	10	22.275	TRUE	0.16
	molecule_3875	molecule_3676	10	22.275	TRUE	0.23
	molecule_3131	molecule_3074	20	46.75	TRUE	0.53
	molecule_3220	molecule_3146	20	31.075	TRUE	0.80
	molecule_3225	molecule_3048	20	47.575	TRUE	0.74

Table 5.25: Stress test on hard instances for MILP-based heuristics

	total_time	1	2	5	10	30	60	120	180	300	900	3600
VPLS	t_{avg}	**0.19**	**1.18**	**4.37**	**9.31**	**29.32**	**55.25**	**102.14**	**155.96**	**216.62**	**500.04**	**1174.23**
	d_{avg}	**1175.68**	1175.68	111.68	71.13	37.15	33.12	25.66	23.84	23.86	20.83	16.28
LocBra	t_{avg}	1.06	2.05	5.03	10.02	30.03	60.03	120.00	180.00	299.96	894.36	3577.55
	d_{avg}	**1175.68**	789.26	**88.92**	88.92	45.09	**29.27**	**25.60**	**21.15**	**13.77**	**11.30**	**6.44**
F3	t_{avg}	1.14	2.16	5.13	10.16	30.15	60.16	120.18	180.13	300.20	900.88	3606.52
	d_{avg}	**1175.68**	**94.13**	94.13	**48.80**	40.66	36.43	31.94	26.42	21.95	20.03	13.81
F2	t_{avg}	1.11	2.18	5.14	10.18	30.17	60.17	120.16	180.16	300.19	900.24	3605.42
	d_{avg}	**1175.68**	1175.68	767.45	90.16	**30.76**	30.09	28.67	25.90	24.03	17.79	7.51

Table 5.26: Stress test on hard instances for IPFP and GNCCP

	it	10	50	100	500	1000	5000	10000	20000
IPFP	t_{avg}	**1.28**	**3.92**	**7.29**	**34.47**	**68.50**	312.13	509.27	**536.43**
	d_{avg}	**24.88**	**18.93**	**19.62**	**19.31**	**19.31**	**19.95**	**19.64**	**19.95**
	d	0.1	0.09	0.08	0.07	0.06	0.05	0.04	0.03
GNCCP	t_{avg}	111.90	177.78	179.78	136.70	360.17	**249.55**	**501.71**	644.10
	d_{avg}	26.61	23.94	19.85	27.13	23.41	25.74	21.24	21.56

Evaluations on hard instances. The results of MILP-based heuristics are presented in Table 5.25, without IPFP and GNCCP heuristics, because they do not share the same parameters with other heuristics. IPFP and GNCCP results are shown in Table 5.26. All MILP-based heuristics have very high d_{avg} that reaches 1175% when $total\_time = 1s$. Increasing the $total\_time$ to $2s$, F3's average deviation drops to 94.13%, while deviations of other heuristics remain high. As the time increases, the deviation starts to decrease for all heuristics, and LocBra reaches the smallest d_{avg} for the range of times between $60s$ till $3600s$. VPLS, F3 and F2 interchangeably switches positions, so VPLS has better d_{avg} when $total\_times$ is set to 120 and $180s$. F3 is the best ($d_{avg} = 21.95\%$) when $total\_time = 300s$. And F2 has better average deviations after LocBra when $total\_times$ is set to $60s$, $900s$ and $3600s$. In terms of average running times, VPLS is the fastest, because it does not consume all the giving running time like the others. Based on Table 5.26, when t_{avg} between $1s$ and $68s$, IPFP has smaller average deviation than LocBra. However, for higher running times LocBra starts to compute more accurate solutions than IPFP. On the other hand, GNCCP seems to be very slow, even when its parameter set to the default value (0.1), where its $t_{avg} = 111s$. Also, GNCCP is outperformed by IPFP and LocBra.

Conclusion. So, IPFP is a very fast heuristic that converges in short amount of time, which is not the case of MILP-based heuristics. However, LocBra solves more efficiently hard instances when granting it more time.

Table 5.27: Stress test on medium instances for MILP-based heuristics

	$total\_time$	1	2	5	10	30	60	120	180	300	900	3600
VPLS	t_{avg}	**0.12**	**1.09**	**3.99**	**7.56**	**16.74**	**26.20**	**49.14**	**65.74**	**99.65**	**162.09**	**259.38**
	d_{avg}	691.68	63.49	37.02	**29.28**	28.50	27.36	25.29	24.94	24.94	24.57	22.23
LocBra	t_{avg}	1.01	1.78	4.17	8.17	24.01	48.15	95.51	144.16	240.19	716.13	2880.15
	d_{avg}	**56.21**	54.35	**31.99**	31.06	**23.59**	**23.80**	25.35	**23.80**	**22.51**	**20.95**	**19.81**
F3	t_{avg}	1.02	1.82	4.21	8.25	24.23	48.26	96.55	144.83	241.43	711.26	1955.51
	d_{avg}	62.04	**36.39**	35.82	34.10	29.70	29.34	28.56	27.42	25.86	23.80	20.59
F2	t_{avg}	1.04	1.99	4.35	8.36	24.38	48.39	96.35	144.34	240.44	721.67	2946.42
	d_{avg}	266.63	60.25	33.98	31.65	25.93	25.93	**25.15**	25.15	23.43	23.08	**19.81**

Table 5.28: Stress test on medium instances for IPFP and GNCCP

	it	10	50	100	500	1000	5000	10000	20000
IPFP	t_{avg}	**0.19**	**0.67**	**1.26**	**5.94**	**11.96**	48.60	85.94	93.07
	d_{avg}	31.93	27.73	27.73	**27.73**	27.73	**27.73**	**27.73**	27.73
	d	0.1	0.09	0.08	0.07	0.06	0.05	0.04	0.03
GNCCP	t_{avg}	10.27	9.23	18.30	11.80	31.62	**17.94**	**31.68**	**56.19**
	d_{avg}	**27.89**	**26.18**	**27.32**	28.09	**26.96**	30.32	28.25	**27.32**

Evaluations on medium instances. Based on the results shown in Table 5.27, LocBra has the smallest average deviations for all $total\_time$ values, except for $2s$, $10s$ and $120s$. Clearly, the gap is reduced between the deviations of all heuristics. These instances are easier to solve than the hard ones. The gap varies between 0% and 2%, so other heuristics are relatively good as LocBra. VPLS is faster as on hard instances. From Table 5.28, it can be seen that IPFP and GNCCP have smaller average deviations than LocBra, when their running times are less than $12s$. Beyond the $12s$, LocBra has d_{avg} smaller than the 27.73% deviation of IPFP and the deviations of GNCCP. The GNCCP heuristic converges way faster on these instances than on hard instances, with a maximum $t_{avg} = 56.12s$.

Conclusion. Similar to the evaluation on hard instances, IPFP is very suitable for small running times, but LocBra achieves better results with higher running times.

Table 5.29: Stress test on easy instances for MILP-based heuristics

	total_time	1	2	5	10	30	60	120	180	300	900	3600
VPLS	t_{avg}	**0.08**	**0.71**	1.20	1.54	1.61	1.56	1.63	1.58	1.56	1.57	1.60
	d_{avg}	503.29	5.29	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
LocBra	t_{avg}	0.66	1.25	1.45	1.13	1.26	1.11	1.11	1.11	1.08	1.09	1.10
	d_{avg}	**3.17**	2.47	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
F3	t_{avg}	0.61	1.01	1.24	1.24	1.24	1.27	1.25	1.22	1.22	1.24	1.26
	d_{avg}	4.60	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
F2	t_{avg}	0.63	0.83	**0.82**	**0.83**	**0.81**	**0.80**	**0.78**	**0.76**	**0.80**	**0.81**	**0.77**
	d_{avg}	4.22	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**

Table 5.30: Stress test on easy instances for IPFP and GNCCP

	it	10	50	100	500	1000	5000	10000	20000
IPFP	t_{avg}	**0.02**	**0.05**	**0.09**	**0.25**	**0.44**	**2.06**	4.24	8.51
	d_{avg}	6.40	6.40	6.40	6.40	6.40	6.40	6.40	6.40
	d	0.1	0.09	0.08	0.07	0.06	0.05	0.04	0.03
GNCCP	t_{avg}	0.95	0.83	1.20	1.49	2.03	2.34	**2.41**	**3.88**
	d_{avg}	**3.51**	**3.52**	**5.28**	**3.88**	**5.28**	**5.28**	**4.59**	**5.28**

Evaluations on easy instances. On easy instances and based on Table 5.29, the MILP-based heuristics have succeeded in finding the optimal solutions all the times, except when $total\_time$ is set to $1s$. The maximum of $0.83s$ was sufficient for F2 formulation to compute all optimal solutions. So, F2 is the fastest among the heuristics. Note that also with $total\_time = 2s$, VPLS and LocBra were not able to compute the optimal solutions. The reason could be the time for computing the UB and exploring the neighborhoods are very small. Based on Table 5.30, surprisingly IPFP and GNCCP did not find the optimal solutions no matter how much are the values of their parameters and their running times increase.

Conclusion. On easy instances, MILP-based heuristics are very effective and better than IPFP and GNCCP, they are even faster than GNCCP. IPFP is the fastest (with small *it* values), but it was not able to find the optimal solutions, and it has found solutions off by 6.40% from the optimal ones.

To get a better visibility on the performances of the heuristics, especially when their running time increases, a chart is drawn showing the average deviations trends w.r.t. the running times (Figure 5.9). The charts show the average deviations of all heuristics on hard (chart a) and medium (chart b) instances. Chart (a) shows that IPFP stops at a running time a bit higher than $500s$, and at that moment it has better average deviations

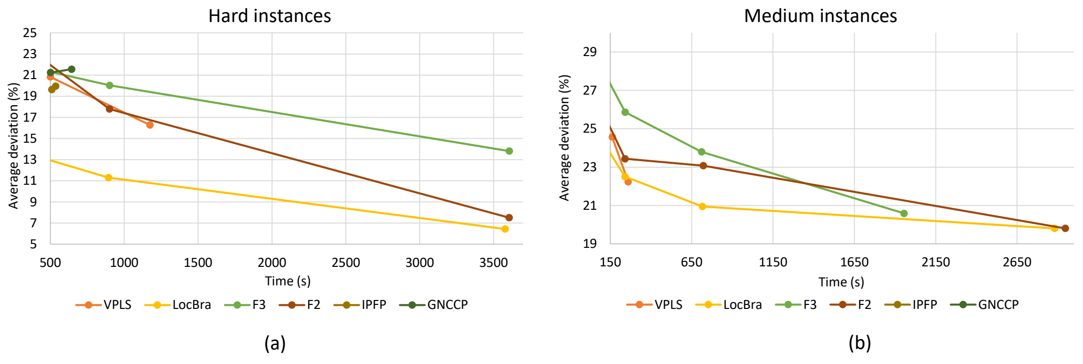


Figure 5.9: Average deviations vs running time for all heuristics, on hard (chart a) and medium (chart b) instances.

than all heuristics, except LocBra. GNCCP stops at a running time around 600s, but it is outperformed by all the other heuristics. Then, VPLS stops somewhere between 1000s and 2000s, where its line is slightly lower than F2's line. The line of F2 goes down faster than F3's line, which means F2 is better when the time increase. The yellow line is for LocBra and it is the lowest for all the time range from 500s till 3600s. This means that LocBra has smaller deviations and solutions closer to the optimal/best ones. On medium instances shown in chart (b), VPLS, until it stops, is better than F2 and F3. Then, the line of F2 is lower than F3, until reaching $\approx$ 1200s. Again, LocBra has the lowest average deviations because its line is always below the other lines. IPFP and GNCCP are not shown here because their running times are not in the selected range.

General conclusions based on the evaluations. Three main conclusions can be taken from the stress test experiment. The first is that IPFP is a very convenient and fast heuristic when dealing with hard and large instances. LocBra and VPLS are not able to compute good solutions in a very short amount of time (1 to 2 seconds). The reason is that CPLEX requires extra time to perform pre-processing steps before solving the MILP formulations. The second conclusion is that VPLS is a MILP-based heuristic with a convenient compromise between running time and solutions quality compared to existing heuristics. The last conclusion, LocBra and VPLS matheuristics are able to compete with existing heuristics and obtain very good results when solving the GED problem.

5.4 Summary and contributions

This chapter has been focused on solving the GED problem in both exact and heuristic contexts, answering the main declared objectives in this thesis. The exact methods consists of designing new MILP formulations, which is the first part of this chapter. The second part consists of designing matheuristics dedicated to the GED problem.

In the exact context three formulations are proposed: VbM is the first proposed formulation inspired by F2 and based on the idea of reducing the number of binary variables.

This has led to introducing new constraints in the formulation. VbM has proven experimentally to be poor when compared to the best MILP formulation F2. The main reason of the poor performance is related to the new constrains that have increased the complexity of the formulation. The second attempt is founded based on the idea of modeling multiple vertices and edges matchings with a single object. This has resulted in an expanded formulation, denoted by ObM, which had an exponential number of constraints. Several attempts are discussed to reduce the number of constraints, but all failed and ObM has remained a really complex model. Finally, a formulation inspired by F2, denoted by F3, is proposed. The particularity of F3 formulation is that its constraints are independent from the number of edges in the graphs. F3 is proven to be very efficient when dealing with dense graphs, and as a good as F2 formulation in the case of normal graphs. This concludes the first part of this chapter.

The second part focuses on using F3 formulation in matheuristic methods to provide heuristics for the GED problem. First, an adapted local branching heuristic is developed over F3 formulation, as it was done before in Chapter 4. Then, it is evaluated against existing heuristics and against an exact method. The results have shown very good results by outperforming the existing heuristics and by computing solutions very close from the optimal/best ones. These conclusions have led to other questions: how about other matheuristics? Could another matheuristic perform better?

To find answers, another matheuristic is picked from the literature that is called *variable partitioning local search* (VPLS) and is proposed by Della Croce et al. (2013). It works similarly to local branching, by defining neighborhoods and performing intensification steps. The neighborhood definition is based on selecting a special set of binary variables and setting them free, while fixing the rest of the variables to the values as in a current feasible solution. The intensification focuses the search in that neighborhood looking for a better solution. The key question is how to select the binary variables to be freed? A method is proposed to define the set of binary variables, based on the notion of spheres. It results in selecting binary variables modeling important and costly vertices in the graph. As for LocBra, VPLS is evaluated against existing heuristics and an exact method. It outperforms the existing heuristics, except in some cases as on HOUSE-REF instances where GNCCP heuristic has performed better in the extended version. But in general, the results are very convenient.

Finally, LocBra and VPLS are tested against each other. It turns out that LocBra is slightly better than VPLS in terms of solutions quality. However, LocBra requires more running time than VPLS to compute those solutions. This extra running time can reach hundreds of seconds between LocBra and VPLS. So, basically LocBra is more accurate than VPLS. But, VPLS can be seen as a compromise between finding good solutions and spending too much time.

As a future work, LocBra and VPLS heuristics can be combined somehow, which might help boosting LocBra without affecting its accuracy. As an example: the neighborhood definition in LocBra can be modified to use the definition as in VPLS. The fact that VPLS has a neighborhood definition based on the structure and the characteristics of the graph instance, could be very useful in improving the intensification phase in LocBra.

Clearly, matheuristics are a good choice to solve the GED problem. So, investigating their efficiency on other GM problems is a new line of positive contribution. Of course, there could be other good metaheuristics and matheuristics that might work well on those problems. As a matter of fact, this shows that OR field has a lot to offer to PR field.

Some of the works in this chapter were published in the following conferences:

- Darwiche, M., Conte, D., Raveaux, R., & T'kindt, V. (2018, February). Formulation linéaire en nombres entiers pour le problème de la distance d'édition entre graphes. In *ROADEF18*.
- Darwiche, M., Raveaux, R., Conte, D., & T'Kindt, V. (2018, April). A New Mixed Integer Linear Program for the Graph Edit Distance Problem. In *ISCO18*.
- Darwiche, M., Raveaux, R., Conte, D., & T'Kindt, V. (2018, August). Graph Edit Distance in the Exact Context. *In Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)* (pp. 304-314). Springer, Cham.

Chapter 6

Conclusion

Concluding remarks

The contributions of this thesis are split into two Chapters: Chapter 4 which is focused on the GED^{EnA} problem and Chapter 5 which is focused on the general GED problem.

In Chapter 4, the first contribution is the distinction made between the GED^{EnA} sub-problem and the GED problem. The sub-problem differs in the instances and cost functions, in which the graphs do not carry attributes. So, their edit operations costs are 0 for substitution and a constant for insertion and deletion. A method (exact or heuristic) that solves the GED problem, can as well solve the sub-problem, but the opposite does not hold. This is the case of JH formulation (Justice and Hero, 2006), which is the best existing one for solving GED^{EnA} instances, but not GED instances. The sub-problem can be seen a lighter version, by reducing the difficulty of determining edges operations. Yet, the sub-problem is still hard as the general one. In fact, GED^{EnA} is also a $\mathcal{NP}$-hard problem. Because cost functions are dependent on graph databases, a review to the most used graph databases is given, with a classification of which databases are suited for both GED^{EnA} and GED problems, or only the GED problem.

The second contribution is the experimentation results of comparing the best existing MILP formulations that solve the GED problem. The formulations are tested, with and without pre-processing procedure. Pre-processing is an algorithm that extracts information from the LP relaxation solution. Then, using these information it tries to fix binary variables in the MILP formulation before giving it to the solver. It aims at simplifying the formulation and makes it easier to the solver, if it works works well. The results of both experiments have shown that JH is ultimately, the best formulation. And also, it turned out that the pre-processing did not help in fixing variables, since the percentage of fixed variables is too low as shown in the results. The reason is probably because the LP relaxation solution is too weak.

Because JH is the best formulation, it is selected in the implementation of an adapted version of local branching heuristic (LocBra) to solve the GED^{EnA} problem. This is the third contribution in the thesis. LocBra is a matheuristic designed to perform local searches in defined neighborhoods in the solution space of a MILP formulation. It combines several heuristic techniques (neighborhood definition, intensification and diversification) in a defined branching scheme by solving small sub-problems using a MILP black-box solver. The key points of LocBra are:

- Neighborhood definition: it is done by considering only variables modeling vertices matching, which leads to a better decomposition of the problem.
- Problem-dependent diversification: to improve the diversification, which is considered as an important step to escape local minima, LocBra adapts a special diversification based on analysis done over the data of the instance at hand. It determines the important variables that guarantee very good diversification. It was shown that this diversification is more efficient than the original one.
- LocBra is very flexible and its performance can be controlled by a set of input parameters.

Next, this new heuristic is intensively evaluated on reference databases against the best heuristics available in the literature. The experiments try to capture different points of view on GM field. They are categorized as follows:

Experiment type	Main comparison indicator	Application
Distance minimization	Deviation	Near-optimal quality
Ranking	Kendall correlation	Similarity search/graph retrieval
Ground-truth matching	Hamming Distance	Result interpretation

The second and third experiments can be seen as the forth contribution of this thesis. Despite their importance, they have not yet been considered when evaluating GED heuristics. They study the performance of the heuristic with a more realistic and application-oriented point of view. The *Ranking* experiment focuses on the application of the graph retrieval of an input graph based on a database of graphs. The *Ground-truth matching* experiment measures the correlation and closeness of solutions computed by the heuristic to the ground-truth solutions, confirming the relevance of the cost functions defined for the database. The results of all the experiments have shown that LocBra is a very competitive and effective heuristic for solving the GED^{EnA} problem. LocBra's solutions are proven to be very close to the optimal and best ones. Also, they have been shown to be highly correlated with the ground-truth solutions. Moreover, LocBra is shown to be suitable for chemical graphs (e.g. MUTA and PAH databases) and graphs extracted from images (HOUSE-NA and HOUSE-A databases). And this concludes the contributions of Chapter 4.

Chapter 5 is devoted to the GED problem, and the contributions are covering exact and heuristic methods. In the exact context, many MILP formulations are proposed based on different ideas and concepts. A first formulation, denoted by VbM (vertex-based model), is proposed with the aim at reducing the complexity of the models by reducing the number of binary variables. But, this has come at the price of increasing the number of constraints in the formulation. The experiments have shown that VbM cannot perform better than F2 formulation (the best one for the GED problem). A second formulation, denoted by ObM (object-based model), is designed based on the idea of multiple vertices/edges matchings in a single compact object. ObM can be seen as an expanded formulation, because it requires multiplying the input data. However, ObM has suffered from an exponential number of constraints, which resulted in a bad performance. Many ideas are suggested to reduce the complexity of the constraints, but none has succeeded in improving the performance of the formulation. F3 is the third proposed formulation, that is considered one of the main contributions of this thesis. It is inspired by F2 formulation, and shares the same objective function. But, it has differences in the sets of variables and constraints. The particularity of F3 formulation is that its constraints are independent from the number of edges in the graphs. The evaluations has proven F3 to be very efficient when dealing with dense graphs, and as good as F2 formulation in the case of normal graphs.

The second part of Chapter 5 starts with the second contribution, which is applying LocBra heuristic with F3 formulation to solve the GED problem. The core of the heuristic, as explained in Chapter 4, is kept the same but only JH is replaced with F3 formulation. This version of LocBra is evaluated against existing heuristics and an exact method. The results have proven, again, the superiority of LocBra over existing heuristics, and its capability in computing near-optimal solutions.

The successful application of local branching matheuristic to the GED problem, has led to the last contribution, which consists in proposing an adapted version of variable partition local search (VPLS) matheuristic. It is a novel heuristic, and it is similar to local branching where it performs local searches in defined neighborhoods. The main idea in VPLS is to define the special set of binary variables, that are required for the neighborhood definition. Those variables are created in the formulation and kept free, while the rest of the variables are set to values based on a given feasible solution. To define the set of special variables, an algorithm is proposed based on the notion of spheres. A sphere is a subgraph containing neighbor vertices and has a cost that is computed based on the matching extracted from a given feasible solution. The sphere with the highest cost is detected and all binary variables modeling vertices belonging to that sphere are put in the set of special variables. Those are the variables that their matchings cost the highest, and defining the neighborhood based on them will result in finding a new solution with better matchings and objective function value. Consequently, the neighborhood definition in VPLS is strongly based on extracting valuable information and characteristics from the graph instance. Finally, the experimentation results have confirmed the good performance of VPLS when compared to existing heuristics. VPLS is compared as well with LocBra, that showed slightly better results in terms of solutions quality over VPLS. However, VPLS was much more faster than LocBra heuristic.

All the aforementioned contributions were communicated to scientific conferences and journals, except the LocBra and VPLS presented in Chapter 5. They will be definitely submitted soon and shared with the research communities. The summary of the publications is given in Table 6.1.

Table 6.1: Summary of publications

Chapter	Publications
4	ROADEF'17(Darwiche et al., 2017a) MIC'17(Darwiche et al., 2017b) CIARP'17(Darwiche et al., 2017c) J. of C&OR(Darwiche et al., 2018c) J. of PRL(Darwiche et al., 2018b) Matheuristics'18(Darwiche et al., 2018f)
5	ROADEF'18(Darwiche et al., 2018a) ISCO'18(Darwiche et al., 2018d) S+SSPR'18(Darwiche et al., 2018e)

Perspectives and future works

This thesis has enforced the bridge between OR and PR fields. It gives an idea about what OR could offer to GM problems in general and the GED problem in particular. It opens the door to a new way of seeing the GED problem, and the possibility of applying efficient modeling and solution techniques that are common in OR field. Also, the thesis shows how to do the following:

- Apply OR techniques to solve a particular problem.
- Analyze the problem properties and cases to extract useful problem-dependent information.
- Make use of these information to enhance the performance of the method, whether it is exact or heuristic.

Nevertheless, there is no guarantee that following those steps will, for sure, result in a good method. Sometimes, what makes sense theoretically may not work and achieve good results experimentally. It is probably due to missing aspects and hidden/unconsidered cases.

In the thesis, both scenarios appear in many places. For instance, the two proposed MILP formulations VbM and ObM, are both based on interesting ideas trying to simplify the models. However, what is thought to be a simplification has led to complicating other aspects, e.g. the constraints in VbM and ObM formulations. On the other hand, F3 formulation has succeeded in accomplishing the objective, which is designing an effective exact method to the GED problem. Anyhow, whether it is the first scenario or the second, there is always a room to improve and increase the efficiency of the methods. Due to the time constraint, it was not possible to invest more time in order to investigate VbM and ObM formulations problems. Though, this fits exactly in the perspectives and the directions for future works.

Regarding the LocBra heuristic, it has shown great capability in solving both the GED and GED^{EnA} problems. However, few improvements are still needed to boost its performance:

1. Parameters tuning: come up with a method to tune the parameters in an adapted fashion. An interesting idea could be to learn a model over features extracted from all solved instances and databases. So the model can predict the right parameter values when executing LocBra on a new instance. Another thing, which is partially done in this thesis, is to fully study the parameters and their influence on the method, and if there is a correlation between them.

2. Speed up LocBra: the heuristic as it is, is able to compete with existing heuristics, but it requires more CPU time than the others. This is drawback in case LocBra is decided to be used in an application that requires fast solutions computation. What could be done to boost the solution of LocBra, is to extract more information from graphs. Then, use these information in the definition of neighborhoods by considering only a small set of binary variables. This set could be predicted by ML techniques. This may lead to visit very interesting neighborhoods in the search space where optimal solutions reside. Another idea is to increase/decrease the size of the neighborhood and see how this affects the performance of the heuristic.

3. Parallelizing LocBra: the heuristic performs local searches in neighborhoods around an initial solution. It could be parallelized to use multiple threads and processes to define neighborhoods around a pool of initial solutions and perform local searches in all of them in parallel. Only good solutions are kept for next iterations. This may help in improving the speed of LocBra, without degrading the quality of the solution.

4. Initial solution: instead of computing an initial solution by solving a MILP formulation, which consumes a considerable amount of time, it could be replaced with a good and fast heuristic such as IPFP. This should be tested to make sure it does not degrade the quality of the solutions.

Even though, VPLS is notably faster than LocBra, the above raised points for LocBra are also legitimate for VPLS heuristic. Adding to them the possibility of improving the procedure of selecting the special variables for neighborhood definitions. Currently, it is based on computing spheres on one of the input graphs. Other ideas could be tested, such as computing spheres on both graphs and combine the information to select vertices from multiple spheres and graphs. Another idea is to include more features when computing the spheres costs, such as vertices degrees, number of edges in the sphere, etc.

Furthermore, LocBra and VPLS heuristics can be combined somehow, which might help boosting LocBra without affecting its accuracy. As an example: the neighborhood definition in LocBra can be modified to use the definition as in VPLS. The fact that VPLS has a neighborhood definition based on the structure and the characteristics of the graph instance, could be very useful in improving the intensification phase in LocBra.

Meanwhile, and since the GED problem is a generalization of other GM problems such as the *Maximum Common Subgraph* (MCS) problem, all the methods developed in this thesis can be tested on the MCS problem. This is an objective to be sought at some point down the road, so new contributions can be added to existing methods and algorithms for solving the MCS problem.

In the end, this research work has, hopefully, achieved its explicit objectives in enriching the GM and GED problems arsenal of methods. As well, it has implicitly realized other objectives such as shedding the light on a new angle for solving the problems through matheuristics and OR techniques. Hopefully, this will leave a good impression and a positive influence in the PR community.

Bibliography

E. Aarts, E. Aarts, and J. Lenstra. *Local Search in Combinatorial Optimization*. Princeton University Press, 2003. ISBN 9780691115221. URL `https://books.google.com.lb/books?id=NWghN9G7q9MC`.

Z. Abu-Aisheh, R. Raveaux, and J.-Y. Ramel. A graph database repository and performance evaluation metrics for graph edit distance. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 138–147. Springer, 2015a.

Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, and P. Martineau. An exact graph edit distance algorithm for solving pattern recognition problems. In *4th International Conference on Pattern Recognition Applications and Methods 2015*, 2015b.

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

Z. Abu-Aisheh, B. Gaüzère, S. Bougleux, J.-Y. Ramel, L. Brun, R. Raveaux, P. Héroux, and S. Adam. Graph edit distance contest: Results and future challenges. *Pattern Recognition Letters*, 100:96–103, 2017.

A. Agresti. *Analysis of ordinal categorical data*, volume 656. John Wiley & Sons, 2010.

T. Akutsu and T. Tamura. On the complexity of the maximum common subgraph problem for partial k-trees of bounded degree. In *International Symposium on Algorithms and Computation*, pages 146–155. Springer, 2012.

H. Almohamad and S. O. Duffuaa. A linear programming approach for the weighted graph matching problem. *IEEE Transactions on pattern analysis and machine intelligence*, 15 (5):522–525, 1993.

K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003. ISBN 9780521825832. URL `https://books.google.com.lb/books?id=1e7Ib04fZAcC`.

L. Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 684–697. ACM, 2016.

P. Baptiste, F. Della Croce, A. Grosso, and V. T'kindt. Sequencing a single machine with due dates and deadlines: an ilp-based approach to solve very large instances. *Journal of scheduling*, 13(1):39–47, 2010.

C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations research*, 46(3):316–329, 1998.

F. Bock. An algorithm for solving travelling-salesman and related network optimization problems. In *Operations Research*, volume 6, pages 897–897. INST OPERATIONS RESEARCH MANAGEMENT SCIENCES 901 ELKRIDGE LANDING RD, STE 400, LINTHICUM HTS, MD 21090-2909, 1958.

S. Bougleux, L. Brun, V. Carletti, P. Foggia, B. Gaüzère, and M. Vento. Graph edit distance as a quadratic assignment problem. *Pattern Recognition Letters*, 87:38–46, 2017.

M. Bourquin, A. King, and E. Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2013.

C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

L. Brun, D. Conte, P. Foggia, M. Vento, and D. Villemin. Symbolic learning vs. graph kernels: An experimental comparison in a chemical application. In *ADBIS (Local Proceedings)*, pages 31–40, 2010.

Q. A. Bui, M. Visani, and R. Mullot. Unsupervised word spotting using a graph representation based on invariants. In *Document Analysis and Recognition (ICDAR), 2015 13th International Conference on*, pages 616–620. IEEE, 2015.

H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, 18(8):689–694, 1997.

H. Bunke. Error correcting graph matching: On the influence of the underlying cost function. *IEEE transactions on pattern analysis and machine intelligence*, 21(9):917–922, 1999.

H. Bunke and G. Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, 1(4):245–253, 1983.

H. Bunke and K. Riesen. Towards the unification of structural and statistical pattern recognition. *Pattern Recognition Letters*, 33(7):811–825, 2012.

H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern recognition letters*, 19(3-4):255–259, 1998.

T. S. Caetano, J. J. McAuley, L. Cheng, Q. V. Le, and A. J. Smola. Learning graph matching. *IEEE transactions on pattern analysis and machine intelligence*, 31(6):1048–1058, 2009.

V. Carletti, P. Foggia, and M. Vento. Performance comparison of five exact graph matching algorithms on biological databases. In *International Conference on Image Analysis and Processing*, pages 409–417. Springer, 2013.

H. Cecotti. Active graph based semi-supervised learning using image matching: application to handwritten digit recognition. *Pattern Recognition Letters*, 73:76–82, 2016.

P.-A. Champin and C. Solnon. Measuring the similarity of labeled graphs. In *International Conference on Case-Based Reasoning*, pages 80–95. Springer, 2003.

L. Chang, X. Feng, X. Lin, L. Qin, and W. Zhang. Efficient graph edit distance computation and verification via anchor-aware lower bound estimation. *arXiv preprint arXiv:1709.06810*, 2017.

X. Chen, H. Huo, J. Huan, and J. S. Vitter. Fast computation of graph edit distance. *arXiv preprint arXiv:1709.10305*, 2017.

M. Cho, K. Alahari, and J. Ponce. Learning graphs to match. In *Computer Vision (ICCV), 2013 IEEE International Conference on*, pages 25–32. IEEE, 2013.

Y. Choi and G. Kim. Graph-based fingerprint classification using orientation field in core area. *IEICE Electronics Express*, 7(17):1303–1309, 2010.

W. Commons. P versus np problem, 2018. URL `https://en.wikipedia.org/wiki/P_versus_NP_problem`.

D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence*, 18 (03):265–298, 2004.

S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

T. H. Cormen. Section 24.3: Dijkstra's algorithm. *Introduction to algorithms*, pages 595–601, 2001.

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

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

G. A. Croes. A method for solving traveling-salesman problems. *Operations research*, 6 (6):791–812, 1958.

G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal, Complex Systems*, 1695(5):1–9, 2006.

G. B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. *New York*, 1951.

M. Darwiche, D. Conte, R. Raveaux, and V. T'Kindt. Evaluation de modèles mathématiques pour le problème de la distance d'édition entre graphes. In *ROADEF2017*, 2017a.

M. Darwiche, D. Conte, R. Raveaux, and V. T'Kindt. The graph edit distance problem treated by the local branching heuristic. In *MIC17 12th Metaheuristics International Conference*, 2017b.

M. Darwiche, R. Raveaux, D. Conte, and V. T'Kindt. A local branching heuristic for the graph edit distance problem. In *Iberoamerican Congress on Pattern Recognition*, pages 194–202. Springer, 2017c.

M. Darwiche, D. Conte, R. Raveaux, and V. T'kindt. Formulation linéaire en nombres entiers pour le problème de la distance d'édition entre graphes. In *ROADEF18*, 2018a.

M. Darwiche, D. Conte, R. Raveaux, and V. T'Kindt. Graph edit distance: Accuracy of local branching from an application point of view. *Pattern Recognition Letters*, 2018b.

M. Darwiche, D. Conte, R. Raveaux, and V. T'Kindt. A local branching heuristic for solving a graph edit distance problem. *Computers & Operations Research*, 2018c.

M. Darwiche, R. Raveaux, D. Conte, and V. T'Kindt. A new mixed integer linear program for the graph edit distance problem. In *ISCO18*, 2018d.

M. Darwiche, R. Raveaux, D. Conte, and V. T'Kindt. Graph edit distance in the exact context. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 304–314. Springer, 2018e.

M. Darwiche, R. Raveaux, D. Conte, and V. T'kindt. Solving a special case of the graph edit distance problem with local branching. In *Matheuristics 2018*, 2018f.

F. Della Croce, A. Grosso, and F. Salassa. Matheuristics: embedding milp solvers into heuristic algorithms for combinatorial optimization problems. In P. Siarry, editor, *The Oxford Handbook of Innovation*, chapter 3. NOVA Publisher, 2013.

M. Dorigo and T. Stützle. The ant colony optimization metaheuristic: Algorithms, applications, and advances. In *Handbook of metaheuristics*, pages 250–285. Springer, 2003.

M. Dorigo, D. de Recherches Du Fnrs Marco Dorigo, T. Stützle, and T. Stützle. *Ant Colony Optimization*. A Bradford book. BRADFORD BOOK, 2004. ISBN 9780262042192. URL `https://books.google.com.lb/books?id=_aefcpY8GiEC`.

N. J. Driebeek. An algorithm for the solution of mixed integer programming problems. *Management Science*, 12(7):576–587, 1966.

A. C. Dumay, R. J. van der Geest, J. J. Gerbrands, E. Jansen, and J. H. Reiber. Consistent inexact graph matching applied to labelling coronary segments in arteriograms. In *Pattern Recognition, 1992. Vol. III. Conference C: Image, Speech and Signal Analysis, Proceedings., 11th IAPR International Conference on*, pages 439–442. IEEE, 1992.

A. A. Elhadi, M. A. Maarof, and A. H. Osman. Malware detection based on hybrid signature behaviour application programming interface call graph. *American Journal of Applied Sciences*, 9(3):283, 2012.

M. A. Eshera and K. S. Fu. An image understanding system using attributed symbolic representation and inexact graph-matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(5):604–618, Sept 1986. ISSN 0162-8828. doi: 10.1109/TPAMI.1986.4767835.

E. Estrada. Graph and network theory in physics. *arXiv preprint arXiv:1302.4378*, 2013.

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

A. Filatov, A. Gitis, and I. Kil. Graph-based handwritten digit string recognition. In *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, volume 2, pages 845–848. IEEE, 1995.

S. Fischer, K. Gilomen, and H. Bunke. Identification of diatoms by grid graph matching. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 94–103. Springer, 2002.

M. Fischetti and A. Lodi. Local branching. *Mathematical programming*, 98(1-3):23–47, 2003.

G. Florez-Puga, P. A. Gonzalez-Calero, G. Jimenez-Diaz, and B. Diaz-Agudo. Supporting sketch-based retrieval from a library of reusable behaviours. *Expert Systems with Applications*, 40(2):531–542, 2013.

C. A. Floudas and P. M. Pardalos. *Encyclopedia of optimization*, volume 1. Springer Science & Business Media, 2001.

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

M. R. Garey and D. S. Johnson. Computers and intractability: A guide to the theory of npcompleteness (series of books in the mathematical sciences), ed. *Computers and Intractability*, 340, 1979.

M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990. ISBN 0716710455.

B. Gaüzère, L. Brun, and D. Villemin. Two new graph kernels and applications to chemoinformatics. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 112–121. Springer, 2011.

P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations research*, 9(6):849–859, 1961.

F. Glover. Tabu search—part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

F. Glover and M. Laguna. Tabu search. In *Handbook of combinatorial optimization*, pages 2093–2229. Springer, 1998.

D. E. Goldberg and J. H. Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

C. Gomila and F. Meyer. Tracking objects by graph matching of image partition sequences. In *Proc. 3rd IAPR-TC15 Workshop Graph-Based Representations in Pattern Recognition*, pages 1–11, 2001.

C. Gomila and F. Meyer. Graph-based object tracking. In *Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference on*, volume 2, pages II–41. IEEE, 2003.

R. E. Gomory et al. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical society*, 64(5):275–278, 1958.

M. Gori, M. Maggini, and L. Sarti. Exact and approximate graph matching using random walks. *IEEE transactions on pattern analysis and machine intelligence*, 27(7):1100–1111, 2005.

Gotha. *Les problèmes d'ordonnancement.* Centre de Recherche en Informatique de Nancy, 1993.

K. Gouda and M. Hassaan. Csi_ged: an efficient approach for graph edit similarity computation. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 265–276. IEEE, 2016.

M. Grandjean. A social network analysis of twitter: Mapping the digital humanities community. *Cogent Arts & Humanities*, 3(1):1171458, 2016.

Y. Han, T. Tan, and Z. Sun. Palmprint recognition based on directional features and graph matching. In *International Conference on Biometrics*, pages 1164–1173. Springer, 2007.

P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2): 100–107, 1968.

R. Hasan and F. Gandon. A machine learning approach to sparql query performance prediction. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2014 IEEE/WIC/ACM International Joint Conferences on*, volume 1, pages 266–273. IEEE, 2014.

M. Hasegawa and S. Tabbone. A local adaptation of the histogram radon transform descriptor: an application to a shoe print dataset. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 675–683. Springer, 2012.

A. Hill and S. Voß. Generalized local branching heuristics and the capacitated ring tree problem. *Discrete Applied Mathematics*, 242:34 – 52, 2018. ISSN 0166-218X. doi: https://doi.org/10.1016/j.dam.2017.09.010. URL `http://www.sciencedirect.com/science/article/pii/S0166218X17304225`. Computational Advances in Combinatorial Optimization.

D. S. Hochbaum. *Approximation algorithms for NP-hard problems.* PWS Publishing Co., 1996.

K. L. Hoffman, M. Padberg, and G. Rinaldi. Traveling salesman problem. In *Encyclopedia of operations research and management science*, pages 1573–1578. Springer, 2013.

M. Jiinger, G. Reinelt, and S. Thienel. Practical problem solving with cutting plane algorithms in combinatorial optimization. *Comb. Optim. Dimacs*, 20:111–152, 1995.

Y. Jin and J. Branke. Evolutionary optimization in uncertain environments-a survey. *IEEE Transactions on evolutionary computation*, 9(3):303–317, 2005.

Y. Jin and B. Sendhoff. Trade-off between performance and robustness: an evolutionary multiobjective approach. In *international conference on Evolutionary Multi-Criterion Optimization*, pages 237–251. Springer, 2003.

H. Johnston. Cliques of a graph-variations on the bron-kerbosch algorithm. *International Journal of Computer & Information Sciences*, 5(3):209–238, 1976.

D. Justice and A. Hero. A binary linear programming formulation of the graph edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(8):1200–1214, 2006.

M. Kammer, H. Bodlaender, and J. Hage. *Plagiarism detection in haskell programs using call graph matching.* PhD thesis, Master's thesis, Utrecht University, 2011.

R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

O. Kostakis, J. Kinable, H. Mahmoudi, and K. Mustonen. Improved call graph comparison using simulated annealing. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1516–1523. ACM, 2011.

J. Kun. A quasipolynomial time algorithm for graph isomorphism the details, Nov 2015. URL `https://jeremykun.com/2015/11/12/a-quasipolynomial-time-algorithm-for-graph-isomorphism-the-details/`.

S. M. Lajevardi, A. Arakala, S. A. Davis, and K. J. Horadam. Retina verification system based on biometric graph matching. *IEEE transactions on image processing*, 22(9): 3625–3635, 2013.

P. Le Bodic, P. Héroux, S. Adam, and Y. Lecourtier. An integer linear program for substitution-tolerant subgraph isomorphism and its use for symbol spotting in technical drawings. *Pattern Recognition*, 45(12):4214–4224, 2012.

D. K. Lê-Huu and N. Paragios. Alternating direction graph matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6253–6261, 2017.

T. S. Lee and D. Mumford. Hierarchical bayesian inference in the visual cortex. *JOSA A*, 20(7):1434–1448, 2003.

M. Leordeanu, M. Hebert, and R. Sukthankar. An integer projected fixed point method for graph matching and map inference. In *Advances in neural information processing systems*, pages 1114–1122, 2009.

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

J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9(4):341, 1973.

Z.-Y. Liu and H. Qiao. Gnccp—graduated nonconvexityand concavity procedure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(6):1258–1267, 2014.

J. Lladós, J. López-Krahe, and E. Martí. Hand drawn document understanding using the straight line hough transform and graph matching. In *Pattern Recognition, 1996., Proceedings of the 13th International Conference on*, volume 2, pages 497–501. IEEE, 1996.

A. Lodi and G. Zarpellon. On learning and branching: a survey. *TOP*, 25(2):207–236, 2017.

H. R. Lourenço, O. C. Martin, and T. Stützle. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer, 2003.

V. Lyzinski, D. E. Fishkind, M. Fiori, J. T. Vogelstein, C. E. Priebe, and G. Sapiro. Graph matching: Relax at your own risk. *IEEE transactions on pattern analysis and machine intelligence*, 38(1):60–73, 2016.

K. Madi, H. Seba, H. Kheddouci, and O. Barge. A graph-based approach for kite recognition. *Pattern Recognition Letters*, 87:186–194, 2017.

D. Maio and D. Maltoni. A structural approach to fingerprint classification. In *Pattern Recognition, 1996., Proceedings of the 13th International Conference on*, volume 3, pages 578–585. IEEE, 1996.

K. Man, K. Tang, and S. Kwong. *Genetic Algorithms: Concepts and Designs*. Advanced Textbooks in Control and Signal Processing. Springer London, 2012. ISBN 9781447105770. URL `https://books.google.com.lb/books?id=RYPuBwAAQBAJ`.

S. Martello, D. Pisinger, and P. Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45(3):414–424, 1999.

D. Mateus, R. Horaud, D. Knossow, F. Cuzzolin, and E. Boyer. Articulated shape matching using laplacian eigenfunctions and unsupervised point registration. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

B. Mayoh, E. Tyugu, and J. Penjam. *Constraint programming*, volume 131. Springer Science & Business Media, 2013.

J. J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1):23–34, 1982.

B. T. Messmer and H. Bunke. Efficient error-tolerant subgraph isomorphism detection. In *Shape, structure and pattern recognition*, pages 231–240. World Scientific, 1994.

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

B. T. Messmer and H. Bunke. A decision tree approach to graph and subgraph isomorphism detection. *Pattern recognition*, 32(12):1979–1998, 1999.

R. Mishra. *Mechanical System Design*. PHI Learning Private Limited, 2009. ISBN 9788120337848. URL `https://books.google.fr/books?id=7dZa2rfhe9cC`.

N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100, 1997.

C. F. Moreno-García, X. Cortés, and F. Serratosa. A graph repository for learning error-tolerant graph matching. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 519–529. Springer, 2016.

J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, 5(1):32–38, 1957.

R. Myers, R. Wison, and E. R. Hancock. Bayesian graph edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(6):628–635, 2000.

G. L. Nemhauser and L. A. Wolsey. Integer programming and combinatorial optimization. *Wiley, Chichester. GL Nemhauser, MWP Savelsbergh, GS Sigismondi (1992). Constraint Classification for Mixed Integer Programming Formulations. COAL Bulletin*, 20:8–12, 1988.

M. Neuhaus and H. Bunke. Edit distance-based kernel functions for structural pattern classification. *Pattern Recognition*, 39(10):1852–1863, 2006a.

M. Neuhaus and H. Bunke. A random walk kernel derived from graph edit distance. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 191–199. Springer, 2006b.

M. Neuhaus and H. Bunke. *Bridging the gap between graph edit distance and kernel machines*, volume 68. World Scientific, 2007.

M. Neuhaus, K. Riesen, and H. Bunke. Fast suboptimal algorithms for the computation of graph edit distance. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 163–172. Springer, 2006.

F. Niedermann. Deep business optimization : concepts and architecture for an analytical business process optimization platform, 2016.

S. Noel and S. Jajodia. Understanding complex network attack graphs through clustered adjacency matrices. In *Computer Security Applications Conference, 21st Annual*, pages 10–pp. IEEE, 2005.

C. Novoa and R. Storer. An approximate dynamic programming approach for the vehicle routing problem with stochastic demands. *European Journal of Operational Research*, 196(2):509–515, 2009.

O. R. S. Operational Research Society. News and notes. *OR*, 13(3):279–286, 1962. ISSN 14732858. URL `http://www.jstor.org/stable/3006901`.

M. A. Osorio, F. Glover, and P. Hammer. Cutting and surrogate constraint analysis for improved multidimensional knapsack solutions. *Annals of Operations Research*, 117(1-4): 71–93, 2002.

E. Ozdemir and C. Gunduz-Demir. A hybrid classification model for digital pathology using structural and statistical pattern recognition. *IEEE Transactions on Medical Imaging*, 32(2):474–483, 2013.

I. K. P. I. K. Park, I. D. Y. I. D. Yun, and S. U. L. S. U. Lee. Models and algorithms for efficient color image indexing. In *Content-Based Access of Image and Video Libraries, 1997. Proceedings. IEEE Workshop on*, pages 36–41. IEEE, 1997.

S. Paul. *Exploring story similarities using graph edit distance algorithms*. PhD thesis, University of Delaware, 2013.

A. Perchant, C. Boeres, I. Bloch, M. Roux, and C. Ribeiro. Model-based scene recognition using graph fuzzy homomorphism solved by genetic algorithm. In *GbR'99 2nd International Workshop on Graph-Based Representations in Pattern Recognition*, pages 61–70, 1999.

E. G. M. Petrakis and A. Faloutsos. Similarity searching in medical image databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):435–447, 1997.

R. Raveaux, E. Barbu, H. Locteau, S. Adam, P. Héroux, and É. Trupin. A graph classification approach using a multi-objective genetic algorithm application to symbol recognition. In *Graph-Based Representations in Pattern Recognition, 6th IAPR-TC-15, International Workshop, GbRPR 2007, Alicante, Spain, June 11-13, 2007, Proceedings*, pages 361–370, 2007. doi: 10.1007/978-3-540-72903-7_33. URL `https://doi.org/10.1007/978-3-540-72903-7_33`.

R. Raveaux, S. Adam, P. Héroux, and É. Trupin. Learning graph prototypes for shape recognition. *Computer Vision and Image Understanding*, 115(7):905–918, 2011.

J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of computer-aided molecular design*, 16(7): 521–533, 2002.

P. Riba, J. Lladãs, and A. Fornés. Handwritten word spotting by inexact matching of grapheme graphs. In *Document Analysis and Recognition (ICDAR), 2015 13th International Conference on*, pages 781–785. IEEE, 2015.

K. Riesen. Structural pattern recognition with graph edit distance. *Advances in Computer Vision and Pattern Recognition. Springer, Cham*, 2015.

K. Riesen and H. Bunke. Iam graph database repository for graph based pattern recognition and machine learning. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 287–297. Springer, 2008.

K. Riesen and M. Ferrer. Predicting the correctness of node assignments in bipartite graph matching. *Pattern Recognition Letters*, 69:8–14, 2016.

K. Riesen, M. Neuhaus, and H. Bunke. Bipartite graph matching for computing the edit distance of graphs. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 1–12. Springer, 2007a.

K. Riesen, M. Neuhaus, and H. Bunke. Graph embedding in vector spaces by means of prototype selection. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 383–393. Springer, 2007b.

K. Riesen, D. Brodić, Z. N. Milivojević, and Č. A. Maluckov. Graph based keyword spotting in medieval slavic documents–a project outline. In *Euro-Mediterranean Conference*, pages 724–731. Springer, 2014.

A. Robles-Kelly and E. R. Hancock. Graph edit distance from spectral seriation. *IEEE transactions on pattern analysis and machine intelligence*, 27(3):365–378, 2005.

J. Rocha and T. Pavlidis. A shape analysis model with applications to a character recognition system. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(4): 393–404, 1994.

I. Rodríguez-Martín and J. J. Salazar-González. A local branching heuristic for the capacitated fixed-charge network design problem. *Computers & Operations Research*, 37(3): 575–581, 2010.

H. D. Røkenes et al. Graph-based natural language processing: Graph edit distance applied to the task of detecting plagiarism. Master's thesis, Institutt for datateknikk og informasjonsvitenskap, 2012.

S. Sahni and T. Gonzalez. P-complete approximation problems. *Journal of the ACM (JACM)*, 23(3):555–565, 1976.

M. Salotti and N. Laachfoubi. Topographic graph matching for shift estimation. In *Proc. 3rd IAPR-TC15 Workshop Graph-Based Representations in Pattern Recognition*, pages 54–63, 2001.

M. Samavati, D. Essam, M. Nehring, and R. Sarker. A local branching heuristic for the open pit mine production scheduling problem. *European Journal of Operational Research*, 257 (1):261–271, 2017.

A. Sanfeliu and K. S. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(3): 353–362, May 1983. ISSN 0018-9472. doi: 10.1109/TSMC.1983.6313167.

M. Seidl, E. Wieser, M. Zeppelzauer, A. Pinz, and C. Breiteneder. Graph-based shape similarity of petroglyphs. In *Workshop at the European Conference on Computer Vision*, pages 133–148. Springer, 2014.

F. Serratosa. Fast computation of bipartite graph matching. *Pattern Recognition Letters*, 45:244–250, 2014.

F. Serratosa. Computation of graph edit distance: reasoning about optimality and speed-up. *Image and Vision Computing*, 40:38–48, 2015a.

F. Serratosa. Speeding up fast bipartite graph matching through a new cost matrix. *International Journal of Pattern Recognition and Artificial Intelligence*, 29(02):1550010, 2015b.

F. Serratosa and X. Cortés. Graph edit distance: moving from global to local structure to solve the graph-matching problem. *Pattern Recognition Letters*, 65:204–210, 2015.

F. Serratosa, A. Solé-Ribalta, and X. Cortés. Automatic learning of edit costs based on interactive and adaptive graph recognition. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 152–163. Springer, 2011.

K. Shearer, H. Bunke, and S. Venkatesh. Video indexing and similarity retrieval by largest common subgraph detection using decision trees. *Pattern Recognition*, 34(5):1075–1091, 2001.

P. D. Simić. Constrained nets for graph matching and other quadratic assignment problems. *Neural Computation*, 3(2):268–281, 1991.

S. Sorlin, C. Solnon, and J.-M. Jolion. A generic graph distance measure based on multivalent matchings. In *Applied Graph Theory in Computer Vision and Pattern Recognition*, pages 151–181. Springer, 2007.

M. Stauffer, A. Fischer, and K. Riesen. Graph-based keyword spotting in historical handwritten documents. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 564–573. Springer, 2016.

M. Stauffer, T. Tschachtli, A. Fischer, and K. Riesen. A survey on applications of bipartite graph edit distance. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 242–252. Springer, 2017.

P. N. Suganthan, E. K. Teoh, and D. P. Mital. Pattern recognition by graph matching using the potts mft neural networks. *Pattern Recognition*, 28(7):997–1009, 1995.

R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine Learning Proceedings 1990*, pages 216–224. Elsevier, 1990.

E.-G. Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

X. Tang, A. Soukhal, and V. T'kindt. Preprocessing for a map sectorization problem by means of mathematical programming. *Annals of Operations Research*, 222(1):551–569, 2014.

J. Till, S. Engell, S. Panek, and O. Stursberg. Empirical complexity analysis of a milp-approach for optimization of hybrid systems. In *IFAC Conference on Analysis and Design of Hybrid Systems*, pages 129–134, 2003.

E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363 (1):28–42, 2006.

V. T'kindt, F. D. Croce, and J.-L. Bouquard. Enumeration of pareto optima for a flowshop scheduling problem with two criteria. *INFORMS Journal on Computing*, 19(1):64–72, 2007.

R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

K. Wang, Y. Wang, and Z. Zhang. On-line signature verification using segment-to-segment graph matching. In *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pages 804–808. IEEE, 2011.

M. Wang, Y. Iwai, and M. Yachida. Expression recognition from time-sequential facial images by use of expression change model. In *Automatic Face and Gesture Recognition, 1998. Proceedings. Third IEEE International Conference on*, pages 324–329. IEEE, 1998.

P. Wang, V. Eglin, C. Garcia, C. Largeron, J. Lladós, and A. Fornés. A coarse-to-fine word spotting approach for historical handwritten documents based on graph embedding and graph edit distance. In *Pattern Recognition (ICPR), 2014 22nd International Conference on*, pages 3074–3079. IEEE, 2014a.

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

J. Wei. Markov edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(3):311–321, 2004.

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.

R. C. Wilson and E. R. Hancock. Graph matching with hierarchical discrete relaxation. In *Advances in Neural Information Processing Systems*, pages 689–695, 1998.

L. Wiskott. Phantom faces for face analysis. *Pattern Recognition*, 30(6):837–846, 1997.

L. Wiskott, N. Krüger, N. Kuiger, and C. Von Der Malsburg. Face recognition by elastic bunch graph matching. *IEEE Transactions on pattern analysis and machine intelligence*, 19(7):775–779, 1997.

X. Wu, X. Zhu, G.-Q. Wu, and W. Ding. Data mining with big data. *IEEE transactions on knowledge and data engineering*, 26(1):97–107, 2014.

M. Zaslavskiy, F. Bach, and J.-P. Vert. A path following algorithm for the graph matching problem. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(12): 2227–2242, 2009.

Z. Zeng, A. K. Tung, J. Wang, J. Feng, and L. Zhou. Comparing stars: On approximating graph edit distance. *Proceedings of the VLDB Endowment*, 2(1):25–36, 2009.

K. Zhang. A constrained edit distance between unordered labeled trees. *Algorithmica*, 15 (3):205–222, 1996.

K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.

Y. Zhang, X. Yang, H. Qiao, Z. Liu, C. Liu, and B. Wang. A graph matching based key point correspondence method for lunar surface images. In *Intelligent Control and Automation (WCICA), 2016 12th World Congress on*, pages 1825–1830. IEEE, 2016.

F. Zhou and F. De la Torre. Factorized graph matching. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 127–134. IEEE, 2012.

F. Zhou, J. Brandt, and Z. Lin. Exemplar-based graph matching for robust facial landmark localization. In *Computer Vision (ICCV), 2013 IEEE International Conference on*, pages 1025–1032. IEEE, 2013.

Appendices

Appendix A

The GED^{EnA} problem: Additional results and evaluations

A.1 Local branching - for the GED^{EnA} problem - experiments results

In this section, the results of additional experiments done on different databases to evaluate the proposed local branching heuristic for solving the GED^{EnA} problem.

A.1.1 Effectiveness of LocBra w.r.t. competitor heuristics

In addition to the experiment presented in Section 4.4.6.1 on MUTA database, other databases were used in the evaluation of LocBra. All the results are reported and discussed here.

A.1.1.1 Evaluations on PAH database

This graph database is constructed from chemical molecules, and contains small to medium instances of graphs. The total number of instances is 8836. The cost functions used are detailed in Section 4.2.

Default versions. The parameters are set for each heuristic as follows:

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 12.25s$, $node\_time\_limit = UB\_time\_limit = 1.75s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
CPLEX-t	$t = 12.48$
CPLEX_LocBra-t	$t = 3.5$
BeamSearch-α	$\alpha = 5$
SBPBeam-α	$\alpha = 5$
IPFP-it	$it = 10$
GNCCP-d	$d = 0.1$

The results are shown in Table A.1. *CPLEX-12.48* has an average deviation of 0.05% which is the smallest among all the heuristics. Next *LocBra* comes with 0.31%. Consequently, *CPLEX-12.48* has performed better than the proposed heuristic. However, an important note is that d_{max} for *LocBra* is 75% against 190.91% for *CPLEX-12.48.* This means that the former provides the closest solutions to the best ones in the worst case. *CPLEX-LocBra-3.5* comes at the third position, with an average deviation less than 1%. The beam-search based heuristics, *IPFP-10* and *GNCCP-0.1* are strongly outperformed by the other MILP-based heuristics with a high average deviations. In terms of CPU time, the beam-search based heuristics seem to be very fast ($t_{avg} < 1s$), while the proposed heuristic is the slowest with $t_{avg} = 3.03s$.

Table A.1: LocBra vs. heuristics on PAH instances

	LocBra	*CPLEX-12.48*	*CPLEX-LocBra-3.5*	*BeamSearch-5*	*SBPBeam-5*	*IPFP-10*	*GNCCP-0.1*
t_{min}	0.06	0.05	0.05	0.00	0.01	0.00	0.17
t_{avg}	3.03	1.97	1.79	**0.01**	0.14	0.03	2.08
t_{max}	12.25	12.48	6.41	0.03	0.37	0.08	6.02
d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
d_{avg}	0.31	**0.05**	0.91	122.65	379.90	127.20	84.43
d_{max}	75.00	190.91	200.00	2400.00	4200.00	2400.00	1000.00
η_I	8716	**8830**	8553	433	100	450	1042

Table A.2: LocBra vs. heuristics with extended running on PAH instances

	LocBra	*CPLEX-LocBra-12*	*BeamSearch-2500*	*SBPBeam-140*	*IPFP-2000*	*GNCCP-0.09*
t_{min}	0.06	0.06	0.01	0.28	0.01	0.24
t_{avg}	3.03	2.59	4.33	4.38	**1.45**	2.63
t_{max}	12.25	14.34	11.55	12.71	13.03	12.33
d_{min}	0.00	0.00	0.00	0.00	0.00	0.00
d_{avg}	0.32	**0.01**	42.70	371.05	117.33	81.01
d_{max}	75.00	40.00	1600.00	4200.00	1600.00	1800.00
η_I	8712	**8833**	3593	100	491	1146

Extended versions. The values of heuristics parameters are:

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 12.25s$, $node\_time\_limit = UB\_time\_limit = 1.75s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
CPLEX_LocBra-t	$t = 3.5$
BeamSearch-α	$\alpha = 2500$
SBPBeam-α	$\alpha = 140$
IPFP-it	$it = 2000$
GNCCP-d	$d = 0.09$

The result of the extended versions are reported in Table A.2. As in the default versions, CPLEX heuristic with a time limit of $12s$ has scored the smallest deviation. So, extending the time for *CPLEX-LocBra-t* to $12s$ to compute an UB, has obtained the smallest deviation with $d_{avg} = 0.01\%$. *LocBra* comes in the second place with $d_{avg} = 0.32\%$, very far from the third heuristic, which is *BeamSearch-2500* ($d_{avg} = 42.7\%$). The positions are conserved when looking at η_I values, with a difference of 121 instances between *CPLEX-LocBra-12* and *LocBra*. *SBPBeam-140* is very poor with $\eta_I = 100$ instances. *IPFP-2000* is the fastest in terms of CPU time, but has $d_{avg} = 117\%$, which reflects a weak accuracy. Note that *LocBra* has $t_{avg} = 3s$, which is better than *BeamSearch-2500* and *SBPBeam-140* average times.

Conclusion. *LocBra*, in both default and extended versions, outperforms the existing heuristics by computing better solutions for PAH instances. However, *CPLEX-t* heuristic is also good at solving PAH instances and performs better in the average case. This due to the presence of very small instances in PAH database, which are very easy to solve by CPLEX to optimality.

A.1.1.2 Evaluations on HOUSE-NA database

It is the version without attributes of CMU-HOUSE graph database, which makes it compatible with the GED^{EnA} problem. More details about the database can be found in Section 4.2.

Table A.3: LocBra vs. heuristics on HOUSE-NA instances

	LocBra	*CPLEX-10*	*CPLEX-LocBra-4*	*BeamSearch-5*	*SBPBeam-5*	*IPFP-10*	*GNCCP-0.1*
t_{min}	0.59	0.41	0.58	0.06	6.58	0.02	7.28
t_{avg}	4.26	4.18	4.36	0.09	7.14	**0.08**	8.01
t_{max}	10.19	10.16	8.27	0.18	8.68	0.25	8.97
d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
d_{avg}	**9.50**	45.67	93.67	25.74	1222.67	146.19	98.80
d_{max}	525.00	716.67	800.00	1000.00	5600.00	850.00	1200.00
η_I	**604**	584	465	546	144	368	452

Default versions. The parameters of each heuristic are:

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 10s$, $node\_time\_limit = 2s, UB\_time\_limit = 4s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
CPLEX-t	$t = 10$
CPLEX-LocBra-t	$t = 4$
BeamSearch-α	$\alpha = 5$
SBPBeam-α	$\alpha = 5$
IPFP-it	$it = 10$
GNCCP-d	$d = 0.1$

The results obtained are given in Table A.3. In terms of deviation, *LocBra* has scored an average of 9.5%, outperforming all the other heuristics. It has the highest η_I with 604 out of 660 instances. Surprisingly, *BeamSearch-5* comes in the second place with $d_{avg} = 25\%$. Perhaps, BeamsSearch has found easily good solutions because it starts its tree with the right vertices, so it converges faster. For this database, the difference between MILP-based methods and LocBra is considerably big, more than 35% by *CPLEX-10* and more than 80% by *CPLEX-LocBra-4*. *SBPBeam-5* seems to have difficulties in solving HOUSE-NA instances, with $d_{avg} = 1222\%$. With regard to the running time, *IPFP-10* is the fastest with $t_{avg} = 0.08s$. Followed by *BeamSearch-5* with $0.09s$. *LocBra* and MILP-based methods seems to be faster than *SBPBeam-5* and *GNCCP-0.1*, with $t_{avg} = 4s$ against 7 to 8 seconds.

Table A.4: LocBra vs. heuristics with extended running time on HOUSE-NA instances

	LocBra	*CPLEX-LocBra-10*	*BeamSearch-350*	*SBPBeam-8*	*IPFP-800*	*GNCCP-0.09*
t_{min}	0.59	0.66	2.33	8.99	0.02	8.51
t_{avg}	4.26	6.71	5.53	9.56	**1.34**	9.80
t_{max}	10.19	16.13	11.20	10.28	9.88	10.14
d_{min}	0.00	0.00	0.00	0.00	0.00	0.00
d_{avg}	**9.20**	43.41	83.78	1220.86	131.45	108.91
d_{max}	525.00	716.67	1100.00	5600.00	850.00	1100.00
η_I	**608**	585	439	144	372	440

Extended versions. The parameters of each heuristic are set as follows:

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 10s$, $node\_time\_limit = 2s, UB\_time\_limit = 4s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
CPLEX-LocBra-t	$t = 10$
BeamSearch-α	$\alpha = 350$
SBPBeam-α	$\alpha = 8$
IPFP-it	$it = 800$
GNCCP-d	$d = 0.09$

Table A.4 presents the results of the extended versions. Basically, the heuristics behaved the same as in the default versions. A small improvement is noticed for *LocBra* with 4 more instances added to η_I and it is reflected by d_{avg} dropping a little bit to 9.2%. Clearly, increasing the beam size for *BeamSearch* did not really help and its d_{avg} jumped from 25% to 83%. As in the default version, *IPFP* remains the fastest with $t_{avg} = 1.34s$, followed by *LocBra* that is faster than the rest of the heuristics.

Conclusion. The presented results are very interesting and shows that *LocBra* is better than all existing heuristics in terms of solutions quality. Moreover, it is also faster than *SBPBeam* and *GNCCP* heuristics on HOUSE-NA instances.

Table A.5: LocBra vs. heuristics on HOUSE-A instances

	LocBra	*CPLEX-2*	*CPLEX-LocBra-1*	*BeamSearch-5*	*SBPBeam-5*	*IPFP-10*	*GNCCP-0.1*
t_{min}	0.30	0.25	0.44	0.02	6.41	0.04	6.55
t_{avg}	0.53	0.40	0.67	0.11	6.84	**0.05**	6.78
t_{max}	2.11	2.06	2.26	0.20	7.57	0.17	8.09
d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
d_{avg}	**0.00**	**0.00**	0.01	6.30	0.91	0.03	**0.00**
d_{max}	0.00	0.00	3.27	60.45	5.54	1.63	0.50
η_I	**660**	**660**	656	313	420	626	656

A.1.1.3 Evaluations on HOUSE-A database

This database contains the same graphs as in CMU-HOUSE database, but it includes the shape context attributes in the vertices costs calculation. This version of CMU-HOUSE database is detailed in Section 4.2. Only one test was realized, because the graph instances are easier and the running time for all the methods are very close even with the default parameters.

Default versions. The parameter values assigned to each heuristic are:

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 2s$, $node\_time\_limit = 0.5s, UB\_time\_limit = 1s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
CPLEX-t	$t = 2$
CPLEX-LocBra-t	$t = 1$
BeamSearch-α	$\alpha = 5$
SBPBeam-α	$\alpha = 5$
IPFP-it	$it = 10$
GNCCP-d	$d = 0.1$

The results are reported in Table A.5. In terms of average deviation, *LocBra*, *CPLEX-2* and *GNCCP-0.1* all have scored 0%. These heuristics were able to solve all the instances and get the best solutions (except for 4 instances by *GNCCP-0.1*). Next, *CPLEX-LocBr-1* has $d_{avg} = 0.01\%$, followed by *IPFP-10* with $d_{avg} = 0.03\%$. It seems that all heuristics have been able to solve all the instances, except *BeamSearch-5* that faced difficulties and scored a high deviation (6.3%). In terms of running time, *IPFP-10* is the fastest with $t_{avg} = 0.05\%$. *LocBra* is also fast with only half a second and beat up the other heuristics (except *CPLEX-2* and *BeamSearch-5*). However, *GNCCP-0.1* and *SBPBeam-5* have the worse average running time, more than $6s$, which makes them really slow on these instances.

Conclusion. For HOUSE-A database, *LocBra* was able to solve efficiently all the instances and with reasonable running times. It is very suitable for this graph database.

A.1.2 Effectiveness of LocBra w.r.t. an exact method

Besides the experiment presented in Section 4.4.6.2, LocBra is evaluated against PAH, HOUSE-NA and HOUSE-A databases. The results are discussed here.

A.1.2.1 Evaluations on PAH database

LocBra parameters are set to the following values: $\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 12.25s$, $node\_time\_limit = UB\_time\_limit = 1.75s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$.

Table A.6 shows the obtained results. The optimal solutions are computed for all instances by CPLEX-∞ ($\eta_I = 8836$). LocBra has found the optimal solutions for 8702 instances. The average deviation of LocBra from the optimal solution is of 0.35%, which is very small. When looking at the 134 instances (out of 8836), for which LocBra failed to find the optimal solution, the average deviation restricted to these instances is about 23%. Nonetheless, it can be concluded that LocBra provides very good solutions on PAH instances. For the running time, CPLEX-∞ is on the average faster than LocBra but in the worst case CPLEX becomes computationally expensive (up to $278.20s$), while the heuristic remains at maximum below than $13s$.

Table A.6: LocBra vs. Exact solution on PAH instances

	CPLEX-∞	*LocBra*
t_{min}	0.09	0.06
t_{avg}	**2.08**	3.03
t_{max}	278.20	12.25
d_{min}	-	0.00
d_{avg}	-	**0.35**
d_{max}	-	100.00
η_I	8836	6715
η'_I	-	**8702**
η''_I	-	0

Conclusion. This experiment shows that PAH instances are easy ones for the JH formulation, so CPLEX is very fast in computing the optimal solutions. In addition, they are easy for LocBra, which has computed solutions with a deviation of 0.35% from the optimal ones.

A.1.2.2 Evaluations on HOUSE-NA database

The parameters of LocBra are: $\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 10s$, $node\_time\_limit = 2s$, $UB\_time\_limit = 4s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$.

Table A.7: LocBra vs. Exact solution on HOUSE-NA instances

	CPLEX-∞	*LocBra*
t_{min}	0.48	0.59
t_{avg}	20.26	**4.26**
t_{max}	2696.80	10.19
d_{min}	-	0.00
d_{avg}	-	**11.01**
d_{max}	-	525.00
η_I	660	440
η'_I	-	**602**
η''_I	-	0

Table A.7 presents the results of the experiment. The average deviation, scored by LocBra, is of 11% from the optimal. Additionally, LocBra did not find the optimal solutions for only 58 instances out of the 660 instances. An important remark is that the average CPU time of LocBra is 5 times smaller than CPLEX-∞ average time. t_{max} increases drastically to reach more than 2000s for the optimal method, while LocBra has a maximum of 10s. The results prove again the closeness, on the average, of LocBra solutions from the optimal solutions.

Conclusion. On HOUSE-NA instances, LocBra is 5 times faster than CPLEX on the average. Also, LocBra reaches a maximum time of $10s$, while CPLEX reaches more than $2000s$ in the worst case. The solutions computed by LocBra are close from the optimal solutions ($d_{avg} = 11\%$).

A.1.2.3 Evaluations on HOUSE-A database

LocBra parameters are set to: $\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 2s$, $node\_time\_limit = 0.5s$, $UB\_time\_limit = 1s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$.

Table A.8: LocBra vs. Exact solution on HOUSE-A instances

	CPLEX-∞	*LocBra*
t_{min}	0.23	0.30
t_{avg}	**0.32**	0.53
t_{max}	2.00	2.11
d_{min}	-	0.00
d_{avg}	-	**0.00**
d_{max}	-	0.00
η_I	660	656
η'_I	-	**660**
η''_I	-	0

The results reported in Table A.8, show that LocBra is as good as the optimal with $d_{avg} = 0\%$ and $\eta'_I = 660$. However, since these instances are very easy, CPLEX-∞ is a bit faster ($0.2s$) than LocBra.

Conclusion. The instance of HOUSE-A are very to solve by CPLEX and LocBra. The average deviation is 0%, which means that LocBra has computed solutions equal to the optimal ones. The CPU times of CPLEX and LocBra are very close, where CPLEX is a bit faster than LocBra (by $0.2s$).

Appendix B

The GED problem: Additional results and evaluations

B.1 VbM experiments results

The formulation VbM was evaluated against MUTA instances in Section 5.2.1.2. The results have shown that F2 formulation is better than VbM formulation. The same experiment is executed again on the three formulations, but this time by adding the pre-processing procedure explained in Section 4.3.2. It is an attempt to see if pre-processing can help VbM formulation in solving better the instances.

B.1.1 Evaluation analysis with pre-processing

One evaluation indicator is added to each formulation, that is the average percentage of fixed variables ($\%varFix$). Table B.1 shows the results obtained, where F2 formulation remains at the first position, followed by F1 and VbM at last. F2 is the best formulation regarding all evaluation indicators, including the $\%varFix$. The gap becomes smaller as the size of the graph increases, to reach 0.39% on subset 70 between the $\%varFix$ obtained by F2 and the values obtained by F1 and VbM formulations. Clearly, pre-processing procedure did not help any of the formulations in improving its results, especially on hard instances. As in the above experiment, the results are filtered by keeping only instances where optimal solutions were found by the three formulations, and they are reported in Table B.2. The results show that F2 formulation was the fastest for all subsets, except subset 30 where F1 is faster, thanks to the pre-processing.

Conclusion. This experiment with pre-processing has shown that F2 is the best formulation compared to F1 and VbM. F2 was able to compute better solutions than the other formulations. This is the same conclusion as in the experiment without pre-processing. So, pre-processing procedure did not help improving the performances of the formulations.

Table B.1: Comparison of VbM, F1 and F2 formulations with pre-processing

	S	10	20	30	40	50	60	70
	t_{min}	0.03	0.16	0.69	1.57	5.29	11.60	22.84
	t_{avg}	0.20	695.97	599.04	778.94	815.74	810.56	839.41
	t_{max}	0.66	902.00	901.29	1087.96	908.59	919.49	936.85
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
VbM	d_{avg}	**0.00**	0.53	22.58	37.63	60.37	76.81	87.10
	d_{max}	0.00	6.67	103.45	127.06	142.86	193.22	655.81
	η	**100**	27	10	10	10	10	10
	η'	**100**	89	10	10	10	10	10
	% varFix	42.40	20.46	14.60	10.74	10.00	10.00	10.00
	t_{min}	0.02	0.06	0.23	0.40	2.01	3.63	9.04
	t_{avg}	0.14	17.76	735.47	788.75	813.07	817.64	828.86
	t_{max}	0.59	185.24	901.32	904.12	912.90	919.85	977.23
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
F1	d_{avg}	**0.00**	**0.00**	2.57	4.58	11.04	13.08	11.22
	d_{max}	0.00	0.00	20.69	26.32	47.85	41.94	85.71
	η	**100**	**100**	22	14	10	10	10
	η'	**100**	**100**	60	33	17	14	23
	% varFix	43.08	19.33	14.33	11.18	10.13	10.00	10.00
	t_{min}	0.03	0.10	0.22	0.34	0.96	1.27	2.23
	t_{avg}	**0.11**	**1.11**	**373.14**	**613.99**	**776.85**	**806.35**	**806.92**
	t_{max}	0.29	5.81	900.56	901.54	903.49	905.15	913.03
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
F2	d_{avg}	**0.00**	**0.00**	**0.05**	**0.06**	**0.29**	**0.02**	**0.59**
	d_{max}	0.00	0.00	2.76	3.23	12.59	1.55	11.88
	η	**100**	**100**	**75**	**39**	**18**	**11**	**11**
	η'	**100**	**100**	**98**	**98**	**97**	**99**	**90**
	% varFix	**48.77**	**24.94**	**17.30**	**13.77**	**11.49**	**10.43**	**10.39**

Table B.2: Comparison of VbM, F1 and F2 formulations with pre-processing - optimal solutions

	S	10	20	30	40	50	60	70
	t_{min}	0.03	0.16	0.69	1.57	5.29	11.60	22.84
	t_{avg}	0.20	151.16	0.79	2.30	5.85	13.25	27.05
VbM	t_{max}	0.66	797.99	1.16	2.76	6.84	16.97	32.45
	η	100	27	10	10	10	10	10
	% varFix	42.40	53.29	100.00	100.00	100	100.00	100.00
	t_{min}	0.02	0.06	0.23	0.40	2.01	3.63	9.04
	t_{avg}	0.14	1.00	**0.53**	1.62	3.14	6.32	13.63
F1	t_{max}	0.59	15.60	0.78	3.45	4.59	9.39	22.10
	η	100	27	10	10	10	10	10
	% varFix	43.08	55.08	100.00	100.00	100	100.00	100.00
	t_{min}	0.03	0.10	0.22	0.34	0.96	1.27	2.23
	t_{avg}	**0.11**	**0.25**	0.59	**0.78**	**1.35**	**3.10**	**3.88**
F2	t_{max}	0.29	0.73	1.47	1.594	2.00	6.40	5.89
	η	100	27	10	10	10	10	10
	% varFix	48.77	58.58	100.00	100.00	100.00	100.00	100.00

B.2 F3 experiments results

In addition to the experiments presented in Section 5.2.3.3, F3 formulation is evaluated on other databases such as PROTEIN and SYNTHETIC-100. PROTEIN database has a density of 16% and contains GED instances. So, evaluating F3 with PROTEIN instances will confirm the conclusions of the experiment on CMU-HOUSE database (both have the same density). The SYNTHETIC-100 database contains large graph instances, and has different subsets and densities. Using this database will show the limits of F3 and F2 formulations.

Table B.3: Evaluation of F3 on PROTEIN database

	S	20	30	40	mixed
F3	t_{min}	0.03	0.09	0.14	0.03
	t_{avg}	26.83	**132.00**	**575.70**	**180.36**
	t_{max}	215.59	856.79	900.52	900.66
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	0.09	**0.00**
	d_{max}	0.00	0.00	1.40	0.00
	η	**100**	**100**	**63**	**96**
	η'	**100**	**100**	**82**	**100**
F2	t_{min}	0.02	0.06	0.09	0.06
	t_{avg}	**18.92**	155.76	647.09	208.50
	t_{max}	330.88	899.99	900.27	900.08
	d_{min}	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	0.01	**0.05**	**0.00**
	d_{max}	0.00	0.19	0.42	0.19
	η	**100**	94	40	89
	η'	**100**	97	74	98

Table B.4: Evaluation of F3 on PROTEIN database - optimal solutions

	S	20	30	40	mixed
F3	t_{min}	0.03	0.09	0.14	0.03
	t_{avg}	26.83	120.83	**209.89**	115.21
	t_{max}	215.59	856.79	737.04	698.46
	η	100	94	36	86
F2	t_{min}	0.02	0.06	0.09	0.06
	t_{avg}	**18.92**	**108.26**	236.43	**105.38**
	t_{max}	330.88	728.63	817.30	670.49
	η	100	94	36	86

B.2.1 Evaluations on PROTEIN database

The results of this experiment are reported in Table B.3. The average deviation indicator does not give a clear idea of which formulation is better. Because, F3 has scored 0% deviations for subsets 20, 30 and *mixed*, while F2 has the smallest deviations for subsets 20, 40 and *mixed*. Instead, looking at η indicator shows that F3 has solved more instances to optimality than F2. The same when looking at η', F3 has computed best solutions for more instances than F2 and for all subsets. F2 has found all optimal solutions only on easy instances (subset 20), but it did not succeed in finding all of them for the other subsets. Except for subset 20, F3 is faster on the average than F2, with a gap of $100s$ on *mixed* subset.

Table B.4 shows the results after selecting only instances solved to optimality by F2 and F3 formulations. F2 seems to be faster than F3 for subsets 20, 30 and *mixed*, but the difference is at most $10s$. F3 is faster than F2 for subset 40, with $t_{avg} = 209.89s$ against $t_{avg} = 236.43s$.

Conclusion. PROTEIN is a dense graph database ($D = 16\%$) and again the second hypothesis is confirmed. F3 formulation is more effective than F2 in solving PROTEIN instances. Mainly, because it has less constraints than F2 formulation.

B.2.2 Evaluations on SYNTHETIC-100 database

The results obtained for subsets 10 and 20 are shown in Table B.5. It was not possible to run F2 formulation on the other subsets, because CPLEX was not able to compute a feasible incumbent solution in $900s$. F2 has reached its limit, therefore other subsets are discarded. Anyway, even instances of subsets 10 and 20 are hard enough, so neither F3 nor F2 were able to find an optimal solution for any instance, i.e. η is always 0 for both subsets and formulations, and t_{avg} is always $900s$. However, F3 has always computed the best solutions, which explains the average deviations at 0% and η' at 100. F2 has scored bad average deviations for subsets 10 and 20 (78.53% and 44.66%) compared to F3 average deviations.

Conclusion. Clearly, F3 has performed better than F2 on very big and high-dense graphs, which makes it more suitable for solving this kind of instances.

B.3 Local branching experiments results

This section presents the results of the additional experiments conducted on many databases during the evaluation of the proposed local branching heuristic.

B.3.1 Effectiveness of LocBra w.r.t. competitor heuristics

Besides the results obtained when testing on PROTEIN instances (Section 5.3.1.1), the same experiments are done on two other databases: MUTA and HOUSE-REF. All the

Table B.5: Evaluation of F3 on SYNTHETIC-100 database

	D	10	20
F3	t_{min}	900.23	901.44
	t_{avg}	900.54	901.78
	t_{max}	900.94	914.35
	d_{min}	0.00	0.00
	d_{avg}	**0.00**	**0.00**
	d_{max}	0.00	0.00
	η	0	0
	η'	**100**	**100**
F2	t_{min}	900.44	901.22
	t_{avg}	900.68	901.98
	t_{max}	901.11	902.62
	d_{min}	39.00	25.64
	d_{avg}	78.53	44.66
	d_{max}	897.09	281.12
	η	0	0
	η'	0	0

results are presented and discussed in this section.

B.3.1.1 Evaluations on MUTA database

Although, it is a GED^{EnA} database, it is still a reference database to evaluate GED methods. Because, it contains different sizes of graphs and the graphs represent chemical molecules, and there is a real application behind.

Default versions. The following are the values of the parameters set for each method.

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 900s$, $node\_time\_limit = UB\_time\_limit = 180s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
CPLEX-t	$t = 900$
CPLEX-LocBra-t	$t = 180$
BeamSearch-α	$\alpha = 5$
SBPBeam-α	$\alpha = 5$
IPFP-it	$it = 10$
GNCCP-d	$d = 0.1$

The results of this experiment are reported in Table B.6. They show that *LocBra* has obtained the smallest average deviations d_{avg} for all subsets. *CPLEX-900* and *CPLEX-LocBra-180* come in the second and third place respectively, with d_{avg} values better than the rest of the heuristics. On easy instances, *LocBra*, *CPLEX-900* and *CPLEX-LocBra-180* were able to solve all instances to optimality, that is why their η_I values are equal to

Table B.6: LocBra vs. heuristics on MUTA instances

	S	10	20	30	40	50	60	70	mixed
LocBra	t_{min}	0.00	0.08	0.48	0.58	1.56	4.38	6.63	0.03
	t_{avg}	0.13	3.49	701.94	784.64	810.29	810.94	811.40	463.55
	t_{max}	0.50	28.35	900.64	900.58	900.00	900.63	900.78	900.64
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	**0.58**	**0.90**	**1.25**	**0.73**	**2.47**	**0.22**
	d_{max}	0.00	0.00	7.14	14.12	10.34	25.42	58.93	6.63
	η_I	**100**	**100**	87	80	**76**	**85**	**70**	**88**
CPLEX-900	t_{min}	0.02	0.08	0.33	0.51	2.29	4.87	8.02	0.05
	t_{avg}	0.07	2.90	442.46	634.98	763.79	811.47	813.00	388.83
	t_{max}	0.17	25.82	900.28	901.09	901.50	901.16	902.06	902.04
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	0.94	1.04	1.89	3.99	4.22	0.59
	d_{max}	0.00	0.00	32.88	15.79	16.11	19.67	19.61	6.59
	η_I	**100**	**100**	**90**	**85**	63	40	47	81
CPLEX-LocBra-180	t_{min}	0.02	0.08	0.22	0.50	1.53	4.12	6.22	0.03
	t_{avg}	0.11	3.18	152.86	159.86	167.37	176.70	183.54	97.40
	t_{max}	0.31	26.49	183.72	190.60	197.29	222.46	284.02	209.56
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	4.21	8.37	10.46	11.81	14.50	2.09
	d_{max}	0.00	0.00	37.97	38.40	28.57	38.85	112.50	21.13
	η_I	**100**	**100**	59	20	11	11	11	56
BeamSearch-5	t_{min}	0.00	0.00	0.01	0.01	0.02	0.04	0.06	0.01
	t_{avg}	**0.00**	**0.00**	**0.01**	**0.03**	**0.07**	**0.11**	**0.18**	**0.09**
	t_{max}	0.07	0.02	0.04	0.11	0.09	0.13	0.22	0.21
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	15.17	36.60	46.56	57.30	68.77	55.49	56.28	20.30
	d_{max}	110.00	124.59	147.37	186.67	200.00	138.38	210.71	112.71
	η_I	35	10	10	10	10	10	10	12
SBPBeam-5	t_{min}	0.01	0.08	0.31	1.11	2.69	4.87	9.02	0.05
	t_{avg}	0.01	0.10	0.45	1.37	3.19	5.56	10.72	3.38
	t_{max}	0.05	0.14	0.54	1.60	3.71	6.85	12.79	12.05
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	20.43	44.90	75.70	80.76	95.19	86.02	79.92	26.17
	d_{max}	90.00	127.87	206.90	200.00	314.29	183.05	280.36	130.99
	η_I	15	10	10	10	10	10	10	10
IPFP-10	t_{min}	0.00	0.01	0.02	0.03	0.06	0.10	0.15	0.01
	t_{avg}	0.01	0.06	0.20	0.30	0.39	0.66	1.05	0.46
	t_{max}	0.08	0.20	0.35	0.59	0.56	1.01	1.49	1.39
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	3.44	10.84	17.76	20.27	20.23	20.30	18.32	6.72
	d_{max}	30.00	80.77	90.41	93.33	66.67	61.02	88.70	49.72
	η_I	69	28	14	11	10	10	11	19
GNCCP-0.1	t_{min}	0.02	0.12	0.38	0.89	1.68	2.88	4.59	0.15
	t_{avg}	0.16	1.30	4.77	11.78	22.08	72.29	111.30	28.53
	t_{max}	0.29	2.52	10.86	31.58	73.46	145.53	255.88	218.99
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	13.29	22.00	26.39	20.66	29.14	16.56	18.87	9.92
	d_{max}	411.43	400.00	188.79	119.12	205.36	174.14	101.79	125.16
	η_I	75	35	8	8	5	11	8	17

100. For subsets 30 (resp. 40), *CPLEX-900* has found smaller distances for 3 (resp. 5) instances than *LocBra*. But, *LocBra*'s average deviations are still smaller for these subsets, which means that *CPLEX-900* has found bad solutions when it did not find the best ones. In addition, for subsets 50, 60 and 70 (hard instances) and *mixed*, *LocBra* has the highest values of η_I, providing all best solutions. The difference is quite important between *CPLEX-900* and *LocBra*, such as on subset 60, where *CPLEX-900*'s $\eta_I = 40$ and *LocBra*'s $\eta_I = 85$. The rest of the heuristics have scored average deviations worse than *LocBra* and CPLEX-based heuristics. *IPFP-10* and *GNCCP-0.1* are better than *BeamSearch-5* and *SBPBeam-5*, in terms of solutions quality. The average deviations of *SBPBeam-5* reaches 95% on subset 50. Regarding the running time, *BeamSearch-5* is the fastest with t_{avg} at most $0.18s$, while other methods such as *LocBra* and *CPLEX-900* reaches $900s$.

Extended versions. The parameter values, given here, are set empirically to extend the running time of the heuristics picked from the literature to reach approximately the $900s$ given to LocBra.

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 900s$, $node\_time\_limit = UB\_time\_limit = 180s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
CPLEX-LocBra-t	$t = 800$
BeamSearch-α	$\alpha = 15000$
SBPBeam-α	$\alpha = 400$
IPFP-it	$it = 20000$
GNCCP-d	$d = 0.03$

Table B.7 shows the results of this experiment. Again, *LocBra* has obtained the best average deviations for all subsets. As well, *LocBra* has the best η_I values for all subsets, except for subset 30 where *CPLEX-LocBra-800* seems to have found better solutions for three instances (91 against 88). Extending the running time of *BeamSearch* and *SBPBeam* did not actually help improving their performances, they still have d_{avg} higher than 90%. It is not the case for *IPFP* and *GNCCP*: the additional running time have helped in improving their results, yet not so much because their average deviations are still far and reaches more than 20% (on subsets 30 and 50) comparing to *LocBra*. Considering the average running time t_{avg}, *LocBra* is the fastest when solving easy instances of subsets 10 and 20. Then, *GNCCP-0.03* becomes the fastest for the rest of the subsets, except for subset 70 where *CPLEX-LocBra-800* is faster.

Conclusion. LocBra heuristic was able to compute better solutions for MUTA instances than the other heuristics. In both the default and the extended versions, LocBra has outperformed CPLEX-based methods and the four heuristics selected from the literature. Yet, LocBra is slower than other existing heuristics, such as BeamSearch and IPFP.

B.3.1.2 Evaluations on HOUSE-REF database

The graphs in this database has attributes on their edges, so it is a GED database.

Table B.7: LocBra vs. heuristics with extended running time on MUTA instances

	S	10	20	30	40	50	60	70	mixed
LocBra	t_{min}	0.00	0.08	0.48	0.58	1.56	4.38	6.63	0.03
	t_{avg}	**0.13**	**3.49**	701.94	784.64	810.29	810.94	811.40	463.55
	t_{max}	0.50	28.35	900.64	900.58	900.00	900.63	900.78	900.64
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	**0.53**	**0.72**	**0.71**	**0.30**	**1.11**	**0.04**
	d_{max}	0.00	0.00	6.67	14.12	8.19	7.06	36.92	2.56
	η_I	**100**	**100**	88	**82**	**85**	**92**	**81**	**97**
CPLEX-LocBra-800	t_{min}	0.03	0.12	0.51	1.06	2.45	5.34	8.44	0.16
	t_{avg}	0.21	3.69	428.55	597.14	716.12	736.40	**746.29**	367.19
	t_{max}	0.53	28.89	801.66	813.26	818.40	845.85	919.52	828.60
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	0.56	0.89	2.46	4.96	5.93	0.93
	d_{max}	0.00	0.00	15.79	13.19	14.88	19.63	35.29	9.33
	η_I	**100**	**100**	**91**	**82**	54	28	20	71
BeamSearch-15000	t_{min}	0.00	0.00	0.03	0.10	0.55	0.24	2.28	0.03
	t_{avg}	8.57	80.65	167.48	279.11	439.68	640.29	938.66	828.52
	t_{max}	31.52	118.71	230.63	419.73	771.90	878.89	1385.11	1800.00
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-
	d_{avg}	1.35	26.66	46.70	50.60	60.04	54.41	49.35	-
	d_{max}	30.00	142.31	165.52	180.00	150.00	145.16	181.54	-
	η_I	88	12	10	10	10	10	10	-
SBPBeam-400	t_{min}	0.76	9.02	39.85	116.11	288.38	548.04	1019.35	1.98
	t_{avg}	0.84	10.02	47.65	139.75	322.43	590.86	1154.86	326.64
	t_{max}	0.96	11.27	54.11	152.34	360.47	657.26	1309.69	1225.92
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	20.43	44.90	75.58	80.39	93.88	85.08	77.01	25.68
	d_{max}	90.00	127.87	206.90	200.00	278.26	173.79	227.69	130.99
	η_I	15	10	10	10	10	10	10	10
IPFP-20000	t_{min}	0.00	0.01	0.02	0.03	0.11	0.10	0.18	0.01
	t_{avg}	1.20	9.62	48.90	115.14	240.54	528.82	903.00	303.21
	t_{max}	8.52	53.83	165.44	456.93	771.64	1620.19	2838.92	1827.03
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	3.44	10.18	15.85	15.98	16.06	13.30	10.49	5.12
	d_{max}	30.00	80.77	90.41	46.67	39.13	37.50	60.00	38.03
	η_I	69	29	15	2	10	12	21	22
GNCCP-0.03	t_{min}	0.03	0.18	0.58	1.26	2.44	4.33	6.65	0.25
	t_{avg}	0.55	6.41	**29.80**	**81.24**	**195.89**	**396.37**	946.25	**185.55**
	t_{max}	1.13	16.81	71.94	167.06	450.41	797.39	2330.57	1398.72
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	3.23	10.67	23.63	20.56	21.42	11.52	13.15	9.24
	d_{max}	90.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
	η_I	81	34	4	7	6	15	20	17

Table B.8: LocBra vs. heuristics on HOUSE-REF instances

	LocBra	CPLEX-10	CPLEX-LocBra-9	BeamSearch-5	SBPBeam-5	IPFP-10	GNCCP-0.1
t_{min}	9.91	9.92	25.38	0.03	7.54	0.03	6.85
t_{avg}	10.04	10.01	47.98	**0.14**	8.50	0.18	9.61
t_{max}	10.11	10.22	150.57	0.54	9.72	0.32	11.70
d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
d_{avg}	**57.43**	212.00	83.05	613.80	317.31	301.50	361.98
d_{max}	1166.18	1791.69	741.70	4332.69	5502.39	34308.00	32426.89
η_I	332	125	271	133	126	308	**439**

Default versions. The default parameters are set to the following:

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 10s$, $node\_time\_limit = 2s$, $UB\_time\_limit = 4s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
CPLEX-t	$t = 10$
CPLEX-LocBra-t	$t = 9$
BeamSearch-α	$\alpha = 5$
SBPBeam-α	$\alpha = 5$
IPFP-it	$it = 10$
GNCCP-d	$d = 0.1$

Based on the results shown in Table B.8, *LocBra* seems to have the smallest average deviation 57.43%, followed by *CPLEX-LocBra-9* with $d_{avg} = 83.05\%$. *CPLEX-10* comes third with $d_{avg} = 212\%$. *BeamSearch-5* has the worst average deviation among all the heuristics with d_{avg} more than 600%. The reason why the average and max deviations are very high for some heuristics is because the optimal solutions values are very small, and the heuristics did not converge to close solutions. It seems that the heuristics have found either the optimal solutions, or feasible solutions that are far from the optimal ones. Regarding the η_I values, *GNCCP-0.1* has scored the highest 439, but this is not reflected in its d_{avg} which is very bad compared to *LocBra* d_{avg}. This means that *GNCCP-0.1* has found the optimal or best solutions for 439 instances, but it has found really bad solutions for the rest of the instances. Again, as in previous experiments, *BeamSearch-5* is the fastest with the smallest average running time.

Table B.9: LocBra vs. heuristics with extended running time on HOUSE-REF instances

	LocBra	CPLEX-100	CPLEX-LocBra-90	BeamSearch-2500	SBPBeam-48	IPFP-7000	GNCCP-0.01
t_{min}	99.93	49.44	89.47	0.05	83.19	0.03	94.00
t_{avg}	100.05	99.17	126.90	77.02	93.25	**14.87**	95.67
t_{max}	100.40	100.45	179.93	120.38	104.16	113.57	110.27
d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
d_{avg}	2.69	67.76	56.81	544.12	345.71	221.76	**0.00**
d_{max}	88.88	510.39	492.62	3146.84	5502.39	34308.00	0.04
η_I	532	296	298	143	126	370	**626**

Extended versions. In the extended version, *LocBra* is set to maximum time limit of $100s$. The rest of the heuristics have the appropriate parameter values to reach the same running time.

LocBra	$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 100s$, $node\_time\_limit = UB\_time\_limit = 30s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$
CPLEX-t	$t = 100$
CPLEX-LocBra-t	$t = 90$
BeamSearch-α	$\alpha = 2500$
SBPBeam-α	$\alpha = 48$
IPFP-it	$it = 7000$
GNCCP-d	$d = 0.01$

The results given in Table B.9 show that for the first time *GNCCP-0.01* was able to perform better than *LocBra*, with $d_{avg} = 0.00\%$ and the highest $\eta_I = 626$. *LocBra* comes in the second place with $d_{avg} = 2.69\%$ and $\eta_I = 532$. Extending the running time of GNCCP was helpful and it performs better than the other heuristics. The same order as in the default version is maintained, so *CPLEX-LocBra-90* comes third, followed by *CPLEX-100*, and *BeamSearch-2500* at the last position. In terms of average running time, the best heuristic is *IPFP-7000* with $t_{avg} = 14.87s$, which is way faster than the rest of the heuristics.

Conclusion. In the default version, LocBra has succeeded in solving efficiently HOUSE-REF instances. However and remarkably, GNCCP was able to perform better than LocBra in the extended version. Yet, the difference is pretty much small with 2.7% on average deviation.

B.3.2 Effectiveness of LocBra w.r.t. an exact method

This section presents the results of additional experiments done to evaluate LocBra against an exact method. The results obtained on MUTA and HOURSE-REF databases are reported here.

Table B.10: LocBra vs. Exact solution on MUTA instances

	CPLEX	LocBra								
S	η_I	t_{min}	t_{avg}	t_{max}	d_{min}	d_{avg}	d_{max}	η_I	η'_I	η''_I
10	100	0.00	0.13	0.50	0.00	0.00	0.00	**100**	**100**	0
20	100	0.08	3.49	28.35	0.00	0.00	0.00	**100**	**100**	0
30	100	0.48	701.94	900.64	0.00	**1.05**	11.21	24	**80**	0
40	100	0.58	784.64	900.58	0.00	**1.79**	14.12	13	**66**	0
50	98	1.72	810.56	900.64	0.00	**3.33**	13.27	10	**43**	0
60	85	4.38	810.94	900.63	0.00	**5.70**	25.42	10	**15**	0
70	35	6.63	811.40	900.78	0.00	**10.78**	58.93	10	**12**	0
mixed	91	0.03	463.55	900.64	0.00	**0.86**	10.29	49	**68**	0

B.3.2.1 Evaluations on MUTA database

LocBra parameters are set to the following values:
$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 900s$, $node\_time\_limit = UB\_time\_limit = 180s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$. The solutions computed by LocBra are evaluated based on the optimal/best solutions that were obtained by solving JH formulation (more details in Section 4.4.6.2). Those solutions were obtained after letting CPLEX solving JH formulation for 10 hours on MUTA instances.

Based on Table B.10, and on very easy instances (subsets 10 and 20), LocBra has found all optimal solutions as *CPLEX*. A positive average deviation of 1% starts to appear on subset 30, and it keeps growing until reaching 10.78% on subset 70. In contrast, the η'_I values decrease with the increase of the size of the graphs. LocBra was not able to compute in the 900s solutions better than the best ones computed by *CPLEX* for all the subsets (i.e. all η''_I are equal to 0).

Conclusion. Those results show that LocBra in 900s can compute solutions at 10.78% far from the optimal/best solutions computed by solving JH formulation during 10 hours. This number is, of course, in the worst case and it is considerably good.

Table B.11: LocBra vs. Exact solution on HOUSE-REF instances

	CPLEX-900	LocBra
t_{min}	66.07	99.93
t_{avg}	416.75	**100.05**
t_{max}	900.67	**100.40**
d_{min}	-	**-46.91**
d_{avg}	-	**2.54**
d_{max}	-	88.88
η_I	633	0
η'_I	-	**549**
η''_I	-	**6**

B.3.2.2 Evaluations on HOUSE-REF database

LocBra parameters are set to the following values:
$\pi = 20$, $\pi\_dv = 30$, $total\_time\_limit = 100s$, $node\_time\_limit = UB\_time\_limit = 30s$, $dv\_max = 5$, $l\_max = 3$, $dv\_cons\_max = 2$. For this database, CPLEX was ran with a maximum time of 900s and not 10 hours. But for this database, the 900s were enough to find optimal solutions for 95% of the instances.

The results shown Table B.11 reveal that LocBra's solutions are very close to optimal/best ones (2.5% on average). As well, LocBra has found better solutions for 6 instances, which explains the minimum deviation of -46.91%. Moreover, it has computed solutions for 549 instances, equal to solutions computed by *CPLEX-900.* All these results are achieved in a maximum running time of 100s, while *CPLEX-900* needed more than 400s on average to solve the instances.

Conclusion. The results have shown that LocBra was able to compute very good quality solutions that are far by 2.5% from the optimal/best solutions. LocBra has achieved these results with a maximum running time of 100s, while CPLEX needed more than 400s on average. Moreover, LocBra in 100s was able to find better solutions than CPLEX in 900s for 6 instances.

B.4 VPLS experiments results

This section presents additional experiments results obtained when evaluating VPLS on two extra databases: MUTA and HOUSE-REF.

B.4.1 Effectiveness of VPLS w.r.t. competitor heuristics

The results of evaluating VPLS over MUTA and HOUSE-REF are presented in this section.

B.4.1.1 Evaluations on MUTA database

Although, it is a GED^{EnA} database, it is still a reference database to evaluate GED methods. Because, it contains different sizes of graphs and the graphs represent chemical molecules, and there is a real application behind.

Default versions. The following are the values of the parameters set for each method.

$VPLS$	$cons\_sol\_max = 5$, $total\_time\_limit = 900s$, $node\_time\_limit = UB\_time\_limit = 180s$,
CPLEX-t	$t = 900$
BeamSearch-α	$\alpha = 5$
SBPBeam-α	$\alpha = 5$
IPFP-it	$it = 10$
GNCCP-d	$d = 0.1$

The results obtained are reported in Table B.12. *CPLEX-900* heuristic has scored the smallest average deviations for all subsets, except subset 70 where *VPLS* has a d_{avg} smaller by 1%. The exact same conclusion can be drawn when looking at η_I values. Next, *IPFP-10* comes third outperforming *GNCCP-0.1*, with a small fall back on subset 60 where *GNCCP-0.1* has a better d_{avg} of 14% against 17.7% by *IPFP-0.1*. Beam-search based methods come at last, with *BeamSearch-5* better on the average deviation than *SBPBeam-5*, but both are far from *CPLEX-900* and *VPLS*. In terms of average running time, *BeamSearch-5* is the fastest.

Extended versions. The parameter values are set empirically to extend the running time of the heuristics to reach approximately the $900s$ as the running time of VPLS.

$VPLS$	$cons\_sol\_max = 5$, $total\_time\_limit = 900s$, $node\_time\_limit = UB\_time\_limit = 180s$,
BeamSearch-α	$\alpha = 15000$
SBPBeam-α	$\alpha = 400$
IPFP-it	$it = 20000$
GNCCP-d	$d = 0.03$

The results presented in Table B.13 show that *VPLS* has performed better than other heuristics. And it has obtained the best deviations and η_I values for all subsets. Extending

Table B.12: VPLS vs. heuristics on MUTA instances

	S	10	20	30	40	50	60	70	mixed
VPLS	t_{min}	0.02	0.02	0.16	0.37	0.86	2.98	5.07	0.05
	t_{avg}	0.05	2.37	147.92	163.04	297.61	225.00	503.80	123.25
	t_{max}	0.12	20.59	446.07	244.73	883.51	598.67	899.97	899.97
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	3.04	4.63	3.86	2.62	**2.43**	0.89
	d_{max}	0.00	0.00	51.72	22.06	18.00	20.34	46.96	16.90
	η_I	**100**	**100**	68	37	44	58	**64**	76
CPLEX-900	t_{min}	0.02	0.08	0.33	0.51	2.29	4.87	8.02	0.05
	t_{avg}	0.07	2.90	442.46	634.98	763.79	811.47	813.00	388.83
	t_{max}	0.17	25.82	900.28	901.09	901.50	901.16	902.06	902.04
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	**0.36**	**0.67**	**0.71**	**1.80**	3.12	**0.18**
	d_{max}	0.00	0.00	6.59	15.79	11.61	11.76	27.12	4.48
	η_I	**100**	**100**	**92**	**91**	**87**	**67**	53	**91**
BeamSearch-5	t_{min}	0.00	0.00	0.01	0.01	0.02	0.04	0.06	0.01
	t_{avg}	**0.00**	**0.00**	**0.01**	**0.03**	**0.07**	**0.11**	**0.18**	**0.09**
	t_{max}	0.07	0.02	0.04	0.11	0.09	0.13	0.22	0.21
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	15.17	36.60	45.71	56.70	66.90	52.17	54.50	19.75
	d_{max}	110.00	124.59	124.14	186.67	200.00	138.38	210.71	112.71
	η_I	35	10	10	10	10	10	10	12
SBPBeam-5	t_{min}	0.01	0.08	0.31	1.11	2.69	4.87	9.02	0.05
	t_{avg}	0.01	0.10	0.45	1.37	3.19	5.56	10.72	3.38
	t_{max}	0.05	0.14	0.54	1.60	3.71	6.85	12.79	12.05
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	20.43	44.90	74.68	80.03	92.84	81.79	77.89	25.53
	d_{max}	90.00	127.87	206.90	200.00	314.29	183.05	280.36	130.99
	η_I	15	10	10	10	10	10	10	10
IPFP-10	t_{min}	0.00	0.01	0.02	0.03	0.06	0.10	0.15	0.01
	t_{avg}	0.01	0.06	0.20	0.30	0.39	0.66	1.05	0.46
	t_{max}	0.08	0.20	0.35	0.59	0.56	1.01	1.49	1.39
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	3.44	10.84	17.07	19.84	18.85	17.70	17.02	6.27
	d_{max}	30.00	80.77	90.41	93.33	66.67	61.02	88.70	49.72
	η_I	69	28	14	11	10	10	10	19
GNCCP-0.1	t_{min}	0.02	0.12	0.38	0.89	1.68	2.88	4.59	0.15
	t_{avg}	0.16	1.30	4.77	11.78	22.08	72.29	111.30	28.53
	t_{max}	0.29	2.52	10.86	31.58	73.46	145.53	255.88	218.99
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	13.29	22.00	25.70	20.24	27.69	14.08	17.64	9.47
	d_{max}	411.43	400.00	188.79	119.12	205.36	160.66	101.79	120.99
	η_I	75	35	7	9	5	10	10	18

Table B.13: VPLS vs. heuristics with extended running time on MUTA instances

	S	10	20	30	40	50	60	70	mixed
VPLS	t_{min}	0.02	0.02	0.16	0.37	0.86	2.98	5.07	0.05
	t_{avg}	**0.05**	**2.37**	147.92	163.04	297.61	**225.00**	**503.80**	**123.25**
	t_{max}	0.12	20.59	446.07	244.73	883.51	598.67	899.97	899.97
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	**0.00**	**0.00**	**0.58**	**0.27**	**0.40**	**0.61**	**0.94**	**0.05**
	d_{max}	0.00	0.00	25.71	5.56	11.39	15.38	27.07	1.39
	η_I	**100**	**100**	**97**	**92**	**91**	**86**	**82**	**96**
BeamSearch-15000	t_{min}	0.00	0.00	0.03	0.10	0.55	0.24	2.28	0.03
	t_{avg}	8.57	80.65	167.48	279.11	439.68	640.29	938.66	828.52
	t_{max}	31.52	118.71	230.63	419.73	771.90	878.89	1385.11	1800.00
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-
	d_{avg}	1.35	26.66	42.46	43.78	53.47	48.55	47.14	-
	d_{max}	30.00	142.31	147.95	180.00	125.81	120.00	169.12	-
	η_I	88	12	10	10	10	10	10	-
SBPBeam-400	t_{min}	0.76	9.02	39.85	116.11	288.38	548.04	1019.35	1.98
	t_{avg}	0.84	10.02	47.65	139.75	322.43	590.86	1154.86	326.64
	t_{max}	0.96	11.27	54.11	152.34	360.47	657.26	1309.69	1225.92
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	20.43	44.90	70.05	71.96	85.91	77.94	74.60	23.91
	d_{max}	90.00	127.87	174.68	200.00	304.76	157.69	227.69	114.57
	η_I	15	10	10	10	10	10	10	10
IPFP-20000	t_{min}	0.00	0.01	0.02	0.03	0.11	0.10	0.18	0.01
	t_{avg}	1.20	9.62	48.90	115.14	240.54	528.82	903.00	303.21
	t_{max}	8.52	53.83	165.44	456.93	771.64	1620.19	2838.92	1827.03
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	3.44	10.18	12.59	10.96	11.55	9.19	9.04	3.96
	d_{max}	30.00	80.77	90.41	46.67	47.62	33.64	52.94	33.57
	η_I	69	29	20	19	18	19	21	24
GNCCP-0.03	t_{min}	0.03	0.18	0.58	1.26	2.44	4.33	6.65	0.25
	t_{avg}	0.55	6.41	**29.80**	**81.24**	**195.89**	396.37	946.25	185.55
	t_{max}	1.13	16.81	71.94	167.06	450.41	797.39	2330.57	1398.72
	d_{min}	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	d_{avg}	3.23	10.67	20.25	15.59	16.97	7.54	11.77	8.12
	d_{max}	90.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
	η_I	81	34	7	19	14	24	22	20

the running time has helped *IPFP-20000* and *GNCCP-0.03* in finding better solutions and decreasing their deviations. However, it is not enough and they are still far from *VPLS*, with a difference reaching more than 20% on average deviation. In addition, *VPLS* is faster than the other heuristics on easy instances (subsets 10 and 20), and very hard instances (subsets 60, 70 and *mixed*). *GNCCP-0.03* is the fastest heuristic on the rest of the subsets.

Conclusion. VPLS heuristic has achieved better results than existing heuristics in terms of solutions quality, but it does not outperform the default behavior of the solver when solving F3 formulation.

B.4.1.2 Evaluations on HOUSE-REF database

Another GED graph database is considered to evaluate VPLS.

Default versions. The default parameters are set to the following:

VPLS	$cons\_sol\_max = 5$, $total\_time\_limit = 10s$, $node\_time\_limit = 2s$, $UB\_time\_limit = 4s$,
CPLEX-t	$t = 10$
CPLEX-LocBra-t	$t = 9$
BeamSearch-α	$\alpha = 5$
SBPBeam-α	$\alpha = 5$
IPFP-it	$it = 10$
GNCCP-d	$d = 0.1$

Table B.14: VPLS vs. heuristics on HOUSE-REF instances

	VPLS	CPLEX-10	BeamSearch-5	SBPBeam-5	IPFP-10	GNCCP-0.1
t_{min}	5.76	9.92	0.03	7.54	0.03	6.85
t_{avg}	9.24	10.01	**0.14**	8.50	0.18	9.61
t_{max}	10.03	10.22	0.54	9.72	0.32	11.70
d_{min}	0.00	0.00	0.00	0.00	0.00	0.00
d_{avg}	**13.32**	221.91	631.32	330.59	313.55	373.44
d_{max}	294.41	1791.69	4332.69	5502.39	34308.00	32426.89
η_I	**523**	120	132	126	300	433

Based on the results shown in Table B.14, *VPLS* seems to be the best heuristic in terms of solutions quality, with the best average deviation at 13.32% and η_I at 523. The second heuristic is *CPLEX-10*, with a $d_{avg} = 221.91\%$ which is very far from the d_{avg} of *VPLS*. In addition, η_I of *CPLEX-10* is 120, which is very small compared to *GNCCP-0.1* ($\eta_I = 433$). Despite the high η_I value of *GNCCP-0.1*, it is outperformed by *IPFP-10* based on the average deviation with a difference of 60%. *BeamSearch-5* is the worst heuristic in terms of solutions quality, but it is the fastest based on the average running time indicator.

Table B.15: VPLS vs. heuristics with extended running time on HOUSE-REF instances

	VPLS	CPLEX-100	BeamSearch-2500	SBPBeam-48	IPFP-7000	GNCCP-0.01
t_{min}	31.92	49.44	0.05	83.19	0.03	94.00
t_{avg}	68.38	99.17	77.02	93.25	**14.87**	95.67
t_{max}	100.04	100.45	120.38	104.16	113.57	110.27
d_{min}	0.00	0.00	0.00	0.00	0.00	0.00
d_{avg}	4.47	67.76	544.12	345.71	221.76	**0.00**
d_{max}	181.60	510.39	3146.84	5502.39	34308.00	0.04
η_I	573	296	143	126	370	**626**

Extended versions. In the extended version, *VPLS* is set to maximum time limit of 100s. The rest of the heuristics have the appropriate parameter values to reach the same running time.

VPLS	$cons\_sol\_max = 5$, $total\_time\_limit = 100s$, $node\_time\_limit = UB\_time\_limit = 30s$,
CPLEX-t	$t = 100$
BeamSearch-α	$\alpha = 2500$
SBPBeam-α	$\alpha = 48$
IPFP-it	$it = 7000$
GNCCP-d	$d = 0.01$

The results given in Table B.15 show that the best heuristic is *GNCCP-0.01*, with the best average deviation of 0.00%, and the highest number of best solutions of 626 out of 660. *VPLS* heuristic comes next with an average deviation of 4.47% and η_I of 573. *CPLEX-100* comes third with smaller average deviation. The gap between the first three heuristics and the others starts to grow from 200% with *IPFP-7000* to exceed 500% with *BeamSearch-2500*. With respect to the average running time, *IPFP-7000* is the fastest heuristic, followed by *VPLS* which is better than the others.

Conclusion. VPLS heuristic, in the default version with a maximum running time of 10s, has performed better than all existing heuristics with their default parameters. However, in the extended version, VPLS came second, right after GNCCP heuristic, which surprisingly was able to solve HOUSE-REF instances efficiently.

B.4.2 Effectiveness of VPLS w.r.t. an exact method

The results of additional experiments done over MUTA and HOUSE-REF databases are reported in this section.

B.4.2.1 Evaluations on MUTA database

VPLS parameters are set to the following values:
$cons\_sol\_max = 5$, $total\_time\_limit = 900s$, $node\_time\_limit = UB\_time\_limit = 180s$.

Table B.16: VPLS vs. Exact solution on MUTA instances

	CPLEX	VPLS								
S	η_I	t_{min}	t_{avg}	t_{max}	d_{min}	d_{avg}	d_{max}	η_I	η'_I	η''_I
10	100	0.02	0.05	0.12	0.00	0.00	0.00	**100**	**100**	0
20	100	0.02	2.37	20.59	0.00	0.00	0.00	**100**	**100**	0
30	100	0.16	147.92	446.07	0.00	**4.09**	51.72	42	**59**	0
40	100	0.37	163.04	244.73	0.00	**5.95**	28.24	16	**32**	0
50	98	0.86	297.61	883.51	0.00	**7.21**	31.58	10	**16**	0
60	85	2.98	225.00	598.67	0.00	**9.98**	30.51	10	**10**	0
70	35	5.07	503.80	899.97	0.00	**11.90**	55.05	10	**12**	0
mixed	91	0.05	123.25	899.97	0.00	**1.96**	17.14	51	**60**	0

Based on Table B.16, and on very easy instances (subsets 10 and 20), VPLS has found all the optimal solutions as *CPLEX*. A positive and slightly high average deviation of 4% starts to appear on subset 30, and it keeps growing until reaching almost 12% on subset 70. In contrast, the η'_I values decrease with the increase of the size of the graphs. All η''_I are equal to 0, which means VPLS did not compute any solution that is better than the ones computed by *CPLEX*.

Conclusion. The solutions computed by VPLS are relatively good compared to the optimal/best ones found by CPLEX. The average deviations vary between 0% and 12% as the graph size increases. Even though the running time of VPLS is set to 900s, the heuristic does not consume all the time. On hard instances the average time is at 504s.

Table B.17: VPLS vs. Exact solution on HOUSE-REF instances

	CPLEX-900	VPLS
t_{min}	66.07	31.92
t_{avg}	416.75	**68.38**
t_{max}	900.67	**100.04**
d_{min}	-	**-46.90**
d_{avg}	-	**4.34**
d_{max}	-	181.60
η_I	633	0
η'_I	-	**604**
η''_I	-	**3**

B.4.2.2 Evaluations on HOUSE-REF database

VPLS parameters values are:
$cons\_sol\_max = 5$, $total\_time\_limit = 100s$, $node\_time\_limit = UB\_time\_limit = 30s$.

The results shown Table B.17 reveal that VPLS solutions are very close to optimal/best ones (4.34% on average). As well, VPLS has found better solutions for 2 instances, which explains the minimum deviation of -46.90%. All these results are achieved in a average running time of $68s$, while *CPLEX-900* needed more than $400s$ on average to solve the instances.

Conclusion. VPLS has computed in $100s$ better solutions for 3 instances than CPLEX in $900s$. The average deviation is relatively small at 4%. In addition, VPLS is very fast compared to CPLEX, where the $t_{avg} = 68s$ for VPLS against $417s$ for CPLEX.

B.4.3 VPLS vs. LocBra experiments results

The results of the comparison of VPLS with LocBra for MUTA and HOURSE-REF databases are presented in this section.

B.4.3.1 Evaluations on MUTA database

The results are reported in Table B.18. Briefly, the smallest average deviations are scored by LocBra on all subsets. However, VPLS average deviations are not far from them, with a maximum 4.56% on subset 60. The same conclusion can be seen when looking at η_I values, where all good values are obtained by LocBra. On the other hand and when looking at the average running times, VPLS is faster than LocBra on all subsets.

Table B.18: VPLS vs. LocBra on MUTA instances

	VPLS							LocBra						
S	t_{min}	t_{avg}	t_{max}	d_{min}	d_{avg}	d_{max}	η_I	t_{min}	t_{avg}	t_{max}	d_{min}	d_{avg}	d_{max}	η_I
10	0.02	**0.05**	0.12	0.00	**0.00**	0.00	**100**	0.00	0.13	0.50	0.00	**0.00**	0.00	**100**
20	0.02	**2.37**	20.59	0.00	**0.00**	0.00	**100**	0.08	3.49	28.35	0.00	**0.00**	0.00	**100**
30	0.16	**147.92**	446.07	0.00	3.37	51.72	67	0.48	701.94	900.64	0.00	**0.35**	7.14	**91**
40	0.37	**163.04**	244.73	0.00	4.31	22.06	41	0.58	784.64	900.58	0.00	**0.21**	10.00	**96**
50	0.86	**297.61**	883.51	0.00	4.34	21.95	43	1.56	810.29	900.00	0.00	**0.54**	9.52	**87**
60	2.98	**225.00**	598.67	0.00	4.56	24.19	31	4.38	810.94	900.63	0.00	**0.47**	7.06	**88**
70	5.07	**503.80**	899.97	0.00	2.92	27.45	51	6.63	811.40	900.78	0.00	**1.83**	36.92	**71**
mixed	0.05	**123.25**	899.97	0.00	1.14	16.90	74	0.03	463.55	900.64	0.00	**0.06**	1.26	**94**

Conclusion. LocBra is better than VPLS in solving MUTA instances. The difference, however, is not that important with a maximum of 5% on the average. In terms of running time, VPLS is way faster than LocBra with a difference reaching the $500s$ on hard instances (subset 60).

B.4.3.2 Evaluations on HOUSE-REF database

Similarly, and based on the results shown in Table B.19, LocBra is better on the average deviation, and worse on the average running time. One difference appears when looking at η_I values, where VPLS has found the best solutions for 617 instances against 584 instances by LocBra.

Conclusion. VPLS was able to compute better solutions than LocBra for 33 instances. The average running time of VPLS is also better than the average running time of LocBra.

Table B.19: VPLS vs. LocBra on HOUSE-REF instances

	VPLS	LocBra
t_{min}	31.92	99.93
t_{avg}	**68.38**	100.05
t_{max}	100.04	100.40
d_{min}	0.00	0.00
d_{avg}	3.62	**1.95**
d_{max}	181.60	62.93
η_I	**617**	584