UNIVERSITÉ FRANÇOIS RABELAIS DE TOURS

ÉCOLE DOCTORALE MIPTIS

Laboratoire d'Informatique (EA 6300)

THÈSE présentée par :

Zeina Abu-Aisheh

soutenue le : 25 mai 2016

pour obtenir le grade de : Docteur de l'université François - Rabelais de Tours

Discipline/ Spécialité : INFORMATIQUE

Approches anytime et distribuées pour l'appariment de graphes

Anytime and Distributed Approaches for Graph Matching

Acknowledgments

First and foremost, I offer my sincerest gratitude to my main advisers Romain Raveaux and Jean-Yves Ramel for their endless help and support during all the phases of the thesis. I thank you Romain for having taught me how to do research step by step. Your endless help, enthusiasm, kindness and patience taught me how to be the best version of me as a researcher and a teacher. Thank you Jean-Yves for being the best director ever with whom I can discuss research at any time and from every conceivable angle. Thank you also for being a helpful friend that listens to me and gives me advice without getting bored. The words cannot express my thanks for all your help. Jean-Yves and Romain, without your support and the collaborative working, this thesis would not have been possible. I will always be grateful to you both.

I also thank Patrick Martineau for introducing me to the distributed and parallel systems domain and for the ideas he has given to me during the meetings in the thesis.

I am grateful to Prof. Christine Solnon and Prof. Francesc Serratosa for having carefully read my thesis and for having provided me their interesting remarks and questions. Thanks also to Prof. Luc Brun and Dr. Kaspar Riesen for being members of the jury and for the interesting remarks and questions they tackled on the day of the defense. It was a pleasure meeting you all in this important phase. Thank you all for being part of this adventure.

I thank Prof. Jean-Charles Billaut for having welcomed me in the laboratory of Tours. Thanks to Prof. Pierre Gaucher, Dr. Christophe Lenté, Dr. Carl Esswein and Dr. Mathieu Delalandre for helping me with the teaching stuff (Monitorat and ATER). I also thank the lovely administrative staff in the lab (particularly my top "3": Betty, Julie and Christelle) for all the open-minded discussions.

Special thanks to the RFAI team for always being kind with me and for all the parties we had together. I particularly thank Alireza, Nathalie, Moncef, Hubert, Mathieu, Nicolas Sidère and Nicolas Ragot for their help and advice.

Being in a big and open-concept office let me meet lots of people whose help was always surprising. Thanks to my office-mates (the old and the new ones): The-Anh, Aymen, Tiang Yang, Fahimeh, Nicolas, Julien, Prof. Romuald, Faiza, Gaëtan and Moncef. I thank all my colleagues for the memorable events, laughers and discussions inside and outside the laboratory. My special thanks go to:

- My friend Faiza: "always lovely, helpful and caring", we both knew how to keep this friendship stronger and stronger even when we have different point of views.

- Anh The-Anh: "for all the interesting discussions, the memorable moments of laughter, singing, walking, etc.".
- The open-minded Fahimeh: "for being truthful, for the nice discussions and for always understanding me".
- Octavio: "for being the best adviser and the most cheerful friend in the lab".
- Frédéric: "for always being here and for never getting bored of any kind of discussions".
- Gaëtan: "his help and attention were really important to me, even more than Faiza".
- Nathalie: "for always being beside me when I needed".
- Tanmoy: "for the unique and unstable friendship that depends on the season".
- Fatma, Kaouthar, Shang Lei, Mostafa and the others "thank you very much".

During this PhD, I participated in international conferences and summer schools where I met people that I would never have met if I had not been a PhD student. "Thank you Nihel, Anna, Diana, Muzzamil, Xavier and Carlos" for the quality time we spent together and for all the confidence and friendship.

I also thank the sport center "OrangeBleu" for making the final few months of the thesis less hard. The classes of Zumba, Stretching and Yoga were always here to let me feel different.

I also thank my friends in Grenoble (Frédéric Wagner, Gaëlle Calvary and Alexandre Demeure) for introducing me to research when I was in Grenoble and for their truthful emails that were always cheering me up in Tours.

The support of my friends that are just like real sisters was one of the key points of completing PhD, I thank you "Nadia, Suzanne, Faiza, Fahimeh, Julie, Eun Jin, Mona, Nihel and Fatma" for the endless help and care and for all the details that we shared. Thank you million times.

I owe a lot to my family, I thank them for the immense love, positive encouragement, and for believing in my ability to succeed specially in the moments when I lost confidence. Thank you my sister "Hanadi" for cheering me up when you knew that I was not feeling good. Thank you my elder brother "Hamoudé" for updating me with the new American and English songs, singers and TV shows. Thank you my younger brother "Firas" for your cheerful personality and for the Palestinian and Jordanian songs that always make me laugh and could definitely change my mood. Above all, I thank you my mother "Amira" for being the best mother and friend ever. Without your faith, prayers, confidence, love and encouragements, I would not have spent a single day in France. You have been the person who inspires me the most. You are the sun shine, the positive energy and number one for me. This thesis is dedicated to you and to the soul of my father "Maher" that is always with us. I love you very much.

I thank God for granting me strength and health to complete PhD successfully.

Résumé

En raison de la capacité et de l'amélioration des performances informatiques, les représentations structurelles sont devenues de plus en plus populaires dans le domaine de la reconnaissance de formes (RF). Quand les objets sont structurés à base de graphes, le problme de la comparaison d'objets revient à un problme d'appariement de graphes (Graph Matching).

Au cours de la dernière décennie, les chercheurs travaillant dans le domaine de l'appariement de graphes ont porté une attention particulière à la distance d'édition entre graphes (GED), notamment pour sa capacité à traiter différent types de graphes. GED a été ainsi appliquée sur des problématiques spécifiques qui varient de la reconnaissance de molécules à la classification d'images.

Les chercheurs se sont focalisés sur les méthodes approchées qui peuvent trouver des solutions sous-optimales, mais souvent sans aucune garantie de précision. Pour cette raison, dans cette thèse, nous nous focalisons sur les algorithmes exacts. La complexité du problème d'appariement de graphes étant NP-difficile, les méthodes exactes proposées ne peuvent être utilisées que sur des petites instances de graphes. Afin de réduire le temps de calcul, deux possibilités sont envisagées : élaguer l'espace de recherche et distribuer les calculs.

Dans cette thèse, nous avons dans un premier temps, proposé un algorithme de recherche arborescente travaillant en profondeur d'abord (Depth-First GED) et nécessitant moins de mémoire et de temps de calcul pour produire une solution. Une évaluation de toutes les solutions possibles est effectuée sans les énumérer explicitement. Les candidats sont éliminés à l'aide des bornes inférieure et supérieure. Pour trouver un compromis entre la vitesse et l'optimalité, nous avons proposé une version améliorée de Depth-First GED (appelée Anytime) qui est capable de délivrer une première solution réalisable très rapidement. Ensuite, en laissant plus de temps, l'algorithme améliore progressivement sa solution initiale dans le but de proposer de meilleures solutions jusqu'à converger vers une solution optimale.

Pour illustrer l'utilisation des méthodes Anytime, nous convertissons notre Depth-First GED en Anytime Depth-First GED. Les propriétés de ces méthodes sont analysées afin de les adapter aux problmes d'appariement de graphes, ceci en tenant en compte des résultats en terme de précision de la solution fournie par rapport à la valeur optimale ou la meilleure solution trouvée par une méthode de l'état de l'art.

Cette thése propose également des solutions initiales pour réduire le temps d'exécution des méthodes de GED exactes à l'aide de techniques de parallélisation et de distribution des calculs. Une approche paralléle et une autre distribuée sont présentées. Ces deux méthodes sont aussi basées sur la méthode Depth-First GED oú l'espace de recherche est décomposé en arbres de recherche qui sont résolus indépendamment en paralléle ou d'une maniére répartie.

Afin d'analyser les performances des méthodes proposées, nous avons non seulement évalué les méthodes GED dans un contexte de classification mais aussi à un niveau plus détaillé en mesurant la qualité de l'appariement. Les méthodes proposées sont comparées avec cinq méthodes de l'état de l'art. Mais en raison de la complexité exponentielle des algorithmes GED, la comparaison est effectuée sous contraintes de restrictions de temps de calcul et d'espace mémoire. De plus, une base de graphes (GDR4GED) avec une vérité terrain dédiée à l'évaluation des méthodes de GED est proposée. Dans cette vérité terrain, nous ajoutons des informations concernant les appariements entre paires de graphes à des bases de données publiques précédemment utilisées dans la littérature. L'ajout d'informations se compose de la meilleure distance d'édition trouvée ainsi que l'appariement "sommet à sommet" et "arc à arc" de chaque paire de graphes. Cette information permet d'évaluer la précision des méthodes de GED exactes et approchées.

De part ces expérimentations proposées, cette thèse remet en cause les réticences à l'utilisation des méthodes exactes d'appariement de grands graphes dans la pratique, ou même dans un contexte de classification. De fait, nous montrons que les méthodes "Anytime", peuvent être efficaces, aussi bien dans un objectif de comparaison que de mise en correspondence de graphes.

Mots-clés: Reconnaissance de formes, appariement de graphes, distance d'édition, Branch-and-Bound, systèmes parallèle et distribué, équilibrage de charge, évaluation de performance, contraintes de temps.

Abstract

Due to the inherent genericity of graph-based representations, and thanks to the improvement of computer capacities, structural representations have become more and more popular in the field of Pattern Recognition (PR). In a graph-based representation, vertices and their attributes describe objects (or part of them) while edges represent interrelationships between the objects. Representing objects by graphs turns the problem of object comparison into graph matching (GM) where correspondences between vertices and edges of two graphs have to be found.

In the domain of GM, over the last decade, Graph Edit Distance (GED) has been given a specific attention due to its flexibility to match many types of graphs. GED has been applied to a wide range of specific applications from molecule recognition to image classification. Researchers have shed light on the approximate methods that can find suboptimal solutions hopefully close to the optimal ones but the gap between optimal and suboptimal solutions has not been deeply studied yet. For that reason, in this thesis, we focus on exact GED algorithms. Unfortunately, exact GED methods have an exponential complexity. Thus, coming up with an exact GED algorithm that can be scaled up to match graphs involved in PR tasks is a great challenge. Two promising ways to cut-off computational time are search space pruning and distributed algorithms. To this end, we first propose a depth-first GED algorithm which requires less memory and search time. An evaluation of all possible solutions is performed without explicitly enumerating all of them. Candidates are discarded using an upper and lower bounds strategy.

To find a trade-off between speed and optimality, we describe how to convert the proposed depth-first GED method into an anytime one that is capable of delivering a first solution very quickly. It also can find a list of improved solutions and eventually converges to the optimal solution instead of providing one and only one solution (i.e., the optimal solution). With the delight of more time, anytime methods can also reach the optimal solution. To illustrate the usage of anytime GM algorithms, we convert our depth-first GED algorithm into an anytime one. We analyze the properties of such methods to solve GM problems and consider the performance in terms of accuracy of the provided solution compared to the optimal or the best one found by a state-of-the-art methods.

This thesis is also considered as a first attempt to reduce the run time of exact GED methods using parallel and distributed fashions. Two parallel and distributed GED approaches are put forward; both of them are based on the depth-first GED method. The search space is decomposed into smaller search trees which are solved independently in a parallel or a distributed manner.

To benchmark the proposed GED methods, we propose not only assessing GED methods in a classification context but also evaluating them in a graph-level one (i.e., evaluating their distance and matching accuracy). Due to the exponential complexity of exact GED algorithms and in order to obtain this kind of information about methods, we propose analyzing the behavior of the eight compared methods under time and memory constraints. In addition to the performance evaluations metrics, we propose a graph database repository dedicated to GED. In this repository, we add graph-level information to well-known and publicly used databases. Added information consists of the best found edit distance of each pair of graphs as well as their vertex-to-vertex and edge-to-edge mappings corresponding to the best found distance. This information helps in assessing the feasibility of exact and approximate GED methods.

This thesis brings into question the usual evidences that claim that it is impossible to use exact error-tolerant GM methods in real-world applications when matching large graphs, or even in a classification context. However, we argue and show that a new type of GM, referred to as anytime methods, can be successful in a graph-level context as well as a classification one.

Keywords: Pattern Recognition, Graph Matching, Graph Edit Distance, Branch-and-Bound, Distributed and Parallel Systems, Load Balancing, Performance Evaluation Metrics, Anytime Graph Matching, Time Constraints.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The comparison between two objects is a crucial operation in Pattern Recognition (PR). Lots of algorithmic tools exist for statistical pattern recognition. Such a fact proves the mathematical efficiency of this approach. However, there are two drawbacks behind choosing such an approach:

- Representing patterns of the same application by vectors limits the length of vectors. That is, all these vectors have the same length even if they differ in their sizes and complexities.
- Vectors do not represent the relationship between each part of a pattern and the other. Such a fact limits their applicability on patterns that are rather simple.

Based on the aforementioned raised points, the use of the structural approach emerged as a powerful approach for complex pattern in Pattern Recognition.

Representing objects by graphs turns the problem of object comparison into a graph matching (GM) one where an evaluation of structural and attributed similarity of two graphs have to be found. Based on that, the parts of objects can be described by vertices that are enriched with properly defined attributes while the edges of graphs represent the relationships between these parts.

Over the last decades, researchers have shed light on GM in several domains in computer science and mathematics: pattern recognition, computer vision, flow analysis, object tracking, person re-identification, image labeling, and the list can be still extended. Recently, GM has been considered as a fundamental problem in PR. The attributes of both vertices and edges play an important role in GM. Combining both symbolic and numerical attributes on vertices and edges makes the extracted vertices and edges more meaningful and highly representative. Unlike the other graphs used in other fields (e.g., shortest path) where the combination of symbolic and numeric attributes is not necessarily needed.

GM problems are all *NP-hard* except for *graph isomorphism*, for which it has not yet been demonstrated if it belongs to *NP* or not [140, 35]. Roughly speaking, GM can be divided into two broad categories: exact GM and error-tolerant GM. Exact GM addresses the problem of detecting identical (sub)structures of two graphs G_1 and G_2 and their corresponding attributes. This category assumes the existence of only noise-free graphs representing objects while in reality objects are usually affected by noise and distortion. Consequently, researchers in PR often shed light on the other category, i.e., error-tolerant GM. By using error-tolerant techniques, one can match graphs, whether identical or not, and a similarity measure as well as vertex-to-vertex matching can be obtained once the matching process is achieved.

In the context of attributed graphs, the problem of error-tolerant GM presents a higher complexity than exact GM as it takes distortion and noise into account during the matching process. Indeed, the exact algorithms dedicated to solving error-tolerant GM are computationally complex [140, 98]. Consequently, lots of works have been employed to approximately solve error-tolerant GM problem. Generally, approximate methods have been investigated based on genetic algorithms [3], linear sum assignment problem [106], concave and convex continuous relaxation [86]. The aforementioned techniques are expected to present a polynomial run-time. However, they cannot ensure the quality of their solutions and are likely to output suboptimal solutions whose quality has not been deeply studied yet.

Another error-tolerant GM approach is achieved by a set of graph edit operations: insertions, deletions and substitutions of vertices as well as edges. The cheapest sequence of operations needed to transform one of the two graphs into another is computed. This approach is referred to in the literature as graph edit distance (GED). Similar to the other error-tolerant GM problems, the complexity of GED is exponential in the number of vertices of the involved graphs. Such a fact limits GED algorithms to match relatively small graphs. To overcome this problem, two main directions have been adopted in the literature. First, few exact methods based on admissible lower bounds have been proposed for pruning the search space and thus for postponing the graph size restriction (e.g., [60, 153]). A widely used exact method for edit distance computation is based on the A^* algorithm [63]. One drawback of such a method is an important memory load in tree traversal for storing pending solutions to be explored. Second, many approximate methods have been put forward to simplify the GED problem (e.g., bipartite GM where certain edge constraints are relaxed [106]). However, they are not as precise as the exact ones. Indeed, there is a trade-off between precision and run time. In order not to lose the matching quality, in this thesis, we focus on exact GED methods aiming at tackling their high memory and time consumption.

Generally speaking, branch-and-bound like methods (i.e., tree-search based methods) always find a sequence of improved solutions towards the final answer. Once a solution is found, it becomes an upper bound. This kind of methods can be transformed into anytime methods [157] by varying the computational time and studying the effect on the outputted answers. Based on that, we propose a depth-first algorithm for GED and transform it into an anytime algorithm. Such a method clearly shows the trade-off between answer quality (i.e., distance and matching accuracy) and run time.

In the meantime, heavy computation tasks have moved from desktop applications to servers in order to spread the computation load on many machines. This paradigm leads to re-design methods and thus improve their scalability and performance. Theory and practice have shown that using distributed clusters is fruitful for reducing run time. Based on this fact, it is also possible to reduce memory consumption and run time of GED methods, and thus to be able to match larger graphs. To this end, we enhance the run time of our anytime GED method through parallel and distributed systems. However, several questions should be raised before designing a parallel or a distributed architecture for GED. First, how to divide the GED problem into subtasks? Second, how to distribute tasks among a certain number of threads? Third, how to make sure that the tasks load is balanced between all threads during the whole matching process? In the thesis we will tackle this problem and discuss it in detail.

Approximate GM methods have been evaluated in a classification context without a precise evaluation of the quality of their outputted answers. Moreover, the scalability of the proposed methods has not been evaluated. Such facts shed the light on the need for performance evaluation metrics that are able to judge the quality of the resulting outputs of GM methods. Thus, in this thesis, we put forward a GM repository along with novel performance evaluation metrics dedicated to testing the precision of GED methods when having various graphs' types and sizes. Four databases, taken from the literature, with different characteristics are integrated in the repository. Each graphs pair is annotated with the best solution found by a literature method.

To detail all these contributions, this thesis is organized as follows:

In Chapter 2, notations, definitions and concepts used in the thesis are presented. We also dig into the details of existing GM problems. Then, we explore the methods dedicated to these problems. A particular focus on GED problem and its associated techniques is given. Based on this study, we summarize the important contributions of our work in light of the limitations of existing GED methods.

In Chapter 3, we propose and explain the interest of anytime algorithms in GM. We describe how to convert a tree-based error-tolerant GM method into an anytime one that is able to find a list of improved solutions and eventually converges to the optimal one instead of providing only one solution (i.e., the optimal solution). As an application of that, we propose a depth-first GED method that outperforms A^* and then we convert it to an anytime algorithm.

In Chapter 4, we focus on the literature of parallel and distributed branch-and-bound (BnB) algorithms aiming at having a *scalable* depth-first GED method. We also outline our parallel and distributed GED algorithms. A theoretical conclusion of both methods is given.

Chapter 5 is dedicated to the experiments of all the methods that are presented in the thesis. This chapter consists of four parts: First, we put forward a graph database repository annotated with low level information like graph edit distances and their matching correspondences. Second, a set of metrics to assess GED is also proposed. Third, the experiments of the methods proposed in the thesis are conducted on the databases of the repository. A discussion is raised after each experiment. Finally, this chapter ends with classification tests and concluding remarks.

In Chapter 6, a more global discussion is provided. We highlight some research points that should be further studied and conclude this thesis. We also point out the possible lines of future research.

Part I

State of the Art

Chapter 2

Strength and Weakness of Actual Graph Matching Methods

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

Contents

Abstract

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

2.1 Definitions and Notations

2.1.1 Graph

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

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

2.1.2 Subgraph

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

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

Figure 2.1 shows a subgraph G_s in a graph G.

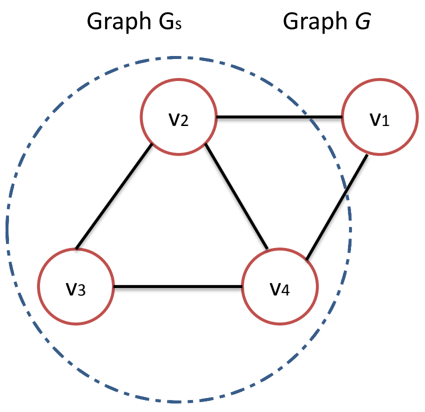


Figure 2.1: A subgraph G_s: $e(v_2, v_3)$, $e(v_3, v_4)$, $e(v_4, v_2)$ of graph G

2.1.3 Directed and Undirected Graphs

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

2.1.4 Attributed Graphs

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

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

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

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

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

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

2.1.5 Graph Size

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

2.1.6 Vertex Degree

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

2.1.7 Graph Density

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

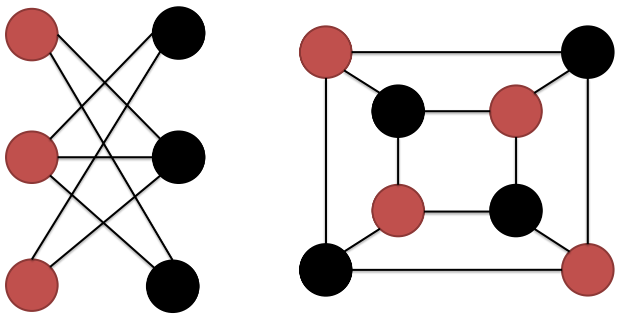


Figure 2.2: Two examples of bipartite graphs

2.1.8 Special Graphs

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

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

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

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

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

2.1.9 Graph Matching

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

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

2.1.10 Exact Graph Matching

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

2.1.10.1 Graph isomorphism

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

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

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

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

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

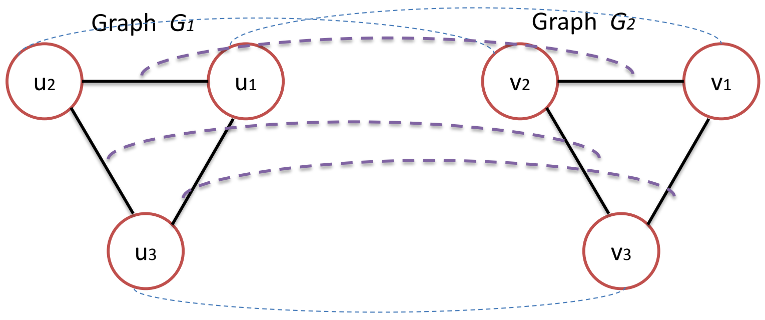


Figure 2.3: Graph isomorphism between G_1 and G_2

2.1.10.2 Induced Subgraph Isomorphism (SGI)

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

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

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

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

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

2.1.10.3 Monomorphism

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

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

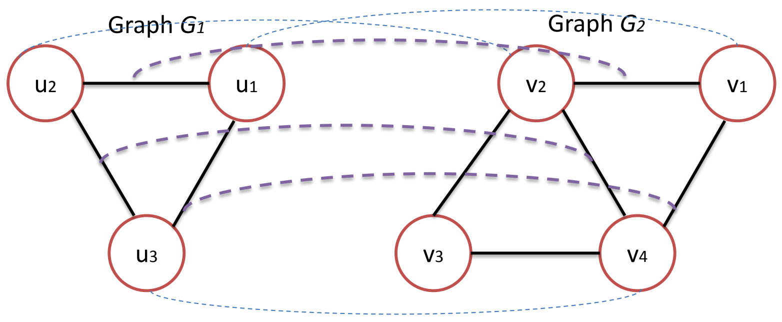


Figure 2.4: Induced subgraph isomorphism

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

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

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

2.1.10.4 Maximum Common Subgraph (MCS)

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

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

2.1.11 Error-Tolerant Graph Matching Problems

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

2.1.11.1 Problem Transformation: from Exact to Error-Tolerant

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

2.1.11.2 Substitution-Tolerant Subgraph Isomorphism

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

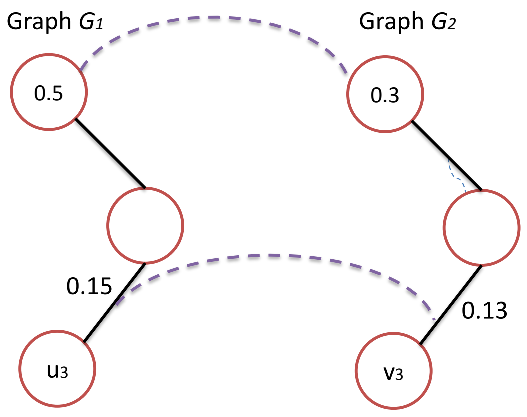


Figure 2.5: Substitution-tolerant subgraph isomorphism problem. G_1 and G_2 are attributed graphs, the mapping takes the difference between the attributes into account. Mappings are also edge-preserving

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

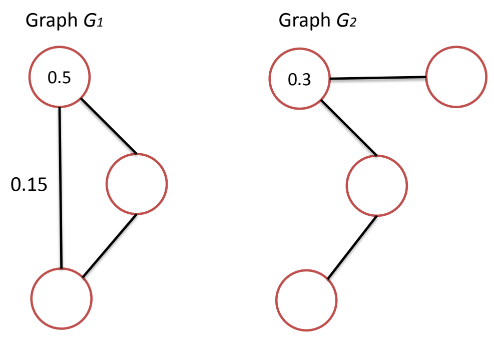
1. $\forall u_i \in V_1,\ \mu_1(u_i) \approx \mu_2(f(u_i))$
2. $\forall (u_i, u_j) \in V_1 \times V_1, e(u_i, u_j) \in E_1 \Leftrightarrow e(f(u_i), f(u_j)) \in E_2$
3. $\forall e(u_i, u_j) \in E_1, \zeta_1(e(u_i, u_j)) \approx \zeta_2(e(f(u_i), f(u_j)))$

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

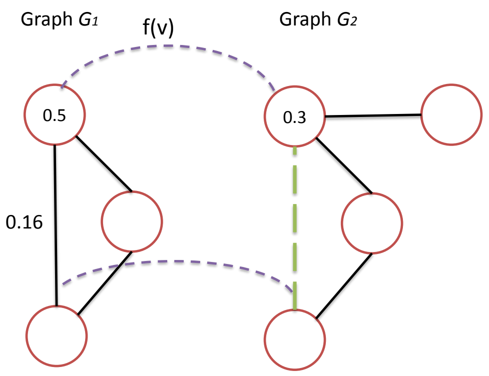
Figure 2.5 depicts the substitution-tolerant subgraph isomorphism problem.

2.1.11.3 Error-Tolerant Subgraph Isomorphism

Error-Tolerant Subgraph Isomporphism [94] takes into account the difference in topology as well as attributes. Thus, it requires that each vertex/edge of graph G_1 is mapped to a distinct vertex/edge of graph G_2 or to a dummy vertex/edge. This dummy elements can absorb structural modifications between the two graphs.



(a) Graphs G_1 and G_2



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

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

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

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

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

2.1.11.4 Error-Tolerant Graph Matching

A significant number of error-tolerant GM algorithms base the definition of the matching cost on an explicit model of the errors (deformations) that may occur (i.e., missing vertices, etc.), assigning a possibly different cost to each kind of error. These algorithms are often denoted by error-correcting or error-tolerant [35, 91].

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

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

Figure 2.7 depicts the error-tolerant graph isomorphism problem.

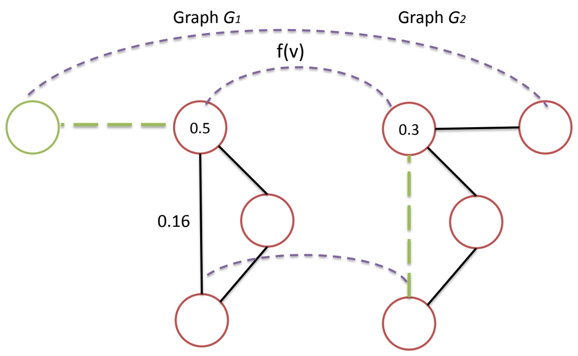


Figure 2.7: Error-Tolerant Graph Isomorphism of the graphs G_1 and G_2 presented in Figure 2.6(a). Note that the dashed vertex and the dashed line on G_1 represent vertex insertion and edge insertion operations respectively. The Dashed line on G_2 depicts an edge insertion operation

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

2.1.11.5 Error-Tolerant Matching Cost

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

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

Definition 11 *Matching Cost Function*

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

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

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

2.1.11.6 Graph Edit Distance

The graph edit distance (GED) was first reported in [136, 117, 60]. GED is a dissimilarity measure for graphs that represents the minimum-cost sequence of basic editing operations to transform a graph into another graph by means classically included operations: insertion, deletion and substitution of vertices and/or edges. Therefore, GED can be formally represented by the minimum cost edit path transforming one graph into another. Edge operations are taken into account in the matching process when substituting, deleting or inserting their adjacent vertices. From now on and for simplicity, we denote the substitution of two vertices u_i and v_k by $(u_i \to v_k)$, the deletion of vertex u_i by $(u_i \to \epsilon)$ and the insertion of vertex v_k by $(\epsilon \to v_k)$. Likewise for edges $e(u_i, u_j)$ and $e(v_k, v_z)$, $(e(u_i, u_j) \to e(v_k, v_z))$ denotes edges substitution, $(e(u_i, u_j) \to \epsilon)$ and $(\epsilon \to e(v_k, v_z))$ denote edges deletion and insertion, respectively. The structures of the considered graphs do not have to be preserved in any case. Structure violations are also subject to a cost which is usually dependent on the magnitude of the structure violation [110]. And so, the meta parameters of each of deletion, insertion and substitution affect the matching process. The discussion around the selection of cost functions and their parameters is beyond the topic of this thesis and will not be discussed in this thesis.

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

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

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

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

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

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

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

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

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

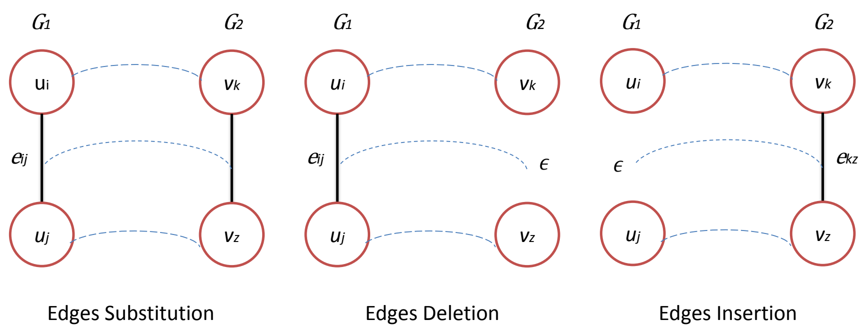


Figure 2.8: Edge mappings based on their adjacent vertices and whether or not an edge between two vertices can be found

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

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

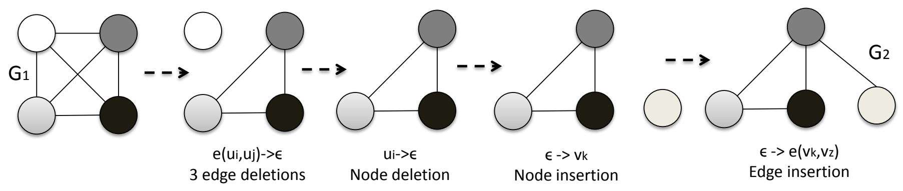


Figure 2.9: Transforming G_1 into G_2 by means of edit operations. Note that vertices attributes are represented in different gray scales

A cost function is associated with each edit operation indicating the change strength an edit operation had done. In fact, GED directly corresponds to the definition of error-tolerant GM, see Definition 10. Thus, GED has also been shown to be NP-hard [153].

Conditions on Cost Functions If no conditions are put on the cost functions for deleting, inserting or substituting vertices and/or edges, then one can have infinite number of complete edit paths λ. For example, one can insert any ed_i (e.g., $\epsilon \to u_i$) and then remove it (i.e., $u_i \to \epsilon$) and thus by doing so with the other edit operations one can end up having infinite number of solutions for $GED(G_1, G_2)$. In order to overcome this problem, some constraints have to be defined for any cost function proposed for solving GED. In [97], three constraints are illustrated. The first constraint, referred to as positivity, is defined as follows:

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

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

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

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

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

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

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

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

It has been shown by Neuhaus and Bunke [97] that for GED to be a metric, each of its elementary operations has to satisfy not only the aforementioned properties but also one more property, referred to as Symmetry. The Symmetry constraint is defined as follows:

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

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

2.1.11.7 Multivalent Matching

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

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

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

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

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

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

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

2.1.11.8 Modeling Graph Matching by Hard and Soft Constraints

In [81], GM is defined by constraint-based modeling language. By using such constraints, any GM problem can be expressed. The constraint-based language, or so-called synthesizer, is designed on top of Comet [139]. Once a user defines the characteristics of the selected problem, a Comet program is automatically created. This program has two modes: Constraint Programming (CP) and Constraint-Based Local Search (CBLS). One of these modes is automatically used depending on the problem's characteristics. For

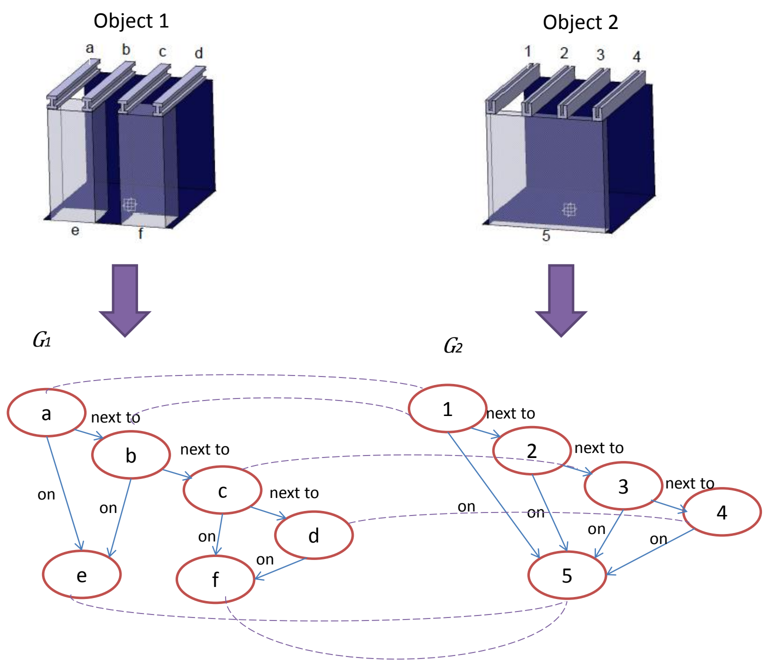


Figure 2.10: Multivalent matching example (taken from [29])

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

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

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

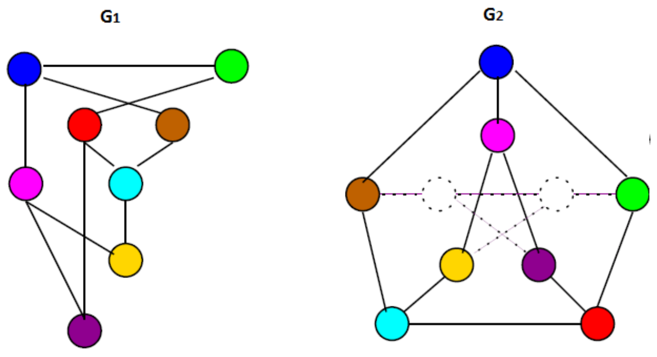


Figure 2.11: An example of two graphs G_1 and G_2. The objective is to decide whether one graph is included into the other. If constraints on edges are hard, the problem is Induced Subgraph Isomorphism. If constraints are both hard and soft ones, the problem is graph Partial Subgraph Isomorphism (Taken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

2.1.11.9 Graph Edit Distance as Quadratic Assignment Problem

GED can be reformulated as a quadratic assignment problems (QAPs) [14]. QAPs belong to the class of NP-hard problems. Over last three decades, extensive research has been done on QAPs. In [115], Sahni and Gonzalez have shown that the QAP is NP-hard and that even finding a suboptimal solution within some constant factor from the optimal solution cannot be done in polynomial time unless P=NP.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.1.12 Matching Difficulties

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

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

2.2 Synthesis of Error-Tolerant Graph Matching Methods

2.2.1 Motivation

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

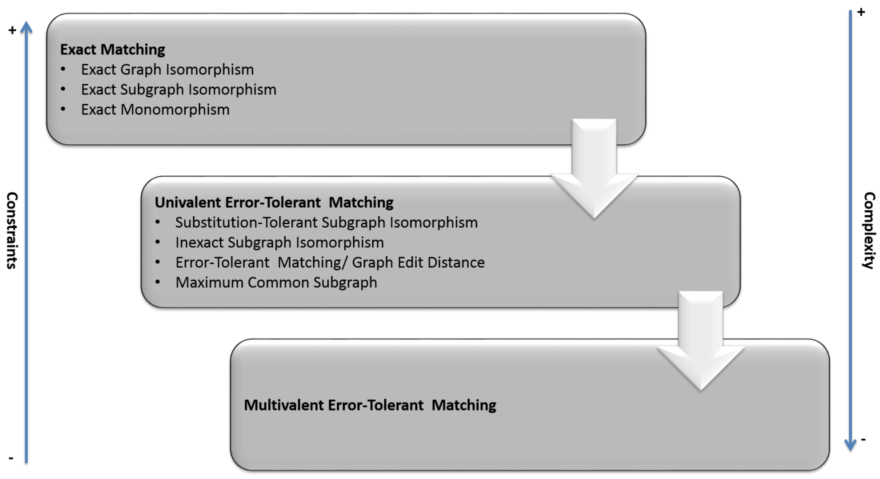

Exact Matching
• Exact Graph Isomorphism
• Exact Subgraph Isomorphism
• Exact Monomorphism
Univalent Error-Tolerant Matching
• Substitution-Tolerant Subgraph Isomorphism
• Inexact Subgraph Isomorphism
• Error-Tolerant Matching/ Graph Edit Distance
• Maximum Common Subgraph
Multivalent Error-Tolerant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

Figure 2.12: Graph matching difficulties according to constraints

Consequently and for all the aforementioned arguments, we focus on univalent error-tolerant GM taking into account the applicability of its proposed methods in various real-world applications. Error-tolerant GM, aims at relaxing, to some extent, the constraints of the extract matching process in such a way that a similarity answer/score is given for matching an attributed source graph with an attributed target graph while penalizing the structure of one or both them. Error-tolerant GM techniques have been widely proposed in the literature. In this section, we survey the methods presented in the literature to solve error-tolerant GM. Since we cannot review all the methods, the list of methods is considered as a non-exhaustive one. We refer the interested reader to two surveys that focused on applications using graphs at different levels [91, 36].

2.2.2 Error-Tolerant Methods in the Literature

In our synthesis, we focus on error-tolerant GM methods that are learning-free (i.e., methods that are not based on a machine learning step). The reason for which we have not focused on such methods is because there are few graph databases with ground truths. Moreover, ground truths constructed by humans cannot be always achieved for some specific structures such as chemical structures. Thus, methods that are based on neural networks (e.g., [54, 130, 77]) and Expectation-Maximization (EM) (e.g., [41, 7, 89]) are not detailed in the synthesis.

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

2.2.2.1 Deterministic Methods

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

Some enhancements have been proposed in the relaxation labeling domain. [61, 74] are the first works applying probability theory to GM where an iterative approach is conducted using a method called probabilistic relaxation. In these works, both a Gaussian error and the use of binary relations are justified to be enough for fully defining the whole structure. The main drawback of the initial formulation of this technique, namely the fact that vertex and edge labels are used only in the initialization of the matching process. Such a drawback was overcome in [141]. A Bayesian perspective is used for both unary and binary vertex attributes in [141]; [59]; and [146]. In [65] this method is further improved by also taking edge labels into account in the evaluation of the consistency measure for matching hierarchical relational models. Bayesian graph [95] has been built up on the idea of probabilistic relaxation. The concept of Bayesian graph has also been successfully applied to trees [135].

Spectral Graph Theory is considered as an approach of great interests for solving GM problems [89, 34, 88, 114]. In this approach, graphs' topology is characterized using eigenvalues and eigenvectors of the adjacency matrix or Laplacian matrix ([147]). The computation of eigenvalues/eigenvectors is a well studied problem that can be solved in polynomial time. However, despite the benefits achieved using all algorithmic tools, spectral methods suffer from two main limitations. First, the weakness in their representation of vertex and/or edge attributes in PR applications, some spectral methods can deal only with unlabeled or labeled graphs under constraints (e.g., only constrained label alphabets or real weights assigned to edges [137].). Second, the sensitivity of eigen-decomposition towards structural errors as they can not cope with graphs affected by noise, such as missing or extra vertices [22].

In the literature, researchers also focused on GED and proposed lots of approaches for solving it. In [23] a distance measure based on the maximal common subgraph of two graphs is proposed. This defined distance is a metric that does not depend neither on the edit distance nor on the costs of the edit operations. The well-known A^* algorithm [101] has been employed for solving GED. A^* along with a lower bound can prune off the underlying search tree by decreasing the number of explored nodes and edges. However, because of the combinatorial explosion of the search tree of GED, the problem is still known to be NP-hard. A linear programming formulation of GED has been reported in [71], such a formulation is applicable only on graphs whose edges are unattributed.

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

Apart from the aforementioned methods in the section, there are also approximate algorithms that work directly on the adjacency matrix of the graphs relaxing the combinatorial optimization to a continuous one like the Path Following algorithm proposed in [152]. Path Following is based on convex-concave relaxations of the initial integer programming problem. The reason of choosing convex/concave relaxations is to approximate in the best way the objective function on the set of permutation matrices. In order to achieve the convex-concave programming formulation, the weighted GM problem is also reformulated as QAP over the set of permutation matrices where the quadratic term encodes the structural compatibility and the linear term encodes local compatibilities, see Section 2.1.11.9. Then this problem is relaxed to two different optimization problems: a quadratic convex and a quadratic concave optimization. Path Following allows to integrate the alignment of graph structural elements with the matching of vertices with similar attributes. Finally, a Graduated NonConvexity and Graduated Concavity Procedure (GNCGCP) [86] has been proposed as a general optimization framework to suboptimally solve the combinatorial optimization problems. One of the special cases in the paper is solving error-tolerant GM. This proposal has two steps. First, graduated nonconvexity which realizes a convex relaxation. Second, graduated concavity which realizes a concave relaxation.

2.2.2.2 Non-Deterministic Methods

The formulation of complex GM problems as combinatorial optimization has been proposed in the literature. Genetic algorithms are considered as examples of such a formulation. Matching is formalized as states (chromosomes) of a search space with a corresponding fitness in [3, 144, 126, 131, 10]. Genetic algorithms start with an initial pool of chromosomes, considered as matching, which evolves iteratively into other generations of matching. Despite the randomness of genetic algorithms in exploring the search space, promising chromosomes can be chosen if one carefully designs genetic algorithms. These chromosomes can then be improved during specific genetic operations. In fact, low cost matching is prioritized which guarantees to have, though not optimal, low cost matching. Furthermore, genetic algorithms are able to efficiently overcome the problem of both huge search spaces and the local minima proposed for approximating GED. However, genetic algorithms are non-deterministic in the sense of having different solutions when running its algorithms several times.

Some generic algorithms have been proposed for solving a variety of GM problems. A greedy algorithm has been proposed in [29]. In the beginning, an empty mapping m is given. Then in order to fill up this set with vertex-to-vertex and edge-to-edge mappings, a greedy way is used to choose mappings that maximize the similarity. To choose the best vertex-to-vertex mappings, vertices that share the same in and out-edges are always privileged. The step of finding the best candidates is repeated until finding a local optimum solution. This algorithm is non-deterministic and thus one need to run it several times and choose the best solution after. In order to improve this algorithm, local search algorithms can be employed [57, 73]. Local search algorithms aim at improving solutions by locally exploring their neighborhoods. In GM, neighborhoods can be obtained by adding or removing vertex-to-vertex mappings in m. To choose the best neighbor to be explored, Tabu search is used [57, 64]. Tabu search prevents backward moves by memorizing the last k moves, such a step overcomes the problem of local optimum from which greedy search algorithms suffer. The authors of [129], inspired by [13], have proposed a reactive tabu search as a generic GM algorithm where k is dynamically adapted. A hashing key is given for each explored path. When the same mapping is explored twice (i.e. when a collision happens in the hash table), the search must be diversified. However, when no collision happens for a certain number of iterations, that is an indicator of the diversity of the mappings and thus the size of k can be decreased. Finally, an Iterated reactive tabu search is proposed in [116] where k executions of reactive search, each of which has $maxMoves/k$ allowed moves, are launched. In the end, the best matching found during the k executions is kept.

2.2.3 Synthesis

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

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

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

2.2.3.1 Large graphs

Based on the graph sizes reported in Table 2.1, we define the term ”large graphs” by dividing it into two categories: exact and approximate large graphs, as shown in Table 2.2. These categories are used as a definition of large graphs in the rest of the thesis. The maximum size of PR graphs is taken from the largest database dedicated to PR graphs, to the best of our knowledge, it is the webpage database [105].

Approximate methods can be grouped into two families:

Name	**Optimality**	**Maximum Graphs Size**	**Graphs type**	**Parallelism Ability**	**Popularity**	**References**
String based methods	Suboptimal	thousands of vertices → Hundreds of Regions	Cyclic/acyclic symbolic graphs	++	+	[66, 8]
Spectral Theory(1)	Optimal	Up to 10	Weighted graphs "same size"	++	++	[137]
Spectral Theory(2)	Suboptimal	Not precised	Directed Acyclic Graphs (DAG)	++	++	[46, 125]
Vertex Assignment	Suboptimal	Up to 128 vertices	Directed attributed graphs	+	++	[104, 69, 106, 58]
Genetic algorithm	Suboptimal	Up to 100 vertices	Various types depending on the application	++	-	[3, 10, 53, 67]
Tabu search based algorithms	Suboptimal	Up to 250 vertices	Various types depending on the application	++	+	[116, 129]
Probabilistic Relaxation	Suboptimal	100 vertices	Various types of graphs	–	?	[141, 47],
GED-A^*	Optimal	Up to 10	Various types depending on the application	+	++	[23]
GED-ILP	Optimal	up to 10	Graphs with unattributed edges	+	++	[71]
Approx GED	Suboptimal	Up to hundreds of vertices	Various types depending on the application	++	++	[90, 106]
IPFP	Suboptimal	Up to 50	Weighted graphs	++	+	[82]
Path Following	Suboptimal	Up to 100	Weighted graphs	++	+	[152]
GNCGCP	Suboptimal	up to 50	Unlabeled graphs	++	+	[86]

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

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

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

1. Reformulation of the problem: The GM problem is truncated into a simpler problem (e.g., reducing the GM problem to an assignment problem [106]). However, solving an approximated or a constraints-relaxed formulation does not lead to an optimal solution of the original problem.

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

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

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

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

These raised questions are still open research questions.

2.3 A Focus on Graph Edit Distance

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

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

Second, it has been shown by Neuhaus and Bunke [97] that GED can be a metric if each of its elementary operations satisfies the three properties of metric spaces (i.e., positivity, symmetry and triangular inequality), see Section 2.1.11.6.

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

Last, but not least, GED has been used in different applications such as Handwriting Recognition (i.e., [48]), Word Spotting (e.g., [108, 143]) and Palm-print classification (e.g., [123]).

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

2.3.1 Graph Edit Distance Computation

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

2.3.1.1 Exact Graph Edit Distance Approaches

A*-based GED A widely used method for edit distance computation is based on the A^* algorithm [101]. This algorithm is considered as a foundation work for solving GED. Algorithm 1 recalls the main steps of the A^* method.

The A^*-based method for exact GED computation proceeds to an implicit enumeration of all possible solutions without explicitly evaluating all of them [111]. This is achieved by means of an ordered tree. Such a search tree is constructed dynamically at run time by iteratively creating successor vertices.

Only leaf vertices correspond to feasible solutions and so complete edit paths. For a node p in the search tree, $g(p)$ represents the cost of the partial edit path accumulated so far, and $h(p)$ denotes the estimated costs from p to a leaf node representing a complete solution. The sum $g(p) + h(p)$ is the total cost assigned to a node in the search tree and is referred to as a lower bound $lb(p)$. Given that the estimation of the future costs $h(p)$ is lower than, or equal to, the real costs, an exact path from the root node to a leaf node is guaranteed to be found [106].

Algorithm 1 Astar ($A*$)

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

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

Algorithm 2 depicts the pseudo code of A^*. This algorithm is taken from [111]. However, in Algorithm 2 we illustrate it differently. The set *OPEN* of partial edit paths contains the search tree nodes to be processed in the next steps. p refers to the current node that will be explored (lines 1 and 2). In the beginning, the first level of the search tree is constructed and inserted in $OPEN$ (lines 3 to 6). To construct the first level of the search tree, u_1 in G_1 is substituted with each v_i in G_2 in addition to the deletion of u_1 (i.e., $u_1 \rightarrow \epsilon$). Algorithm 3 represents how the children of node p are generated. The substitution (lines 3 to 6) or the deletion of a vertex (lines 7 and 8) are considered simultaneously, which produces a number of successor nodes in the search tree. These successors are then saved in a list *Listp*. Back to Algorithm 2, the most promising partial edit path $p \in$ *OPEN*, i.e., the one that minimizes $lb(p)$, is always chosen first (lines 8 and 9). This procedure guarantees that the complete edit path outputted by the algorithm is always optimal, i.e., its cost is the minimum among all possible competing paths. If all vertices of G_1 have been processed (line 11), the remaining vertices of the second graph are inserted in a single step (lines 15 to 19). Finally, when all the branches of the tree have been pruned, if p, whose cost is the minimum, is a complete node, p and its cost $g(p)$ are outputted as an optimal solution of $GED(G_1, G_2)$ (line 13). Note that pendingV_1 and pendingV_2 represent a set of V_1 and V_2 that has not yet been matched.

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

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

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

Algorithm 2 Astar GED algorithm (*A**)

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

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

Algorithm 3 GenerateChildren

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

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

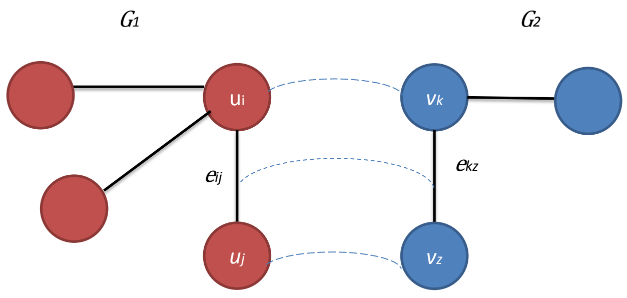


Figure 2.13: Notations recall: G_1 and G_2 are two graphs, e_{ij} is an edge between two vertices v_i and v_j in G_1 while e_{kz} is an edge between two vertices v_k and v_z in G_2

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

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

Algorithm 4 Insertion

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

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

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

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

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

Algorithm 5 deletion

Input: A tree node q and a vertex u_i in G_1

Output: A tree node q

```
1: $q$.add($u_i \to \epsilon$)
2: $ListE_{u_i}$ = edges($u_i$)
3: for $e_{ij} \in ListE_{u_i}$ do
4:     $q$.add($e_{ij} \to \epsilon$)
5:     pending$E_1$(q).remove($e_{ij}$)
6: end for
7: pending$V_1$(q).remove($u_i$)
8: Return($q$)
```

on A^*.

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

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

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

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

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

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

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

Algorithm 6 substitution

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

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

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

2.3.1.2 Approximate Graph Edit Distance Approaches

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

Beam Search A modification of *A**, called Beam-Search (*BS*), has been proposed in [90]. The purpose of *BS*, is to prune the search tree while searching an exact edit path. Instead of exploring all edit paths in the search tree, a parameter x is set to an integer x which is in charge of keeping the x most promising partial edit paths in the *OPEN* set. Such an algorithm cannot always ensure the exact matching to be found. When $x = 1$, *BS* becomes a greedy search algorithm.

Bipartite Matching As previously mentioned in Section 2.1.11.9, GED has been reformulated as an instance of QAP. In [106], Riesen et al have also reformulated the assignment problem as finding an exact matching in a complete bipartite GM. However, they have reduced the QAP of GED computation to an instance of a Linear Sum Assignment Problem (LSAP). LSAPs as well as QAPs formulate an assignment problem of entities. Unlike QAPs, LSAPs are able to optimize the permutation $(\varphi_1, \cdots, \varphi_{n+m})$ with respect to the linear term ($i.e \sum_{i=1}^{n+m} c_{i\varphi(i)}$), see Section 2.1.11.9. That is, the matrix C is the only matrix that is considered and the quadratic term (i.e., $\sum_{i=1}^{n+m} \sum_{j=1}^{n+m} c(a_{\varphi_i \varphi_j} \to b_{\varphi_i \varphi_j})$) is omitted from the objective function. By doing so, the edges between vertices are neglected since that the matrices A and B are the ones that refer to the edges of G_1 and G_2, respectively. In order to reduce GED into a LSAP, local rather than global relationships are considered.

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

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

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

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

Recently, two new versions of BP to compute GED, called Fast Bipartite method (*FBP*) and Square Fast Bipartite method (*SFBP*), have been published in [121] and [122], respectively.

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

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

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

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

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

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

FBP and *SFBP* have a restriction since edit costs have to be defined such that the edit distance is a distance function [122] that is equal to *BP*. Thus three restrictions have to be satisfied:

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

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

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

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

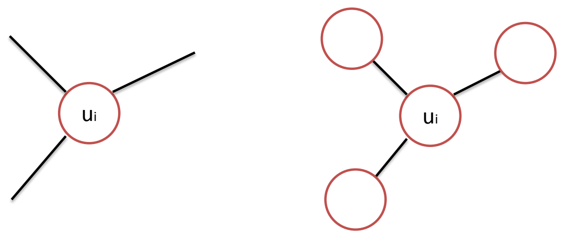


Figure 2.14: Examples of the local substructures: λ_i^{deg} and λ_i^{clique}. Note that $\lambda_i^{deg} = 3$ and $\lambda_i^{clique} = 6$

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

In [109], a greedy swap GED is proposed, see Algorithm 7. As a first step, *BP* is used to compute a first distance d_{best} as well as a mapping m (line 1). Then for each pair of vertex assignments (i.e., $(u_i \rightarrow u_{p_i})$ and $(u_j \rightarrow u_{p_j})$) in m, the partial cost $cost_{orig} = cost(u_i \rightarrow u_{p_i}) + cost(u_j \rightarrow u_{p_j})$ is calculated. This cost is compared with the new mapping cost $cost_{swap}$ which is based on swapping the former vertex assignments (i.e., $(u_i \rightarrow u_{p_j})$ and $(u_j \rightarrow u_{p_i})$) (lines 7 and 8). In order to decide whether the new swapped mappings are beneficial or not, a parameter θ multiplied by $cost_{orig}$ is compared with the absolute value of $cost_{orig}$-$cost_{swap}$ (line 9). If this value is smaller than the defined threshold, the swapping is performed, a new mapping $\bar{m}$ is generated and a new distance $d_{(\bar{m})}$ is derived (lines 10 and 11). If $d_{(\bar{m})}$ is smaller than d_{best}, d_{best} is updated and replaced by $d_{(\bar{m})}$. Moreover, m is replaced by $\bar{m}$. The steps from line 5 to 18 are repeated searching for a better m and so d_{best}. The parameter *swapped* is used as an indicator that tells if there were some changes that have been made when executing the two for loops. If *swapped* equals *true*, the steps are re-executed on the new m. Once m becomes stable (i.e., when *swapped* equals *false*), d_{best} and m are outputted as a best answer that can be found by Greedy-Swap. Note that d_{best} is not necessarily the optimal solution since *BP* is based on local search and that's why the Greedy-Swap algorithm belongs to approximate GED methods.

Based on the same idea of [109], a Beam-Search version of *BP*, called *BP-Beam*, is proposed in [112]. This work focuses on investigating the influence of the order in which the assignments are explored, such a process is considered as a post search process on the distance quality. As in [109], the original node assignment $d_{(m)}(G_1, G_2)$ is systematically varied by swapping $(u_i \rightarrow v_{p_i})$ and $(u_j \rightarrow v_{p_j})$. For each swap it is verified whether (and to what extent) the derived distance approximation stagnates, increases or decreases. For a systematic variation of mapping m, a tree search is used. As usual, in tree-search based methods, a set *OPEN* is employed that holds all of the unprocessed tree nodes. Each tree node is a triple ($\bar{m}$,q,$d_{(\bar{m})}(G_1, G_2)$) where $\bar{m}$ is the matching, q is its depth in the search

Algorithm 7 Greedy-Swap (G_1,G_2)

Input: Non-empty attributed graphs $G_1 = (V_1, E_1, \mu_1, \zeta_1)$ and $g_2 = (V_2, E_2, \mu_2, \zeta_2)$ where $V_1 = \{u_1, ..., u_{|V_1|}\}$ and $V_2 = \{v_1, ..., v_{|V_2|}\}$ and a parameter θ

Output: A minimum cost edit path (d_{best}) from G_1 to G_2

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

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

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

In [113], a search procedure based on a genetic algorithm referred to as *BPGA* is proposed for improving the accuracy of *BP*. After calculating a first upper bound $(m, d_m(G_1, G_2))$ using *BP*, an initial population $P(0)$ is build by computing N random order variations of m (i.e., $m_1^{(0)}, \cdots, m_N^{(0)}$). Each variation $m_i^{(0)} \in P(0)$ is computed by assigning a mutation probability to each vertex-to-vertex mapping (i.e., $u_i \to v_{p_i}$). This probability tells whether a mapping can be deleted or not. In the case of deleting a vertex-to-vertex mapping, an infinity cost is assigned to it (i.e., c$(u_i \to v_{p_i})$=∞). Given this modification in a matrix entity, a new mapping $m_i^{(0)}$ and its underlying distance are generated. Note that $m_i^{(0)}$ does not contain $u_i \to v_{p_i}$ anymore. As mentioned before, N mappings are generated through mutation procedure and thus the aforementioned procedure is repeated N times to obtain an initial population $P(t) = \{m_1^{(t)}, \cdots, m_N^{(t)}\}$. Afterwards, a subset E, called parents, of $P(0)$ is created aiming at obtaining a second population $P(t+1)$. Parents' selection is achieved by selecting the best K approximations whose distances are the minimum. These parents are inserted into $P(t+1)$ without any modification. Then, to generate the other $N - |E|$ mappings, the following strategy is repeated $N - |E|$ times. Two mappings $\bar{m}$, $\bar{\bar{m}} \in E$ are randomly selected and combined in one mapping m where $C_m = \max(\{c_{\bar{i},j}, c_{\bar{\bar{i}},j}\})$ where $\bar{c}$ and $\bar{\bar{c}}$ are the cost matrices of $\bar{m}$ and $\bar{\bar{m}}$, respectively. Any prevented mapping (i.e., a mapping whose cost is infinity) in $\bar{m}$ and $\bar{\bar{m}}$ is also prevented in the merged mapping m. The selection of parents $E \subseteq P(t)$ and the generation of $N - |E|$ new mappings are repeated again and again. The algorithm is stopped when the best distance has not been modified for ι iterations where $\iota \leq t$. Since any genetic algorithm is non-deterministic, the computation of *BPGA* is repeated s times. Afterwards, the best distance along with its matching are outputted. Thus, when comparing *BPGA* to *BP*, *BPGA* increases the run time by parameters $s.t.N$.

Compared to the original BP, the improvements proposed in [45, 112, 109, 113] increase run times. However, they improve the accuracy of the BP solution.

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

Definition 15 *The Modified Hausdorff (H)*

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

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

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

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

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

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

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

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

2.4 Performance Evaluation of Graph Matching

In this section, a focus on the existing benchmarks for GM methods is given. Table 2.3 synthesizes the graph databases presented in the literature. One may notice that exact GM methods have been evaluated at matching level. However, error-tolerant GM methods have been evaluated at the classification level rather than the matching one. Yet, in the literature, some approximate GED methods have been evaluated at the matching level but unfortunately on graphs whose sizes are not bigger than 16 vertices [107]. One may ask the following question: why error-tolerant GM algorithms have not been tested at the matching level? Is it because there is really no need for that and that the only need for error-tolerant GM methods is to classify graphs? For instance, one of the databases repository called CMU [2] is devoted to error-tolerant GM with ground truth information. However, graphs have the same number of vertices and thus the scalability measure cannot be assessed. Indeed, one may clearly see that there is a lack of performance comparison measures dedicated to the scalability of error-tolerant GM methods, whether exact or approximate ones.

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

Table 2.3: Synthesis of graph databases

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

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

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

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

Method	Reference	Problem type	Graphs type	Distributed?
*A**	[111]	Exact GED	symbolic and numeric attributes	NO
BLP	[71]	Exact GED	symbolic and numeric attributes only on vertices	NO
BS	[98]	Approximate GED	symbolic and numeric attributes	NO
BP	[106]	Approximate GED	symbolic and numeric attributes	NO
SWAP-BP	[112]	Approximate GED	symbolic and numeric attributes	NO
BP-Beam	[109]	Approximate GED	symbolic and numeric attributes	NO
IBP-Beam	[45]	Approximate GED	symbolic and numeric attributes	NO
BPGA	[113]	Approximate GED	symbolic and numeric attributes	NO
FBP	[121]	Approximate GED	symbolic and numeric attributes	NO
SFBP	[122]	Approximate GED	symbolic and numeric attributes	NO
H	[48]	Approximate GED	symbolic and numeric attributes	NO

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

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

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

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

Part II

Optimization Techniques for Graph Matching

Chapter 3

Toward an Anytime Graph Matching Approach

After growing wildly for years, the field of computing appears to be reaching its infancy.
John Pierce.

Contents

Abstract

As seen in the conclusion of the aforementioned chapter, there is a need for an optimized and exact GED method. Thus, in this chapter, we propose and explain the interest of a new GM methods family, referred to as anytime GM methods. This family allows exact error-tolerant GM methods to be applicable in real-world applications. In this chapter, we describe how to convert an error-tolerant GM method into an anytime one [157] that is able to find a list of improved solutions and eventually converges to the optimal solution instead of providing only one solution (i.e., an approximate or an optimal solution). The approach we adopt uses a variant of an exact Depth-First search GED method to find a first suboptimal solution quickly, and then continues the search to find improved solutions. In addition of getting rid of memory bottleneck, this approach improves the upper bound while exploring the search tree. When the time available to solve an optimization problem is limited or uncertain, this creates an anytime heuristic search algorithm that allows a flexible trade-off between search time and solution quality. With more time, the method can improve its solution and finally reach the optimal solution.

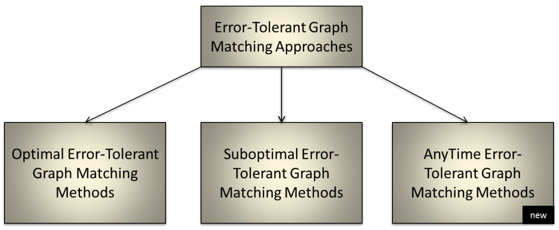

Error-Tolerant Graph Matching Approaches
Optimal Error-Tolerant Graph Matching Methods
Suboptimal Error-Tolerant Graph Matching Methods
AnyTime Error-Tolerant Graph Matching Methods
new

Figure 3.1: Adding a new family of GM approaches

3.1 Motivation

More and more sophisticated applications need to achieve huge computation tasks within strict elapsed time limits in order to be useful. For instance, interactive systems typically need to complete their task within a few seconds so as it would be acceptable by users. On the other hand, powerful data structures such as attributed graphs that are used to represent complex entities always require more and more computational resources. Then, a trade-off between accuracy and computational cost (i.e., execution time and consumed memory) has to be found and new methods such as anytime version of algorithms have to be defined. The main idea of an anytime algorithm comes from the simple observation that there is no reason to stop an algorithm after a first solution is found, specially when it is possible to find better solutions with the delight of more time. By continuing the search, a sequence of improved solutions can then be found and eventually with additional time the algorithm can even converge to an optimal solution.

In the literature, many different error-tolerant GM algorithms are available [35, 140]. However, the exact computation of GM is then restricted to small graphs or applications where time limit is exponential in the number of vertices of the involved graphs restricting their applicability to graphs of rather small size or to specific applications where time limit is not a problem.

In this part of thesis, we would like to take advantage of both exact and approximate error-tolerant GM methods by merging them together to propose a third type of GM methods that we call "*Anytime GM*", see Figure 3.1.

Roughly speaking, GED methods can also be categorized into two groups. First, methods that are fast enough but that can only find one and only one feasible solution (such as *BP* [106], *SFBP* [121]). Second, methods that are tree-search based methods (such as *BS* [90]) that can provide more than only one solution while traversing the search tree during the matching process. Thus, this last kind of GM methods (i.e., tree-based method) becomes of great interest since computational time and even explored search space can be manageable with the impact of the quality of the provided matching solution. From here comes the primary motivation of the chapter saying that tree-based methods for GM computation can be turned into anytime methods by varying the computational time and studying the effect on the outputted answers. In this chapter, we propose the definition of an anytime algorithm of GM based on a variant of Depth-First search GED computation that does not consume so much memory. By managing time and memory at the same time, the proposed method is then as much scalable as possible.

3.2 Formulating Search-Based Graph Edit Distance Algorithms as Anytime Ones

In this section, the main characteristics of anytime algorithms [157] are recalled explaining how they can respond to the constraints (i.e., time and memory).

For the reasons mentioned in Section 2.2, we shed a spot of light on the problem of GED. For more details about GED, see Section 2.2.

As previously seen, the most famous classical technique that was used to solve the GED problem is called A^* [101]. A^* is a search-tree based algorithm that updates the solution once it finds a better one. Various heuristics based on A^* have been studied in [84] to allow a trade-off between solution quality and search time. The possibility of continuing a non-admissible[1] A^* search after the first solution is found was suggested by Hansen with an approach called bandwidth heuristic search [62]. After some years of silence, this idea has been extended as a strategy for creating an anytime A^* algorithm. A description of this extension can be found in [156]. Note that the term non-admissible solution refers to a solution that overestimates the cost of reaching the goal.

In [158], Zilberstein and Russel have defined the desirable properties of anytime algorithms. These properties can be summarized as follows:

- Interruptability: after some small amount of setup time, a suboptimal solution can be provided by stopping the algorithm at time t.
- Monotonicity: the quality of the result increases in function of computational time.
- Measurable quality: we can always measure the quality of a suboptimal result.
- Diminishing returns: at the beginning of anytime algorithms, the improvement of solutions can be remarkably observed. However, this improvement decreases over time.
- Preemptability: an anytime algorithm can be suspended and resumed with minimal overhead.

Anytime algorithms have trade-off between quality and execution time, see Figure 3.2. Anytime algorithms can find a best-so-far solution which is generated after some initialization time at the beginning of the execution. From Figure 3.2, one can see that the quality of the solution is improved when increasing the execution time. Users have the choice to stop the algorithm at anytime and thus get an answer that is satisfying, or they can run the algorithm until its completion when it is important to find the optimal solution.

The setup time (in Figure 3.2) corresponding to the time needed by an anytime algorithm to get a first solution is a crucial point. This setup time has to be compared with the

[1]A search algorithm is described as admissible if it is able to find an optimal path or solution [99].

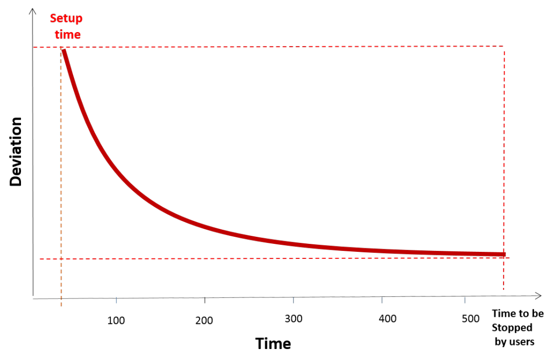


Figure 3.2: Characteristics of anytime algorithms

delay requested by approximate methods that solve the same problem. Another important point when dealing with anytime algorithms is to know when an anytime algorithm should be interrupted (by the system or the user) to get the best-so-far answer. Thus, algorithms should be equipped with appropriate stopping criteria based on a monitoring of actual performances when the time of an optimal interruption is not known in advance. A study of this specific point will be proposed in the experiments.

3.2.1 Tree-Search Methods with Time and Memory Consideration

3.2.1.1 Considering Time with Anytime Algorithms

Anytime algorithms are dedicated to problem-solving under time constraints. After a setup time, anytime algorithms are able to provide a solution whenever they are stopped. Then, the quality of the solution has to improve with additional computation time [158]. For difficult search problems (e.g., GED computation), A^* and Depth-First Search *DFS* may take too long time to find an optimal solution. However, a suboptimal solution that is found relatively quickly can be useful. The most common approach to transform a search algorithm, such as A^*, into an anytime algorithm consists in the following three changes [62].

- A non-admissible evaluation function, $lb_0(n) = g(n) + h_0(n)$, where the heuristic $h_0(n)$ is not admissible, is used to select nodes for expansion in an order that allows good, but possibly suboptimal, solutions to be found quickly.
- The search continues after a solution is found, in order to find improved solutions.
- An admissible evaluation function (i.e., a lower-bound function), $lb(n) = g(n) + h(n)$, where $h(n)$ is admissible, is used together with an upper bound on the optimal solution cost given by the cost of the best solution found so far, in order to prune the search space and detect convergence to an optimal solution.

Based on this idea, many researchers have explored the effect of weighting the terms $g(n)$ and $h(n)$ in the node evaluation function differently, in order to allow A^* to find a bounded-optimal solution with less computational effort. In the approach called Weighted A^* (WA^*) [84], the node evaluation function is defined as $lb_0(n) = g(n) + \omega^* h(n)$, where the weight $\omega > 1.0$ is a parameter set by the user. If $\omega > 1.0$, the search is not admissible and the first solution found may not be optimal, although it is usually found much faster. A weighted heuristic accelerates the search for a solution because it makes nodes closer to a goal seem more attractive, giving the search a more depth-first aspect and implicitly adjusting a trade-off between search effort and solution quality. Weighted heuristic search is more effective for search problems with close-to-optimal solutions, and can often find a close-to-optimal solution in a small fraction of the time it takes to find an optimal solution. Some variations of weighted heuristic search have been studied. For example, an approach called dynamic weighting adjusts the weight with the depth of the search [75]. Weighted heuristic search has been used with other search algorithms besides A^*, including memory-efficient versions of A^* such as IDA^* and RBFS [76], see next section, as well as Learning Real-Time A^* ($LRTA^*$) [124].

3.2.1.2 Considering Memory with Anytime Algorithms

As previously mentioned, the scalability of A^* is limited by the memory required to store the lists of open path inside the search tree. This also limits the scalability of Anytime A^* and so we have to take care of this point in the conception of our new GED algorithms and try to create a linear-space anytime algorithm.

Considering memory aspect, *DFS* algorithms are very effective for some tree-search problems since they overcome the memory bottleneck from which A^* methods suffer. DFS algorithms are anytime by nature [154], as they systematically explore leaf nodes of a state space. They quickly find a solution that is suboptimal, and then continue to search for an improved solution until an optimal solution is found. They can even use the cost of the best solution found so far as an upper bound to prune the search space. So, the *DFS* strategy seems to correspond to a simple and efficient approach for converting an exact algorithm of GED into an anytime algorithm that offers a trade-off between search time, memory consumption and quality of the provided solution when more time is available.

Several variants of A^* have been developed that use less memory, including algorithms that require only linear space in the depth of the search space. One of the most known algorithms is Recursive Best-First Search (*RBFS*) [76]. *RBFS* is a general heuristic search algorithm that expands frontier nodes in best-first order, but saves memory by determining the next node to expand using stack-based backtracking instead of by selecting nodes from an Open list.

At a first glance, *RBFS* looks similar to a recursive implementation of DFS. However, it differs from *DFS* since the nodes are branched in a best-first way by using a special backtracking condition. The complete path to the current node being branched is maintained on the recursion stack as well as all siblings of each node on this path. A tracking of the f=g+h cost of the best alternative available path is kept from any ancestor of the current node, which is passed as an argument to the recursive function. The arguments of *RBFS* are a node n to be explored and a bound B that is the best so-far f value of an unexplored node n_2 that has the same parent as the node n. As in Weighted A^* algorithms, a pruning stage happens when the f-cost of the children of n is greater than or equal to B. When pruning a branch, backtracking to the parent n is performed and so a best-first exploration is conducted to expand the new promising node. *RBFS* determines whether a child node is being visited for the first time or not. By doing such an action, the backtracking of already visited branches is avoided.

Based on the aforementioned remarks, we have defined an efficient optimized Anytime GM algorithm that avoids high memory consumption. The proposed algorithm is described in this chapter starting with the description of an optimized version of a depth-first search algorithm. Thereafter, this algorithm is reformulated as an anytime one that is able to provide successive improved solutions according to available time.

3.3 Optimized Depth-first Graph Edit Distance

To overcome the bottleneck of A^*, we propose a novel algorithm that reduces the used memory space using a different exploration strategy (i.e., depth-first instead of best-first). This approach also reduces the computation time as the unfruitful nodes are pruned by a lower and upper bounds strategy. A preprocessing strategy is included to avoid the re-computation of vertices and nodes matching costs. Edges and vertices costs matrices are constructed during an initialization step and the list V_1 is sorted to speed up the search for the best edit path to be explored.

The elements of the depth-first GED (DF) algorithm are described in Sections 3.3.1 to 3.3.5. Moreover, a pseudo-code is presented in Section 3.3.6.

3.3.1 Structure of Search-Tree Nodes

DF is a tree-search based algorithm in which each node corresponds to a matching state in the GM problem. Figure 3.3 illustrates an example of a tree node or so-called partial edit path.

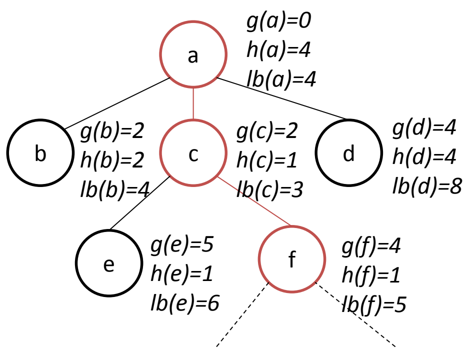


Figure 3.3: An example of a partial edit path p whose explored nodes so far are a, c and f. $lb(*) = g(*) + h(*)$

Each tree node p, in the search tree, is then identified by the following elements:

- $matchedVertices(p)$ and $matchedEdges(p)$: the vertices and edges that have been matched so far in both G_1 and G_2. These sets can contain substitution ($u \rightarrow v$), deletion ($u \rightarrow \epsilon$) and/or insertion ($\epsilon \rightarrow v$) of vertices and edges, correspondingly.

- *pendingV*$_i(p)$ and *pendingE*$_i(p)$ with $i \in \{1, 2\}$: these sets represent vertices and edges of both G_1 and G_2 (i.e., V_1, V_2, E_1 and E_2) that are not substituted, deleted or inserted yet where *pendingV*$_1(p)$ and *pendingV*$_2(p)$ represent pending V_1 and pending V_2, respectively whereas *pendingE*$_1(p)$ and *pendingE*$_2(p)$ represent pending E_1 and pending E_2, respectively.
- *children(p)*: for any node p, the exploration is achieved by choosing the next most promising vertex u_i of *pending-vertices*$_1(p)$ and matching it with all the elements of *pending-vertices*$_2(p)$ in addition to the deletion of this node (i.e., $u_i \rightarrow \epsilon$). All these mappings are referred to as *children(p)*.
- $h(p)$: the estimated future cost from node p does not overestimate the complete solution. The calculation of the lower bound is described in Section 2.3.1.1.
- $g(p)$: the cost of *matched-vertices*(p) and *matched-edges*(p). Both h and g depend on the attributes as well as the structure of the involved sub-trees. The cost functions involved with each dataset permit to calculate insertions, deletions and substitutions of vertices and/or edges.

3.3.2 Preprocessing

Before starting the Branch-and-Bound (BnB) part, important data structures have to be initialized to speed up the tree search exploration. Preprocessing includes two steps: cost matrices construction and vertices-sorting strategy.

3.3.2.1 Cost Matrices

The vertices and edges cost matrices (C_v and C_e) are constructed, respectively. This step aims at speeding up branch-and-bound by getting rid of re-calculating the assigned costs when matching vertices and edges of G_1 and G_2.

Let G_1 =(V_1,E_1,L_{V_1},L_{E_1},μ_1,ζ_1) and G_2 =(V_2,E_2,L_{V_2},L_{E_2},μ_2,ζ_2 be two graphs with $V_1 = (u_1, ..., u_n)$ and $V_2 = (v_1, ..., v_m)$. A vertices cost matrix C_v, whose dimension is $(n + 2)$ X $(m + 2)$, is constructed as follows:

$$C_v = \left|\begin{array}{cccc||cc} c_{1,1} & \cdots & \cdots & c_{1,m} & c_{1\leftarrow\epsilon} & c_{1\rightarrow\epsilon} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ c_{n,1} & \cdots & \cdots & c_{n,m} & c_{n\leftarrow\epsilon} & c_{n\rightarrow\epsilon} \\ \hline c_{\epsilon\rightarrow 1} & \cdots & \cdots & c_{\epsilon\rightarrow m} & \infty & \infty \\ c_{\epsilon\leftarrow 1} & \cdots & \cdots & c_{\epsilon\leftarrow m} & \infty & \infty \end{array}\right|$$

where n is the number of vertices of G_1 and m is the number of vertices of G_2 .

Each element $c_{i,j}$ in the matrix C_v corresponds to the cost of assigning the i^{th} vertex of the graph G_1 to the j^{th} vertex of the graph G_2. The left upper corner of the matrix contains all possible node substitutions while the right upper corner represents the cost of all possible vertices insertions and deletions of vertices of G_1, respectively. The left bottom corner contains all possible vertices insertions and deletions of vertices of G_2, respectively whereas the bottom right corner elements cost is set to infinity which concerns the substitution of $\varepsilon - \varepsilon$.

Similarly, C_e contains all the possible substitutions, deletions and insertions of edges of G_1 and G_2. C_e is constructed in the very same way as C_v.

3.3.2.2 Vertices-Sorting Strategy

As GED aims at transforming G_1 into G_2, it is important to sort V_1 in order to start with the most promising vertices that will speed up the exploration of the search tree while searching for the exact GED. C_v is used as an input of the vertices-sorting phase. First, *BP* is applied to establish the initial edit path *EP* which can be used as a first upper bound [106]. Second, the edit operations of *EP* are sorted in ascending order of the matching cost where *sortedEP* $= \{u \rightarrow v\}\ \forall u \in V_1 \cup \{\epsilon\}$. At last, from *sortedEP*, each $u \in V_1$ is inserted in $sortedV_1$.

After these preprocessing steps the tree exploration can start.

3.3.3 Branching and Selection Strategies

The solution space is organized as an ordered tree which is explored in a depth-first way. In depth-first search, each node is visited just before its children. In other words, when traversing the search tree, one should travel as deep as possible from node i to node j before backtracking. The exploration starts with the root node and so the first most promising vertex u_1 in the sortedV_1 set and will generate some edit paths by matching u_1 with all vertices of G_2 in addition to deleting u_1 (i.e., $u_1 \rightarrow \epsilon$); note that each edit path has its own structure. This step constructs the first level in the search tree (i.e., the root's children). For any node p, in the search tree, the children are sorted in an ascendant way according to *lb(q)* of each child node q. These children are then added to *OPEN*. Since the children are ascendantly sorted, the exploration is achieved by choosing the first element in *OPEN*, which represents the next most promising vertex u_i of $pendingV_1(p)$ and matching it with all $pendingV_2(p)$ in addition to the deletion option (i.e., $p \rightarrow \epsilon$) and so on. And so, each node is visited just before its children.

3.3.4 Reduction Strategy

Pruning, or bounding, is achieved thanks to *h(p)*, *g(p)* and a global upper bound *UB* obtained at node leaves. Formally, for a node p in the search tree, *lb* is taken into account and compared to *UB*. That is, if *g(p)+h(p)* is less than *UB* then p can be explored. Otherwise, the encountered p will be removed from *OPEN* and the next promising node is evaluated and so on until finding the best *UB* that represents the optimal solution of *DF*. This algorithms differs from A^* as at any time t, in the worst case, *OPEN* contains at most $|V_1|.|V_2|$ tree nodes and hence the memory consumption is not exhausted.

3.3.5 Upper and Lower Bounds

As mentioned before, an initial upper bound edit path *UB* and its cost *UBCOST* can be computed by *BP* or remained unset. This choice can be seen as a parameter. Afterwards and while traversing the search tree, *UB* is replaced by the best *UB* found so far (i.e., a complete path whose cost is less than the current *UB*). After finishing the traversal of the search tree (i.e., when *OPEN* equals Φ), the best *UB* is outputted as an optimal solution of *DF*. Encountering upper bounds when performing a depth-first traversal efficiently prunes the search space and thus helps in finding the optimal solution faster than A^*.

As for the lower bound $lb(p) = g(p) + h(p)$, several ones are proposed in Section 5.3.3.1 and the best one is used for the rest of the experiments.

3.3.6 Pseudo Code

As depicted in Algorithm 8, *DF* starts by a preprocessing step (lines 3 to 6), then an upper bound *UB* along with its distance *UBCOST* are calculated by *BP* or set to ∞ (line 5). The traversal of the search tree starts by generating the root's children (line 7). Algorithm 9 illustrates how *children(p)* are generated. The most promising vertex u_1 is popped up from $sortedV_1$ (line 1). Consequently, vertex u_1 is substituted with all the vertices in $pendingV_2(p)$ (line 3). Each of the mappings is added to a list *Listp*. In addition to substitutions, the deletion of u_1 (i.e., $u_1 \rightarrow \epsilon$) is added to *Listq* (lines 6 and 7). Thus, *Listp* contains the children of *p*. Back to Algorithm 8, *Listp* is sorted in an ascending order and inserted in *OPEN* (lines 8 to 11). Since each child inserted in *OPEN* are ordered, the most promising child *p* will be first selected (line 13). Then the children of *p* are generated and sorted as mentioned above (lines 14 and 25). If *ListP* is empty, then each $v_i \in V_1$ is either substituted or deleted. Consequently, if Pending$V_2(p)$ is not empty, each $v_i \in$ Pending$V_2(p)$ will be inserted in *p* (lines 16 to 19). *UB* and *UBCOST* are updated whenever a better solution is encountered (lines 20 to 23). This algorithm guarantees to find the optimal solution of $GED(G_1, G_2)$. The edge operations are taken into account in the matching process when substituting, deleting or inserting their corresponding vertices, see Figure 2.8.

3.4 Anytime Depth-First Graph Edit Distance

This section describes how we convert the previous algorithm into an anytime one that can produce an instant matching and dissimilarity measure between two graphs, or if given the luxury of additional time, can increase precision of this matching and dissimilarity measure. After the anytime algorithm finds a first solution, and each time it finds an improved solution, it saves (or outputs) the solution and continues the search. We convert *DF* into an anytime one while respecting the anytime properties. As in *DF*, the anytime version contains the preprocessing step (see Section 3.3.2), the reduction strategy (Section 3.3.4), the selection and branching strategy 3.3.3 as well as the lower and upper bounds (see Section 3.3.5). However, the anytime version of GED differs in the sense of exporting the complete edit paths along with their distances while exploring the search

Algorithm 8 Depth First GED algorithm (DF)

Input: Non-empty attributed graphs $G_1 = (V_1, E_1, \mu_1, \zeta_1))$ and $G_2 = (V_2, E_2, \mu_2, \zeta_2)$ where $V_1 = \{u_1, ..., u_{|v_1|}\}$, $V_2 = \{u_2, ..., u_{|v_2|}\}$, μ and ζ are the attributes associated with vertices and edges respectively.

Output $UBCOST$ = a minimum cost edit path from g1 to g2 & UB = sequence of edit operations.

```
1:  $p \leftarrow$ root node of the tree with all vertices and edges of $G_1$ and $G_2$ as pending lists
2:  $OPEN \leftarrow \Phi$, $UB \leftarrow \phi$, $UBCOST \leftarrow \infty$
3:  Generate $C_v$, $C_e$
4:  **Optional:**                                        ▷ Steps below are optional
5:        ($UB$ , $UBCOST$) $\leftarrow BP(G_1, G_2)$
6:        $sortedV_1 \leftarrow$ sortVertices$V_1$($V_1$,$V_2$)        ▷ in ascending order of $BP(G_1, G_2)$
7:  $Listp \leftarrow$ GenerateChildren($p$,$C_v$,$C_e$,sorted$V_1$($p$),pending$V_2$($p$))
8:  $Listp \leftarrow$ SortAscending($Listp$)                  ▷ according to $g(p)+h(p)$
9:  **for** $p \in Listp$ **do**
10:     $OPEN$.AddFirst($p$)
11: **end for**
12: **while** $OPEN$ != $\Phi$ **do**
13:     $p \leftarrow OPEN$.popFirst()          ▷ Take first element and remove it from $OPEN$
14:     $Listp \leftarrow$ GenerateChildren($p$)
15:     **if** $Listp = \Phi$ **then**
16:         **for** $v_i \in$ pending$V_2$($p$) **do**
17:             $q \leftarrow$ insertion($\epsilon$ , $v_i$)                    ▷ i.e., $\{\epsilon \rightarrow v_i\}$
18:             $p$.AddFirst($q$)
19:         **end for**
20:         **if** $g(p) <$ UBCOST **then**
21:             $UBCOST \leftarrow g(p)$, UB $\leftarrow p$
22:         **end if**
23:     **else**
24:         $Listp \leftarrow$ SortAscending($Listp$)              ▷ according to $g(p)+h(p)$
25:         **for** $q \in Listp$ **do**
26:             **if** $g(q) + h(q) < UB$ **then**
27:                 $OPEN$.AddFirst($q$)
28:             **end if**
29:         **end for**
30:     **end if**
31: **end while**
32: Return($UBCOST$ , $UB$)                                  ▷ export final results
```

Algorithm 9 GenerateChildren

Input: a tree node p, the vertices and the edges matrices (C_v and C_e), sorted$V_1(p)$ and pending$V_2(p)$.

Output: a list *Listp* whose elements are the children of p.

```
1: $u_1 \leftarrow sortedV_1$.popFirst($p$)
2: for $v_i \in$ pending$V_2(p)$ do
3:     $q \leftarrow$ substitution($u_1$ , $v_i$)        ▷ i.e., $\{u_1 \rightarrow v_i\}$
4:     $Listp$.AddFirst($q$)        ▷ $q$ is a tree node
5: end for
6: $q \leftarrow$ deletion($u_1$)        ▷ i.e., $\{u_1 \rightarrow \epsilon\}$
7: $Listp$.AddFirst($q$)
8: Return($Listp$)
```

tree. Moreover, it also has a capability to be interrupted at approximately anytime t. Such a fact keeps the output accessible at anytime.

3.4.1 Pseudo code

As one may see through Algorithm 10, the steps of AnyTime DF (ADF), referred to as *ADF*, resemble DF. However, two additional lines are added to the pseudo code (see lines 6 and 23). Moreover, a time-out and memory-out interruptions are added. By adding lines 3 and 23, the algorithm is able to keep tracking the solutions found while exploring the search tree. The first complete solution found by BP is first exported (line 4). Afterwards, when exploring the search tree and finding a solution that is better than the upper bound found so far, an exportation of the new upper bound is done (line 23). Note that the output is available at anytime after the setup time (line 5). As long as the time and memory are not violated and there are nodes to explore in *OPEN*, the exploration step continues (line 13). As in DF, the edge operations are taken into account in the matching process when substituting, deleting or inserting their corresponding vertices.

3.5 Theoretical Observations

DF and so its anytime version (ADF) speed up the computations of GED for two reasons. First, the upper and lower bounds strategy which reduces the search tree size as one may get rid of exploring unfruitful nodes. Second, the preprocessing step which helps in starting by the most promising vertices in G_1 in addition to getting rid of the recalculation of both vertex-to-vertex and edge-to-edge mappings. Unlike A^*, DF does not exhaust memory as the number of pending edit paths that are stored in the set $OPEN$ is $|V_1|.|V_2|$ in the worst case.

Anytime DF guarantees to find the optimal solution of $GED(G_1, G_2)$ if no time limit is set. It also provides better and better solutions and guarantees to find the optimal solution if enough time is available.

However, one should also be aware of the drawback that DF and its anytime version

Algorithm 10 The Anytime Version of Depth First GED algorithm (ADF)

Input: Non-empty attributed graphs $G_1 = (V_1, E_1, \mu_1, \zeta_1))$ and $G_2 = (V_2, E_2, \mu_2, \zeta_2)$ where $V_1 = \{u_1, ..., u_{|v_1|}\}$, $V_2 = \{u_2, ..., u_{|v_2|}\}$, μ and ζ are the attributes associated with vertices and edges respectively.

Output accessible at anytime: $UBCOST$ = a minimum cost edit path from g1 to g2 & UB = sequence of edit operations.

```
1:  $p \leftarrow$ root node of the tree with all vertices and edges of $G_1$ and $G_2$ as pending lists
2:  $OPEN \leftarrow \Phi$, $UB \leftarrow \phi$, $UBCOST \leftarrow \infty$
3:  Generate $C_v$, $C_e$
4:  Optional:                                                ▷ Steps below are optional
5:        ($UB$ , $UBCOST$) $\leftarrow BP(G_1, G_2)$
6:        Export $UBCOST$ and $UB$
7:        $sortedV_1 \leftarrow$ sortVertices$V_1$($V_1$,$V_2$)      ▷ in ascending order of $BP(G_1, G_2)$
8:  $Listp \leftarrow$ GenerateChildren($p$,$C_v$,$C_e$,sorted$V_1$($p$),pending$V_2$($p$))
9:  $Listp \leftarrow$ SortAscending($Listp$)                  ▷ according to $g(p)+h(p)$
10: for $p \in Listp$ do
11:     $OPEN$.AddFirst($p$)
12: end for
13: while $OPEN$ != $\Phi$ & TimeOut != TRUE & MemoryOut != TRUE do
14:     $p \leftarrow OPEN$.popFirst()          ▷ Take first element and remove it from $OPEN$
15:     $Listp \leftarrow$ GenerateChildren($p$)
16:     if $Listp = \Phi$ then
17:         for $v_i \in$ pending$V_2$($p$) do
18:             $q \leftarrow$ insertion($\epsilon$ , $v_i$)                   ▷ i.e., $\{\epsilon \rightarrow v_i\}$
19:             $p$.AddFirst($q$)
20:         end for
21:         if $g(p)$ < UBCOST then
22:             $UBCOST \leftarrow g(p)$, $UB \leftarrow p$
23:             Export $UBCOST$ and $UB$
24:         end if
25:     else
26:         $Listp \leftarrow$ SortAscending($Listp$)          ▷ according to $g(p)+h(p)$
27:         for $q \in Listp$ do
28:             if $g(q) + h(q) < UB$ then
29:                 $OPEN$.AddFirst($q$)
30:             end if
31:         end for
32:     end if
33: end while
34: Return($UBCOST$ , UB)                                       ▷ export final results
```

have which lies in exploring a part of the search tree for a long time. The solutions found while exploring this part of the tree look good locally but are tremendously less favorable when compared to other solutions in other parts of the search tree. Since DF and ADF are depth-first algorithms, it takes time to take steps backward specially if the error is made in a very early stage during the matching process.

Chapter 4

Parallel and Distributed Approaches for Graph Edit Distance

Life is not accumulation, it is about contribution. Stephen Covey

Contents

Abstract

In the aforementioned chapter, an optimized Branch-and-Bound GED algorithm was proposed (*i.e DF*). This algorithm works well on relatively small graphs. To optimize it more and thus scale up to match larger graphs, one may think of a parallel exploration of the search tree. To the best of our knowledge, parallel GED methods have never been proposed in the literature. Thus, in this chapter, we first report approaches that have been proposed in the literature to solve BnB in a fully parallel or distributed manner. Then, in the rest of this chapter, we propose a parallel and a distributed exact GED algorithms.

4.1 Parallel Computing versus Distributed Computing

In both parallel and distributed computings, a big task is partitioned into small sub-tasks each of which may have a different unit. In parallel systems, these units are referred to as *threads* while they are referred to as *processes* in distributed systems depends on whether or not these units have a shared memory.

In parallel computing, threads may have access to a shared-memory to exchange information between them. The variables, objects, and data structures in that environment are accessible to all threads. In a shared memory model, different threads may execute different statements, but any statement can affect the shared environment.

Distributed computing often assumes autonomous computing agents; each process has its own private memory. Information is exchanged by passing input/output (I/O) messages between threads. In 1985, Andrew Tanenbaum and Robert van Renesse offered the following definition for an operating system controlling a distributed environment [133]:

A distributed operating system is one that looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs).

There are two predominant ways of organizing computers in a distributed system:

- The client-server architecture [127] which consists of a system, called server, that delivers and manages most of the resources and several systems, called clients, who consumes these resources. The server handles multiple clients at the same time. It also dispatches work to clients and/or waits a request from clients. Neither the server nor clients share their resources with each other.
- The peer-to-peer architecture (P2P) [119] differs from the client-server architecture in the sense that peers, or participants, share a part of their own resources without going through a separate server. Such an architecture is necessary in file sharing applications. These shared parts are accessible by all peers directly since all of them have the same capabilities and responsibilities.

There are different design options dedicated to construct a distributed or a parallel computer system. Flynn [51] proposed a famous processing taxonmy which depends on the data and the operations applied on this data:

- **Single Instruction Single Data Stream (SISD):** This term refers to a computer architecture in which a single processor executes a single instruction stream to control the functioning of data stored in a single memory. This is the case of single thread. Older generation mainframes, minicomputers and workstations employ SISD instruction and data streams.
- **Single Instruction Multiple Data Stream (SIMD):** A local memory is assigned to each thread, and a portion of data is given to each thread. All the threads execute the same instruction on their own portions of data. Example of such applications: graphics processing, video compression, medical image analysis, etc. Moreover, most modern computers, particularly those with graphics thread units (GPUs) employ SIMD instructions and execution units.
- **Multiple Instruction Single Data Stream (MISD):** This term refers to parallel threads that are given multiple instructions which are executed on different portions of data. Carnegie-Mellon C.mmp computer (1971) is one of the few parallel computers which employ MISD instruction and data streams.

- **Multiple Instruction Multiple Data Stream (MIMD):** Each thread runs its different data and its different instructions. As an example of this design, we mention multicore threads and multi-threaded processes.

  A subcategory of MIMD is called Single Program Multiple Data Stream (SPMD) [9]. SPMD executes copies of the same program.

Our interest in this chapter is to propose a parallel and a distributed extension of our algorithm *DF* to be able to match large graphs. One of the best computing designs that suits *DF* is SIMD where a portion of the data (i.e., sub-search tree) is given to each thread/process and all threads execute the proposed method on their associated sub-search trees.

Both parallel and distributed computings can have the simultaneous execution of the same task on multiple threads/processes in order to obtain faster results. Figure 4.1 gives an idea about the parallel computing techniques (POSIX threads (PThreads) [25], Open Multi-Processing (OpenMP) [31, 1] and CUDA GPUs [100]) and the distributed models (Message Passing Interface (MPI) [102], Unified Parallel C (UPC) [138], Fortress [5] and MapReduce [40]) based on their architecture. Note that MPI, UPC and Fortress can be configured to work on a single machine with shared memory or on multi-machines where each process has its own separate memory.

We synthesize the aforementioned techniques taking into account these following criteria:

- **Architecture:** Parallel or distributed computing architecture.
- **Communication method:** The communication way between threads (e.g., shared memory, network, files system, etc.).
- **Fault Tolerance:** Is the parallel method able to continue its intended operation without failing completely when partial failure happens?
- **Scalability:** The capability of a system or a process to handle a growing amount of work (e.g., larger datasets) or to be easily expanded or upgraded, when required, to accommodate that amount of work by adding more machines to the distributed system.
- **Compute-Intensive versus Data-Intensive**: Parallel and distributed computing models can be generally classified as either compute-intensive, or data-intensive.
  - **Compute-intensive approach:** is used to describe application programs that have a high computational complexity. Such applications devote most of their execution time to computational requirements as opposed to I/O, and typically require small volumes of data.
  - **Data-Intensive approach:** processing large volumes of data typically terabytes or petabytes in size and typically referred to as *Big Data* [32].
- **Easiness to program:** is it easy for a user/programmer to write a parallel/distributed code using the model X?

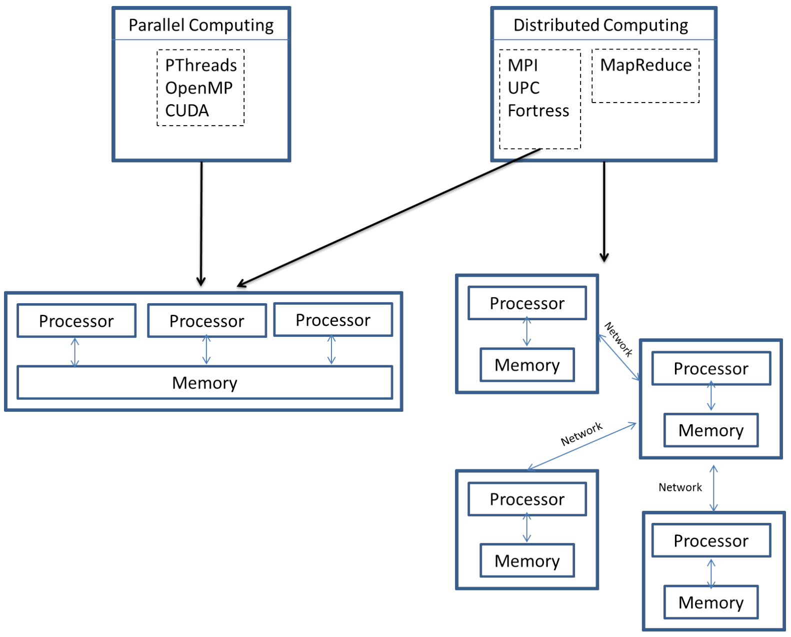


Figure 4.1: Parallel and distributed computing models and their supported system architecture (taken from [128])

Table 4.1 depicts a synthesis of the SPMD models. When taking a look at parallel models (i.e., threads and CUDA GPUs), we see that threads are easier to handle since all threads share the same CPU memory. Despite the fact that CUDA GPUs share memory, they need GPU-CPU memory transfer and that turns GPU programs to be hard to manage. On the side of distributed models, Hadoop MapReduce has an advantage over MPI. In Hadoop MapReduce, if one node, or thread, fails, a master program selects another free node to do its task. MPI when it faces such a problem, it needs to re-execute all the processes if such a problem happens.

If the problem is affordable (i.e., can be solved in one machine), it is much faster for a model to have one machine and to spread/send information between the CPUs locally with a shared memory rather than distributing the work between several machines. However, for a system to be scalable, it is more powerful to have a fully distributed parallel computing model.

For a detailed survey on the synthesis of parallel and distributed computing models based on more criteria, we kindly refer the interested reader to [128].

4.2 Parallel and Distributed Branch and Bound Approaches

To cope with the inherent complexity of GED, the use of parallel or disributed computing is argued. However, the parallel execution of a combinatorial optimization problem is not trivial. The main questions to be answered when trying to build/propose a parallel or a distributed version of a GED algorithm appear to us as follows:

Characteristics	**Threads and OPEN-MP**	**GPU**	**MPI, UPC and Fortress**	**Hadoop MapReduce**
Architecture	Parallel	Parallel	Distributed	Distributed
Communication Method	Memory	Memory + PCI Express bus between CPU and GPU	network	Distributed File System
Fault Tolerance	Stopping the execution of the program and re-executing all the jobs	Stopping the execution of the program and re-executing all the jobs	Stopping the execution of the program and re-executing all the jobs	Thread failure: another thread re-executes its job
Data-Intensive or Compute-intensive	Compute-intensive	Data-intensive and recently it has also become compute-intensive	Compute-intensive and data-intensive can be in parallel	Data-intensive
Scalability	−−	−−	++	++
Easiness to Program	−	−−	++	++

Table 4.1: SPMD models.

1. What should be parallelized? what is time consuming? *h(p)*, *g(p)*, edge operations or tree node exploration?
2. How many subtask(s) should be associated to each thread?
3. What is the estimated time per subtask?
4. What is the needed memory per subtask?
5. What is the number of CPUs (in case of the parallel approach) and/or machines (in case of the distributed approach) that are needed?
6. When does the parallelism, or distribution start? In other words, how many subtasks one may need to generate before the parallelism or distribution starts?
7. How to efficiently distribute the search tree nodes of the irregular search tree among a large set of threads? [1]
8. How to keep all threads busy?
9. How and when to update Upper Bound if it is upgraded?

Since *DF* is a BnB algorithm, the aforementioned raised questions will be answered after exploring the state-of-art methods dedicated to solving BnB. Processing the search tree in parallel and distributed fashion has been studied for decades [134, 56]. In this section we shed spot of light on parallel BnB (in Section 4.2.1) and distributed BnB (in Section 4.2.2). Then, in Section 4.2.3, we synthesize all of these works.

[1]Irregular search tree indicates that the number of children of each node differs from the others because of the pruning strategy. Therefore, the depth of a sub-tree is different than the depth of the other sub-trees. This is known as the irregularity of branch-and-bound problems [56].

4.2.1 Parallel Branch-and-Bound Approaches

In [103], a master-slave parallel formulation of depth-first search for solving the 15-puzzle problem [99] was proposed on plenty of parallel architectures. Each thread takes a disjoint part of the search space. Once a thread finishes its assigned part, a donor thread gives some unexplored nodes of the search space to this requester thread. To split the work between the donor and the requester threads, the half-split strategy was adopted. The selection of such a strategy was based on the work of [78] where different splitting strategies have been analysed. As stated in [78], half-split leads to an overall high efficiency for the shared-memory architecture. The selection of nodes to be donated was also discussed in the paper. Figure 4.2 shows a search tree that was partially explored. The nodes selection depends on the nature of the search space (whether it is a regular or a non-regular one). Under the formulation of [103], the scalability of different architerctures was evaluated and the most scalable load balancing schemes were determined in [79]. In [103], the number of threads, the size of the problem and the speed are mentioned. However, the initial number of nodes to be generated before parallelism starts is not defined.

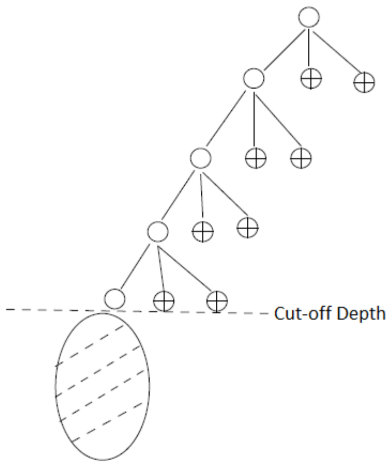


Figure 4.2: A search tree that is partially explored and stopped to be splitted. Note that the nodes ”+” are the ones that can be given to threads (i.e., requesters)

Chakroun et al in [28] have been put forward a template that transforms the unpredictable and irregular workload associated to the explored BnB tree into regular data-parallel kernels [2] optimized for the single instruction multi-data based execution model of GPUs. At first, the pool of sub-problems is selected from the tree and off-loaded to the GPU where the *branching operator* is applied. Each thread only generates one child of its parent node according to its unique identifier. Apart from the pool of parent nodes, a pool containing the number of children of each parent node is registered. A second kernel is then in charge of parallel evaluation of $g(p) + h(p)$ as well as parallel elimination of unpromising branches thanks to $UBCOST$. Besides equivalent operations, the pruning operator on top of GPU reduces the time of transferring the resulting pool from the GPU

[2]Kernels are functions that are executed N times in parallel by N CUDA GPUs threads.

to the CPU since the non-promising generated sub-problems are kept in the GPU memory and deleted there. This approach has been tested on the Flow-Shop scheduling problem [132] on 20 machines, each of which has GPU card which contains 480 CUDA cores (15 multiprocessors with 32 cores each). In fact, this approach solved the irregularity of BnB however it makes the explorations take longer time since in the first kernel each machine generates only one child at each time while the second kernel eliminates branches.

A producer-consumer hybrid depth-first/best-first was proposed to solve the Flow-Shop problem [33]. The master thread keeps generating the tree nodes at a predetermined level (i.e., level i) and saves them to a work pool. Then, each worker threads takes a node with the minimum lower bound $lb(p)$ from the pool and explores it in a depth-first way. Generating and exploring nodes are repeated until finding the solution of the problem. Synchronization between the master and the workers is implemented with semaphores, especially for the work pool management. The master thread has to stop generating candidate nodes if no more free slot is available in the pool. A similar situation occurs for an idle worker, it has to wait if the pool is empty (i.e., no item is available). Experiments were conducted on a 16-core multithread. Results showed that the execution time is proportional to the number of explored nodes of the flow-shop problem. That is, the smaller the number of explored nodes, the less the execution time. Moreover, increasing the pool size (i.e., the list where generated nodes are saved) at level i can significantly improve the performance. Results also showed that increasing the number of threads can reduce the execution time. This model is interesting since it takes advantage of both breadth-first and depth-first algorithms. However, when the master generates all nodes up to a level i, it may generate some misleading nodes (i.e., nodes that do not lead to the optimal solution).

A dynamic work-stealing [3] depth eager scheduling method for solving Traveling Salesman Problem was proposed in [96]. In the beginning, a depth parameter is set to 2, which means all the nodes whose level in the search space is 2 are considered atomic and are processed by threads with no further subdivision. Then each thread works on its associated problems. Threads are organized in a tree. When a thread runs out of work it requests work from some threads that it knows (i.e., first from its children, if any, and, if that fails, from its parent.). This balances the computational load as long as the number of tasks per thread is high. When a thread finds a better solution, it propagates it to its entire tree of threads. Experiments were conducted on a 1152-processor machine. Results showed that the average time per task decreases by over 50% when the dynamic depth increment is changed from 1 to 2 which is the desired effect of splitting large atomic tasks. They have also studied the effect of speed-up when increasing the number of threads (64 threads, 256 threads, 512 threads, and 1024 threads). The communication of this approach is asynchronous [4], and thus threads only communicate if they succeed in updating the upper bound. An eager scheduling approach is used to make the tasks somehow balanced depending on the difficulty of each branch, this scheduling is presented in [26]. However, the propagation of the upper bound takes time and the search tree may not be pruned as

[3]Work-stealing refers to the case when a thread runs out of work it requests, or steals, work from another thread that is not idle.

[4]Asynchronous communication indicates that no thread waits another thread to finish in order to start its new task [15].

soon as possible since the propagation is done in an asynchronous way.

An OPEN-MP approach has been put forward in [42]. The proposed approach addressed the Knapsack Problem [92]. T threads are created and established by the master program. Moreover, the master program generates tree nodes and put them in a queue. Then, T tree nodes are removed from the queue and assigned to each thread. Each thread only takes one node, generates its children and at the end of its exploration inserts the children in the shared list. The best solution found so far must be modified carefully where only one thread can change it at any time. The same thing is done when a thread tries to insert a new tree nodes in the global shared queue. Such a model slows down the exploration of the search tree because of the access to the critical region by all threads at each iteration.

A parallel BnB approach dedicated to solving the problem of the maximum labeled clique was proposed in [93]. Two parallelism levels are used: Bit and Thread parallelisms. A bitset encoding for SIMD-like parallelism is used to represent the graph and so its vertices. For thread parallelism, the recursive part of the sequential algorithm is transformed into tree-like form. A first level in the search tree is constructed where each tree node represents a vertex in the graph to be expanded. These tasks (i.e., expanding vertices) are placed on queue. Each thread takes one task to solve. When the queue becomes empty, the idle thread can steal work from other threads. A single shared variable (i.e., incumbent), is updated carefully by threads using an atomic. This approach was compared to a sequential approach proposed in the same paper and another mathematical programming approach proposed by Carrabs et al [120]. The parallel approach was always faster than the mathematical programming one sometimes by four or five orders of magnitude. Moreover, the parallel approach always outperformed the sequential approach on both trivial and non-trivial instances.

In [4], a breadth-first/depth-first algorithm for solving inexact subgraph matching was proposed. This algorithm aims at solving subgraph isomorphism. At each iteration, each thread takes a node from its queue to be solved by expanding it in a depth-first search way until its branch is explored, updating the local best permutation and the corresponding degree of mismatch and eliminating test. Afterwards, a global permutation with its corresponding degree of mismatch is updated and given to all threads when all of them finish solving their chosen nodes or problems, then, a next node is chosen by each thread. A thread becomes inactive when it has no node left in its local queue. Load balancing is performed if the number of inactive threads with empty queues is above a threshold T. This algorithm is iterative and so each thread explores its node until reaching a leaf node. Therefore, some threads may be idle for a certain amount of time waiting the others to finish their exploration in order to start the next iteration. Moreover, the best permutation and the best degree of match are only updated at the end of each iteration, such a fact will not prune the search space as fast as possible.

4.2.2 Distributed Branch-and-Bound Approaches

In [12], a one-iteration MPI approach was proposed for three phase electrical distributed networks. In the beginning, a specific number of nodes are generated by the master process. When this number is reached, no more nodes could be generated. The master then gives, or sends, a node to each slave. Then, each slave starts the exploration of the search tree in a depth-first way. Once a slave finds a better upper bound, it sends to the master that updates all slaves. Once a slave finishes its exploration of its assigned node, it sends a message to the master asking for a new node and process continues. The drawback of such an approach is that once all the nodes generated by the master are given to all threads, some threads might become idle because they finished their associated nodes. Such a fact does not allow this approach to use all its resources at each time.

In addition to the parallel approach that was proposed in [42] (see Section 4.2.1), an MPI approach for BnB was also introduced. This work is similar to [12]. However, the way of exploring the tree is left to the user so one can choose either depth-first, a best-first or a breadth-first.

A generic library, referred to as DryadOpt, has been put forward in [20] for distributed BnB. This library is implemented on top of DryadLINQ [151], a distributed data-parallel execution engine similar to Hadoop and Map-Reduce. Two standard ways are used to explore the search tree (breadth-first and depth-first). In the beginning, T nodes are generated by DryadOpt in a breadth-first way. Then, these nodes are divided randomly into N parts, where N represents the number of machines. Each machine explores its associated part in a depth-first way. As in Hadoop, DryadLINQ has a restrictive communication and so threads that belong to different machines cannot communicate. In order to overcome that, a budget time t_i is set per machine. All machines have a similar budget time. Once they exceed it, they stop their exploration. Then the resulting work of all machines is merged, divided into K parts and distributed again randomly to all machines in order to start a new round and so on. The algorithm ends when finishing the exploration of all nodes. Thus, the number of rounds, or iterations, cannot be known except at run time. DryadOpt is not deterministic. If, for example, in the first round data corruption occurs after having been associated to a machine, DryadOpt must re-execute the computation in order to have N parts and thus give a new copy of the data to the machine. However, DryadOpt will not be able to give the same data to the machine. To this end, a non-determinism may occur. Moreover, the upper bound cannot be shared except at the end of each round. Such a fact will not let machines prune their search trees as soon as possible.

4.2.3 Discussion

In this section, a synthesis is made to better understand the aforementioned parallel and distributed BnB methods. To achieve such a synthesis, the following criteria are taken into account:

- **Thread/process Tasks:** The type of tasks that was associated to each thread/process (e.g., a sub-tree or a branch exploration, children generation, etc.).
- **Tasks synchronization:** The term *synchronous tasks* refers to threads that cannot take a next task to execute until waiting the other threads to finish their execution. In other words, all threads have to start handling the new task at the same time. On the other hand, in *asynchronous* tasks, when a thread finishes its assigned task,

it saves its current state and takes the next task to be executed without waiting the other threads.

- **Massively Parallel:** The term *massive* parallel refers to phases without (a lot or any) communication. Since communication is costly, a massively parallel approach is the one that does not need so much communication.
- **Number of threads/processes:** The maximum number of threads/processes that still provides good speedup and on which the algorithm was tested.
- **Addressed Problem:** The type of problem addressed by each paper.

Table 4.2 shows that no method was dedicated to solving GED. In fact, even if [4] has tackled subgraph error-tolerant GM problem, it did not take attributes into account. Moreover, the search tree is not pruned as soon as possible since it is based on a synchronous communication, see Section 4.2.2.

The approaches in [103], [33] and [96] are of great interest since the communication is asynchronous and thus there is no need to stop a thread if it did not finish its tasks, unless another thread ran out of tasks as in [103] and [96]. In [33], however, work stealing is not integrated. Thus, when there are no more problems to be generated by the master thread, some threads might become idle for a certain amount of time while waiting the other threads to finish their associated tasks. For *GED*, the work-stealing is important to keep the amount of work balanced between all threads. In the literature, this process is referred to as *Load Balancing*, see Section 4.3.3.

Even if DryadOpt [20] was massively parallel, the best upper bound was not shared with the other machines except at the end of each round. In fact, that is a problem in DryadLINQ [151] and Hadoop [145] which is due to the restricted communication of both of them.

At the beginning of Section 4.2, a few number of questions was raised. After having explored the state-of-the-art of BnB parallel and distributed approaches, we provide some answers here:

Regarding subtasks (see Question 1), in the GED problem, *g(p)*, *h(b)* and edge operations tree nodes are less time-consuming tasks than tree nodes exploration. Thus based on that, subtasks can be tree nodes.

Questions 2, 3, 4 and 5 are linked together and are dependent on the difficulty of the subtasks to be solved. There is no rule or advice provided in the literature to decide what is the number of processors an application has to have or what is the number of subtasks that has to be created before parallelism or distribution starts.

Concerning the distribution of the subtasks (see Question 8), the choice here depends on the regularity and predictability of the duration/difficulty to solve a subtask. In the case of GED, the decomposition, or distribution, is irregular due to the bounding/pruning. Such a thing cannot be known except at run time. Thus, a load balancing strategy seems to be mandatory to optimize the distribution of subtasks and to avoid having idle threads.

Concerning the questions related to the division of the problem into subtasks (see Questions 6 and 7), it seems that the classical way is to give each worker a branch of

Reference	**Thread/process Tasks**	**Communication Pattern**	**Massively Parallel?**	**# threads/processes**	**Addressed Problem**
[103]	Thread: Sub-tree	Asynchronous	No	128 threads	15-puzzle
[28]	One kernel for children generation and another for children elimination	Synchronous	No	480 CUDA cores (15 multiprocessors with 32 cores each)	Flowshop
[33]	Thread: Sub-tree	Asynchronous	Yes	1 machine with 16-core multi-thread	Flow-Shop
[96]	Thread: Sub-tree	Asynchronous	No	1152 threads	Traveling Salesman
[42] (two approaches)	Thread/Process: Children generation	Synchronous	No	32 threads	Knapsack Problem
[4]	Thread: Tree branch	Synchronous	Yes	1024 threads	Minimum-distance between two unattributed graphs
[93]	Thread: Tree branch	Asynchronous	No	2 cores, 4 threads	maximum labeled clique
[12]	Process: Sub-tree	Asynchronous	No	267 machines each has 4 cores	Three-phase electrical distribution networks
[20]	Process: Sub-tree	Synchronous	Yes	128 machines each has 4 cores	the Steiner tree problem

Table 4.2: Synthesis of Parallel and Distributed methods for BnB

the tree or a part of the sub-tree to be explored in a parallel or distributed way. The problem is then to estimate the time needed to explore each branch. Finally (see Question 8), the upper bound UB is a necessary variable to be shared with all workers. Thus we have to choose an architecture that allows an easy and fast way to share/communicate UB between workers. On this basis, the next sections present the way(s) we propose to parallelize and distribute the GED algorithm.

4.3 Load Balancing for a Parallel Graph Edit Distance

4.3.1 Motivation

As mentioned in Section 4.2.3, the search tree of GED contains nodes that represent partial edit paths. When thinking of a parallel and/or a distributed approach of DF, these edit paths can be given to threads as tasks to be solved. Such a step, divides the GED problem into smaller problems. The GED problem is irregular in the sense of having an irregular search tree where the number of nodes differs, depending on the ability of $lb(p)$ to prune the search tree. Based on that, as already explained, it becomes hard to estimate the time needed by threads to explore a sub-tree. Likewise, the number of CPUs and/or machines have to be adapted to the amount and type of data that have to be analyzed. Some experiments in Chapter 5 illustrate this point and are followed by a discussion.

As seen in [33], a breadth-first strategy is performed before a depth-first one starts. However, when generating nodes using breadth-first, one may generate unfruitful nodes.

Instead, one may think of A^* since it starts exploring nodes that lead to the optimal solution, if $lb(p)$ is carefully chosen. But, there are two key issues: First, how many nodes shall be generated by A^* before a *DF* procedure starts? Second, how to divide, or dispatch, the nodes between threads?

The method proposed by [103] requires lots of communication between threads. First, for the upper bound and second for threads to steal some work from the other threads. However, that will not be a bottleneck for solving *GED* in a parallel. Roughly speaking, only *UBCOST* and *UB* have to be shared with all threads. In fact, at the beginning of *DF*, *UBCOST* and so *UB* might be modified. However, after a certain amount of time, they will not be updated that often, see Figure 3.2. Furthermore, a work-stealing is beneficial to cope with the irregularity of the *DF*'s search tree. Thus, when any thread finishes all its assigned nodes, a certain amount of nodes can be given to it by another thread. Such a procedure is referred to as *load balancing* in the literature. The following question should be addressed "Which nodes can be taken, or stolen, from one thread to another when a thread runs out of tasks?".

Before digging into the details of this approach, the load balancing problem is formally defined and presented to establish the basement of an efficient parallel GED algorithm. Moreover, an overview of our proposal is given.

4.3.2 From Graph Edit Distance to Load Balancing

GED is a discrete optimization problem that faces the combinatorial explosion curse. The complexity class of GED was proven to be NP-hard where the computation complexity of matching is exponential in the number of vertices of the involved graphs [153].

Generally speaking, the combinatorial optimization problem is characterized by an unpredictably varying unstructured search space. The search space is represented as an ordered tree.

The initial and leaf tree nodes correspond to the initial state and the final acceptable state in the search tree, respectively. Each edge represents a possible way of state change. A path from the initial node to a leaf tree node is a feasible solution to the optimization problem. A combinatorial optimization problem is essentially the problem of finding a minimum-cost path from an initial node to a goal node in the search tree which represents a partial solution. More concretely in GED, each tree node is a sequence of edit operations. Leaf nodes are complete edit operations sequences (edit path) while intermediate nodes are partial solutions representing partial edit path. An example of a search tree corresponding to GED computation is shown in Figure 4.3.

The parallelism of combinatorial optimization problems is not trivial. In a parallel combinatorial search application, thread evaluates candidate solutions from a set of possible solutions to find one that satisfies at best a problem-specific criterion. Each thread searches for optimal solutions within a portion of the solution space. The shape and size of the search space usually change as the search proceeds. Portions that encompass the optimal solution with high probability will be expanded and explored exhaustively, while portions that have no solutions will be discarded at run-time. Consequently, tree nodes are generated and destroyed at run-time. To ensure parallel efficiency, tree nodes have to

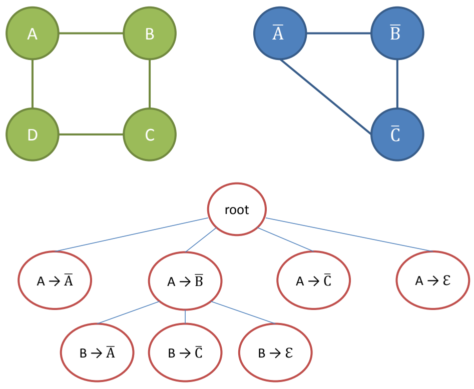


Figure 4.3: An incomplete search tree example for solving the GED problem. The first floor represents possible matchings of vertex A with each vertex of the second graph (in blue). Each tree node is a partial solution which is to say a partial edit path

be dispatched at run-time. Hence, the patterns of workload changes of the threads are difficult to predict.

The parallel execution of combinatorial optimization problems relies on load balancing strategies to divide the search space iteratively at run-time. From the viewpoint of a load distribution strategy, parallel optimizations fall in the asynchronous category. A thread initiates a balancing operation when it becomes lightly loaded or overloaded. The objective of the data distribution strategies is to ensure a fast convergence to the optimal solution such that all the tree nodes are evaluated as fast as possible. To achieve this goal our proposal makes sure that all threads have more or less the same amount of load while ensuring threads to explore promising tree nodes first.

4.3.3 Load Balancing Problem

A parallel program is composed of multiple threads, each **thread** is a processing unit which performs one or more works. Multiple threads can exist within the same process and share resources such as memory. In particular, the threads of a process share its instructions (executable code) and its context (the values of its variables at any given moment). A **work** is defined by a workload and it is the smallest unit of concurrency the parallel program can exploit. In GED, we define a work as a tree node to be expanded.

Load balancing algorithms can be broadly categorized into static and dynamic. Static load balancing algorithms distribute works to threads once and for all, in most cases relying on a priori knowledge about the works and the system on which they run, while dynamic algorithms bind works to threads at run-time. A very detailed definition of load balancing models can be found in [149].

Creating a parallel program involves first decomposing the overall computation into works and then assigning the works to threads. The decomposition together with the assignment steps is often called partitioning. The assignment on its own is referred to static load balancing.

4.3.3.1 Static Load Balancing

Static load balancing algorithms rely on the estimated execution times of works and inter-thread communication requirements. It is not satisfactory for parallel programs that are of the dynamic and/or unpredictable kind. For instance, in GED, we cannot predict the time needed by a sub-tree to be fully explored.

We are given a set of works $\mathcal{W}$ and a set $\mathcal{A}$ of parallel threads. A work j in the set of works $\mathcal{Q}$ which is processed on a thread k has a workload measure $\omega_{k,j}$. The goal is to assign works to threads such that the maximum thread load is minimized. In equation 4.1, an integer linear programming formulation of the static load balancing formulation is provided. In equation 4.1, variable $x_j^k = 1$ if a work j is processed on thread k and 0 otherwise. $\omega_{\max} = \max_{k \in \mathcal{A}} \sum_{j \in \mathcal{Q}} \omega_{k,j} x_j^k$ is the maximum load of a thread. $\overline{\omega} = \frac{1}{|\mathcal{A}|} \sum_{k \in \mathcal{A}} \sum_{j \in \mathcal{Q}} \omega_{k,j}$ is the average workload of the threads $\mathcal{A}$.

Static load balancing formulation

$$\min_{\mathbf{x}} \left(\omega_{\max} - \overline{\omega} \right) \tag{4.1a}$$

$$\text{subject to} \quad \sum_{k \in \mathcal{A}} x_j^k = 1 \quad \forall j \in \mathcal{Q} \tag{4.1b}$$

$$\sum_{j \in \mathcal{W}} \omega_{k,j} x_j^k \leq \omega_{\max} \quad \forall k \in \mathcal{A} \tag{4.1c}$$

$$x_j^k \in \{0,1\} \quad \forall (j,k) \in \mathcal{A} \times \mathcal{W} \tag{4.1d}$$

$$\tag{4.1e}$$

Static load balancing problem is closely related to a problem called $P||C_{\max}$ [43] in scheduling theory. It is known to be an NP-hard problem when the number of threads is greater or equal to 2. Consequently, the static load balancing problem is a NP-hard problem too.

4.3.3.2 Dynamic Load Balancing

The execution of a dynamic load balancing algorithm requires some means for maintaining a consistent view of the system state at run-time. Generally, a dynamic load balancing algorithm consists of three components: a load measurement rule, an initiation rule, and a load balancing operation.

Load Measurement Dynamic load balancing algorithms rely on the workload information of threads. The workload information is typically quantified by a load index, a non-negative variable taking on a zero value if the thread is idle, and taking on increasing positive values as the load increases [149]. Since the measure of load would occur frequently, its calculation must be very efficient.

Initiation Rule This rule dictates when to initiate a load balancing operation. The execution of a balancing operation incurs non-negligible overhead; its invocation decision must weight its overhead cost against its expected performance benefit. An initiation policy is thus needed to determine whether a balancing operation will be profitable.

Load Balancing Operation This operation is defined by three rules: location, distribution and selection rules. The location rule determines the partners of the balancing operation, i.e., the threads to involve in the balancing operation. The distribution rule determines how to redistribute workload among threads in the balancing domain. The selection rule selects the most suitable data for transfer among threads.

4.3.4 Dynamic Load Balancing Models

Consider a parallel program running on a parallel computer. The parallel computer is assumed to be composed of T identical threads, labelled from 1 through T. Threads are interconnected by a shared memory. Threads communicate through shared variables. The parallel computation comprises a large number of works, which are the basic units of workload. Works may be dynamically generated and consumed for load balancing as the computation proceeds. We distinguish between the computational operation and the balancing operation. At any time, a thread can perform a computational operation or a balancing operation. This dynamic workload model is valid in situations where some threads are performing balancing operations, while the others are performing computational operations. The situation is common in parallel tree-structured computations, such as combinatorial optimizations.
Let t be a time variable, representing global real time. We quantify the workload of a thread k at time t by ω_k^t. Let ϕ_i^{t+1} denote the amount of workload generated or finished between time t and $t+1$. Let $\mathcal{I}(t)$ denote the set of thread performing balancing operations at time t. Then, the workload change of a thread at time t can be modelled by the following equation:

$$\omega_k^t = \begin{cases} balance_{j\in\mathcal{A}(k)}(\omega_k^t, \omega_j^t) + \phi_k^{t+1} & \text{if k} \in \mathcal{I}(t); \\ \omega_k^t + \phi_k^{t+1} & \text{otherwise} \end{cases}$$

where $balance()$ is a load balancing operator and $\mathcal{A}(k)$ is a set of threads that are within the load balancing domain of thread k. This model dynamic workload model, given by [149], is generic because the operator $balance()$, the balancing domain $\mathcal{A}()$ of a thread, and the set of threads in load balancing at a certain time t, $\mathcal{I}(t)$, are left unspecified. The operator $balance()$ and the balance domain $\mathcal{A}()$ are set by load balancing operations; the set $\mathcal{I}(t)$ is determined by the initiation rule of the load balancing algorithm. The choice of $\mathcal{I}(t)$ is independent on the load balancing algorithm where any initiation policy can be used in conjunction with any load balancing operation in implementation.

4.3.4.1 Dynamic Load Balancing Formulation

We denote the overall workload distribution at certain time t by a vector $\Omega^t = (\omega_1^t, \omega_2^t, \cdots, \omega_P^t)$. Denote its corresponding equilibrium state by a vector $\overline{\Omega}^t = (\overline{\omega}^t, \overline{\omega}^t, \cdots, \overline{\omega}^t)$, where $\overline{\omega}^t = \sum_{i=1}^{P} \frac{\omega_k^t}{P}$. The workload variance, denoted by υ^t, is defined as the deviation of Ω^t from $\overline{\Omega}^t$; that is,

$$\upsilon^t = \|\Omega^t - \overline{\Omega}^t\|^2 = \sum_{i=1}^{P} (\omega_k^t - \overline{\omega}^t)^2 \tag{4.2}$$

The optimization problem is introduced by minimizing the workload variance at any time t. Variable of the problem is the set of threads performing balancing operations at time t.

$$\min_{\mathcal{I}(t)} \upsilon^t \quad \forall t \in [0, \infty[\tag{4.3}$$

In equation 4.3, the variable t is integer and represents the discrete time rated by the initiation rule of the load balancing algorithm.

The objective of the data distribution strategy is to ensure a fast convergence to the optimal solution such that all the tree nodes are evaluated as fast as possible.

4.3.5 Decomposition and Load Balancing Procedures

Now that we have explained the load balancing problem and models, we can describe how we have used these models to define a parallel version of our *DF* algorithm. Our algorithm referred to as parallel *DF* (*PDFS*) consists of three main steps: Decomposition, Branch-and-Bound and Load Balancing. Figure 4.4 pictures the whole steps of *PDFS*.

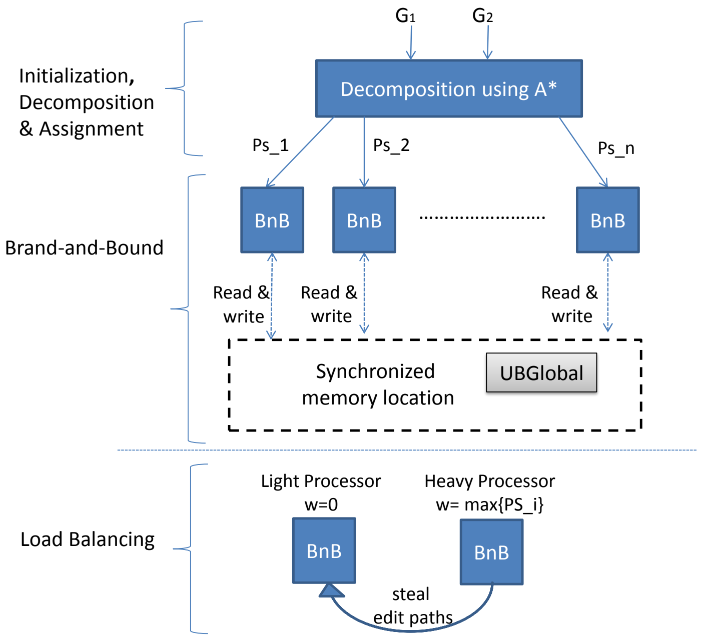


Figure 4.4: The main steps of *PDFS*

4.3.5.1 Initialization, Decomposition and Assignment

Decomposition Before starting the parallelism a decomposite approach is applied aiming at distributing the workload or subproblems among threads. For that purpose, N edit paths are first generated using A^* by the main thread and saved in the heap. Afterwards, the N edit paths are sorted as an ordered tree starting from the node whose $lb(p)$ is minimum up to the most expensive one. Note that N is a parameter of *PDFS*.

Assignment Let $\mathcal{Q}$ be the set of partial solutions outputted by A^*. Assigning partial solutions to parallel threads is equivalent at solving the static load balancing problem stated in equation 4.1. Due to the complexity of the problem, we chose to avoid an exact computation and we adopted an approximated algorithm. The greedy algorithm called "Graham's Rule". Algorithm 11 depicts the strategy we have followed. Once the partial edit paths are sorted in the centralized heap (line 1), the local list *OPEN* of each thread is initialized as an empty set (lines 2 to 4). Each thread receives one partial solution at a time, starting from the most promising partial edit paths (line 8). The threads keep taking edit paths in that way until there is no more edit path in the centralized heap (lines 6 to 10).

Algorithm 11 Dispatch-Tasks

```
Input: A set of partial edit paths $\mathcal{Q}$ generated by $A^*$ and $T$ threads.
Output: The local list $OPEN$ of each thread $T_i$
 1: $\mathcal{Q} \leftarrow$ sortAscending($\mathcal{Q}$)
 2: for $T_{index} \in T$ do
 3:     $OPEN_{T_{index}} \leftarrow \{\phi\}$
 4: end for
 5: $i$=0                                  ▷ a variable used for thread's indices
 6: for $p \in \mathcal{Q}$ do
 7:     $index = i \ \% \ |T|$
 8:     $OPEN_{T_{index}}$.addTask($p$)
 9:     $i$++
10: end for
11: Return $OPEN_{T_{index}} \quad \forall\, index \in [1, |T|]$
```

Each thread maintains a local heap to keep the assigned edit paths for exploring edit paths locally. Such an iterative way guarantees the diversity of nodes difficulty that are associated to each thread.

4.3.5.2 Branch and Bound

When the GED problem is decomposed and assigned to different threads, each thread executes a serial BnB algorithm for exploring its edit paths locally. In this section, the BnB and load balancing procedures are detailed.

Branching Procedure Initially each thread has only its assigned edit paths in its local heap tree *OPEN* (i.e., the set of the edit paths), found so far. This procedure is similar to *DF*, see Section 3.3. The exploration starts with the first most promising vertex u_1 in *sorted*-V_1 in order to generate children of the selected edit path. Then, the children are added to *OPEN*. Consequently, a minimum edit path (p_{min}) is chosen to be explored by selecting the minimum cost node (i.e., $min(g(p) + h(p))$) among thechildren of p_{min} and so on. Threads backtrack to continue the search for a good edit path if p_{min} equals ϕ. In this case, we try out the next child of *parent*(p_{min}) and so on.

Bounding Procedure As in *DF*, pruning, or bounding, is achieved thanks to $h(p)$, $g(p)$ and an upper bound *UBCOST* obtained at node leaves. However, in *PDFS*, *UBCOST* and *UB* are shared between all threads since it is saved in a shared memory.

Selection Rule A systematic evaluation of all possible solutions is performed without explicitly evaluating all of them. The solution space is organized as an ordered tree which is explored in a depth-first way. In depth-first search, each node is visited just before its children. In other words, when traversing the search tree, one should travel as deep as possible from node i to node j before backtracking.

4.3.5.3 Load Balancing

Load Measurement Each thread T_i provides some information about its workload or weight index ω_i. Obviously, the number of edit paths of a thread can be a workload index. However, this choice may not be accurate since BnB computations are irregular with different computational requirements. Several workload indices can be adapted. One could think of $h(p)$. Formally:

$$w_i = \sum_{j=1}^{|OPEN_{T_i}|} h(p_j) \quad \forall i \in T_i$$

where $OPEN_{T_i}$ is the local OPEN of thread T_i. However, $h(p)$ can be hard to interpret, it can be small either because p is close to the leaf node or because p is a very promising solution. To eliminate this ambiguity one can count the number of vertices in G_1 that have not been matched yet. Formally:

$$w_i = \sum_{j=1}^{|OPEN_{T_i}|} |pendingV_1(p(j)| \quad \forall i \in T_i$$

In our approach, we have selected the latter.

Initiation Rule An initiation rule dictates when to initiate a load balancing operation. Its invocation decision must appear when a thread workload index ω_i reaches a zero value that is to say if the thread is idle.

$$if\ \omega_i^t = 0 \rightarrow \mathcal{I}(t) \text{ where } i \in \{1, 2, \cdots, P\}$$

Load Balancing Operation In parallel BnB computations, each thread solves one or more sub-problems depending on the decomposite procedure. In our problem, two threads are involved in the load balancing operation: heavy and idle/light threads referred to as T_H and T_L, respectively. When a thread becomes idle or light, the heaviest thread T_H will be in charge of giving to T_L some edit paths to explore. All the edit paths of the heavy thread are ordered using their $lb(p)$. The heavy thread divides the best edit paths between it and the light thread. This procedure is done through a peer-to-peer communication since both threads have a shared memory. Algorithm 12 presents the load balancing step which guarantees the exploration of the best edit paths first since each thread holds some promising edit paths. In the beginning, ω_H and ω_L are calculated (lines 1 and 2). At each iteration, a verification of whether or not ω_L is less than $\frac{\omega_H}{2}$ is achieved (line 3) since the objective is to divide the workload between T_H and T_L. $OPEN_{T_H}$ is arranged in the same way as DF (i.e., in a depth-first way). In order to ensure that T_H and T_L take both most and less promising edit paths as well as edit paths whose search tree level is lowest and highest, at each iteration, T_H gives its second located edit path p to T_L and removes it from its sub-tree $OPEN_{T_H}$ (line 4). T_L then inserts p to its $OPEN_{T_L}$ (line 5). Note that T_L inserts edit paths in the same order as they were in $OPEN_{T_H}$ (i.e., by putting the edit path that should be explored first in the first cell and so on). Before starting the next iteration, the new ω_L is calculated (line 5). Algorithm 12 terminates when ω_L becomes equal or greater than ω_H.

Algorithm 12 Load-Balancing

```
Input: A heavy thread $T_H$ and a light thread $T_L$
Output: the local lists $OPEN$ of $T_L$ and $T_H$ (i.e., $OPEN_{T_L}$) and $OPEN_{T_H}$)).
1: $\omega_H \leftarrow T_H$.getWorkLoad()
2: $\omega_L \leftarrow T_L$.getWorkLoad()
3: while $\omega_L < \frac{\omega_H}{2}$ do
4:     $p \leftarrow OPEN_{T_H}$.getandRemove(2)
5:     $OPEN_{T_L}$.addLast($p$)
6:     $\omega_L \leftarrow T_L$.calculateWorkLoad()
7: end while
8: Return $OPEN_{T_L}$ and $OPEN_{T_H}$
```

4.3.5.4 Thread Communication

All threads share C_v, C_e, *sorted*-V_1 (read-only), *UB* and *UBCOST* (read/write). Since all threads try to find a better *UBCOST* and *UB*, a memory coherence protocol is required on the shared memory location of *UBCOST* and *UB* [83]. When two threads simultaneously try to update *UBCOST*, a synchronization process is applied in order to make sure that only one thread can access the resource at a given point in time.

4.3.6 Pseudo Code

Algorithm 13 summarizes all the aforementioned procedures of *PDFS*. As in *DF*, *PDFS* starts by generating C_v and C_e (line 1). Since *PDFS* is a parallel version of *DF*, C_v and C_e are put in a shared memory so that all threads can access to them. *BP* is run and its solution (i.e., *UB* and *UBCOST*) is used as an upper bound and put in shared variables UB_{shared} and $UBCOST_{shared}$ (line 2). Then, the vertices of G_1 are sorted as explained in Section 3.3.2.2 (line 3). A^* with *lb2* is executed and stopped once N partial edit paths are generated and saved in a list $\mathcal{Q}$ (line 4). The edit paths of $\mathcal{Q}$ are then divided between all threads T using the Dispatch-Tasks procedure, see Algorithm 11. At the beginning of *PDFS*, there is no heavy or light thread and thus their indices are equal to -1 (line 6). Each thread T_i runs a BnB procedure on its editpaths that are saved in its local sub-tree $OPEN_{T_i}$ (lines 7 to 9).

Algorithm 13 Parallel *DF* (*PDFS*)

Input: Non-empty attributed graphs $G_1 = (V_1, E_1, \mu_1, v_1)$ and $G_2 = (V_2, E_2, \mu_2, v_2)$ where $V_1 = \{u_1, ..., u_{|v_1|}\}$ and $V_2 = \{v_1, ..., v_{|v_2|}\}$, a parameter N which is the number of the first generated partial edit paths and threads T.
Output: A minimum distance $UBCOST_{shared}$ and a minimum cost edit path (UB_{shared}) from G_1 to G_2 e.g., $UB = \{u_1 \rightarrow v_3,\ u_2 \rightarrow \epsilon\ ,\ \epsilon \rightarrow v_2\}$

```
1:  Generate $C_v$ and $C_e$
2:  ($UB_{shared}$,$UBCOST_{shared}$) ← BP($G_1,G_2$)
3:  $sortedV_1$ ← sortVertices$V_1$($V_1$,$V_2$)
4:  $\mathcal{Q} \leftarrow A^*(N)$
5:  $\{OPEN_T\}$ ← Dispatch-Tasks($\mathcal{Q}$,$T$)
6:  $H \leftarrow -1$, $L \leftarrow -1$   ▷ The indices of the heaviest and the lightest thread, respectively
7:  parallel for $T_i \in T$ do
8:      Call Partial-Branch-and-Bound($OPEN_{T_i}$,$T$,$UBCOST_{shared}$, $UB_{shared}$)
9:  end parallel for
10: Return ($UBCOST_{shared}$,$UB_{shared}$)
```

Algorithm 14 depicts the Partial-Branch-and-Bound that each thread T_i executes on its $OPEN_{T_i}$. This algorithm differs from *DF* in the sense of putting UB_{shared} and $UBCOST_{shared}$ in a place accessible to all threads and thus memory coherence is required, see Section 4.3.5.4. Thus, when thread T_i finds a better $UBCOST_{shared}$ it acquires a lock so as to insure that there is no other thread that modifies $UBCOST_{shared}$ at the same time. This kind of access to the variables in the shared memory is denoted by *atomic access*. Thus, in order to achieve atomic access, locks are introduced (lines 13 and 14), thread T_i acquires the lock preventing other threads from accessing to UB_{shared} and $UBCOST_{shared}$. Once it finishes the update of these shared variables, it releases the lock. When $OPEN_{T_i}$ becomes empty, T_i will be considered as the light thread since its workload ω_i is zero (line 27). In such a case, a search for the heaviest thread is done through *findHeaviestThread* procedure (line 28). Algorithm 15 illustrates the process of finding the heaviest thread via the search for the biggest ω value. Once T_H is found, the load balancing procedure starts (line 33), see Algorithm 12. Load balancing is also atomically accessed. That is, only one thread at a time t can access to it to steal some works from the heaviest thread.

If both ω_H and ω_L have a workload that equals zero, *PDFS* will terminate by outputting UB_{shared} and $UBCOST_{shared}$ as an optimal solution since there is no more edit-path to explore (lines 29 to 31).

4.3.7 Advantages and Drawbacks

PDFS's memory complexity is $|T|.|V_1|.|V_2|$. *PDFS* has some additional time when compared to *DF* because of tasks decomposition, threads initialization, work stealing and load balancing procedures. Thus, if the GM problem is small (i.e., when matching small graphs), *DF* is faster than *PDFS* to find the optimal solution. However, when matching larger graphs, *PDFS* can converge faster to the optimal solution since all threads work in a collaborative way.

PDFS has an advantage over *DF* since it explores different parts of the search tree as it gives each part of it to a thread. Such a fact allows a wider exploration of the search space and thus more easily escapes local optima. Unlike *DF* which may find a local optimum that look good locally but not globally and thus it can hardly escape it specially when the size of the search tree is huge.

PDFS, however, only works on one machine and cannot scale up to work on several machines. Thus in the next section, a distributed *DF* will be presented.

4.4 A Distributed Graph Edit Distance Algorithm

In order to have a massively distributed algorithm, one may need to reduce communication between machines. Thus, in this chapter, a distributed and optimized depth-first algorithm for exact GED computation is proposed without the notion of load balancing. The proposed architecture looks similar to the architecture proposed in [12]. However, instead of simply using MPI as a model, we build MPI over Hadoop MapReduce [145] in order to take advantage of the fault tolerance of Hadoop. So if a thread fails another thread re-executes its tasks without needing to re-execute the program, see Table 4.1. However, Hadoop is a model with restricted communication patterns. Thus, in order to allow threads to send messages and notify the other threads when finding better *UBCOST* and *UB*, a message passing tool is adopted [70]. Moreover, our proposed algorithm uses both parallelism (at the core level) and distributed execution (at the machine level).

As in *PDFS*, the search tree is cleverly pruned thanks to $lb(p)$ and a shared *UBCOST* between all threads. In addition, tree branches, or partial edit paths, are explored in a completely distributed manner to speed up the tree traversal. A search tree decomposition is performed first to fragment the GM problem into smaller ones that are solved in a distributed manner. Our distributed approach, referred to as *D-DF*, consists of a single *job*.

Algorithm 14 Partial-Branch-and-Bound

Input: A local list $OPEN_{T_i}$, the set of threads T, $UBCOST_{shared}$ and UB_{shared}
Output: A minimum distance $UBCOST_{shared}$ and a minimum cost edit path (UB_{shared}) from G_1 to G_2

```
1: $p_{min} \leftarrow \phi$
2: $UBCOST \leftarrow$ read($UBCOST_{shared}$)
3: while true do
4:     if $OPEN_{T_i}$ != $\{\phi\}$ then
5:         $p \leftarrow OPEN_{T_i}$ .popFirst() ▷ Take the first element and remove it from $OPEN_{T_i}$
6:         $Listp \leftarrow$ GenerateChildren($p$)
7:         if $Listp = \{\phi\}$ then
8:             for $v_i \in$ pending$V_2(p)$ do
9:                 $q \leftarrow$ insertion($\epsilon$ , $v_i$)                    ▷ i.e., $\{\epsilon \to v_i\}$
10:                $p$.AddFirst($q$)
11:            end for
12:            acquire_lock()
13:            if $g(p)$ < UB then
14:                $UB_{shared} \leftarrow p$
15:                $UBCOST_{shared} \leftarrow g(p)$
16:            end if
17:            release_lock()
18:        else
19:            $Listp \leftarrow$ SortAscending($Listp$)          ▷ according to $g(p)$+$h(p)$
20:            for $q \in Listp$ do
21:                if $g(q) + h(q) < UB$ then
22:                    $OPEN_{T_i}$.AddFirst($q$)
23:                end if
24:            end for
25:        end if
26:    else
27:        $L \leftarrow i$
28:        $H \leftarrow$ findHeavisetThread($T$)
29:        if $\omega_H = 0$ then
30:            Return ($UBCOST_{shared}$, $UB_{shared}$)
31:        else
32:            acquire_lock()
33:            $OPEN_{T_i} \leftarrow$ Load-Balancing($T_L$,$T_H$)
34:            release_lock()
35:        end if
36:    end if
37: end while
```

Algorithm 15 findHeavisetThread

Input: All threads T

Output: The index of the heaviest thread (H)

```
1: $H \leftarrow -1$                                    ▷ the index is initialized to -1
2: $maxWorkLoad \leftarrow -1$
3: for $T_i \in T$ do
4:     $\omega_i \leftarrow T_H$.calculateWorkLoad()
5:     if $\omega_i > maxWorkLoad$ then
6:         $maxWorkLoad \leftarrow \omega_i$
7:         $H \leftarrow i$
8:     end if
9: end for
10: Return $H$
```

Definition 16 *Job*

A job is a distributed procedure which has one or more *workers (i.e.,* threads that are assigned to tasks). Each worker takes a task or a bunch of tasks to be solved. The whole process is referred to as job.

The master thread, which has a special copy of the program assigns problems to be solved, or so called tasks, to workers (i.e., both master and slave threads). Each of these workers is the responsible of its assigned tasks. Thus, once a worker finishes a task, it sends its result to the master. See Figure 4.5.

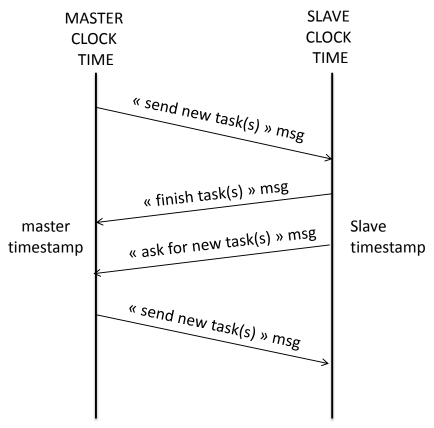


Figure 4.5: Messages sent from master to workers and vice versa

In order to schedule tasks associated to each worker, two scheduling approached can be integrated: Static and Dynamic Scheduling. In Static Scheduling, each worker can be given a certain number of tasks, in this case workers send their results back to the master node once they finish all their associated tasks. On the other hand, Dynamic Scheduling may associate one or more tasks to each worker, associating tasks occurs at run time.

4.4.1 Distributed Depth-first GED Approach

Algorithm 16 represents the three main steps of *D-DF*. First, the master matches G_1 and G_2 using BP and outputs both UB and its $UBCOST$ and the vertices V_1 are then sorted (lines 2 and 3). Second, A^*, see Section 2.3.1.1, is executed and stopped once N partial edit paths are generated (line 5), see Section 2.3.1.1. Afterwards, these partial edit paths ($\mathcal{Q}$) are sorted in ascending order and inserted to *OPEN* (lines 6 to 9). The master also saves UB and its $UBCOST$ in a place/space accessible by all workers W (lines 10 and 11). Each worker also copies C_v and C_e in its local memory (lines 13 to 15). Finally, the master distributes the work (i.e., $\mathcal{Q}$) among workers, each worker takes one edit path from the master at a time (line 17). This step differs from *PDFS* since each thread takes one and only one edit path at a time t instead of having a predefined list of edit paths. The reason is that *D-DF* does not have a load balancing procedure and thus allowing each thread to take one and only one edit path will keep the threads balanced as much as possible. Workers start the exploration of their associated partial edit paths (line 18). If a worker finishes its assigned partial edit path, it sends a message to the master asking for a new edit path (lines 19 to 21). When finishing all the partial edit paths, saved in $File_{OPEN-shared}$, the program outputs UB_{shared} and $UBCOST_{shared}$ as an optimal solution of matching G_1 and G_2 (line 23).

Algorithm 17 demonstrates the function *Partial-Depth-First-GED* that each worker w executes on its assigned partial edit path p. Note that each p is given to an available worker by the master. The procedures of this algorithm are similar to *DF*, the only difference is that $UBCOST$ and UB are saved in a shared space that is accessible by all workers W. These shared variables are referred to as $UBCOST_{shared}$ and UB_{shared}. All the workers read the value stored in $UBCOST_{shared}$ through *read* message (line 3). They also put a watch on $UBCOST_{shared}$ via *Set-Watch* message so as to be awaken when any change happens to its value (line 4). All the workers solve their associated partial edit path. Whenever worker w succeeds in finding a better value of its $UBCOST$, it updates both $UBCOST_{shared}$ and UB_{shared} through *update* messages (lines 14 and 16), the master then sends a notification via *notify-Worker* message to all the other workers (line 17). Workers read the new value, update their local UB and continue solving their problems. Moreover, workers re-establish, or reset, the watch for data changes through *Reset-Watch* message (lines 18 to 21). The update of $UBCOST_{shared}$ is achieved carefully as only one worker can change $UBCOST_{shared}$ at any time t. That is, if two workers want to change $UBCOST_{shared}$ at the same time, one of them is delayed by the master for some milliseconds before entering the critical point. The final answers (i.e., the exact matching and its distance) are found in UB_{shared} and $UBCOST_{shared}$ respectively when all workers finish their associated tasks.

Figure 4.6 illustrates the messages shared between the master and the workers when updating $UBCOST_{shared}$.

4.4.2 Advantages and Drawbacks

D-DF is a fully distributed approach where each worker accomplishes its task without waiting for each other. Moreover, the search tree is cleverly pruned. As soon as any worker

Algorithm 16 Distributed *DF* (*D-DF*)

Input: Non-empty attributed graphs $G_1 = (V_1, E_1, \mu_1, v_1)$ and $G_2 = (V_2, E_2, \mu_2, v_2)$ where $V_1 = \{u_1, ..., u_{|v_1|}\}$ and $V_2 = \{v_1, ..., v_{|v_2|}\}$. A parameter N which is the number of the first generated partial edit paths and workers W.
Output: A minimum distance $UBCOST_{shared}$ and a minimum cost edit path (UB_{shared}) from G_1 to G_2 e.g., $UB = \{u_1 \to v_3,\ u_2 \to \epsilon\ ,\ \epsilon \to v_2\}$

```
1: Generate $C_v$ and $C_e$
2: ($UB$ , $UBCOST$) $\leftarrow$ BP($G_1, G_2$)
3: $sortedV_1 \leftarrow$ sortVertices$V_1$($V_1$,$V_2$)
4: $OPEN \leftarrow \{\phi\}$
5: $OPEN \leftarrow A^*(N)$
6: $OPEN \leftarrow$ SortAscending($OPEN$)
7: for $q \in \mathcal{Q}$ do
8:     $OPEN$.AddFirst($q$);
9: end for
10: $UBCOST_{shared} \leftarrow UBCOST$ {put $UBCOST$ in an accessible place to all workers}
11: $UB_{shared} \leftarrow UB$ {put $UB$ in an accessible place to all workers}
12: $File_{OPEN-shared} \leftarrow OPEN$
13: parallel for $w \in W$ do
14:     $C_{v_w}$,$C_{e_w} \leftarrow$ copy($C_v$,$C_e$)
15: end parallel for
16: parallel for $w \in W$ do
17:     Get-Next-Task: $p \leftarrow File_{OPEN-shared}$.popFirst()
18:                     Call Partial-Depth-First-GED($p$,$W$,$UBCOST_{shared}$, $UB_{shared}$ )
19:     if $File_{OPEN-shared}$ is not empty then
20:         Repeat Get-Next-Task          {Asking the master for a new edit path}
21:     end if
22: end parallel for
23: Return ($UBCOST_{shared}$, $UB_{shared}$).
```

Algorithm 17 Partial-Depth-First-GED

```
Input: An edit path $p$, the set of workers $W$, $UBCOST_{shared}$ and $UB_{shared}$
 1: $OPEN \leftarrow \{\phi\}$, $p_{min} \leftarrow \phi$
 2: $OPEN$.addFirst($p$)
 3: $UBCOST \leftarrow$ read($UBCOST_{shared}$)
 4: Set watch on $UBCOST_{shared}$
 5: while $OPEN$ != $\{\phi\}$ do
 6:     $p \leftarrow OPEN$.popFirst()          ▷ Take first element and remove it from $OPEN$
 7:     $Listp \leftarrow$ GenerateChildren($p$)
 8:     if $Listp = \{\phi\}$ then
 9:         for $v_i \in$ pending$V_2(p)$ do
10:             $q \leftarrow$ insertion($\epsilon$ , $v_i$)          ▷ i.e., $\{\epsilon \rightarrow v_i\}$
11:             $p$.AddFirst($q$)
12:         end for
13:         if $g(p) <$ UB then
14:             $UB \leftarrow p$
15:             $UBCOST_{shared} \leftarrow g(p) + h(p)$
16:             $UB_{shared} \leftarrow p$
17:             MASTER: notify-all-workers $w \in W$
18:             for $w \in W$ do
19:                 $UBCOST \leftarrow$ read($UBCOST_{shared}$)
20:                 Reset watch on $UBCOST_{shared}$
21:             end for
22:         end if
23:     else
24:         $Listp \leftarrow$ SortAscending($Listp$)          ▷ according to $g(p)+h(p)$
25:         for $q \in Listp$ do
26:             if $g(q) + h(q) < UB$ then
27:                 $OPEN$.AddFirst($q$)
28:             end if
29:         end for
30:     end if
31: end while
```

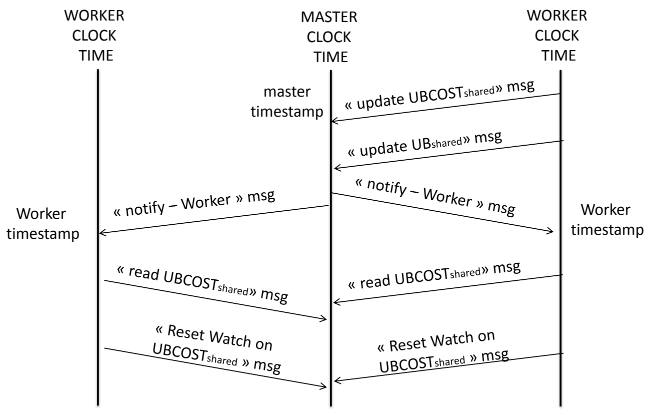


Figure 4.6: Messages shared between the masters and the workers when a worker updates $UBCOST_{shared}$.

finds a better $UBCOST_{shared}$, it sends the new value to the master. Then, the notification to all the other workers is achieved by the master. Finally, all the workers receive the new value. Such operations help in pruning the workers' search trees as fast as possible.

D-DF is a single-job approach and thus the drawback behind such an approach is that some workers might become idle because there is no more edit path in $File_{OPEN-shared}$ while the other ones are still working as they have not finished their assigned edit paths. Figure 4.7 illustrates an example of three workers; two of them were stopped since their *g(p)+h(p)* is greater than *UB* while one of them continues its exploration.

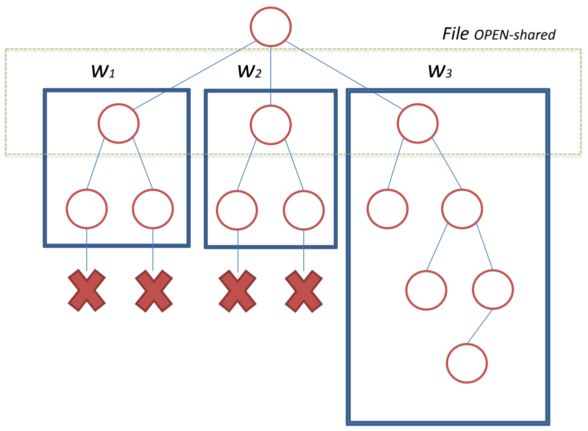


Figure 4.7: Unbalanced search space. Three workers w_1, w_2 and w_3 are given some edit paths. w_1 and w_2 were stopped since their *g(p)+h(p)* is greater than *UB* while the exploration of the search space of w_3 continues

To overcome such a problem, this algorithm can be transformed into multi-jobs, or multi-iterations, algorithm where the master distributes again some edit paths to workers. This can be a future extension of this algorithm.

Chapter 5

New Metrics and Datasets for Performance Evaluation of Graph Edit Distance

Science is not about making predictions or performing experiments. Science is about explaining. Bille Gaede

Contents

Abstract

As explained in Chapter 2, most of the publicly available repositories with associated ground truths are dedicated to evaluating graph classification or exact GM methods. However, to evaluate the accuracy of GED methods, information is required not only at class level but also at matching level even during classification experiments. Thus, the matching correspondences as well as the distance between each pair of graphs have to be directly evaluated. This chapter consists of three parts: First, a graph database repository annotated with graph-level information like graph edit distances and their matching correspondences is proposed and described. Second, a set of metrics to assess GED methods in a more precise way is put forward. Third, the experiments of the GM methods proposed in the thesis are conducted on these new databases with the help of the proposed metrics. A discussion is raised after each experiment. This chapter ends with classification tests and concluding remarks.

5.1 Motivation

As it was presented in the previous chapters, many exact and approximate approaches have been proposed for solving GED [111, 71, 106, 48]. As a first step to evaluate such approaches, one needs to find repositories dedicated to evaluating GM in general or GED in particular.

Graph repositories have been made publicly available for the community [105, 150]. However, to the best of our knowledge, most of these repositories have been put forward for classification and clustering experiments. Moreover, only high-level information has been given to the community such as the class labels of the objects represented by graphs. When evaluating classification, the matching quality is evaluated indirectly through a recognition rate which highly depends on the classifier and does not allow a clear analysis of the matching algorithms. On the other hand, graph-level information has not been provided. For instance, the exact matching between vertices and edges of each pair of graphs, see Table 2.4.

We believe that providing graph-level information is of great interest for understanding the behavior of GED methods in terms of accuracy and speed as a function of graph properties like size and attribute types. Hence, instead of proposing yet a completely new graph database repository, we propose adding graph-level information for well-known and publicly used databases. For that purpose, the GREC, Mutagenicity, Protein, and CMU databases were selected [105, 2]. Added information consists of the best found edit distance for each pair of graphs as well as their vertex-to-vertex and edge-to-edge matching corresponding to the best found distance. This information helps in assessing the feasibility of exact and approximate methods.

Our repository, called Graph Data Repository For Graph Edit Distance (GDR4GED), aims at making a first step towards a GM repository that is able to assess the accuracy of error-tolerant GM methods. All the graph databases and their added information are publicly available[1]. Moreover, we propose novel performance evaluation metrics that aim at comparing a set of GED approaches based on several significant criteria. For instance, deviation and solution optimality. All the provided criteria could be assessed under time and memory constraints.

5.2 GDR4GED: A Graph Set Repository with Graph-Level Ground Truth for Graph Edit Distance Evaluation

This section is divided into four main parts. First, in Section 5.2.1 we direct attention to some important remarks that have to be considered before starting the evaluation of GED methods. Second, in Sections 5.2.2 and 5.2.3 we focus on the description of the selected graph databases, their justification as well as the details of the information added to each database. Third, in Section 5.2.4, the protocol followed for building GDR4GED is depicted. Last, but not least, performance evaluation metrics are defined in Section 5.2.6.

[1]http://www.rfai.li.univ-tours.fr/PublicData/GDR4GED/home.html

5.2.1 Meta Parameters

One difficulty when evaluating GED methods comes from the cost functions used for each graphs database and so their associated meta parameters, see Section 2.1.11.6. For each database, two non-negative meta parameters associated to GED are included (τ_{vertex} and τ_{edge}) where τ_{vertex} denotes a vertex deletion or insertion cost whereas τ_{edge} denotes an edge deletion or insertion costs. A third meta parameter α is integrated to control whether the edit operation cost on the vertices or on the edges is more important. From each GM pair, we derive two notions: a distance between each pair of graphs and vertex-vertex and edge-edge matching or so-called edit sequence. When comparing a set of methods, one has to choose the same database(s), with the same cost functions as well as their associated parameters.

5.2.2 Databases

We aim at evaluating scalability of GED approaches (i.e., when increasing the number of vertices). To this end, we decomposed each database into disjoint subsets, each of which contains graphs that have the same number of vertices. Moreover, we aim at studying the behavior of each algorithm on different types of attributes.

As shown in Table 2.3, most of the public datasets consist of synthetic graphs that are not representative of PR problems concerning GM under noise and distortion. We shed light on the IAM graph repository which is a widely used repository dedicated to a wide spectrum of tasks in pattern recognition [105]. Moreover, it contains graphs of both symbolic and numeric attributes which is not often the case of other datasets. In addition to IAM, we also include another database called CMU [2] houses since it has numeric attributes on vertices and is widely used in GM thanks to its vertex-to vertex ground truth. This database consists of 111 geometric graphs.

5.2.2.1 GREC Database

The GREC database used in our experiments is taken from IAM. GREC consists of a subset of the symbol database underlying the GREC2005 competition [85]. The images of GREC represent symbols from architecture, electronics, and other technical fields. Distortion operators are applied to the original images in order to simulate handwritten symbols. In Figure 5.1 five drawing samples of GED are given. Each of the samples represents a distortion level. Depending on the distortion level, either erosion, dilation, or other morphological operations are applied.

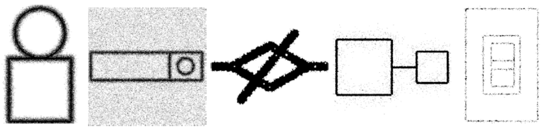


Figure 5.1: GREC: A sample image of each distortion level

The primitive lines of the symbols are then divided into sub-parts. The ending points of these sub-parts are then randomly shifted within a certain distance, maintaining connectivity. Each vertex represents a sub-part of a line and is attributed with its relative length (ratio of the length of the actual line to the length of the longest line in the symbol). Connection points of lines are represented by edges attributed with the angle between the corresponding lines. The dataset consists of 6 classes and 30 instances per class. This dataset is useful as both vertices and edges are labeled with numeric attributes. In addition, it holds graphs whose sizes vary from 5 to 25 vertices.

Database Interest GREC is composed of undirected graphs of rather small size (i.e., up to 25). In addition, continuous attributes on vertices and edges play an important role in the matching process. Such graphs are representative of pattern recognition problems where graphs are involved in a classification stage.

Cost Function The vertices of graphs from GREC dataset are labeled with (x, y) coordinates and a type (ending point, corner, intersection or circle). The same accounts for the edges where two types (line, arc) are employed. The Euclidean cost model is adopted accordingly. That is, for vertex substitutions the type of the involved vertices is compared first. For identically typed vertices, the Euclidean distance is used as vertex substitution cost. In case of non-identical types on the vertices, the substitution cost is set to $2\tau_{vertex}$, which reflects the intuition that vertices with different type label cannot be substituted but have to be deleted and inserted, respectively. For edge substitutions, the dissimilarity of two types is measured with a Dirac function returning 0 if the two types are equal, and $2\tau_{edge}$ otherwise. Both τ_{vertex} and τ_{edge} are non-negative parameters. This dataset consists of 1,100 graphs where graphs are uniformly distributed between 22 symbols. The resulting dataset is split into a training and a validation set of size 286 each, and a test set of size 528.

Additionally to (x, y) coordinates, the vertices of graphs from the GREC database are labeled with a type (ending point, corner, intersection, circle). For edges, two types (line, arc) are employed. The cost functions of vertex and edge substitutions, deletions and insertions are defined as follows:

- $c(u \rightarrow \epsilon) = c(\epsilon \rightarrow v) = \alpha.\tau_{vertex}$
- $c(u_i \rightarrow v_k) = \begin{cases} \alpha.|\mu(u_i) - \mu(v_k)|, & \text{if labels are similar} \\ \alpha.2.\tau_{vertex}, & \text{otherwise} \end{cases}$
- $c(e_{ij} \rightarrow e_{kz}) = 2.(1-\alpha).Dirac(\mu(e_{ij}), \mu(e_{kz}))$
- $c(e_{ij} \rightarrow \epsilon) = c(\epsilon \rightarrow e_{kz}) = (1-\alpha).\tau_{edge}$

where $u_i \in V_1$, $v_k \in V_2$, $e_{ij} \in E_1$ and $e_{kz} \in E2$. μ is a function which returns the attribute(s) of each vertex/edge. $(* \rightarrow \epsilon)$ and $(\epsilon \rightarrow *)$ denote vertex/edge deletion and insertion, respectively. In our experiments, we have set τ_{vertex}, τ_{edge} and α to 90, 15 and 0.5 respectively. As in GREC, these meta parameters' values are taken from the thesis of Riesen [110].

GREC Decomposition We decompose the database into subsets, each of which contains graphs that have the same size. We focus on the following subsets: (GREC5, GREC10, GREC15 and GREC20) aiming at evaluating the two methods when increasing the size of the involved graphs (i.e., the number of vertices). Due to the large number of mappings considered and the exponential complexity of the tested algorithms, we select 10 graphs of the train set of each subset of GREC. The train set is representative of all graph distortions and the selection of only 10 graphs per subset ends up having 100 pairwise comparisons which is significant for such a kind of experiments. Furthermore, we add another subset GREC-mix that contains different number of vertices (i.e., 10 graphs with different number of vertices).

5.2.2.2 Mutagenicity Database

Mutagenicity is the capacity of a chemical or physical agent to cause permanent genetic alterations. The Mutagenicity database was originally prepared by the authors of [72]. Graphs are simply transformed into attributed graphs where vertices represent atoms and are labeled with the number of the corresponding chemical symbol whereas edges represent the covalent bonds and are labeled with the valence (i.e., the combining power of atoms) of the linkage. For simplicity, we denote this database as MUTA.

Database Interest MUTA is representative of GM problems where graphs have only symbolic attributed. MUTA gathers large graphs up to 70 vertices.

Cost Function Edge substitutions are free of cost (i.e., edge cost equals Zero). For vertex substitutions, the dissimilarity of two chemical symbols is calculated with a Dirac function returning 0 if the two symbols are equal, and $2\tau_{vertex}$ otherwise. Mathematically saying, the cost functions of operations on vertices and edges are defined as follows:

- $c(u_i \rightarrow \epsilon) = c(\epsilon \rightarrow v_k) = 2.\alpha.\tau_{vertex}$
- $c(u_i \rightarrow v_k) = \begin{cases} 0, & \text{if they have the same symbols} \\ \alpha.2.\tau_{vertex}, & \text{otherwise} \end{cases}$
- $c(e_{ij} \rightarrow \epsilon) = c(\epsilon \rightarrow e_{kz}) = (1-\alpha).\tau_{edge}$
- $c(e_{ij} \rightarrow e_{kz}) = 0$

where (τ_{vertex}, τ_{edge},α) values of Mutagenicity are set to (11,1.1,0.25). These meta parameters' values are also taken from [110].

Database Decomposition We also shrunk the train set of MUTA by selecting 10 graphs in each of the following subsets (MUTA-10, MUTA-20, MUTA-30 ... MUTA-70). In the MUTA train set, the number of train graphs of MUTA-50, MUTA-60 and MUTA-70 is less than 10. Hence, to complete the subsets, we took some graphs from the test or the validation subsets. We also added another subset, denoted by MUTA-mix, that contains

10 graphs of various number of vertices. As in GREC, 100 comparisons are carried out in each subset.

5.2.2.3 Protein Database

The Protein database was first reported in [72]. The graphs are constructed from the Protein Data Bank and labeled with their corresponding enzyme class labels from the BRENDA enzyme database [68]. The Proteins database consists of six classes (EC1, EC2, EC3, EC4, EC5, EC6), which represent Proteins out of the six enzyme commission top level hierarchy (EC classes). The Proteins are converted into graphs by representing the secondary structure elements of a Protein with vertices and edges of an attributed graph. Vertices are labeled with their type (helix, sheet, or loop) and their amino acid sequence (e.g., TFKEVVRLT). Every vertex is connected with an edge to its three nearest neighbors in space. Edges are labeled with their type and the distance they represent in angstroms.

Database Interest This database contains numeric attributes on each vertex as well as a string sequence that is used to represent the amino acid sequence.

Cost Function For vertex substitutions, the two selected vertices' types of the two graphs g_1 and g_2 are compared first. That is, if two types are identical, string edit distance (SED) [142] is carried out on the amino acid sequences of the vertices to be substituted. Given two amino acid sequences (s_1 and s_2), the corresponding cost of substitutions, insertions, and deletions of symbols in s_1 and s_2 is defined as follows:

$\mathrm{c}(c_i \to c_j) = \mathrm{c}(c_i \to \epsilon) = \mathrm{c}(\epsilon \to c_i) = 1$, $\mathrm{c}(c_i \to c_i) = 0$, s.t. $c_i, c_j \in s_1, s_2$ and $c_i \neq c_j$

The cost functions of matching operations are defined as follows:

- $c(u_i \to \epsilon) = c(\epsilon \to v_k) = \alpha.\tau_{vertex}$
- $c(u_i \to v_k) = \begin{cases} \mathrm{SED}(\mu(u_i), \mu(v_k)), & \text{if labels are similar} \\ \alpha.\tau_{vertex}, & \text{otherwise} \end{cases}$
- $c(e_{kz} \to \epsilon) = c(\epsilon \to e_{kz}) = (1-\alpha).\tau_{edge}$
- $c(e_{ij} \to e_{kz}) = \begin{cases} 0, & \text{if labels are similar} \\ \alpha.\tau_{edge}, & \text{otherwise} \end{cases}$

where (τ_{vertex},τ_{edge},α) are set to (11,1,0.75), taken from [110], and SED is string edit distance [142].

Database Decomposition 10 graphs were selected from Protein-20, Protein-30 and Protein-40. In addition, 10 graphs were picked up from the aforementioned Protein subsets and were put in the mixed database referred to as Protein-mix. 100 pairwise mappings per subset are conducted and integrated in the repository.

5.2.2.4 CMU Database

The CMU model house sequence is made up of a series of images of a toy house that has been captured from different viewpoints. 111 images in total are publicly available. 660 comparisons are carried out.

Graphs Construction A manual identification of corner features, or points, was done to represent vertices on each of the rotated images. Then, the Delaunay triangulation was applied on the corner-features in order to identify edges and finally transform the images into graphs. Vertices are labeled with (x,y) coordinates while edges are labeled with the distance between vertices. Each graph has 30 vertices and matched with graphs representing the same image rotated at 10, 20, 30, 40, 50, 60, 70, 80, and 90 degrees. Since vertex-to-vertex mappings are given, the accuracy of the final matching sequence of any method p can be computed by verifying whether or not the matched vertices are correct. Figure 5.2 demonstrates two CMU houses subjected to rotations. Then, a matching is applied on their generated graphs.

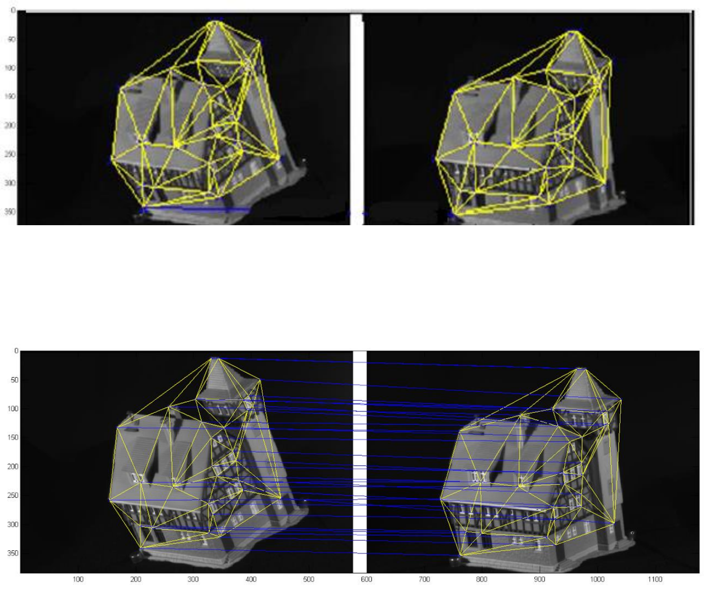


Figure 5.2: Graph partitions on Delaunay triangulation and their matching correspondences

Database Interest Unlike the aforementioned databases, the CMU database has a model house sequence that consists of a series of images. Each of which has been manually annotated by a human being providing its key points (corner features). That is, vertex-to-vertex matching (i.e., graph-level information) between each pair of graphs has been created by humans. We believe that it would be interesting to see if GED methods are able to provide solutions that can be compared to humans' point of view "whether they are far or close to them".

Cost Function We empirically set (τ_{vertex},α) to (∞,0.5). The cost functions of matching vertices and edges are similar to [155]. We formalize them as follows:

- $c(u \to \epsilon) = c(\epsilon \to v) = \alpha.\tau_{vertex}$
- $c(u \to v) = 0$
- $c(e_{ij} \to \epsilon) = c(\epsilon \to e_{kz}) = (1-\alpha).\mu(e_{kz})$
- $c(e_{ij} \to e_{kz}) = (1-\alpha).|\mu(e_{ij}) - \mu(e_{kz})|$

Table 5.1 summarizes the characteristics of all the selected datasets as well as the meta parameters, needed by GED methods, that have been associated to them.

Dataset	**GREC**	**Mutagenicity**	**Protein**	**CMU houses**
Size	4337	1100	600	111
Vertex labels	x,y coordinates	Chemical symbol	Type and amino acid sequence	x,y coordinates
Edge labels	Line type	Valence	Type and length	Distance between points
$\overline{vertices}$	11.5	30.3	32.6	30
$\overline{edges}$	12.2	30.8	62.1	79
Max vertices	25	71	40	30
Max edges	30	112	146	79
$(\tau_{vertex}, \tau_{edge}, \alpha)$	(90,15,0.5)	(11,1.1,0.25)	(11,1,0.75)	$(\tau_{vertex},\alpha) = (\infty,0.5)$

Table 5.1: The characteristics of GREC, Mutagenicity, Protein and CMU houses datasets

We are aware of the necessity of having a graphs database with larger graphs. Meanwhile, GDR4GED does not have such a database. However, such databases will be added soon and will be made publicly available. To overcome this limitation, experiments on synthetic graphs of different sizes have been conducted in the thesis, see Section 5.3.8.

5.2.3 Added Graph-Level Information

For each graphs pair (G_i, G_j), in addition to the initial content of the databases the following graph-level information is provided:

- The optimal or sub-optimal solution provided by the most *accurate* GED method for $d(G_i, G_j)$. Since GED is a minimization problem, the most accurate method is the one whose distance is the smallest among all the other methods.
- The name of the most accurate GED method.
- The solution status (i.e., optimal or sub-optimal).
- The edit path sequence corresponding to the best solution found so far.

Table 5.2 illustrates an example of two graphs taken from GREC-5 and their added information in GDR4GED. All these information are available as csv files in our website.

G_1 **Name**	G_2 **Name**	**Method**	**Distance**	**Optimal**	**Matching**
image3_23	image3_25	$BS-100$	135.178	false	*Vertex:*0 $\rightarrow$ 0=37.476/ *Vertex:*1 $\rightarrow$ 1=6.519/ *Vertex:*2 $\rightarrow$ 2= 32.070/ *Vertex:*4 $\rightarrow$ 4=34.409/ *Vertex:*3 $\rightarrow$ 3=24.703/ *Edge:*2 $\leftrightarrow$ 3 $\rightarrow$ 2 $\leftrightarrow$ 3 =0.0/ *Edge:*0 $\leftrightarrow$ 4 $\rightarrow$ 0 $\leftrightarrow$ 4=0.0

Table 5.2: Graph-level information (taken from the file GREC5-lowlevelinfo.csv)

5.2.4 Graph Edit Distance Methods for Building the Ground Truth

In order to build the GED ground truth included in GDR4GED, a variety of methods were selected. From the related works, we chose three exact methods and four approximate methods. Moreover, both *ADF* and *D-DF* were included. Of course, it is worth clarifying that there are other papers that have been recently published in the scope of GED, outstanding with respect to their scientific contributions. However, because they are recent, we have not been able to be exhaustive in the experiments.

5.2.4.1 Exact Methods

On the exact method side, first, A* algorithm applied to GED problem [111] is a foundation work. It is the most well-known exact method and it is often used to evaluate the accuracy of approximate methods. Second, *ADF* is also a depth-first GED that beats A^* in terms of run time and precision as seen in the previous chapters. Last but not least, *D-DF* was integrated in the experiments since it is a distributed algorithm.

Table 5.3 shows the five exact algorithms included in the tests:

Acronym	**Reference**	**Description of the method**
A^*	[106]	A^* algorithm, $h(p) = lb2$
ADF	this thesis	Depth-first algorithm
D-DF	this thesis	Distributed GED depth-first algorithm

Table 5.3: Exact methods used to build the ground truth of GDR4GED

5.2.4.2 Approximate Methods

On the approximate method side, we can distinguish three families of methods, tree-based methods, assignment-based methods and set-based methods. For the tree-based methods, the truncated version of A* (i.e., *BS*) was chosen. This method is known to be one of the most accurate heuristic from the literature. Among the assignment-based methods, we selected *BP*. In [106], authors demonstrated that *BP* is a good compromise between speed and accuracy. Finally, we picked a recent set-based method. An approach based on the Hausdorff matching has been recently proposed in [49]. All these methods cover a good range of GED solvers and return a vertex-to-vertex matching as well as a distance between two graphs G_1 and G_2 except the Hausdorff matching which only returns a distance between two graphs. The approximate methods that are included in the experiments are depicted in Table 5.4.

Acronym	**Reference**	**Type**	**Description of the method**	**Parameter(s)**
BP	[106]	Upper bound	the Bipartite GM *BP*	∅
BS-x	[90]	Upper bound	Beam search approach of A^* along with a bipartite heuristic.	max. number of open paths $x \in \mathbb{N}^*$
H	[48]	Lower bound	Modified Hausdorff distance applied to graphs	∅

Table 5.4: Approximate methods used to build the ground truth of GDR4GED

5.2.4.3 Experimental Settings

To build the ground truth of GDR4GED, the experiments were conducted on a 5-node cluster of machines running JAVA Runtime Environment 1.7 and Hadoop MapReduce version 1.4.0. Each node contains a 4-core Intel i7 processor 3.07GHz, 8GB memory and one hard drive with 380GB capacity. Hadoop was allocated 20 workers (4 workers per machine), each with a maximum JVM memory size of 1GB. For sequential algorithms, evaluations are conducted on one node.

5.2.4.4 Evaluation Protocol

As GM methods have high complexity, it is sometimes difficult to have enough resources and time to run many methods on large data sets, to wait for a final answer given by each method and to get good quality statistics. Thus, as we have proposed an Anytime GM method, it seems also interesting for us to evaluate the resources and time needed by each method to output a solution. To obtain this kind of information about methods, we propose analyzing the behavior of the different methods under a time constraint (C_T) as well as a memory constraint (C_M). In the experiments, C_T is set to 300 seconds while C_M is set to 1GB.

5.2.5 Availability of GDR4GED

In the previous sections of this chapter, we have proposed a new repository called GDR4GED. This repository is made publicly available [2] including the additional graph-level annotation (i.e., the best solution found by a GED method) for each pair of graphs. For the reason of testing the scalability of GED methods, we divided GREC, MUTA, Protein and CMU into subsets that can also be found on the website.

New performance evaluation metrics have been put forward to assess GED methods. Due to the high complexity of GED methods, we propose evaluating them under time and memory constraints.

[2]http://www.rfai.li.univ-tours.fr/PublicData/GDR4GED/home.html

5.2.6 Performance Evaluation Metrics

In this section, we define and put forward the new metrics used to evaluate any GED method that can be used under both time and memory constraints (i.e., C_T and C_M). We divide the metrics into two groups depending on whether a metric separately evaluates each pair of graphs or takes all the pairs of graphs together.

5.2.6.1 Metrics for each pair of Graphs

Deviation: The error committed by each method p over the reference distances is a very important characteristic to evaluate GM methods. For each pair of graphs matched by method p, we provide the following deviation measure:

$$dev(G_i,G_j)^p = \frac{|d(G_i,G_j)^p - GT_{G_i,G_j}|}{GT_{G_i,G_j}},\ \forall(i,j) \in [\![1,m]\!]^2, \forall p \in \mathcal{P} \tag{5.1}$$

where m is the number of graphs. $d(G_i,G_j)^p$ is the distance obtained when matching G_i and G_j using method p while GT_{G_i,G_j} corresponds to the ground truth provided in our csv files. GT_{G_i,G_j} represents the best distance among all methods p for matching graphs G_i and G_j. As all comparisons are evaluated under C_T, GT_{G_i,G_j} does not necessarily have to be the optimal distance. In other words, GT_{G_i,G_j} represents the best solution found under C_T. This solution could be optimal when C_T is reasonable to solve the given matching problem. For each method p, the mean $dev(G_i,G_j)^p$ is computed. Note that the less the deviation the more precise the algorithm. In other words, 0% refers to the best case while 100% refers to the extreme case.

Matching Dissimilarity: Let EP refer to vertex-to-vertex mappings between G_1 into G_2. We aim at finding how dissimilar are two EPs (i.e., EP_p and EP_q) that correspond to matching G_i and G_j using methods p and q. Thus, the idea here is to see how far an EP_i obtained when matching a pair of graphs from the best EP that is saved in the GDR4GED repository. To this objective, we count the differences between the two solutions, we exclude edge correspondences in order to only concentrate on vertex correspondences since edge mappings/correspondences can be deduced by their adjacent vertices, see Figure 2.8.

Mathematically saying, the distance between EP_p and EP_q is defined as:

$$\mathrm{d}(EP_p,EP_q) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}\delta(EP_p(i),EP_q(j))}{\sum_{i=1}^{n}\sum_{j=1}^{m}1} \tag{5.2}$$

where $n = |EP_p|$, $m = |EP_q|$ and $\delta(EP_p(i),EP_q(j))$ is the well-known Kronecker Delta function [148]:

$$\delta(EP_p(i),EP_q(j)) = \begin{cases} 0, & \text{if } EP_p(i) = EP_q(j) \\ 1, & \text{if } EP_p(i) \neq EP_q(j) \end{cases} \tag{5.3}$$

Note that even if two methods have a deviation that is similar, their matching can be different. Figure 5.3 represents two solutions whose distances are close, however their matching can be different since x_1 belongs to a different place in the search space.

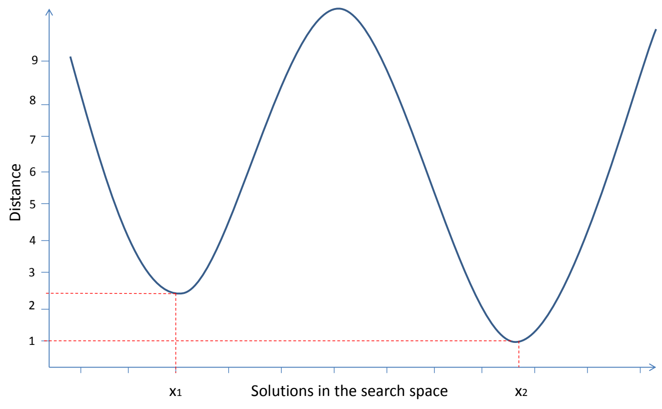


Figure 5.3: Two local optima x_1 and x_2. $f(x_1)$ is close to $f(x_2)$ however the matching of both of them is different since they belong to different parts in the search space

Similar to the deviation metric, the less the matching dissimilarity the better precise the algorithm. In other words, 0% refers to the best case while 100% refers to the extreme case when the matching of an algorithm differs completely from the best matching that is saved in the ground truth GT.

Best Found Solutions: We count the number of times the best known solution is found by method p. This number indicates that method p was able to find the best known solution, not necessarily the optimal one. The solution is supposed to be the best one when compared to all the involved methods. Note that the more best solutions found by method p, the better the method.

5.2.6.2 Metrics for a full Dataset

Number of Explored Nodes: We propose measuring the number of explored nodes in the tree search for each comparison $d(G_i, G_j)$. This number represents the number of moves in the search tree needed to obtain the best found solution. The mean number of explored nodes in the tree search is calculated as follows:

$$\overline{\#explored\_nodes}_p = \frac{1}{m \times m} \sum_{i=1}^{m} \sum_{j=1}^{m} expnd(d(G_i, G_j)^p) \tag{5.4}$$

where $m \times m$ is the total number of comparisons per subset, m is the number of graphs to be matched and $expnd(d(G_i, G_j)^p)$ is the number of explored nodes obtained when matching graphs G_i and G_j of subset s using method p. Note that p has to be a tree-search based algorithm (e.g., A^* [111] and BS-x [90]).

Number of Unfeasible Solutions: For each method p, we measure the number of times method m was not able to find any EP. This case can happen when C_T or C_M is violated before finding a complete solution. (i.e., when there is only incomplete matching correspondences). Lower bound methods [48] always give unfeasible solutions as they only output distances without matching sequences.

Number of Time-Out and Out-Of-Memory Cases: For each subset, we count the number of times method p violates C_T and C_M respectively.

Run time: We measure the overall time in milliseconds (ms), for each GED computation. The mean run time is calculated per subset s and for each method p.

Time-Deviation Scores: To sum up advantages and drawbacks of each method p, a projection of p on a two-dimensional space ($\mathbb{R}^2$) is achieved by using *speed-score* and *deviation-score* features defined in equations 5.7 and 5.8 where speed and deviation are two concurrent criteria to be minimized. First, for each database, the mean deviation and the mean time is derived as follows:

$$\overline{dev_k^p} = \frac{1}{m \times m} \sum_{i=1}^{m} \sum_{j=1}^{m} dev(g_i, g_j)^p \quad \forall p \in P \quad \forall k \in \#subsets \tag{5.5}$$

$$\overline{time_k^p} = \frac{1}{m \times m} \sum_{i=1}^{m} \sum_{j=1}^{m} time(G_i, G_j)^p \; and \; (i,j) \in [\![1, m]\!]^2 \quad \forall k \in \#subsets \tag{5.6}$$

where $dev(G_i, G_j)$ is the deviation of each $d(G_i, G_j)$ and $time(G_i, G_j)$ is the run time of each $d(G_i, G_j)$. To obtain comparable results between databases, mean deviations and times are normalized between 0 and 1 as follows:

$$\overline{deviation\_score^p} = \frac{1}{\#subsets} \sum_{i=1}^{\#subsets} \frac{\overline{dev_i^p}}{max\_dev_i} \tag{5.7}$$

$$\overline{speed\_score^p} = \frac{1}{\#subsets} \sum_{i=1}^{\#subsets} \frac{\overline{time_i^p}}{max\_time_i} \tag{5.8}$$

where $max\_dev_i = \max(\overline{dev_i^p})$ and $max\_time_i = \max(\overline{time_i^p})$ $\forall p \in \mathcal{P}$

5.3 Evaluations of the Proposed Methods

In the next sections we will evaluate the methods proposed in the thesis against the state-of-the-art methods using the metrics proposed in the repository. It is worth clarifying that the ground truth GT_{G_i,G_j} was built using the methods mentioned in Section 5.2.4. In our experiments, we compare our methods to some methods that were not used to build GT_{G_i,G_j}. For this reason, we upgraded GT_{G_i,G_j} taking the results of these methods into account. As a short-term perspective, we intend to upgrade the publicly available version of GDR4GED considering all the methods included in our experiments.

5.3.1 Studied Methods

We compare *ADF*, *PDFS* and *D-DF* to six other graph edit distance algorithms from the literature. From the related works, we chose three exact methods and four approximate methods.

5.3.1.1 Exact Methods

On the exact method side, first, A* algorithm applied to GED problem [111] is a foundation work. It is the most well-known exact method and it is often used to evaluate the accuracy of approximate methods. In addition, a binary linear programming formulation of GED published in [71] is also added. This formulation is implemented and solved using CPLEX-12 mathematical solver. Last but not least, a naive parallel *PDFS*, referred to as *naive-PDFS*, is implemented and compared to *PDFS*. The basis of *naive-PDFS* is similar to *PDFS*. However, *naive-PDFS* does not include neither the assignment phase (see Section 4.3.5.1) nor the load balancing phase (see Section 4.3.5.3). Instead of the assignment phase, a random assignment is applied where T_1 takes the first N/T tree nodes, T_2 takes the second N/T tree nodes and so on, where N is the number of the edit paths generated by A^* and $T$are the threads integrated in solving the GED problem. *naive-PDFS* does not include a balancing strategy which means that if a thread T_i finished its assigned nodes, it would be idle during the rest of the execution of *naive-PDFS*.

Table 5.5 shows all the exact algorithms included in the tests:

Acronym	**Reference**	**Description of the method**
A^*	[106]	A^* algorithm, $h(p) = lb2$
JHBLP	[71]	the Binary linear programming formulation of GED proposed by Justice and Hero
ADF	this thesis	Anytime depth-first algorithm
naive-PDFS	this thesis	Naive parallel depth-first algorithm
PDFS	this thesis	Parallel depth-first algorithm with static and dynamic load balancing techniques
D-DF	this thesis	Distributed depth-first algorithm

Table 5.5: Notations corresponded to each exact GED method

5.3.1.2 Approximate Methods

As for the approximate GED methods, we have included all the methods mentioned in Table 5.4. In addition, *FBP* which is a fast version of *BP* is also added [121]. Note that the Degree Centrality (λ_i^{deg}) has been integrated in both *BP* and *FBP*, see Section 2.3.1.2. Moreover, in our implemented version of *FBP*, the three restrictions on the edit costs were not included. Since H is a lower bound of GED, in the ideal case, it should be compared to other methods only when the optimal solution is found. Moreover, unlike other methods, H is unable to output a matching. Table 5.6 summarizes all the included approximate methods.

Acronym	**Reference**	**Type**	**Description of the method**	**Parameter(s)**
BP	[106]	Upper bound	the Bipartite GM *BP*	∅
FBP	[121]	Upper bound	Square Fast *BP*	∅
BS-x	[90]	Upper bound	Beam search approach of A^* along with a bipartite heuristic.	max. number of open paths $x \in \mathbb{N}^*$
H	[48]	Lower bound	Modified Hausdorff distance applied to graphs	∅

Table 5.6: Notations corresponded to each approximate GED method

5.3.2 Environment

PDFS and *naive-PDFS* are implemented using Java threads. The evaluation of both algorithms are conducted on a 24-core Intel i5 processor 2.10GHz, 16GB memory. For sequential algorithms, evaluations are conducted on one core.

Since *D-DF* is a distributed algorithm, it has a different environment. Thus, the evaluation of *D-DF* is conducted on a 5-node cluster of machines running Hadoop MapReduce version 1.4.0. Each node contains a 4-core Intel i7 processor 3.07GHz, 8GB memory and one hard drive with 380GB capacity. Hadoop was allocated 20 workers (4 workers per machine), each with a maximum JVM memory size of 1GB.

5.3.3 Parameters Selection for the Proposed Methods

This study aims at understanding the impacts of parameters on our algorithms. For the sake of clarity and for better synthesis, parameters are only studied on the GREC dataset. However, for the rest of the experiments in this chapter, all the included methods are evaluated on all datasets.

5.3.3.1 Lower Bounds

A^*, *ADF* and the extensions of *ADF* (i.e., *PDFS*, *naive-PDFS* and *D-DF*) have lb as a parameter. To solve the problem of estimating $h(p)$ and so $lb(p)$ for the costs from the current node p to a leaf node, one can map the unprocessed vertices and edges of graph G_1 to the unprocessed vertices and edges of graph G_2 such that the resulting costs are minimal. This mapping should be done in a faster way than the exact computation and should return a good approximation of the true future cost. Note that the smaller the difference between $h(p)$ and the real future cost, the fewer nodes will be expanded by *A**. In this part of the section, three $h(p)$s and so $lb(p)$s are discussed. Two of them (i.e., $lb2$ and $lb3$) are based on well-known upper bound and lower bound GED algorithms, respectively.

***lb1*:** In the simplest scenario of an A^* algorithm, the estimation function $h(p)$ of the future costs for the current node p is always set to zero i.e., $h(p) = 0$.

***lb2*:** The idea is to reformulate the assignment algorithm as a problem of finding an exact matching in a complete bipartite GM on the unmapped vertices and edges yet to estimate the future costs. Let p be a node at a given position in the search tree, and let the number of unprocessed vertices of the first graph G_1 and the second graph G_2 be n_1 and n_2, respectively. To obtain a lower bound of the exact edit cost, we accumulate the costs of the $min\{n_1, n_2\}$ least expensive of these vertex substitutions, the costs of $max\{0, n_1 - n_2\}$ vertex deletions and $max\{0, n_2 - n_1\}$ vertex insertions. Any of the selected substitutions that is more expensive than a deletion followed by an insertion operation is replaced by the latter. This is performed by an assignment algorithm based on *BP*, see Section 2.3.1.2. The complexity of such a method is $O(max\{n_1, n_2\}^3)$. The unprocessed edges of both graphs are handled analogously. Obviously, this procedure allows multiple substitutions involving the same vertex or edge and, therefore, it possibly represents an invalid way to edit the remaining part of G_1 into the remaining part of G_2. However, the estimated cost certainly constitutes a lower bound of the exact cost.

***lb3*:** This lower bound is inspired by the modified Hausdorff Distance [48], see Section 2.3.1.2. The idea here is to use the modified Hausdorff Distance as a heuristic function that estimates the future costs while applying it on the remaining or unprocessed vertices and edges of G_1 and G_2. The complexity of this lower bound is $O((n_1 + n_2)^2)$.

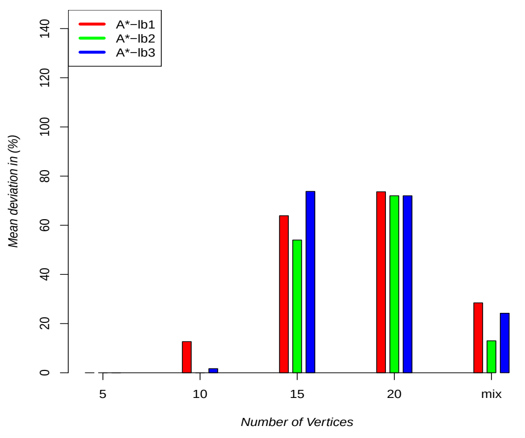
Table 5.7 summarizes these lower bounds:

Acronym	**Description of the method**
lb1	No lower bound (i.e., $h(p) = 0$).
lb2	Vertex assignment and edge assignments in a separated manner, each of which applies *BP*.
lb3	Modified Hausdorff GED applied on remaining vertices and edges.

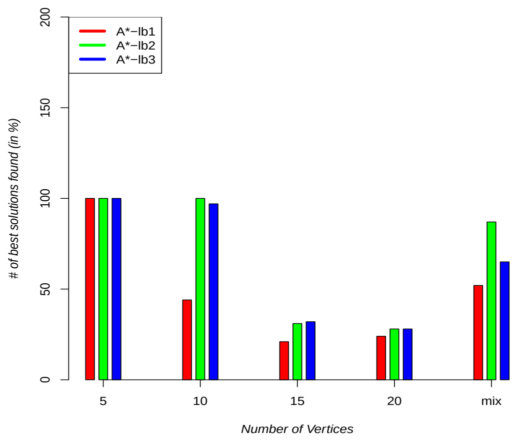
Table 5.7: Lower bounds' notations

Since *PDFS* and *D-DF* are extensions of *ADF*, *ADF* is the only method that is integrated in the *lb* study. Different variants of A^* and *ADF* were evaluated, each of which has *lb1*, *lb2* or *lb3*. In this part of the experiments, C_T is set to 300 seconds while C_M is set to 1GB.

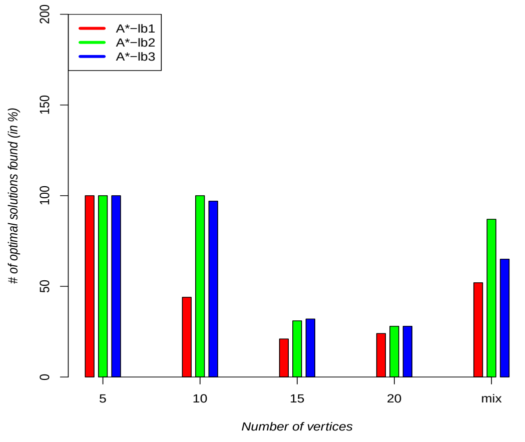
Results and Discussion Figures 5.4 and 5.5 show the effect of *lb1*, *lb2* and *lb3* on A^* and *ADF*, respectively. Results showed that *lb2* is the best lower bound for both A^* and *ADF* as it had the least deviation and the maximum number of best found solutions on all subsets. Moreover, the choice of *lb2* made both A^* and *ADF* run faster. Figures 5.4(f) and 5.5(f) empirically demonstrate that *lb2* represented the minimum run time and deviation scores. Indeed, having an efficient lower bound helps in discarding exploring unfruitful nodes that do not lead to a better, or an improved, solution. One can see such a fact in Figures 5.4(c) and 5.5(c). Moreover, we can clearly see that the simplest scenario *lb1* explored mores nodes when compared to *lb2* and *lb3* as it did not estimate the remaining cost of each node. As a result, in the rest of the paper *lb2* is used as a lower bound of each of A^*, *ADF*, *PDFS* and *D-DF*.



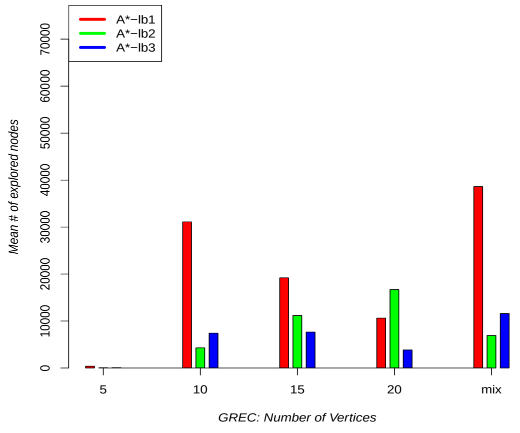
(a) Deviation



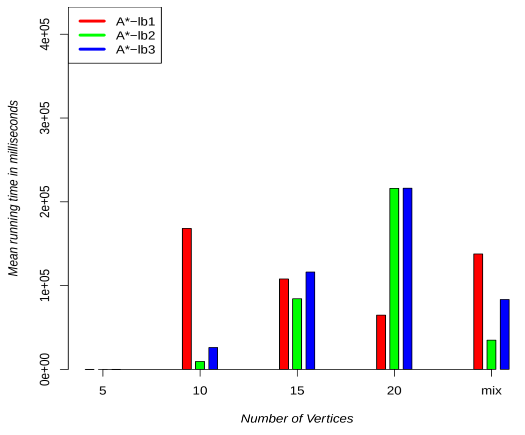
(b) Number of best found solutions



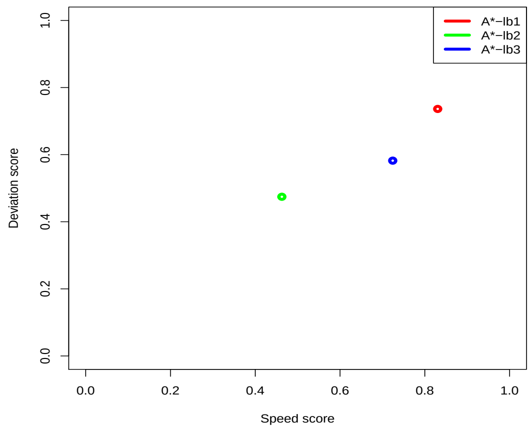
(c) Number of optimal solutions



(d) Number of explored nodes

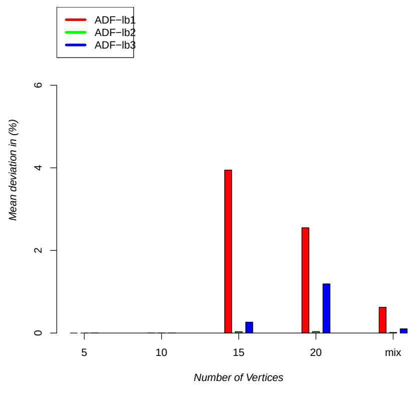


(e) Average run time

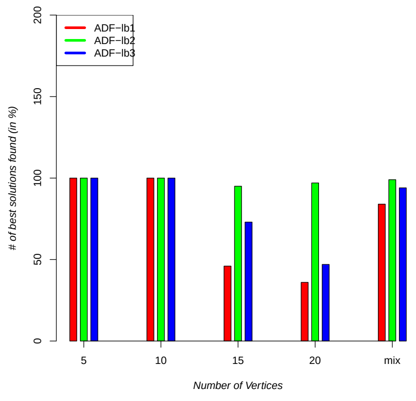


(f) Time-Deviation scores

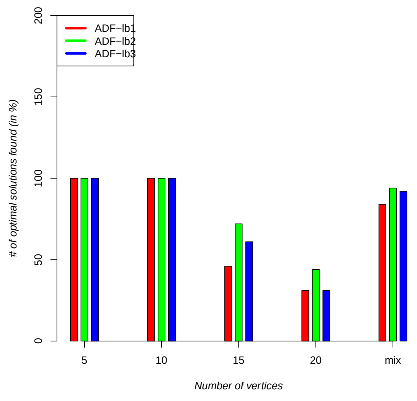
Figure 5.4: Lower bound study on A^*



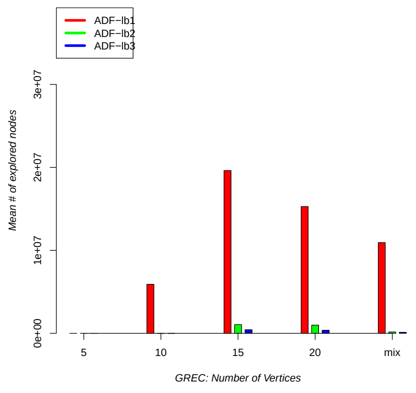
(a) Deviation



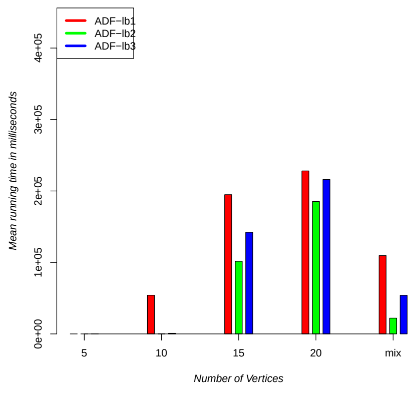
(b) Number of best found solutions



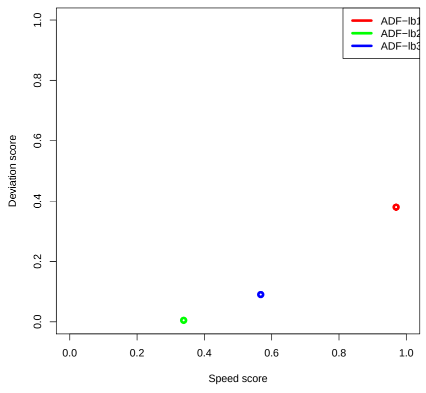
(c) Number of optimal solutions



(d) Number of explored nodes



(e) Average run time



(f) Time-Deviation scores

Figure 5.5: Lower bound study on *ADF*

5.3.3.2 PDFS' Parameters

In this section, parameters of both *naive-PDFS* and *PDFS* are studied. Finally a choice between *naive-PDFS* and *PDFS* is made.

Number of Threads We study the effect of increasing the number of threads T on both the accuracy and speed of *PDFS* as well as *naive-PDFS*. T is varied from 2 to 128 threads.

Figure A.1 displays the effect of the number of threads T on the performance of *PDFS*. One may notice that increasing the number of threads resulted in increasing the chance to find a better solution (see Figure A.1(b)), more optimal solutions (see Figure A.1(c)) a smaller deviation (see FigureA.1(a)) as we explored more nodes in a parallel manner (see Figure A.1(d)). Thus, the overall run time decreased (see Figure A.1(f)) and the number of time-outs decreased (see Figure A.1(e)). Since the machine on which we run this test has a 24-core processor, there is a threshold when increasing the number of threads. For example, on 128 threads the deviation became bigger and the number of time-outs increased. On a 24-core machine, 32 and 64 threads had got the best results. Note that increasing the number of threads also increased the load balancing, one can see that through Figure A.1(g). In the rest of the experiments, the number of edit paths will be set to 100 while the number of threads will be set to 64.

For *naive-PDFS*, the same experiment was conducted. At the end, the number of threads was set to 128.

Parameter *N* The effect of several values of N, described in Section 4.3.5.1, is studied. We expect *PDFS* to perform better when increasing N up to a threshold where the accuracy of the algorithm will be degraded under the same C_T. Five values of N are chosen: -1, 100, 250, 500 and 1000, where N=-1 represents the decomposition of the first floor in the search tree with all possible branches, N= 100 and 250 moderately perform load balancing while N=500 and 1000 is the exhaustive case where threads have much less time dedicated to load balancing since each thread will be assigned sufficient number of works before the parallelism starts.

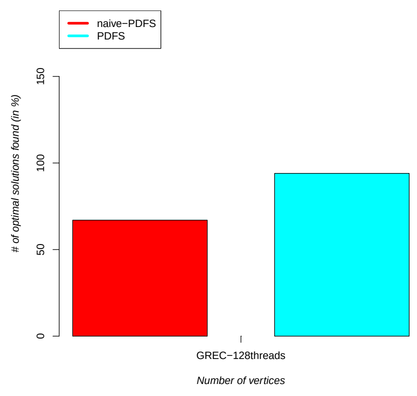
Figure A.2 demonstrates the effect of the number of initial edit paths N on the performance of *PDFS*. On can remark that N equals 100 illustrated the best choice in terms of the number of best found solutions, number of optimal solutions and deviation. One can notice that even though N equals 100 spent much more time on load balancing, it was still 2.3 times more precise than N equals 1000, see Figure A.2 since the latter spent much more time on decomposing the problem into 1000 sub-problems rather than exploring the search space as fast as possible. N equals 1000 represented the least precise results.

For *naive-PDFS*, the same experiment was conducted. In the end, N was also set to 100.

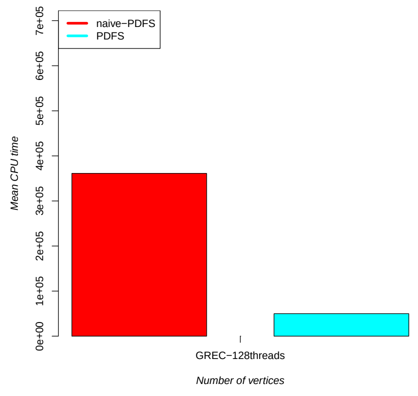
Static and Dynamic Load Balancing Procedures In this section we compare both *PDFS* and *naive-PDFS* in order to choose the best one to be compared to the state-of-the-art methods. Both algorithms were executed on 128 threads so as to be comparable.

Since *PDFS* and *naive-PDFS* are considered as extensions of *ADF*, *lb2* is used as a lower bound of both of them.

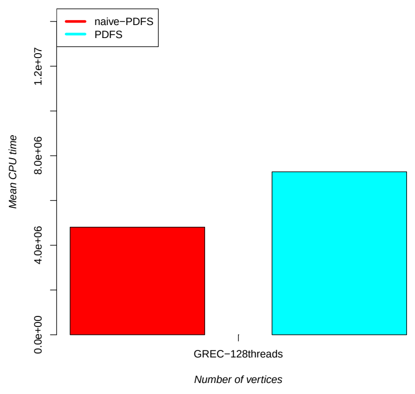
The results in Figure 5.6 show that *PDFS* beats *naive-PDFS* with 26 more optimal solutions. In fact, *PDFS* minimized the variance, stated in equation 4.3, which was not the case of *naive-PDFS*. One can also observe that *PDFS* is fully parallelized where the CPU time was doubled compared to *naive-PDFS*. For all these criteria and for the rest of the chapter, *PDFS* is selected and compared to the other GED methods.



(a) Number of optimal solutions



(b) Variance



(c) CPU time

Figure 5.6: The effect of the number of edit paths on the performance of *PDFS* when executed on the GREC dataset

5.3.3.3 D-DF's Parameters

Number of Machines We study the effect of increasing the number of machines, from 2 to 5 machines where 5 is the maximum number of machines in our cluster, on both accuracy and speed.

Figure B.1 displays the effect of the number of machines (M) on the performance of

D-DF. M was varied from 2 to 5 where 5 represents the maximum number of machines dedicated to the experiments. One may notice that increasing the number of machines resulted in increasing the chance to find a better solution (see Figure B.1(b)), more optimal solutions (see Figure B.1(c)) and smaller deviation (see FigureB.1(a)) as distribution scheme explore more nodes (see Figure B.1(d)). Thus, the overall run time decreased (see Figure B.1(f)) and the number of time-outs decreased too (see Figure B.1(e)). We authors believe that running *D-DF* on a larger cluster (e.g., 100 machines) tremendously decreases the run time and therefore enhances the deviation and increases the number of optimal solutions. Accordingly, M has been set to 5 in order to use all the available resources of the cluster.

Parameter *N* The effect of several values of N, described in Section 3.3, is studied. We expect *D-DF* to perform better when increasing N up to a threshold where the accuracy of the algorithm will be degraded under the same C_T. As in *PDFS*, five values of N are chosen: 20, 100, 250, 500 and 1000, see Section 5.3.3.2.

Figure B.2 demonstrates the effect of the number of initial edit paths N on the performance of *D-DF*. In order to efficiently distribute edit paths among workers, N should ensure that all workers will always be busy and never become idle during the execution of *D-DF*.

When N was equal to 20, some workers finished their work and thus became idle instead of exploring new edit paths. This fact affected the best found solutions and the average CPU time, see Figure B.2. On the other hand, when N was equal to 500 or 1000, partial edit paths became small and vast resulting in an increase of the communication overhead between the master and workers. The communication overhead of 1000 can be visibly seen in Figures B.2(d), B.2(e), B.2(f) and B.2(g) as when the communication overhead increased, the average CPU time decreased, the number of explored nodes decreased and the number of time-out cases incredibly increased. The best choice was illustrated when N was equal to 100 or 250. Both values had similar results in terms of the number of time-out cases, number of optimal solutions and deviation. However, the latter was superior to the former in terms of CPU usage. As a result, in the rest of experiments, N has been set to 250.

5.3.4 Protocol

In this section, the experimental protocol is presented and the objectives of the experiment are described.

The objective of the experiments is four-fold:

- Testing the methods proposed in the thesis as well as the methods of the state-of-the-art under soft and hard time constraints when compared to the other algorithms proposed in the state-of-the-art.
- Studying the trade-off between quality and time of our anytime algorithm *ADF* against the state-of-the-art methods.

- Studying the behavior of some algorithms when increasing the number of vertices and graphs' density.
- Classifying graphs using both exact and approximate GED methods under time and memory constraints.

5.3.4.1 Tests under Soft Time Constraints

The aim of this experiment is to illustrate the error committed by approximate methods over exact methods. In an ideal case, no time constraint should be imposed to reach the optimal solution. Due to the large number of mappings considered and the exponential complexity of the tested algorithms, we allowed a maximum of **300 seconds**. This time constraint is large enough to let the methods search deeply into the solution space and to ensure that many nodes will be explored. The key idea is to reach the optimality, whenever it is possible, or at least to get as close as possible to the *Graal*, the optimal solution. This use case is necessary when it is important to accurately compare images represented by graphs even if the execution time is long.

5.3.4.2 Tests under Hard Time Constraints

The goal is to evaluate the accuracy of exact methods against approximate methods when time matters, that is to say in a context of very limited time. Thus, for each dataset, we select the slowest graph comparison using an approximate method among BP and H as a first time constraint. Unlike BP and H, BS is not included as it is a tree-search which outputs the best solution found so far even on a limited time constraint. Mathematically saying, the time constraint C_T is defined as follows:

$$C_T = \max_{m,i,j}\{time_m(G_i, G_j)\} \tag{5.9}$$

Where $m \in \mathcal{M}_s \ /BS$, $(i, j) \in [\![1, k]\!]^2$ and $time$ is a function returning the run time of method p for a given graph comparison. This way ensures that BP and H could solve any instance. When the time limit is over, the best solution found so far is outputted by BS as well as the exact GED methods. So time and memory limits play a crucial role in our experiments since they impact such methods. Based on the previous equation, we display in Table 5.8 the time limits used for each dataset in the thesis.

Dataset	**GREC**	**MUTA**	**Protein**	**CMU**
C_T (ms)	400	500	400	500

Table 5.8: Time constraints for accuracy evaluation in limited time

C_M is set to 1GB during all the experiments. Among all the aforementioned methods, we expect A^* to violate C_M specially when graphs get larger.

In a small C_T context, there are four key issues that we have taken into account:

- The number of threads in *PDFS* is set to 3. The reason is that since the time constraint is quite small, we did not want to lose time on decomposing the workload among a big number of threads.

- Since C_T is quite small, the *lb* computation time at each tree node is not minor. Consequently, we have removed the lower bound from all of *BS*, A^*, *ADF* and *PDFS* as *lb* is not trivial to compute.
- The number of initial edit paths is limited to the first level of the search tree. In other words, the only nodes that are generates are the substitutions of u_1 with each vertex of V_2 in addition to the deletion of u_1 (i.e., $u_1 \rightarrow \epsilon$).
- *D-DF* cannot be evaluated in a speed test context due to several reasons. First, the time Hadoop takes to initialize its job. Second, the time *D-DF* takes to distribute the first tasks (i.e., initial edit paths) among all the available workers. Finally, the time each worker needs to read its assigned edit path. The sum of all these times approximately equals to 20,000 ms. All these distribution aspects prevent *D-DF* from being evaluated along under small C_T. As *D-DF* is a distributed version of *ADF*, instead, *ADF* can be used for the speed test.

5.3.4.3 Anytime Tests

The objective of the experiments is first to study how anytime methods can provide a trade-off between quality and time according to the difficulty of the matching to realize. For that, we illustrate curves representing the evolution of the quality of provided solutions with the delight of time. Second, we would like to test the performance of anytime GM methods when comparing them to approximate ones under C_T. To this end, two metrics are chosen:

- The deviation metric depicted in Section 5.2.6.
- The setup time needed by the anytime algorithm to output an initial solution (i.e., the first complete solution found when exploring the search tree). Only *ADF* and *ADF-UB* are able to find one or more solutions while exploring the search tree. This time is compared with the time taken by approximate algorithms like *BS1*, *BP* and *FBP* which output one and only one complete solution. From now on, this measured time will be called "*setup time*".

5.3.5 Testing the Proposed Methods under Soft Time Constraints

In this section, we compare the state-of-the-art's methods to our algorithms under a big C_T (i.e., 300 seconds). *JHBLP* was only evaluated on MUTA since it cannot match graphs that have attributes on edges, see Section 2.3.1.1.

5.3.5.1 Results on GREC

Figures C.1(a) and C.1(b) indicate that *PDFS* and *D-DF* had the least deviations (around 0% on all subsets) and the best mappings/matching when compared to the other algorithms. However, *D-DF* was slightly better on GREC-15 and GREC-MIX where the gap between them was 4.5% on GREC-15 and 1.9% on GREC-MIX.

PDFS and *D-DF* outperformed the other methods in terms of the best found solutions, see Figure C.1(c). This can be visibly seen when scaling up to match larger graphs as GREC-15 and GREC-20. *D-DF* found the maximum number of best found solutions, it also defeated *PDFS* with 5 more better solutions.

One also can see the importance of *PDFS* and *D-DF* when matching more complex graphs (e.g., GREC-20), in this case both algorithms outperformed *ADF* in terms of the number of optimal solutions and time-out cases. As depicted in Figure C.1(d), *ADF* outperformed A^*. For instance, on GREC-15 the number of optimal solutions found by *ADF* was 72 while only 46 solutions were found by A^*. On GREC-20, the number of optimal solutions decreased, A^* found 28 optimal solutions while *ADF* found 44 ones. Both *PDFS* and *D-DF* found more optimal solutions when compared to *ADF*. However, the results of both algorithms were quite similar, for instance on GREC-15 *PDFS* won with one more optimal solution while on GREC-20 *D-DF* found one more optimal solution. Thus, the number of time-out cases in both *PDFS* and *D-DF* was the least, see Figure C.2. *PDFS* and *D-DF* explored more nodes than *ADF* since they explore the search tree in parallel and distributed fashions, respectively. *D-DF*, in average, explored 720200 nodes more than *PDFS*, see Figure C.1(f).

Regarding run time, *PDFS* was always faster than *ADF*. However, that was not the case of *D-DF* which was slower than both *ADF* and *PDFS*. This is because of the distribution issues of *D-DF*, see Section 5.3.4.2. Thus, for some less difficult graph pairs, *ADF* took few milliseconds (ms), while *D-DF* took approximately 20,000 ms. Although *BS* and *BP* are approximate, they were able to find solutions with low deviation and low speed scores, as depicted in Figure C.2(d). A^* outputted some unfeasible solutions before violating C_T (on GREC-20) and C_M (on GREC-15 and GREC-MIX), see Figures C.1(e), C.2(a) and C.2(b). Note that H always outputs unfeasible solutions since it is a lower bound algorithm.

Conclusion On the exact methods side, *PDFS* and *D-DF* were the best algorithm in terms of deviation, matching dissimilarity and number of best found solutions. *PDFS* and *ADF* however were faster than *D-DF* due to the distribution issues of *D-DF*. On the approximate methods side, *BS-100* proved to be the most precise one in terms of deviation and matching dissimilarity and among the fastest algorithms. Finally on the exact versus approximate methods side, *PDFS* and *D-DF* were the most precise algorithms. In terms of run time, *BS-10* and *BS-100* were the fastest.

5.3.5.2 Results on MUTA

Figures C.3, C.4, C.5 and C.6 and C.7 illustrate the results of all methods on MUTA when C_T is equal to 300 seconds.

JHBLP was the best in terms of deviation, matching dissimilarity, number of best found solutions, and optimal solutions, see Figures C.3(a), C.3(b), C.4(a) and C.4(b), respectively. The only weakness we could notice regarding *JHBLP* was its high memory consumption when graphs hold 70 vertices, see Figure C.6(b). For this subset, *JHBLP* could not find any solutions for half of the instances, as depicted in C.5(a). In fact, this is due to the complexity continuous relaxation of *JHBLP* is $O(3n^7)$ where n is the number of vertices of the included graphs. We can conclude that *JHBLP* fails to match large graphs. Thus, On MUTA-70, *BS-10* got the best deviation (i.e., 19.91%) while the deviation of *JHBLP* was 50%.

PDFS and *D-DF* visibly outperformed *ADF* in terms of deviation where the average deviation of *ADF*, *PDFS* and *D-DF* was 32.37%, 27.25% and 21.50 %, respectively (see Figure C.3(a)). On the other hand, the average number of optimal solutions of A^*, *ADF*, *PDFS*, *D-DF* and *JH* was 20.5%, 21.6%, 21.8% and 63.1%, respectively, see Figure C.4(b). Thus, on MUTA, *JH* was the best algorithm among the exact GED algorithms.

The number of time-out cases of *D-DF* was less than *ADF*, see subsets MUTA-20 and MUTA-MIX in Figure C.6(b). However, from MUTA-20 to MUTA-70, *D-DF* was slower than both *ADF* and *PDFS* which is due to the time needed by this distributed algorithm for the distribution issues, see Section 5.3.4.2. Even if *D-DF* explored more nodes in a fully distributed manner, see Figure C.5(b), exploring more nodes in a distributed way helps in updating the upper bound, see Figure C.3(a).

The deviation of *FBP* and *BP* was quite high (61% and 64%, respectively). This fact confirms that the more complex the graphs the less accurate the answer achieved by approximate methods. *FBP* and *BP* consider only local, rather than global, edge structure during the optimization process [106] and so when graphs get larger, their solutions become far from the optimal one.

On MUTA-70, despite the outperformance of *D-DF* over *BP*, *H* and *ADF*, it did not outperform *BS* in terms of deviation, as shown in Figure C.7(a). We have argued that the implemented versions of *BS* and *ADF* use *lb2*. The major difference between these algorithms is the search space in addition to the Vertices-Sorting strategy which is adapted in *ADF* and not in *BS*. Since *BP* did not give a good estimation on MUTA, it was also irrelevant when sorting the vertices of g_1 resulting in the exploration of misleading nodes in the search tree. Since the graphs of MUTA are relatively large, backtracking nodes took time. MUTA contains symbolic attributes while *ADF*, *PDFS* and *D-DF* are designed for rich attributed graphs. However, the average deviation of *BS* compared to *PDFS* and *D-DF* was relatively similar. *JH* represented the best trade-off between deviation and run time followed by *BS-100*, see Figure C.7(b).

Conclusion On the exact methods side, *JHBLP* was the best algorithm in terms of deviation, matching dissimilarity, run time, number of best found solutions and number of optimal solutions. That was the case up to the graphs whose number of vertices is 60. On MUTA-70, *JHBLP*, outputted unfeasible solutions due to the complexity of the continuous relaxation of *JHBLP*. Thus, in this case *D-DF* outperformed it. On the approximate methods side, *BS-100* proved to be the most precise one in terms of deviation and matching dissimilarity and was the fastest algorithms. Finally on the exact versus approximate methods side, *JHBLP* was the most precise algorithms and fastest algorithm, followed by *BS-100* on MUTA-70.

5.3.5.3 Results on Protein

Figures C.8 and C.9 point out that *PDFS* and *D-DF* were among the algorithms whose deviation is the minimum. In average, the deviation of both *PDFS* and *D-DF* was 0.28% and 0.755%, respectively. The gap between these methods increases when taking into consideration the matching dissimilarity where in average *PDFS* obtained 38.5% while *D-DF* obtained 41%. One can see that on all subsets of Protein (except Protein-30), the matching dissimilarity of *PDFS* was the lowest. The reason for which *D-DF* got the lowest matching dissimilarity on Protein-30 is that it got 3 more optimal solutions, see Figure C.8(d).

Similar to A^* and H, *BS-100* outputted unfeasible solutions on Protein-40 and Protein-mix and thus *BS-1* beat *BS-100* in terms of deviation on Protein-40. However, the matching dissimilarity of *BS-1* was 57.61% which is far from the best matching dissimilarity obtained by *PDFS* (i.e., 38.87%).

Conclusion One can see that on Protein deviation cannot be taken as a significant evaluation performance metric since most of the algorithms obtained quite low deviation. In terms of deviation, *BS-1*, *BS-10*, *BP*, *ADF*, *PDFS* and *D-DF* obtained low values. However, *BP*, *BS-1* and *BS-10* were the fastest.

5.3.5.4 Results on CMU

Figures C.10 and C.11 illustrate the results under soft constraints ($C_T = 300$ seconds). As seen in Figures C.10(a) and C.10(b), *D-DF* and *PDFS* were the most precise algorithms in terms of deviation and matching dissimilarity. However, *D-DF* was more accurate since its matching dissimilarity was 0.5% while the one of *PDFS* was 1.13%. The third precise algorithm was *ADF* since its deviation was 2.8% and its matching dissimilarity was 5%. The number of best found solutions of *D-DF* was the biggest (i.e. 98%), see Figure C.10(c), followed by *PDFS* whose number of best solutions was 95%.

PDFS was also able to find 20.30% of optimal solutions when compared to *ADF* and A^* which were able to find 18.33% of optimal solutions. *D-DF*, however, beat *PDFS* since its average number of optimal solutions was 22.57%. Although *D-DF* explored much more nodes in the search space, the average time-out was 77.42% while it was 81.66% in both *ADF* and A^* and 79.0% in *PDFS*. This fact highlights the importance of parallel and distributed algorithms. Figure C.11(d) summarizes the previous results. On the approximate methods side, *BS-1* was the best one with a deviation that equals 7% while *BP* and *FBP* were the worst ones with a deviation that equals 27%.

Conclusion By looking at all the previous results, we can see that in general *D-DF* beat *PDFS*. This is theoretically true since *D-DF* is a scalable algorithm and thus its processes are totally independent. On the other hand, threads in *PDFS* share resources which can slow them down. However, practically, on Protein, *PDFS* was performing better than *D-DF*. In fact, since both algorithms are run on different architectures, the number of workers is different (i.e., 128 threads in *PDFS* sharing the same resources versus 20 processes in

D-DF with independent resources). Such a fact cannot make the comparison in this part possible.

5.3.5.5 Focusing on Difficult Graph Edit Distance Instances

As seen in the previous sections, unlike *PDFS*, the run time of *D-DF* is sometimes bigger than *ADF*. This is due to the fact that *D-DF* ultimately needs some pre-processing time before starting the exploration of the search tree in a distributed fashion, see Section 5.3.4.2. Thus, one can conclude that *D-DF* is worthwhile when having more difficult problems. For example matching G_i with G_i is an easy instance that does not need to be distributed. Based on that, in this section, we eliminate the easy GM instances from GREC, MUTA, Protein and CMU datasets. To filter these databases, we run *ADF* on each pair of graphs and stop it after 300 seconds. If an optimal solution could not be found within 300 seconds, then the GM problem is considered as difficult.

Table 5.9 depicts the number of difficult problems per subset on each of the aforementioned databases.

GREC5		**GREC10**	**GREC15**		**GREC20**	**GREC-mix**	
0		0	28		58	6	
MUTA10	**MUTA20**	**MUTA30**	**MUTA40**	**MUTA50**	**MUTA60**	**MUTA70**	**MUTA-mix**
0	90	90	90	90	90	90	90
Protein20		**Protein30**		**Protein40**		**Protein-mix**	
84		90		90		90	
CMU							
549							

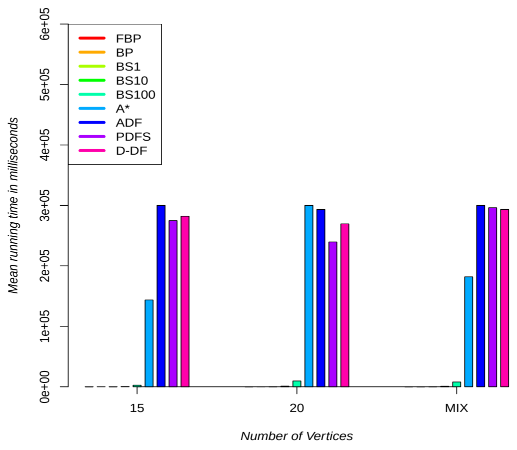
Table 5.9: Number of difficult problems on each dataset

Figure 5.7 shows the run time of each of the methods under time constraints. *D-DF* was always faster or equal to *ADF* (in average it was faster with a percentage that equals to 6% on GREC, 3% both MUTA and CMU, and 1% on CMU. At a first glance, one can think that A^* was faster than both *ADF* and *D-DF*. However, because of its memory bottleneck, as seen in the previous sections, it was unable to output feasible solutions. *BP* was the fastest algorithm where its run time in average was 12.3 ms on GREC, 11 ms on MUTA).

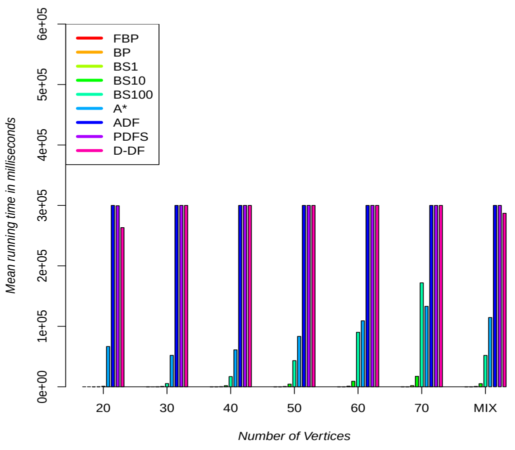
5.3.6 Testing the Proposed Methods under Hard Time Constraints

In these tests, the ground truth of GDR4GED is used for the two metrics (deviation and matching dissimilarity), see Section 5.2.6.1.

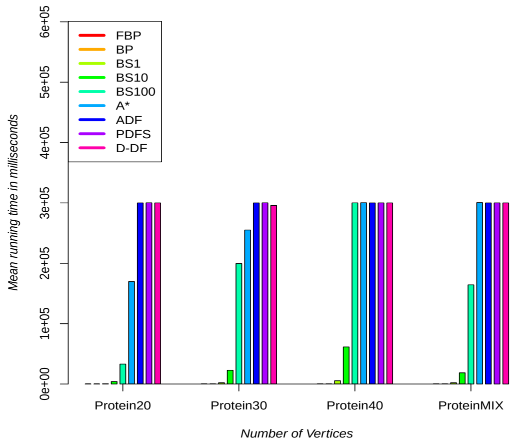
Under hard time constraints, A^* did not have time to output feasible solutions and thus it got the highest average deviation rates (around 35% on GREC, 80% on MUTA, 80% on Protein and 81% on CMU), see Figure D.1. *JHBLP* was also unable to output feasible solutions on each of MUTA-30, MUTA-40, MUTA-50, MUTA-60, MUTA-70 and MUTA-MIX where its solutions and it had a deviation that was equal to 100%. This is due to the setup time taken by mathematical solver to solve the continuous relaxation of the GED problem in polynomial time $\mathcal{O}(n^{3.5})$ with the interior point method [30]. Despite the fact that *PDFS* was among the slowest algorithms, it obtained the least deviation



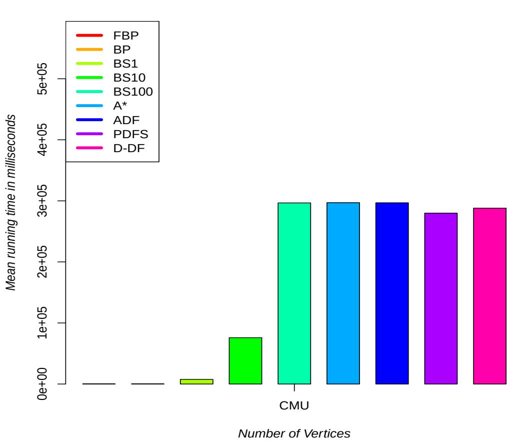
(a) GREC



(b) MUTA



(c) Protein



(d) CMU

Figure 5.7: Difficult problems: Run time (C_T = 300 seconds)

(1.75% on GREC, 2.27% Protein, 13% CMU and 18% on MUTA), see Figure D.1. *BS-100* outputted unfeasible solutions on MUTA-50, MUTA-60, MUTA-70, MUTA-MIX, Protein and CMU due to the hard C_T. *BP* got relatively small average deviation on GREC, Protein and CMU (4.01% on GREC, 2.44% on Protein and 25.56% on CMU), see Figure D.1. However, it had quite high dissimilarity matching, see Figure D.2.

In order to better understand the behavior of *ADF*, *PDFS* and *BP*, Table 5.10 summarizes their average best found solutions, deviation and matching dissimilarity on GREC, MUTA, Protein and CMU. One can see that, for instance, the deviation gap between *BP* and *PDFS* was 2.26% on GREC, 22.37% on MUTA, 0.17% on Protein and 12.56% on CMU. While the deviation gap is relatively small, the matching dissimilarity gap is bigger (8.9% on GREC, 46.58% on MUTA, 7.75% on Protein and 18.37% on CMU). Such a fact reveals that the error made by matching is always higher than the error made by distance.

	Avg # Best Found Solutions			**Avg Deviation**			**Avg Matching Dissimilarity**		
	BP	**DF**	**PDFS**	**BP**	**DF**	**PDFS**	**BP**	**DF**	**PDFS**
GREC	42.8%	84.6%	95.6%	4.016%	2.23%	1.75%	26.5%	18.51%	17.6%
MUTA	15.5%	61.8%	76.12%	40.37%	19.75%	18%	68.7%	32.855%	22.125%
Protein	86%	90%	95.8%	2.44%	2.34%	2.27%	28.5%	25.43%	20.75%
CMU	90.75%	97.20%	99.39%	25.56 %	15.37%	13.0%	37.0454%	20.252%	15.67%

Table 5.10: Speed Test: The average number of best found solutions, deviation and matching dissimilarity of GREC, MUTA, Protein and CMU

One can also see that, on Protein and GREC, *BP* got high number of best found solutions, small deviation and small matching dissimilarity. In fact, in such a hard time constraint, *ADF* and so *PDFS* were not always able to improve the upper bound found by *BP*. Moreover, in the case of Protein, that could be because of the time the Protein's cost function takes to calculate vertex-to-vertex distances preventing *ADF* and *PDFS* from improving *UB* when time matters, see Section 5.2.2.3.

5.3.7 Anytime Tests

5.3.7.1 Environment

The evaluations of anytime tests are conducted on a 8-core Intel R(Xeon) CPU 2.66GHz and 8GB memory. A memory constraint was set to 1GB. The time constraint is varied from 5 milliseconds (ms) to 500 ms on all databases.

5.3.7.2 Databases

In the experiments, three databases have been included (Protein-40, MUTA-70, GREC-20 and CMU). The reason for having chosen these subsets is because they are the biggest ones in the GDR4GED repository.

5.3.7.3 Methods

Table 1 summarizes the methods included in the experiments. In all the aforementioned methods, memory consumption is not exhausted. As it was previously mentioned, the memory complexity of the anytime *ADF* and *DF-UB* algorithms is relatively small thanks to the depth-first search where the number of pending nodes is $|V_1|.|V_2|$ in the worst case. A^* could have been also added to the experiments, however, its memory complexity is exponential and so it will not be able to keep exploring the search tree and thus outputting feasible (i.e., complete) solutions before timeout. The algorithm of [71], described in Section 2.3.1.1, was implemented and solved using CPLEX-12 mathematical solver. This method was also unable to output feasible solutions in 500 ms or less. In fact, this is due to the complexity of continuous relaxation of *JHBLP* is $O(3n^7)$ where n is the number of vertices of the included graphs.

In both *BP* and *FBP*, λ_i^{deg} is used as a node centrality when matching vertices.

Acronym	**Reference**	**Details**
ADF-UB	This thesis	AnyTime GED with an initial UB equals to *BP*, h(p)=0.
ADF	This thesis	AnyTime GED without an initial UB, h(p)=0.
BS-1 and *BS-100*	[98]	Beam Search with OPEN size = 1 and 100 respectively
BP	[106]	The bipartite GM
FBP	[121]	The fast version of *BP*

Table 5.11: Methods included in the experiments

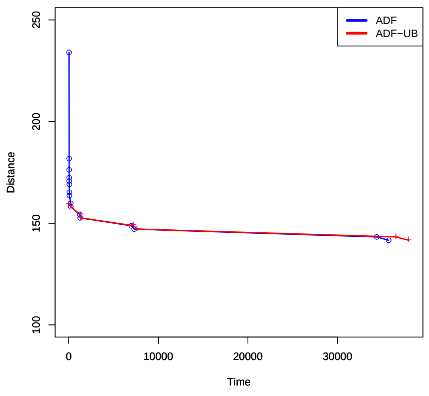
5.3.7.4 Results and Discussions

Figure 5.8 depicts the list of improved solutions (i.e., distances) found by the anytime methods *ADF-UB* and *ADF* on three random pairs of graphs that are taken from GREC-20, MUTA-70 and Protein-40. One can observe that in the first few milliseconds *ADF-UB* does not output any solution while *ADF* succeeds in outputting several solutions, however, with the delight of time both of them reach the same distance. Such a fact reveals the importance of anytime algorithms, since they are able to improve their solutions in a few milliseconds.

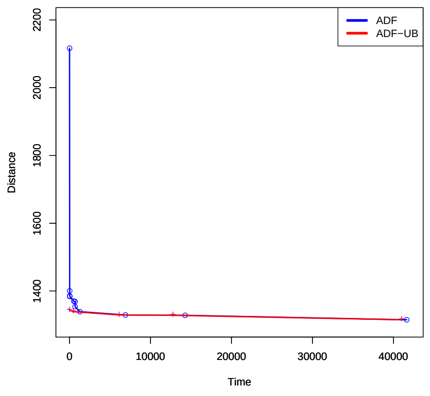
Figure 5.9 illustrates the deviation on Protein-40 when varying the available time up to 40 ms (see Figure 5.9(a)) and 500 ms (see Figure 5.9(b)). One can observe that *ADF* was the fastest method as it outputs solutions just after few milliseconds, followed by *BS1* and *BS100*. Both *FBP* and *BP* solve the linear assignment problem with the help of the Hungarian methods. This fact cannot let them output solutions rapidly for relatively large graphs when time matters. Since *ADF-UB* computes *BP* as a UB so its first solution is highly dependent on *BP*. When we add more time (as illustrated in Figure 5.9(a)), *BP* and *FBP* output satisfied deviations. Unlike the latter methods, *ADF-UB* is an anytime algorithm and so it keeps improving its solutions until we suspend the algorithm.

Figure 5.10 points out the results on CMU. The same remarks as Protein can be seen, however, the deviation of *BP* and *FBP* have a low quality even when increasing time constraint (see Figure 5.10(b)). On the other hand, *ADF-UB* succeeds at improving the deviation when increasing the time constraint. On both Protein and CMU, *BS100*'s deviation is approximately equal to 100%, this is because it did not find feasible solutions before timeout.

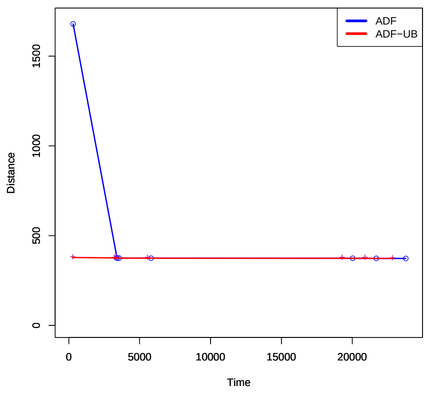
As for MUTA-70, Figure 5.11(a) shows that when time matters, *FBP* was surprisingly faster in outputting solutions, followed by *BP* and then *ADF-UB*. We have argued that: First, MUTA has less dense graphs than Protein's and CMU's graphs where the average $|V|/|E|$ ratio is 30.3/30.8 on MUTA, 32.6/62.1 on Protein and 30/79 on CMU, such that $|V|$ and $|E|$ are the total numbers of vertices and edges, respectively. Second, the edges of MUTA are unattributed, unlike Protein where type and length are provided on attributes. Third, the cost functions of MUTA consumes less time when compared to the ones of Protein. For all these reasons, solving the edges assignment problem on MUTA is faster



(a) MUTA-70



(b) GREC-20



(c) Protein-40

Figure 5.8: Random pairs of graphs taken from different graph databases illustrating the improvement of found solutions with the delight of time

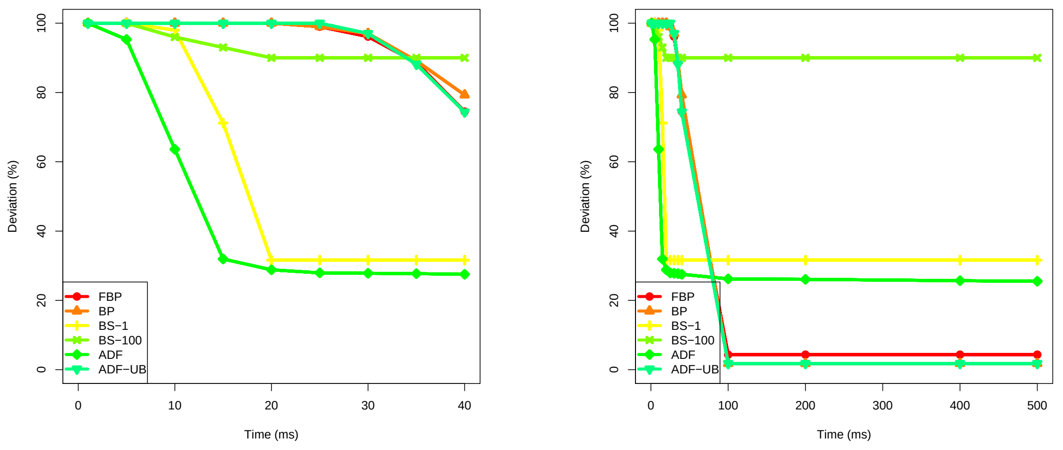


Figure 5.9: Anytime test: Deviation on Protein: Left (up to 40 ms), Right (up to 500 ms)

than Protein and CMU. Thus, *FBP* and *BP* were able to output their solutions faster than *ADF*. Since *FBP* has a reduced complexity when compared to *BP*, it always performs faster. After 27 ms, *ADF* outperformed *BP* and so *ADF-UB*. However, after 70 ms, *ADF-UB* beat it. Since both algorithms are anytime ones, they keep improving their solutions found so far while the other algorithms remain stable, except *BS100* since additional time is beneficial for it.

Figure 5.12 shows the effect of *lb2*, mentioned in Table 5.7, on the performance of *ADF*, *ADF-UB*, *BS1* and *BS100*. Results demonstrated that after 4000 ms, *BS100-LB*, *ADF-LB* and *ADF-UB-LB* had the smallest deviation. Among these algorithms, *ADF-UB-LB* was the most accurate. One can conclude that with more time, the existence of $h(p)$ is important since it helps in pruning tree branches. In all the experiments, the behavior of *BS100* looks similar to A^* which was also unable to output feasible solutions.

The results of GREC are illustrated in Figure 5.13. Similar to MUTA, the $|V|/|E|$ ratio of GREC is 11.5/12.2 and so as MUTA the edge assignment problem is easier than the one of Protein. Thus, as on Protein, *FBP*, *BP* and so *ADF-UB* are faster than *ADF*.

For a better understanding of the performance of anytime GEDs, Table 2 directs attention on the average setup time. We study the average setup time on two databases on which anytime algorithms behaved differently. On Protein, *ADF* has proven to be faster than the approximate algorithms. In average, *ADF* only needed 12.88 ms to output a solution. However, this was not the case on MUTA where *FBP* was the fastest with only 15.60 ms in average.

As seen in the previous experiments, the GED anytime algorithms sometimes outperformed the approximate GED algorithms (such as *BP* and *FBP*). This conclusion brings into question the usual evidences saying that it is impossible to use exact methods of GM in real world applications when matching large graphs.

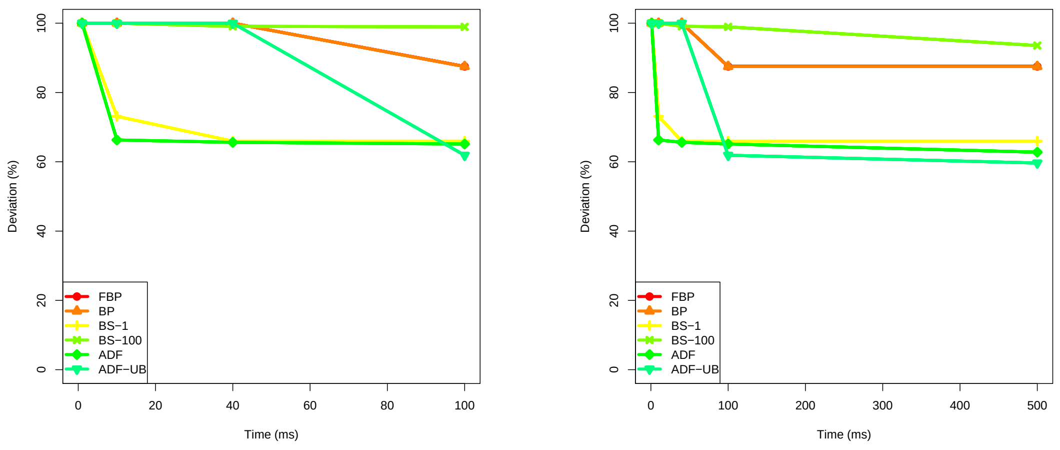


Figure 5.10: Anytime test: Deviation on CMU: Left (up to 100 ms), Right (up to 500 ms)

	Protein				
	FBP	**BP**	**BS1**	**ADF**	**ADF-UB**
Setup Time (ms)	47.60	49.81	12.10	12.88	50.02
Deviation at Setup Time	4.344	1.789	31.597	31.490	1.789
	MUTA				
Setup Time (ms)	15.60	17.55	3152.56	24.70	18.35
Deviation at Setup Time	37.874	42.254	4.169	44.169	42.254

Table 5.12: The average setup time and deviation on Protein and MUTA

5.3.8 Graph Density Tests

In the previous section, we have argued that *FBP* and *BP* are faster when graphs are less dense while *ADF* is faster when graphs are more dense. In order to prove that, a graph database dedicated to graph density is generated. To have different densities, random graphs are created using the Erdos-Renyi model [44]. Figure 5.14 illustrates an example of an Erdos-Renyi graph [3].

Every possible edge is created with the same constant probability. These densities represent the probability of drawing an edge between two arbitrary vertices. For instance, 0.1 indicates that the probability of having an edge between two vertices is 0.1 and so on.

This part of experiments is divided into two parts:

- Density Scalability: For the same number of vertices, different densities are generated aiming at studying the behavior of GED methods when increasing density. Thus, we generate one database that consists of four subsets (0.1, 0.2, 0.4 and 0.6), each of which has 100 graphs whose size is 100 vertices. These subsets represent four different densities
- Vertex Scalability: The purpose of such a test is to see how GED methods behave

[3]taken from http://www.ece.umn.edu/ ~mihailo/software/graphsp/performance.html

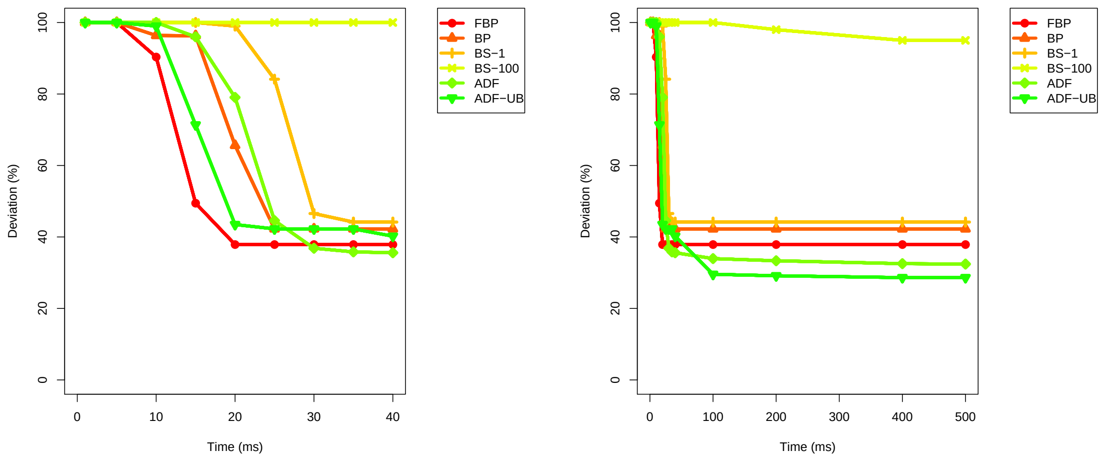


Figure 5.11: Anytime test: Deviation on MUTA: Left (up to 40 ms), Right (up to 500 ms)

when having low dense graphs (e.g., 0.1) and high dense graphs (e.g., 0.4) while increasing the number of vertices from 100 to 1000. To this end, 4 databases are generated each of which concerns a specific number of vertices (100-vertices, 200-vertices up to 1000 vertices). For each database, we generate two subsets of 100 graphs each of which has 0.1 and 0.4 as graph density, receptively.

In both density and vertex scalability tests, we are interested in the setup time of *FBP*, *BP* and *ADF*. Thus, for *ADF*, we stop the algorithm after a first complete solution is generated; this solution is compared to the answers provided by both *FBP* and *BP*. Since the graphs in this section are large (i.e., up to 1000 vertices), the preprocessing step of *ADF* and the first upper bound calculated by *BP* are not integrated, see Section 3.3.2. C_T is set to 300 seconds and C_M to 5GB due to the complexity of the problems.

5.3.8.1 Results of Density Scalability Test

Figures 5.15(a) and 5.15(b) represent the results of deviation and run time, respectively. Results showed that when the density of graphs increases the deviation of *BP* and *FBP* decreases and their run time increases. For instance, on the 0.1 subset, the deviations of *FBP*, *BP* and *ADF* were 0.93%, 1.72%, 2.186%, respectively. However, when the density increased, *ADF* had proven to be the most effective algorithm with a deviation that was equal to 1.34%, while the deviation of each of *FPB* and *BP* was 17.03%, 51.97%, receptively. These results prove that the most complex the graphs the less precise the results obtained by approximate GED algorithms. This is also the case of the run time. For instance, on the 0.1 density subset, both *FBP* and *BP* were twice as fast as *ADF* while *ADF* was twice as fast as *FBP* and *BP* on denser graphs (i.e., graphs whose density is 0.6). One can also notice that when density increased, the run times of all these algorithms increased because of the difficulty of the matching problems.

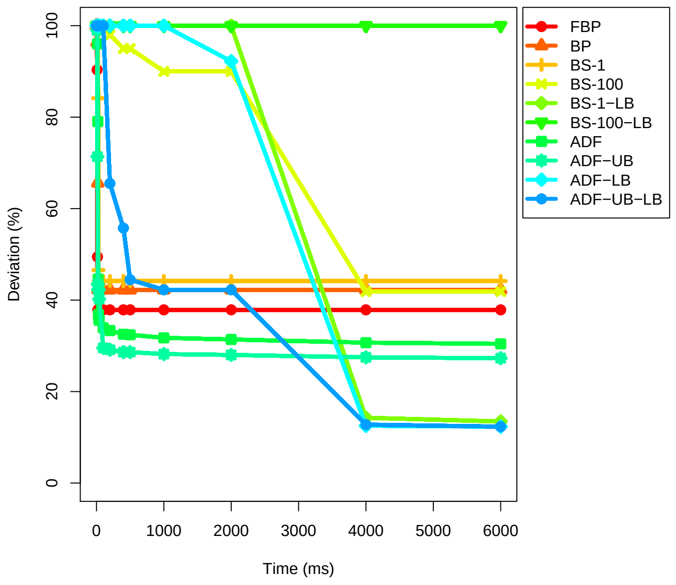


Figure 5.12: Anytime test: Deviation on MUTA: (up to 6000 ms)

5.3.8.2 Results of Vertex Scalability Test

Figure 5.16 depicts the results of vertex scalability when density is equal to 0.1. Figure 5.16(a) shows that the first solution found by *ADF* is very close to the solutions of *FBP* and *BP*. For example, in the case of 100 vertices, the deviations of *FBP*, *BP* and *D-DF* were 4.53%, 5.31%, 0.031%, respectively. However, when number of vertices is equal to 500, the deviation of *ADF* was beaten by *FBP* and *BP* with a very low percentage (i.e., 0.47%). On the run time side, *ADF* was tremendously faster in outputting a first solution. *FBP* was faster than *BP*, for instance, on 500 vertices, the average run time of *FBP*, *BP* and *ADF* was 273877 ms, 10833 ms, respectively. However, the run time of *FBP* and *BP* exceeded C_T when graphs are larger than 500 vertices. Thus, these methods were not tested on graphs whose sizes are larger than 500 vertices, see Figure 5.16(c). We can observe that the complexity of the problem increased approximately twice when increasing the number of vertices. For instance, the run time of *ADF* on 900 vertices was 174580 ms while it was 233924 ms on 1000 vertices. As a conclusion from Figure 5.16 , *ADF* proved to be the algorithm whose run time is remarkably low when compared to *FBP* and *BP*.

Regarding the results of vertex scalability test when the density is equal to 0.4%, one can observe that the deviation of *FBP* and *BP* became bigger due to the increase of the density, see Figure 5.17(a). *ADF* was the most precise algorithm with a deviation that is equal to 1.5% and 0.5% on 100 vertices and 200 vertices, respectively. *BP* was the least precise algorithm with a deviation that was equal to 16.5% and 33.6% on 100 vertices and 200 vertices, respectively. Regarding the run time results, one can see a tremendous growth in run time of *FBP* and *BP* when increasing the number of vertices from 100 to

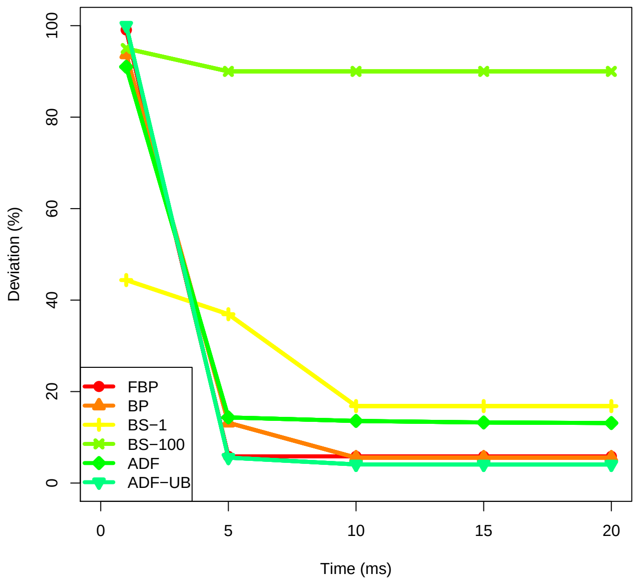


Figure 5.13: Anytime test: Deviation on GREC

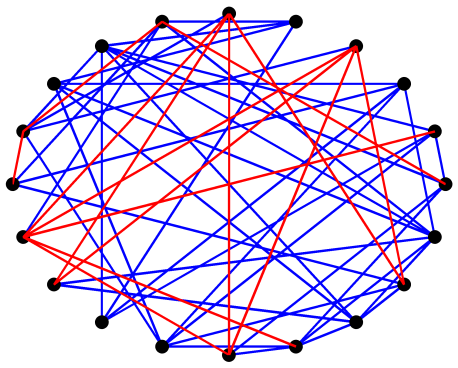
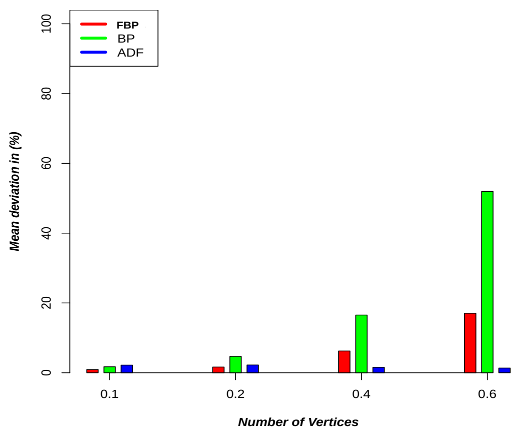


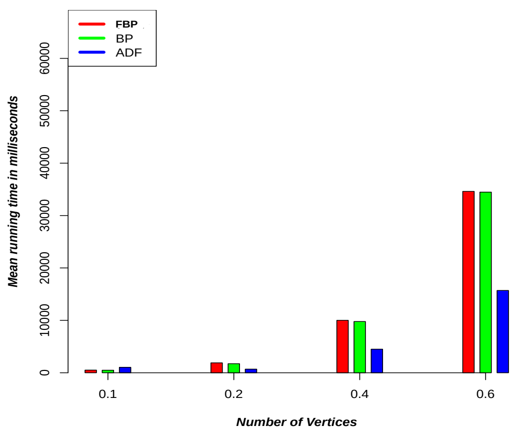
Figure 5.14: An example of an Erdos-Renyi graph

200 vertices. On 200 vertices, the average run time of *FBP* and *BP* is 377757 ms and 377979.49 ms, respectively. That is, both of them exceeded C_T on only 200 vertices while *ADF* took only 4926.04ms on 200 vertices, see Figure 5.17(b).*ADF*, however, continued to match graphs up to 500 vertices with a run time that was equal to 269979 ms, see Figure 5.17(c). After 500 vertices, it exceeded C_T.

Conclusion In the previous experiments, *ADF* was executed and stopped after a first complete solution is generated. *ADF*, however, had remarkably low deviations and run times when compared to *FBP* and *BP*. Despite the enough time *FBP* and *BP* had, they



(a) Deviation



(b) Running-Time

Figure 5.15: Graph Erdos-Renyi's density test: Deviation and run time

were not able to output solutions that are as precise as the ones found by *ADF* (except on non-complex graphs). Thus from the density and vertex scalability tests, one can conclude that:

- Approximate methods like *FBP* and *BP* are not precise at all when graphs are dense.
- Approximate methods can output fruitful solutions only when attributes help in guiding the matching process.

5.4 Classification under Time and Memory Constraints

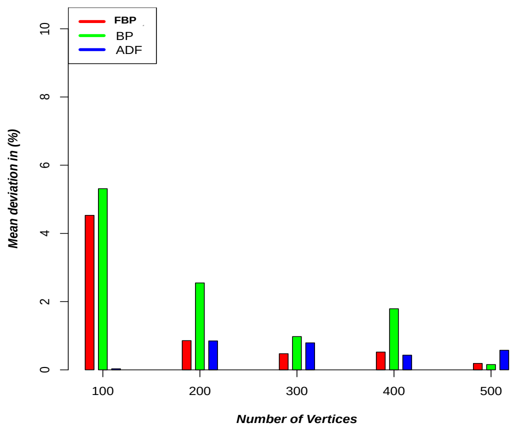
In the previous chapters, graph-level evaluations of methods were performed. In other words, methods are evaluated based on their distances and matching dissimilarities. In this chapter, a classification-level evaluation is performed. Methods are compared based on their classification rate. This test is considered as a use case of the test described in Section 5.3.4.2 where many distances have to be quickly computed.

5.4.1 Included Methods

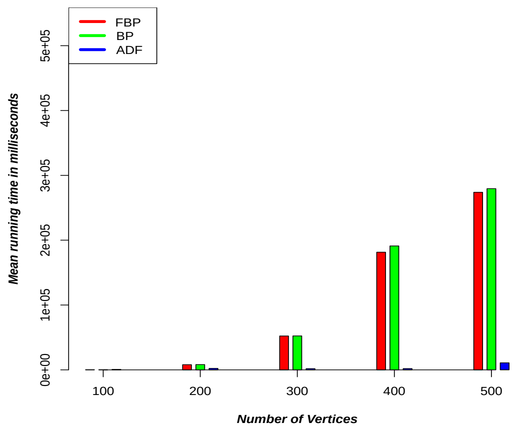
Table 5.13 shows the methods included in the classification experiments. Different versions of *ADF* as well as A^* are integrated and tested. We recall that *D-DF* cannot be included in a classification context because of its implementation' constraints, see Section 5.3.4.2.

5.4.2 Environment and Constraints

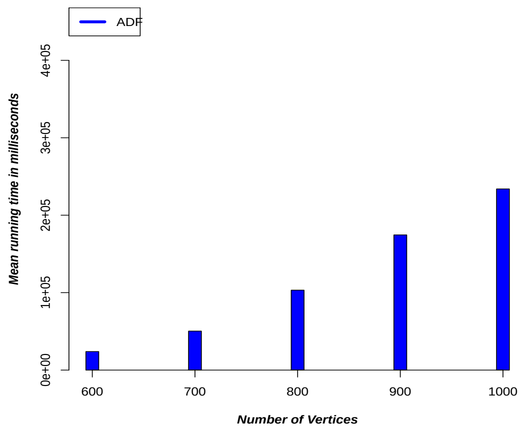
The evaluation of all algorithms are conducted on a 24-core Intel i5 processor 2.10GHz, 16GB memory. The methods are evaluated under time and memory constraints. As in the



(a) Deviation



(b) Run time (all methods)



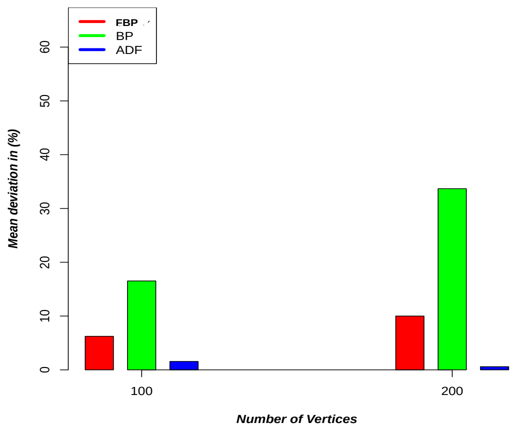
(c) Run time (ADF)

Figure 5.16: Vertex scalability test (Erdos-Renyi density = 0.1): Deviation and run time

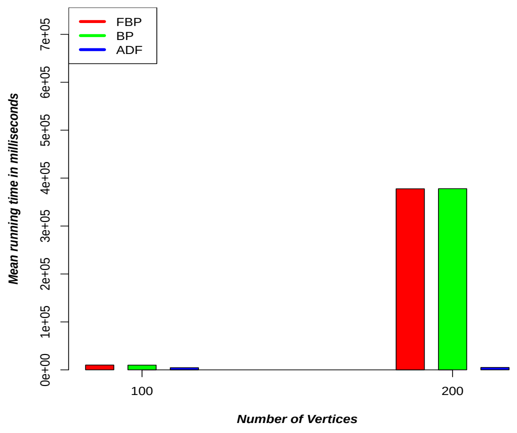
speed test, we have chosen the time limits used for each of GREC, MUTA and Protein, as depicted in Table 5.8. Regarding C_M, it was set to 1GB during the whole experiment.

5.4.3 Included Datasets

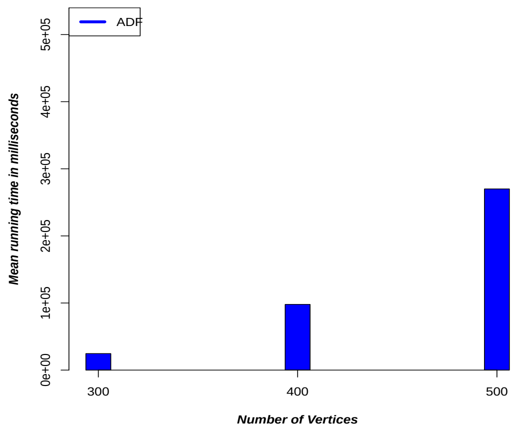
In order to test the classification rate, one need to choose some graphs' databases dedicated to classification. For that purpose, the IAM database [105] is selected. Since we are interested in classification, we will take the whole dataset without dividing it into subsets as what have been previously done. In this section, we provide some information about the learning dataset of each of GREC, MUTA and Protein.



(a) Deviation



(b) Run time (all methods)



(c) Run time (ADF)

Figure 5.17: Vertex scalability test (Erdos-Renyi density = 0.4): Deviation and run time

5.4.3.1 GREC

This data set consists of 1,100 graphs where graphs are uniformly distributed between 22 symbols. The resulting data set is split into a training and a validation set of size 286 each, and a test set of size 528.

5.4.3.2 Mutagenicity

4,337 elements are represented in this data set (2,401 mutagen elements and 1,936 non-mutagen elements) which are divided into: a training set of size 1,500, a validation set of size 500, and a test set of size 2,337.

Acronym	**Details**
$ADF\text{-}\overline{UB}\text{-}\overline{LB}$	*ADF* without upper bound and with h(p)=0.
$ADF\text{-}\overline{UB}\text{-}LB$	*ADF* without *UB* and with h(p)=*lb2*.
$ADF\text{-}UB\text{-}\overline{LB}$	*ADF* with an initial *UB* equals to *BP*, h(p)=0.
DF-UB-LB	*ADF* with an initial *UB* equals to *BP* and *lb2*
PDFS-UB-LB	*PDFS* with an initial *UB* equals to *BP* and *lb2*
$A^*\text{-}\overline{LB}$	the A^* algorithm with *lb2*
A^*	the A^* algorithm without *lb2*
BS-1, *BS-10* and *BS-100*	Beam Search with *OPEN* size = 1, 10 and 100, respectively
BP	The bipartite GM
FBP	The fast version of *BP*
H	The hausdorff algorithm.

Table 5.13: Methods included in the classification experiments

5.4.3.3 Protein

600 Proteins are uniformly distributed over 100 classes. The resulting data set is split into a training, a validation and a test set of size 200 each.

5.4.4 Protocol

In this experiment, 1-NN classifier is used to assess the quality of the resulting edit distances because it directly uses the edit distances without any additional classifier training.

Algorithm 18 The 1NN−classifier of each graph G_x in the test set

Input: The set S of labeled graphs (i.e., train set): $\{(G_1, C_1), \cdots, (G_n, C_n)\}$ and the unknown graph G_x **Output:** the class label assigned to G_x

```
1: $d_{min} = \infty$
2: $j = 0$
3: for $i = 0$ to n do
4:     $d_{ix}$=GraphEditDistance($G_x$,$G_i$)
5:     if $d_{ix} < d_{min}$ then
6:         $i = j$
7:         $d_{min} = d_{ix}$
8:     end if
9: end for
10: $C_x = C_j$
```

Algorithm 18 shows the pseudo code of the 1NN-classifier of each graph G_x in the test set. The classification of each test graph is performed by evaluating its edit distance from every graph in the training set and assigning it the class of the closest training graph (lines 3 to 8). Note that under time and memory constraints, *BS* and A^* sometimes output unfeasible distances (i.e., ∞) because they are not able to find a feasible solution before halting. 1GB has been set as a memory constraint for all algorithms. This memory constraint always concerns A^*.

Three metrics are selected: classification rate, response time, number of unfeasible solutions and number of optimal solutions. The average of each of the aforementioned metrics is calculated by dividing it by the number of test graphs.

5.4.5 Results and Discussion

Table 5.14 depicts the results on GREC and Protein. *BP ADF* and *PDFS* are the methods whose classification rates are the best on GREC and Protein.

On GREC, *ADF* with all its variants obtained the same classification rate as *BP* (i.e., 0.985) even the one without upper and lower bounds (i.e., *ADF*-$\overline{UB}$-$\overline{LB}$). That shows that *ADF* can also be used to classify graphs even without being obliged to wait for the final, or optimal, solution. *DF-UB-LB* was the fastest compared to all the variants and the one whose number of optimal solutions is the biggest. This fact shows the importance of *UB* and *LB* to make the algorithm faster. Accordingly, and since *PDFS* is an extension of *ADF*, not all the variants of *PDFS* are tested. That is, only *PDFS-UB-LB* has been included in the tests. *PDFS-UB-LB* was 29% faster than *DF-UB-LB*. Despite the fact that *H* was the worst algorithm when evaluating its distances as well as the matching dissimilarity, it was among the algorithms whose classification rate were high. One can see that, on GREC, *H* beat both *BS-10* and *BS-100*. A^*-$\overline{LB}$ obtained better classification rate than A^*, that is because the number of unfeasible solutions found by A^* is higher.

On Protein, one can see a different behavior, where *DF-UB*-$\overline{LB}$ was the fastest while *DF-UB-LB* was the slowest. That is because of the time consumed to calculate distances using the cost functions of Protein. Thus, as in GREC, *PDFS-UB*-$\overline{LB}$ was included in the tests. Despite the slowness of *DF-UB-LB*, it was also the best algorithm in terms of classification rate and the average number of optimal solutions. *PDFS-UB*-$\overline{LB}$ was 36% faster than *DF-UB*-$\overline{LB}$ and was able to find more optimal solutions. Even though *BS* took relatively enough time to classify graphs (compared to *ADF*), it was way far from the results obtained by *ADF*. A^* was not able to find feasible solutions of each pair of graphs. That was not the case of all the variants of *ADF* as they were always able to output feasible solutions before halting. Despite the fact that *FBP* was the fastest algorithm, it was unable to find the best classification rate. One can see that *ADF*-$\overline{UB}$-$\overline{LB}$ obtained a better classification rate than the one found by *FBP*.

On MUTA and since C_T is set to 500ms, we kept only *ADF*-$\overline{UB}$-$\overline{LB}$ and A^*-$\overline{LB}$ since computing *LB* and *UB* are costly on such a database. Results showed that *ADF*-$\overline{UB}$-$\overline{LB}$ was twice as slow as *BP*, however, both of them succeeded in finding the best classification rate (i.e., approximately 0.7). *PDFS*-$\overline{UB}$-$\overline{LB}$ was also able to find the same classification rate and was 40% faster than *ADF*-$\overline{UB}$-$\overline{LB}$.

	GREC			
Methods	**Classification Rate**	**Response Time (ms)**	**# Unfeasible Solutions**	**# Optimal Solutions**
ADF-$\overline{UB}$-$\overline{LB}$	0.985	171401.54	0.0	0.139
ADF-UB-$\overline{LB}$	0.985	163979.45	0.0	0.141
ADF-$\overline{UB}$-***LB***	0.985	140675.0	0.0	0.468
DF-UB-LB	0.985	140525.48	0.0	0.469
PDFS-UB-LB	**0.985**	**99850.79**	0.0	0.678
A^*-$\overline{LB}$	0.890	358158.76	0.218	0.065
A^*	0.530	222045.94	0.443	0.439
BS1	0.985	69236.34	0.0	0.0
BS10	0.943	83928.21	0.009	0.0
BS100	0.587	83928.20	0.287	0.0
BP	0.985	62294.60	0.0	0.0
FBP	**0.985**	**27922.65**	0.0	0.0
H	0.9621	63563.74	1.0	0.0
	Protein			
ADF-$\overline{UB}$-$\overline{LB}$	0.445	128469.57	0.0	0.00755
ADF-UB-$\overline{LB}$	0.52	124361.61	0.0	0.00765
ADF-$\overline{UB}$-***LB***	0.40	147371.86	0.0	0.00738
DF-UB-LB	0.52	145779.68	0	0.007875
PDFS-UB-$\overline{LB}$	**0.52**	**80038.33**	0.0	0.00855
A^*-$\overline{LB}$	0.29	1065106.80	0.011	0.00595
A^*	0.265	194021.88	0.005	0.0045
BS1	0.24	129571.76	0.244	0.0
BS10	0.26	139294.88	0.035	0.0
BS100	0.26	141265.41	0.007	0.0
BP	**0.52**	**59041.84**	0.0	0.0
FBP	0.385	39425.69	0.0	0.0
H	0.43	71990.62	1	0.0
	Mutagenicity			
ADF-$\overline{UB}$-$\overline{LB}$	0.70089	1139134.29	0.0	0.001411
PDFS-$\overline{UB}$-$\overline{LB}$	**0.70089**	**760861.518**	0.0	0.04856
A^*-$\overline{LB}$	0.4574	856793.020	0.4	0.00001
BS1	0.55840	1015688.00	0.37155	0.0
BS10	0.55540	1256793.020	0.5067	0.0
BS100	0.55540	1383838.66	0.7238	0.0
BP	0.700042	528546.64	0.0	0.0
FBP	**0.700042**	**376135.51**	0.2	0.0
H	0.58964	525610.25	1.0	0

Table 5.14: Classification results on GREC, Protein and MUTA. The best results on the side of exact and approximate GED methods are marked in bold style

Part III

Conclusion

Chapter 6

Conclusion and Future Works

In literature and in life we ultimately pursue, not conclusions, but beginnings. Sam Tanenhaus

Contents

Abstract

In this very final chapter, we recall the main contributions of the thesis and review the advantages and drawbacks of the proposed works as well as their possible extensions.

6.1 Conclusion

This thesis lies in the scope of graph-based pattern recognition methods. We focused on the optimization of GM techniques in order to make them usable on real-world PR applications. That is, the proposed methods should be able to process highly attributed graphs in a reasonable time on a reasonable machine (i.e., under time and memory constraints).

Roughly speaking, in the literature there are two categories of GM known as exact GM and error-tolerant GM. The algorithms dedicated to solving exact GM have to answer the question of whether or not two graphs, or subgraphs of them, are identical. Thus, the graphs to be matched have to be strictly correspondent or partially correspondent in terms of structure as well as attributes. Exact GM methods fail to match real-world graphs that are noisy and deformed due to graph extraction process. For these reasons, error-tolerant GM emerged as a powerful and flexible GM problem where any kind of graphs can be included in the matching process. By using error-tolerant techniques, one can match graphs, whether identical or not, and a similarity measure as well as the matching sequence can be obtained after the matching process. Thus, exact GM can be seen as a special case of error-tolerant GM. The similarity measure of error-tolerant GM can be obtained through objective functions that aim at increasing the similarity between two graphs. For an algorithm dedicated to error-tolerant GM to obtain the exact matching sequence, huge resources are needed specially when working on large graphs. To this end, researchers tend to optimize their proposed approaches in order to be able to match relatively large graphs.

Graph edit distance (GED) has received attention as an important way to measure the similarity between pairwise graphs error-tolerantly. The basic idea of GED is to find the best set of transformations that can transform graph G_1 into graph G_2 by means of edit operations on graph G_1. Each path that transforms G_1 into G_2 is called an edit path. A partial edit path is an edit path that partially transforms one graph into another. Classically, the allowed operations are inserting, deleting and/or substituting vertices and their corresponding edges.

In the literature, error-tolerant GM methods have often been evaluated in a classification context and less deeply assessed in terms of the accuracy of the found solution. To evaluate the accuracy of error-tolerant GM methods, graph-level information is required at matching level (i.e., matching quality and similarity deviation) and not only at class level. Most of the publicly available repositories with associated ground truths are dedicated to evaluating graph classification or exact GM methods and so the matching correspondences as well as the distance between each pair of graphs is not directly evaluated. As a first contribution in the thesis, a performance evaluation tool for GED methods is proposed. This contribution consists of two parts: First, a graph database repository, called GDR4GED, annotated with graph-level information like graph edit distances and their matching correspondences has been made publicly available for some representative graph databases. The proposed new metrics allow to evaluate and characterize GED methods by evaluating the matching correspondences as well as the distance between each pair of graphs. Because of the high complexity of GED methods, we proposed to evaluate them under time and memory constraints. The aim of this contribution is to make GED methods better comparable against each other and to provide information about their applicability on real-world problems. For that reason, we highly encourage the community not only to use the information provided in GDR4GED, but also to integrate their algorithms' answers when obtaining more accurate results.

The next contribution of this thesis falls in the GM families. As stated before, two main families have been found in the literature: exact and error-tolerant GM. In this thesis, we proposed adding a new GM family, called anytime GM. In order to demonstrate the benefit of having such a family, a new optimized algorithm which is based on depth-first search is put forward. This algorithm speeds up the computations of graph edit distance thanks to its upper and lower bounds pruning strategy and its preprocessing step. Moreover, this algorithm does not exhaust memory as the number of pending edit paths that are stored in the set *OPEN* is relatively small thanks to the depth-first search where the number of pending nodes is $|V1|.|V2|$ in the worst case where $|V_1|$ and $|V_2|$ are the numbers of vertices in G_1 and G_2, respectively. Accordingly, *DF* outperformed A^* in terms of speed, precision and classification rates, as it was illustrated in the thesis. *DF* is able to provide not only one solution but successive solutions for better and better quality according to available resources. The anytime version of *DF*, denoted by *ADF*, is able to find an initial, possibly suboptimal, solution quickly, keep it in the memory and then continue searching for improved solutions until convergence to a provably optimal solution. The simplicity of the approach makes it very easy to use, it is also widely applicable. It can be used not only when optimal solution is desired, but also when we want to see the evolution of the quality of the suboptimal solutions found at each time t. One can remarkably notice the interests of anytime GM methods compared to approximate methods as demonstrated in the experiments under time constraints. Generally speaking, Anytime GM provides an attractive approach to challenging GM problems, especially when the time and memory available to compare graphs are limited or uncertain and when we are interested in improving the best solution found so far.

To go one step further, to be able to match larger graphs with better quality, we also propose parallel GED algorithms. Always starting from *DF*, we speed up its computations by proposing a multithreaded implementation with an efficient load balancing strategy. This algorithm is called *PDFS*. In *PDFS*, each thread gets one or more partial edit paths and all threads solve their assigned edit paths in a fully parallel manner. A work stealing process is performed whenever a thread finishes all its assigned threads. Moreover, synchronization is applied in order to ensure upper bound coherence. *PDFS* has a bottleneck since that it cannot be run on several machines. To cope with this problem, a distributed version can be of great interest so as to scale up and to match larger graphs.

As a distributed version of *DF*, we have proposed another exact GED algorithm, referred to as *D-DF*, which is a distributed GED that is also based on *DF*. *D-DF* is implemented on the top of Hadoop with a message passing tool. The reason for having chosen Hadoop is to take advantage of its fault tolerance. *D-DF* first starts with a preprocessing step and the distribution of partial edit paths among workers. Each worker gets one partial edit path and all workers solve their assigned edit paths in a fully distributed manner. In addition, a notification process is integrated. When any worker finds a better upper bound, it notifies the master to share the new upper bound with all workers.

The main drawback behind *D-DF* is that it has a single job. When there is no C_T, some workers may work while others become idle after finishing the exploration of their assigned partial edit paths. To overcome this drawback and as future work, we aim at including the load balancing into the process of *D-DF* where all workers work without becoming idle. Moreover, two ideas can be applied for both *DF* and *D-DF*. First, coming up with a better lower bound and thus making the calculations faster. Second, learning to sort the vertices of each dataset in a way that minimizes its deviation. Such an extension of *DF* and *D-DF* can beat the approximate approaches when matching symbolic graphs.

To evaluate *ADF*, *PDFS* and *D-DF*, we have proposed evaluating both exact and approximate GED approaches, using the metrics proposed in GDR4GED under soft and hard time constraints. Soft constraints are devoted to accuracy tests while hard constraints are devoted to speed tests, respectively. One could ask a question: "how many milliseconds do we need as soft time constraints?". To answer this question, we measured the time needed by the approximate methods (i.e., the bipartite matching method *BP* [106] and the hausdorff-based method *H* [49]) to output solutions for each pair of graph. The maximum time needed by *BP* and *H* to output a solution on the selected databases was: 500 ms on MUTA and 400 ms on GREC, Protein and CMU. As for the soft time constraint, 300 seconds was fixed for all the selected subsets. This amount of time can ensure that the tree-based approximate GED algorithms as well as the exact algorithms output solutions can be compared to the algorithms whose complexities are less (such as *BP* and *H*).

PDFS was compared to other methods under both hard and soft time constraints. However, that was not the case of *D-DF*. Because of distribution aspects, *D-DF* was not evaluated under a hard time constraint. Hence, *DF* had replaced it for such a test as it represented the sequential version of *D-DF*. All the algorithms evaluated under the soft time constraint without any exception. In the case of evaluating the methods under hard constraints, the lower bound was removed from *PDFS* because it was time consuming.

Results showed that under soft time constraints *PDFS* and *D-DF* had the minimum deviation and matching dissimilarity and the maximum number of best found solutions. Both of them explored more nodes than *ADF* in parallel and distributed fashions and thus helped in speeding up the exploration of the search tree. Since our goal was to elaborate methods dealing with rich and complex attributed graphs, *BS* was slightly superior to *PDFS* and *D-DF* in terms of best found solutions and matching dissimilarity when evaluated on the MUTA dataset. Experiments has also demonstrated that *PDFS* always outperformed *ADF* in terms of speed, however, that was not the case of *D-DF* because of distribution aspects. *D-DF*, however, was faster than *ADF* when matching difficult problems. Results has also indicated that there is always a trade-off between deviation and running time. In other words, approximate methods are fast, however, they are not as accurate as exact methods. On the other hand, *ADF*, *PDFS* and *D-DF* took longer time but led to better results (except on MUTA). By limiting the run-time, our exact method provides (sub)optimal solutions and becomes an efficient upper bound GED approximation.

By taking a look at the results of the experiments, it is hard to tell whether *PDFS* or *D-DF* is better for solving GM problems. *PDFS* has been tested on a 24-core machine while *D-DF* was tested on 5-machines, each of which is a 4-core machine and thus these methods cannot be compared due to the differences in their approaches and mainly machines' characteristics.

As for the tests under hard time constraints, *PDFS* was among the slowest algorithms, however, it was always the most precise one. While *BP* was the most precise approximate GED method, its matching was high when compared to *ADF* and *PDFS*. Such a fact reveals the necessity of not only evaluating the distances outputted by approximate algorithm but also their matching correspondences.

One could ask a question: "When to use *PDFS* and when to use *D-DF*?". As it was stated before, *PDFS* is limited by the number of cores the machine has. *D-DF* is more scalable as one can add more machines and thus have more precise solutions in terms of distance and matching. This is not actually the case of *PDFS* as we cannot keep increasing the number of the threads, there will be a moment where the performance of *PDFS* will be degraded (as shown in the experiments when changing the number of the threads).

To the best of our knowledge, *PDFS* and *D-DF* were the first attempts for reducing the running time of exact GED using parallel and distributed fashions.

In the experiments of anytime GED, we have focused on both the deviation when varying the timeout and the minimal time needed by our anytime algorithm to get a first solution, often suboptimal specially on large graphs, as well as the time needed by some suboptimal GM methods on different graph datasets. Results showed that there is a trade-off between time and quality. Even if the fast bipartite matching *FBP* [121] and *BP* [106] were faster when graphs were sparser; *ADF* was faster when graphs were denser. It is remarkable that anytime algorithms are also effective when we accept some additional time that grantees better solutions to be found. Merging *ADF* and *BP* as in *ADF-UB* is also beneficial since anytime *ADF* will improve the solutions found by *BP*. On the selected datasets, the experiments showed that (*ADF*) *ADF-UB* have beaten all approximate methods by just waiting 100 ms per graphs comparison. The proposed anytime GM methods bring into question the usual evidences that claims that it is impossible to use exact methods of GM in real-world applications when matching large graphs, or even in a classification context.

As a last part of the thesis, a classification test under hard time constraints and a memory constraint is performed. This test is considered as a high-level experiment where classification accuracy and not distance or matching is evaluated. *PDFS*, *ADF* and the other sequential algorithms are evaluated on GREC, Protein and MUTA. 1-NN classifier is used to assess the quality of the resulting edit distances because it directly uses the edit distances without any additional classifier training on these databases. Results showed that *ADF* with all its versions and *PDFS* have proved to be fruitful in a classification context. Both algorithms along with the cubic algorithm *BP* had the highest classification rate. *H*, when evaluated in a graph-level way was the worst. However, when evaluated in a high-level manner, it has showed to perform well. Such a result highlights the importance of evaluating GED methods, or generally error-tolerant GM methods in a graph-level context.

6.2 Perspectives and Future Challenges

As a future work of GDR4GED, we will further expand this repository by integrating other publicly available graph databases that have (un)attributed graphs with different topologies, densities, etc. Both exact and approximate methods will be integrated. Moreover, it will be also interesting to improve the quality of the best found solutions for all the pairs of graphs on all the datasets.

At the level of *DF*, it can still be optimized using some ideas:

- Coming up with a better lower bound and thus making the calculations faster.
- Learning to sort the vertices of each pair of graphs in a way that minimizes its deviation.

Such an extension of *DF* can beat the approximate approaches when matching graphs of MUTA and Protein. Since *PDFS* and *D-DF* are based on *DF*, these improvements would also be added to them.

Since *PDFS* is limited to one machine, an extended version of it that can scale up from a single machine algorithm to a multi-machines one is highly effective. Such a version can be able to scale up and converge faster to the optimal solution. To do that, Message

Passing Interface can be a suitable distributed computing model since it proved to be efficient for scalable algorithms that need to share some information between processes.

As for *D-DF*, its main drawback was due to the fact that it is a single job algorithm. When there is no C_T, some workers may work while others become idle after finishing the exploration of their assigned partial edit paths. To overcome this drawback and as future work, *D-DF* can be transformed into a multi-iteration method where all workers work without becoming idle.

To the best of our knowledge, *ADF* is the first attempt of introducing anytime methods in GM. One big challenge for anytime algorithms is to be able to automatically decide when to stop the computation because the actual solution is of sufficient quality or will not be improved significantly even with the delight of more time. Different techniques can be proposed to take this decision like analyzing the past evolution of solutions. In future works, more tests will be conducted to understand better the effect of graph's structures on approximate and anytime algorithms. Moreover, others heuristic search methods or anytime version (like CBS [154]) can be adapted to solve the GM problem and could be compared to the first anytime method proposed in the thesis.

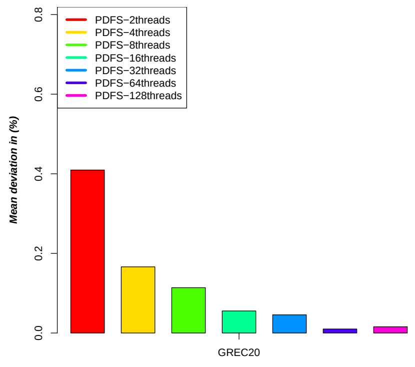
Generally speaking, selecting the most appropriate GED algorithm and so GM algorithm to the own requirements is not a trivial task. To answer this question, one should study the behavior of the selected algorithms on various graphs that differ in their topology, their sizes and the type of attributes on edges as well as vertices (symbolic, numerical, etc.). All algorithms should be compared on the same databases.

This thesis mainly focused on speeding up exact GED methods and so evaluating them under memory and time constraints. For all the aforementioned algorithms, and instead of fixing the time constraints, we can propose a way to automatically interrupt sequential, parallel, distributed or anytime GED algorithms when the quality of the actual solution is sufficient for the targeted application or when even with much more time, the quality of the solution will not increase significantly.

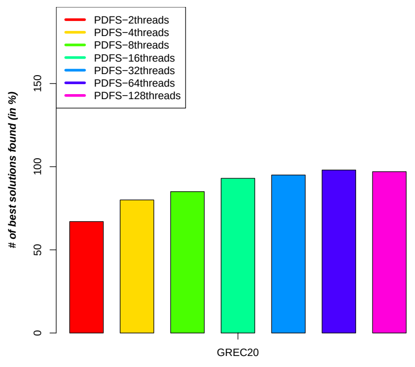
Appendices

Appendix A

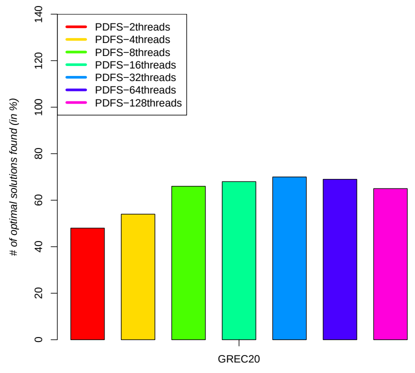
Parameters Selection for the Parallel Graph Edit Distance Method



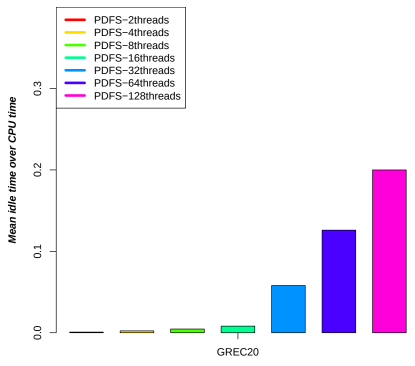
(a) Deviation



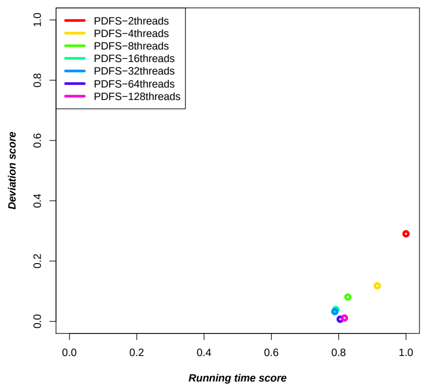
(b) Number of best found solutions



(c) Number of optimal solutions

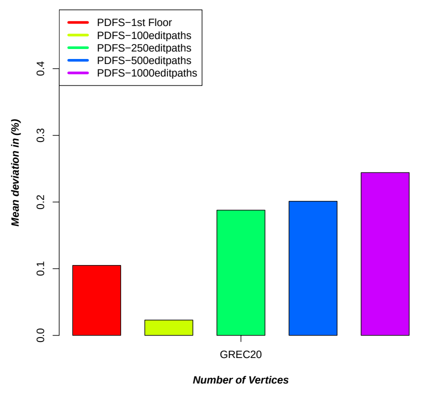


(d) Idle time over CPU time

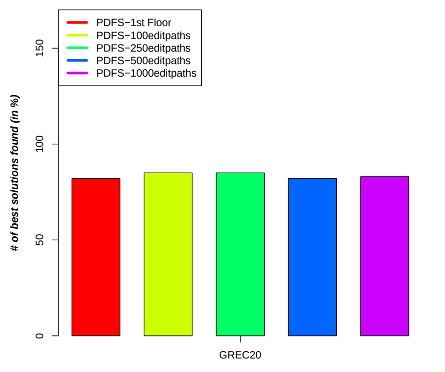


(e) Time-deviation scores

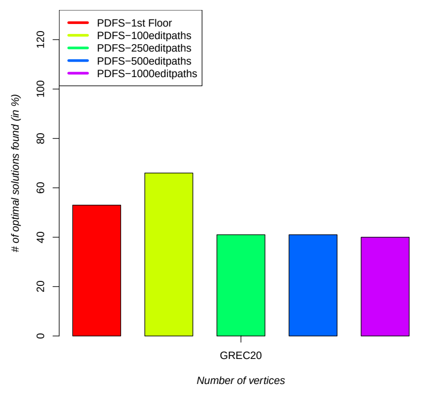
Figure A.1: The effect of the number of threads on the performance of *PDFS* when executed on GREC-20. N was equal to 100



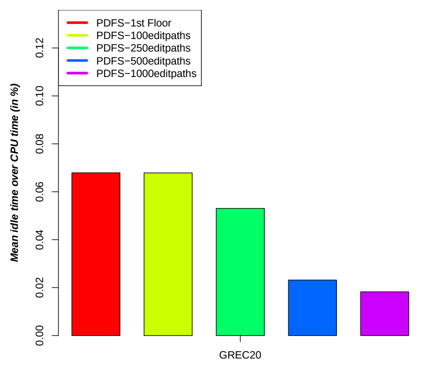
(a) Deviation



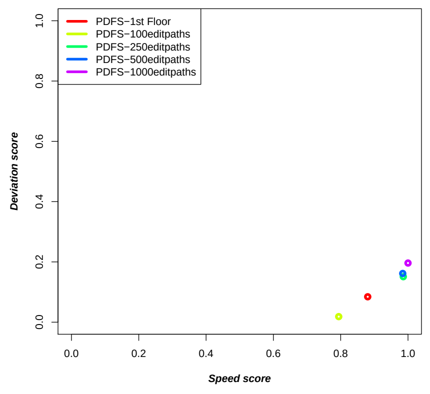
(b) Number of best found solutions



(c) Number of optimal solutions



(d) Average idle time over CPU time

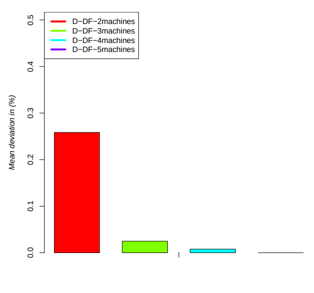


(e) Time-deviation scores

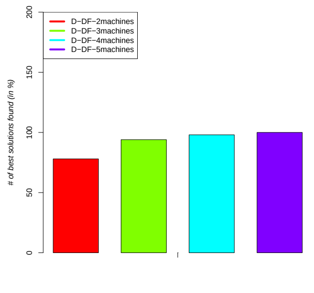
Figure A.2: The effect of the number of edit paths N on the performance of *PDFS* when executed on GREC-20. The number of threads was 64

Appendix B

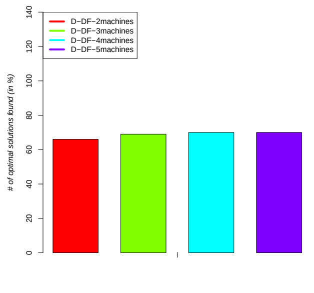
Parameters Selection for the Distributed Graph Edit Distance Method



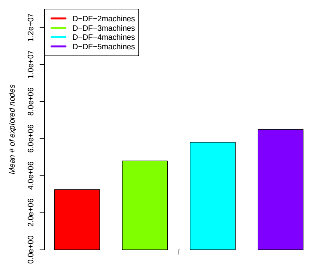
(a) Deviation



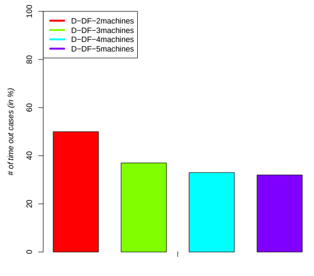
(b) Number of Best Found Solutions



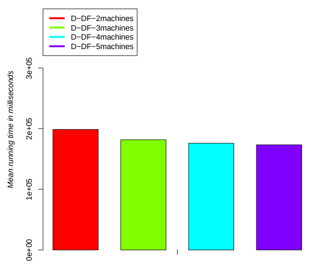
(c) Number of Optimal Solutions



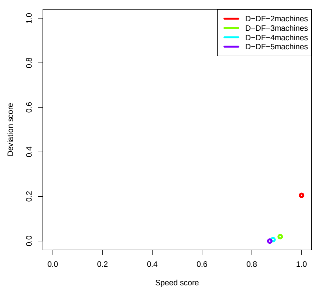
(d) Number of Explored Nodes



(e) Number of Time-Outs

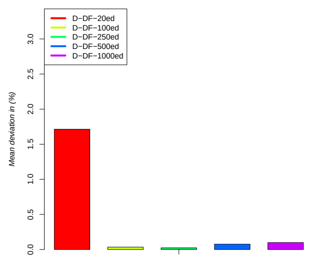


(f) Run time

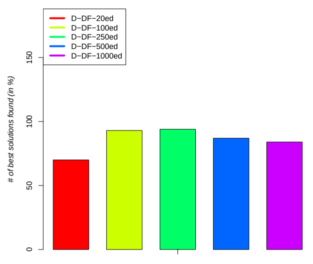


(g) Time-Deviation Scores

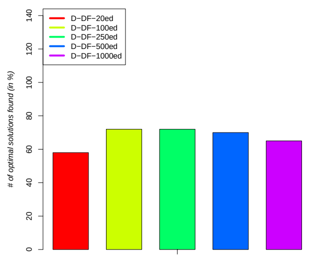
Figure B.1: The effect of the number of machines on the performance of *D-DF*



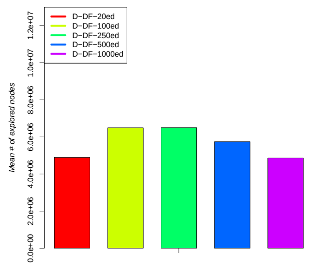
(a) Deviation



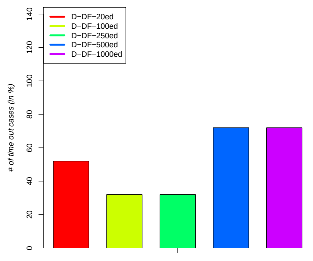
(b) Number of Best Found Solutions



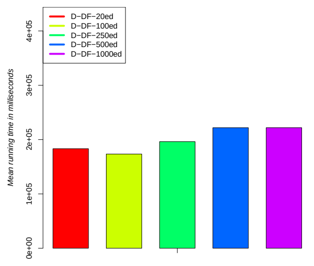
(c) Number of Optimal Solutions



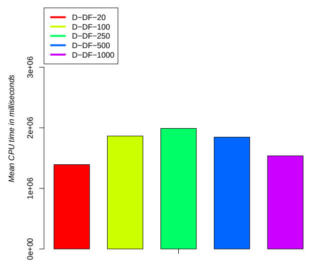
(d) Number of Explored Nodes



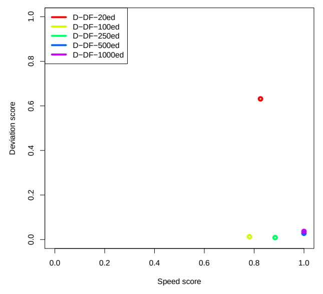
(e) Number of Time-Outs



(f) Average run time



(g) Average CPU Time

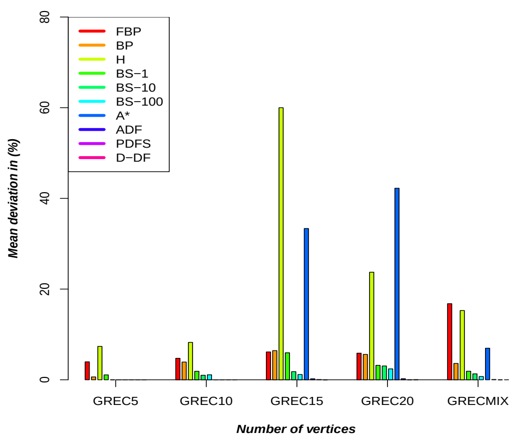


(h) Time-Deviation Scores

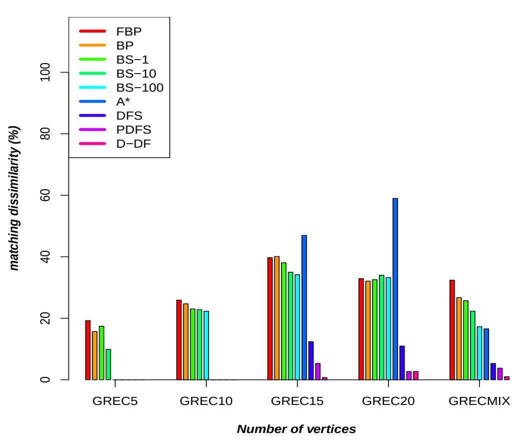
Figure B.2: The effect of the number of edit paths on the performance of *D-DF* when executed on the GREC dataset

Appendix C

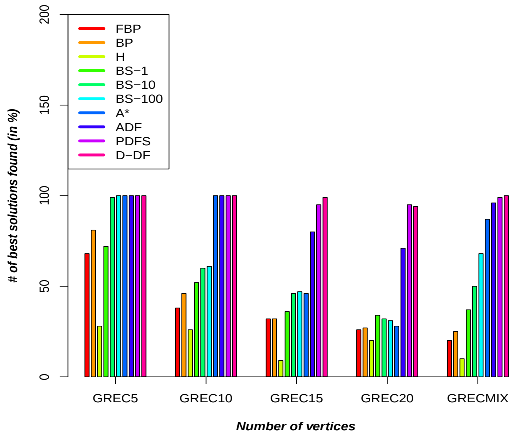
Tests under Soft Time Constraints



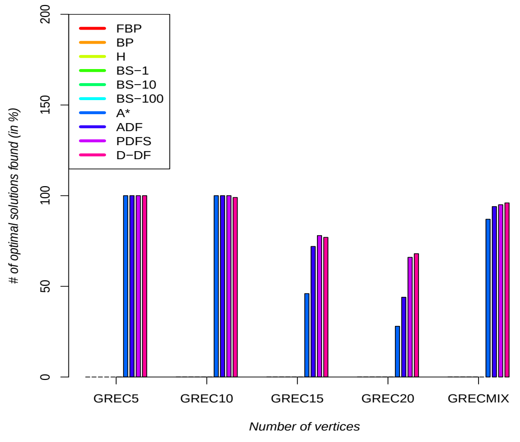
(a) Deviation



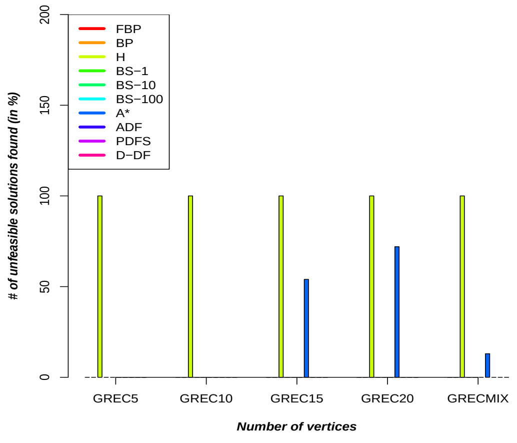
(b) Matching dissimilarity



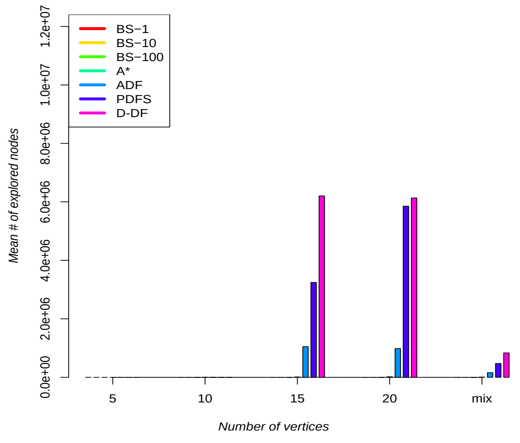
(c) Number of Best Found Solutions



(d) Number of optimal solutions

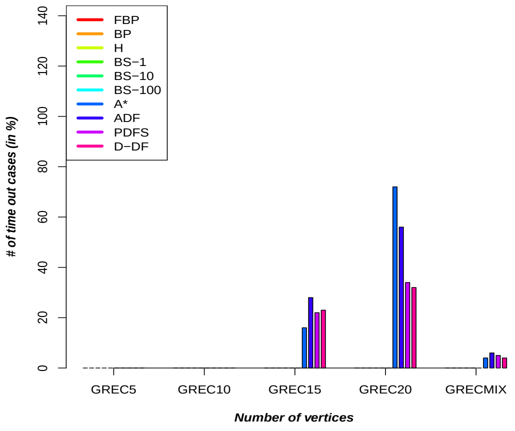


(e) Number of unfeasible solutions

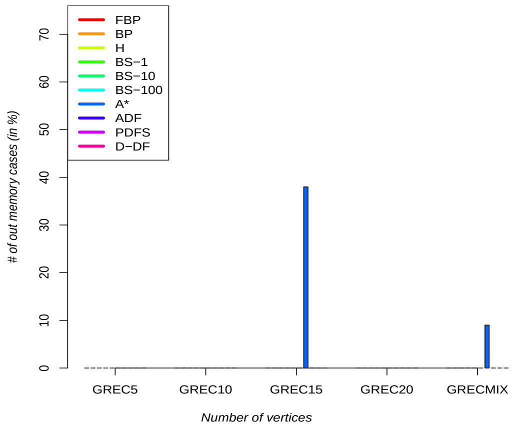


(f) Number of explored nodes

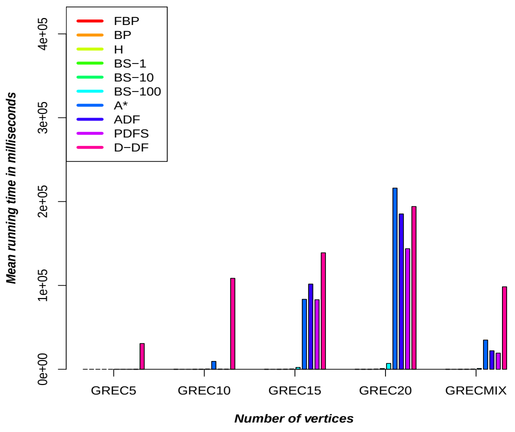
Figure C.1: Test under soft time constraints: Results on GREC



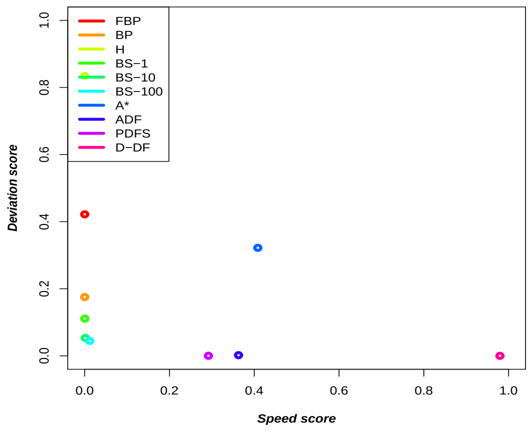
(a) Number of time-outs



(b) Number of out-of-memory cases

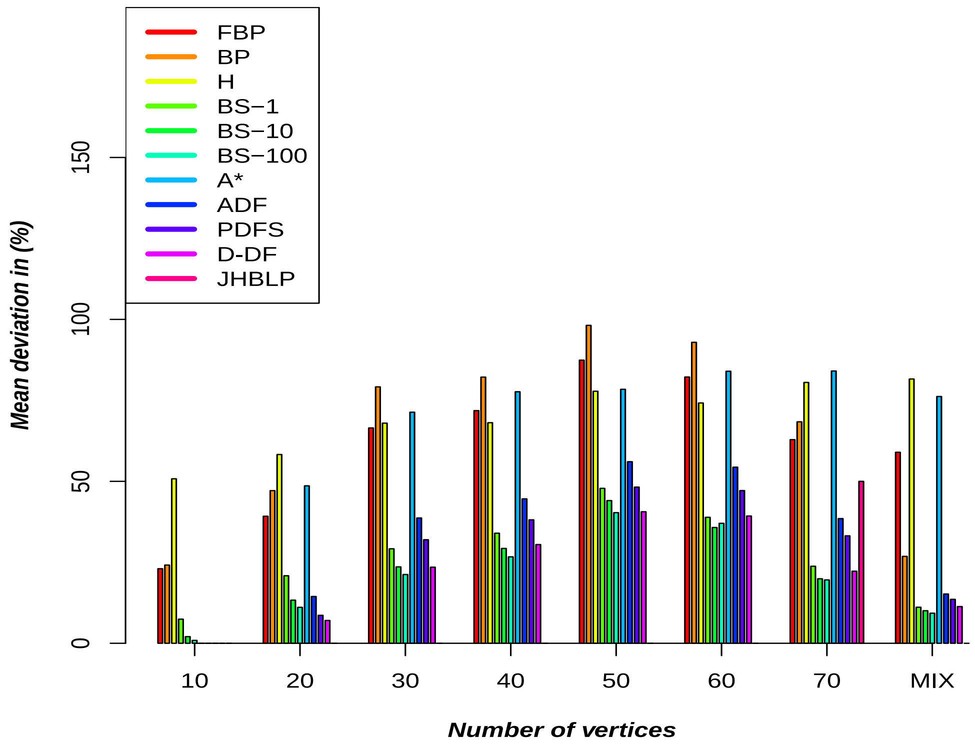


(c) Run time

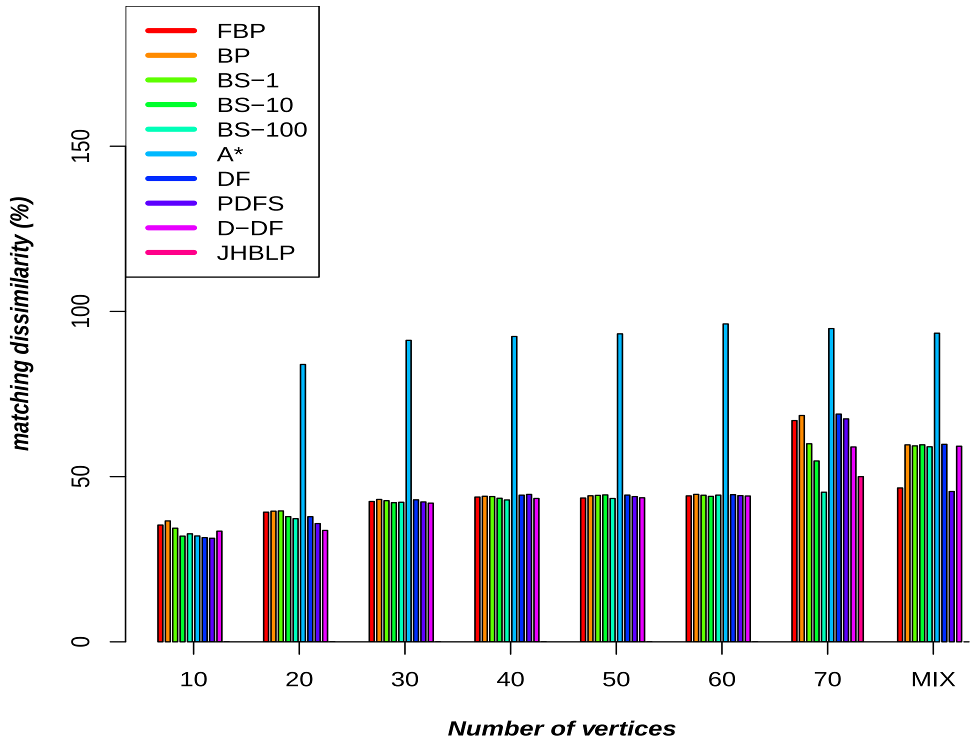


(d) Speed-deviation scores

Figure C.2: Test under soft time constraints: Results on GREC

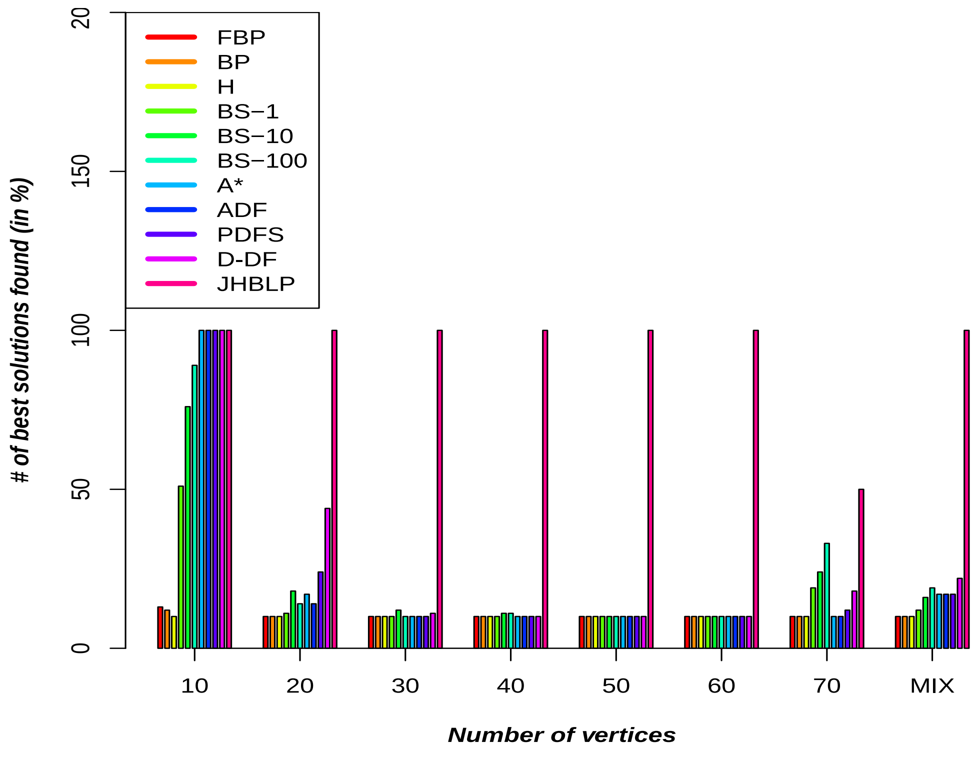


(a) Deviation: Note that the deviation of *JHBLP* was equal to zero on all subsets (except MUTA-70). For this reason, its bar does not appear in the histogram

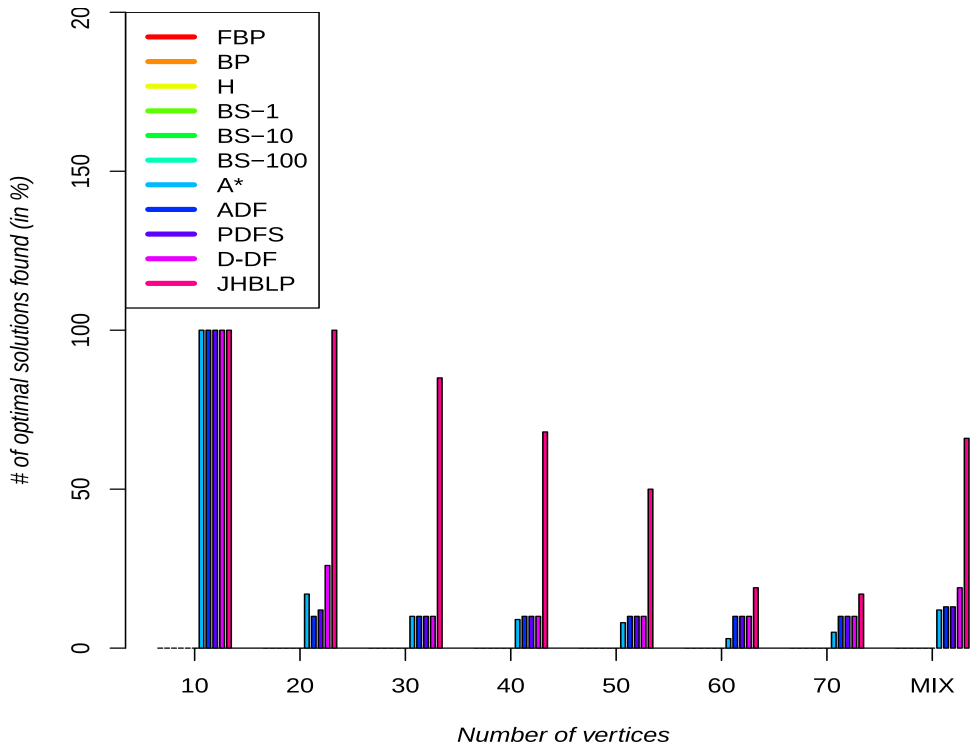


(b) Matching Dissimilarity

Figure C.3: Test under soft time constraints: Deviation and Matching Dissimilarity on MUTA

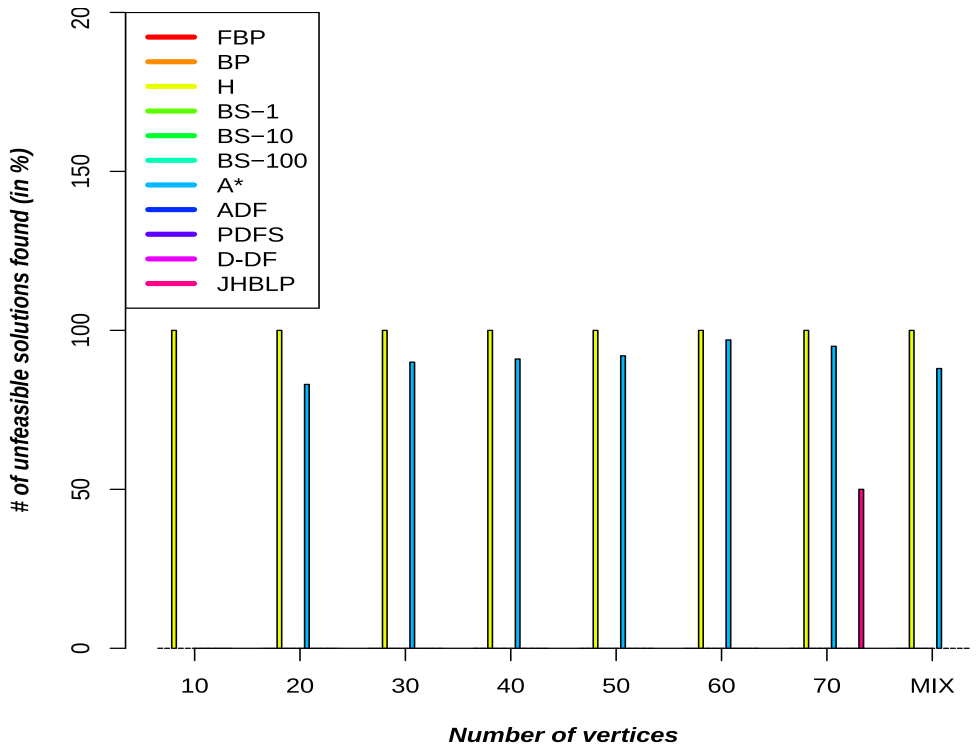


(a) Number of best found solutions

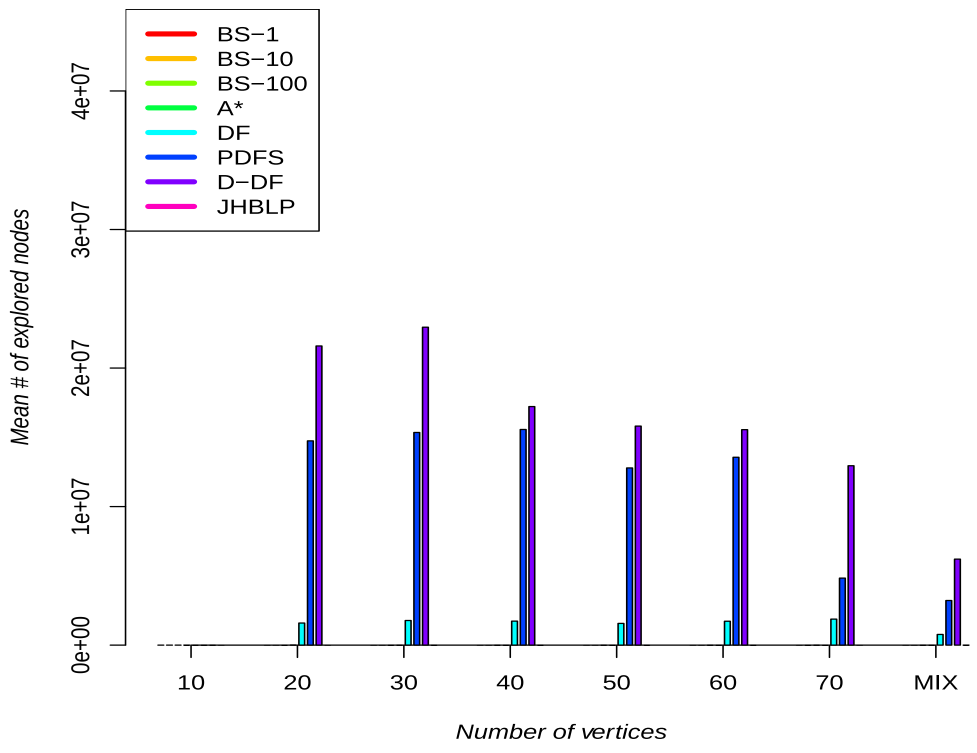


(b) Number of optimal solutions

Figure C.4: Test under soft time constraints: Number of best found solutions and optimal solutions on MUTA

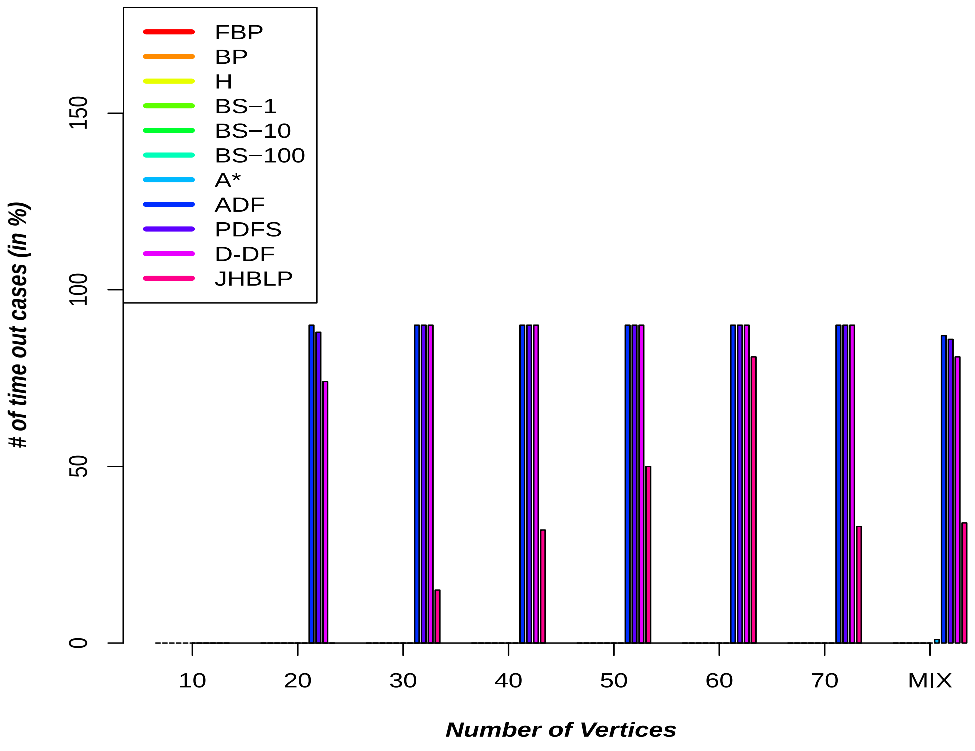


(a) Number of unfeasible solutions

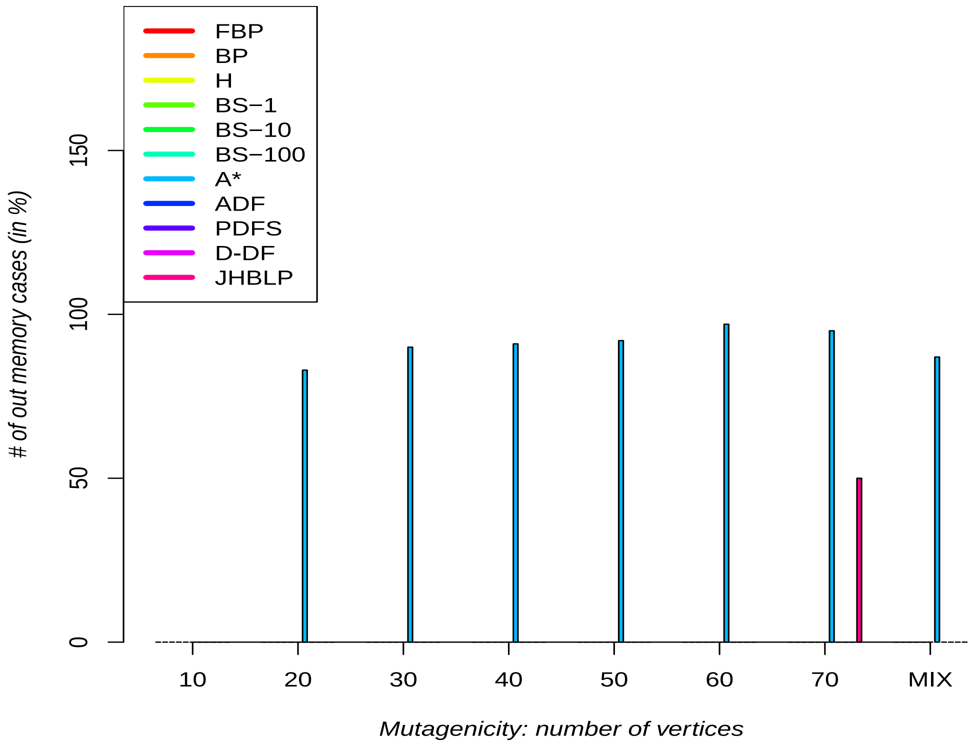


(b) Number of explored nodes. The number of nodes explored by *JHBLP* cannot be seen since it is very small when compared to the other methods

Figure C.5: Test under soft time constraints: Number of unfeasible solutions and explored nodes on MUTA

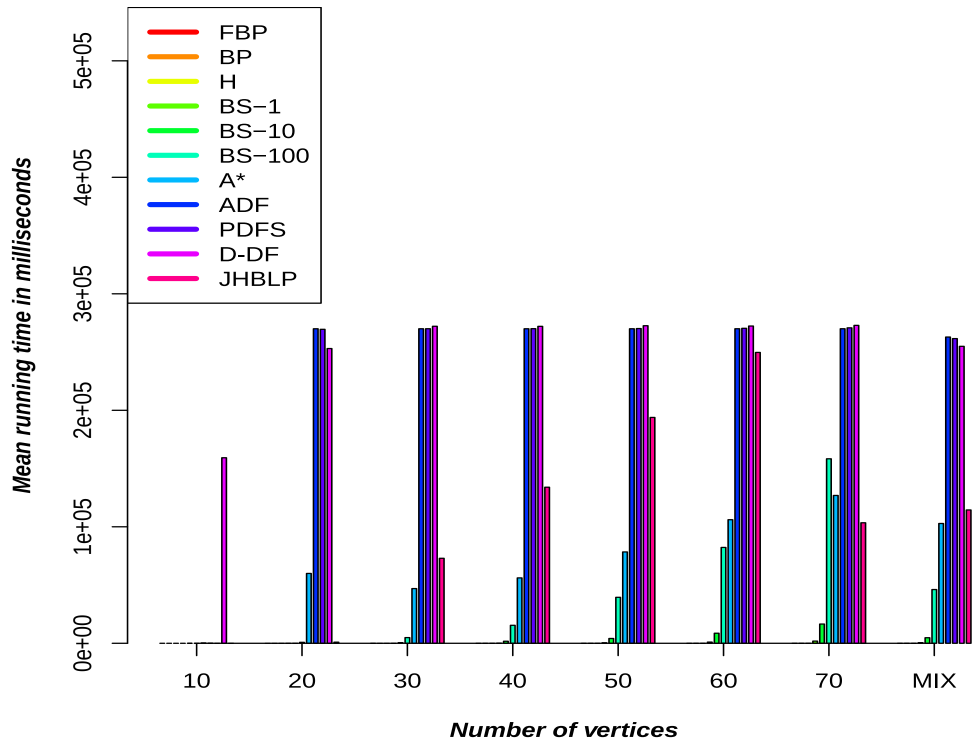


(a) Number of time-outs

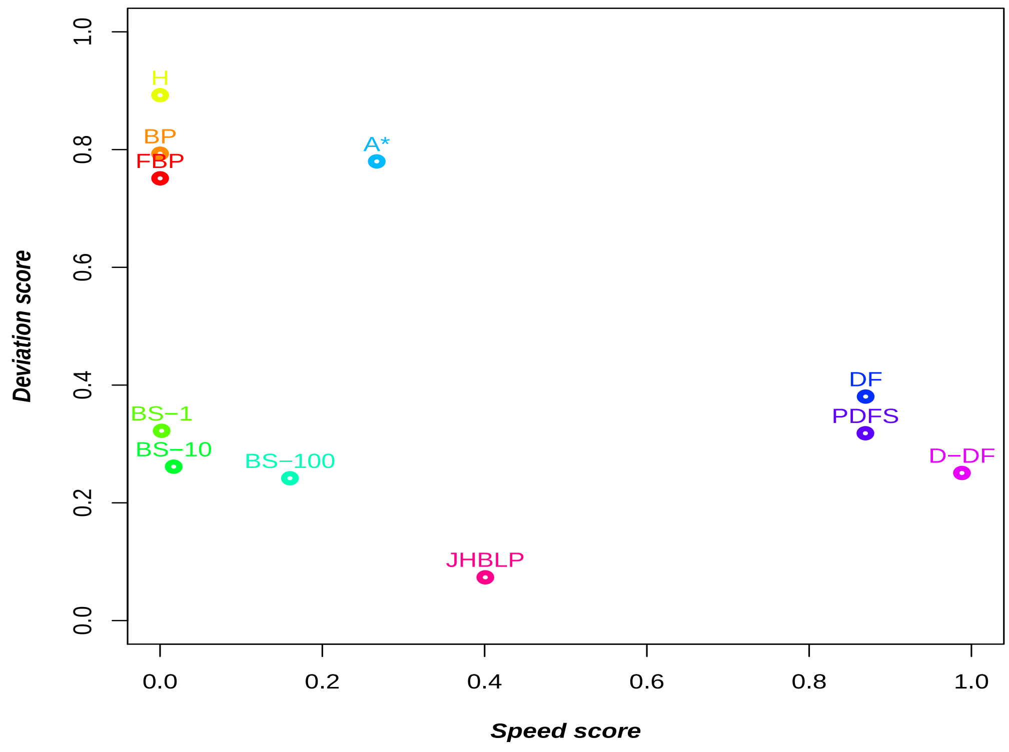


(b) Number of out-of-memory cases

Figure C.6: Test under soft time constraints: Number of time-outs and memory-outs on MUTA

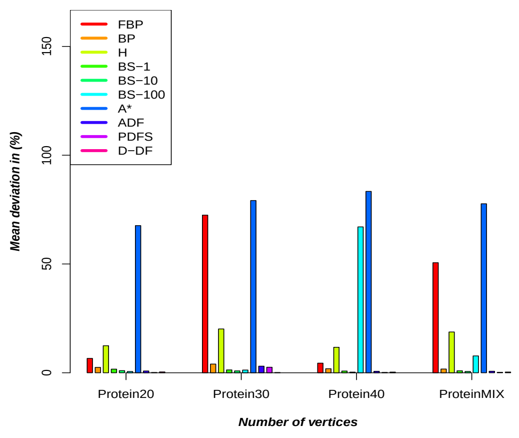


(a) Run time

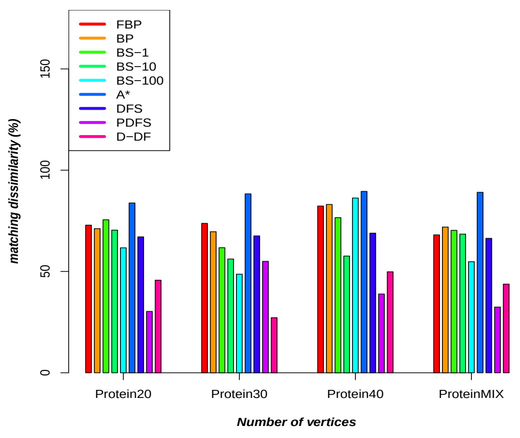


(b) Time-deviation scores

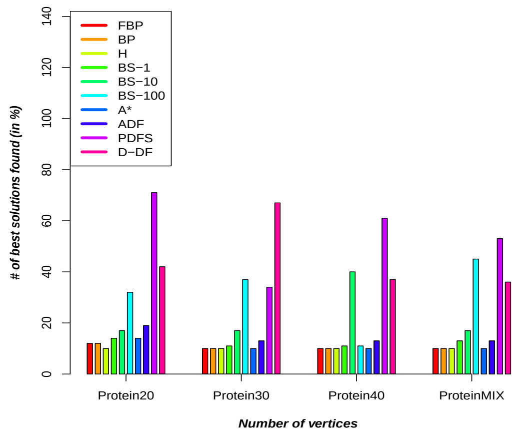
Figure C.7: Test under soft time constraints: Run time and Time-Deviation scores on MUTA



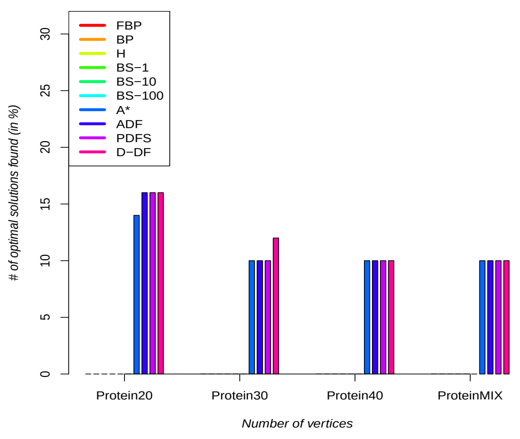
(a) Deviation



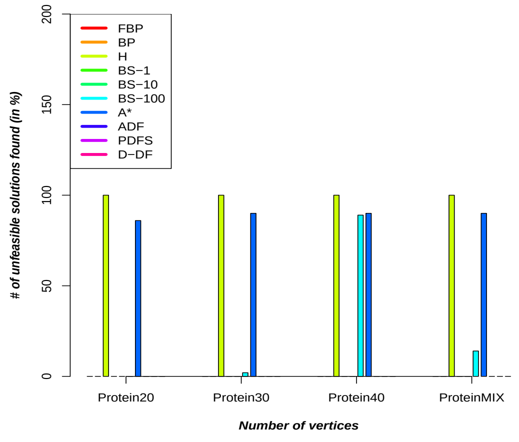
(b) Matching dissimilarity



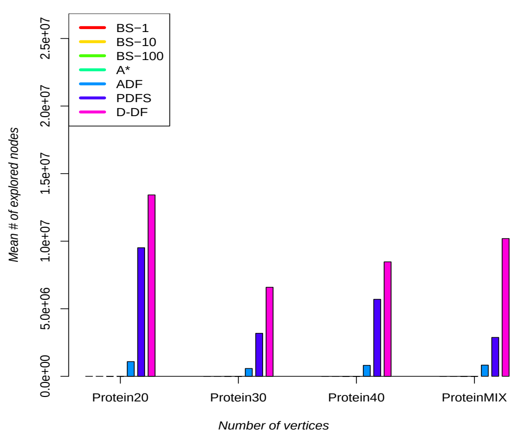
(c) Number of best found solutions



(d) Number of optimal solutions

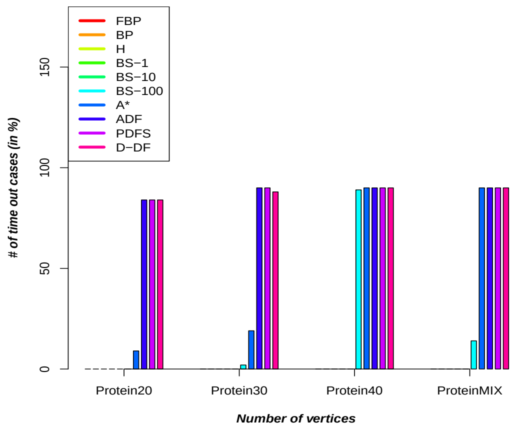


(e) Number of unfeasible solutions

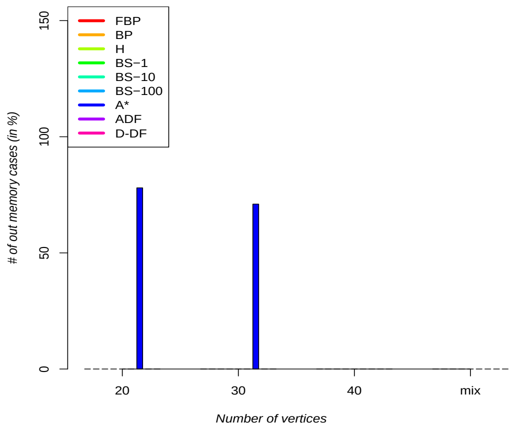


(f) Number of explored nodes

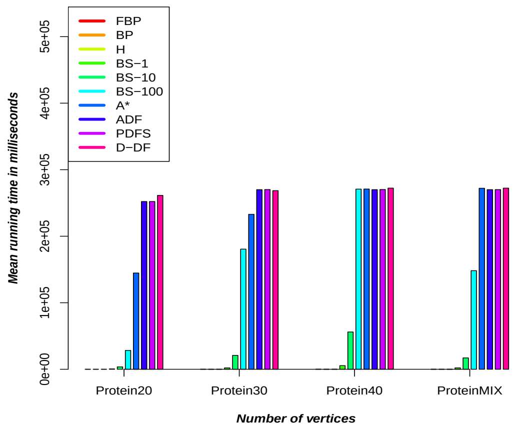
Figure C.8: Test under soft time constraints: Results on Protein



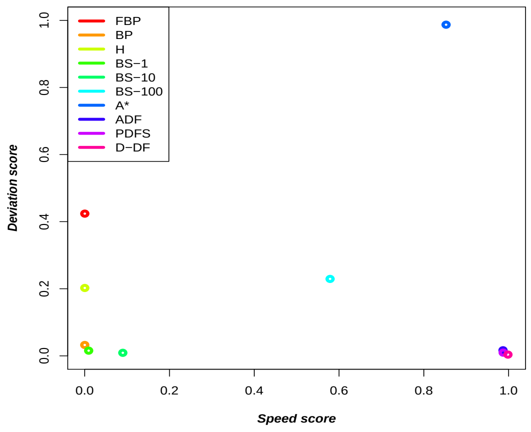
(a) Number of time-outs



(b) Number of out-of-memory cases

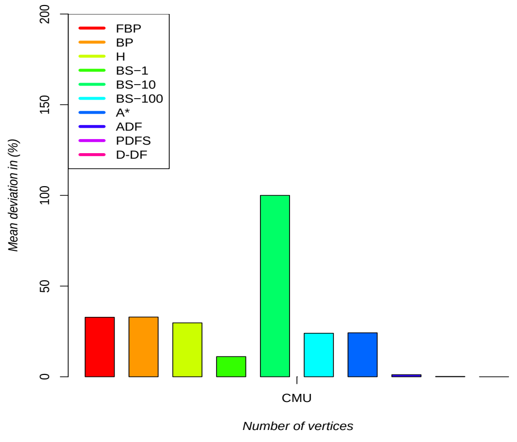


(c) Run time

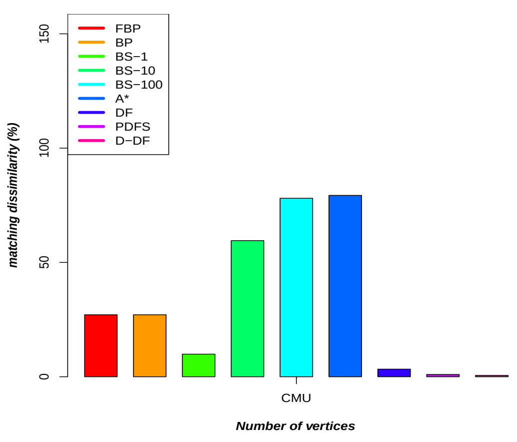


(d) Time-deviation scores

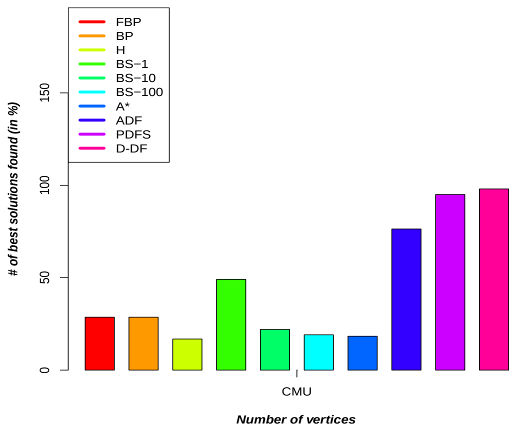
Figure C.9: Test under soft time constraints: Results on Protein



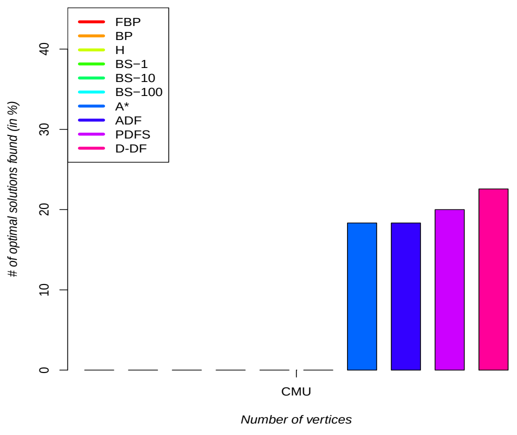
(a) Deviation



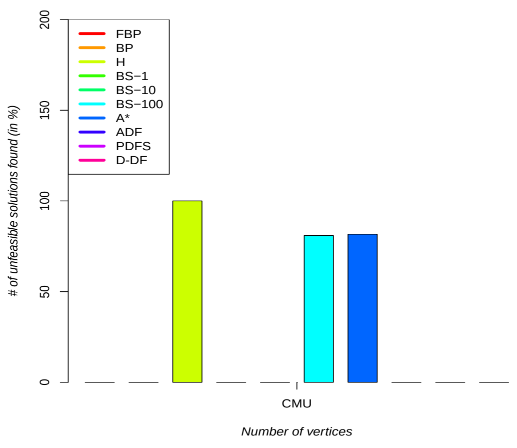
(b) Matching dissimilarity



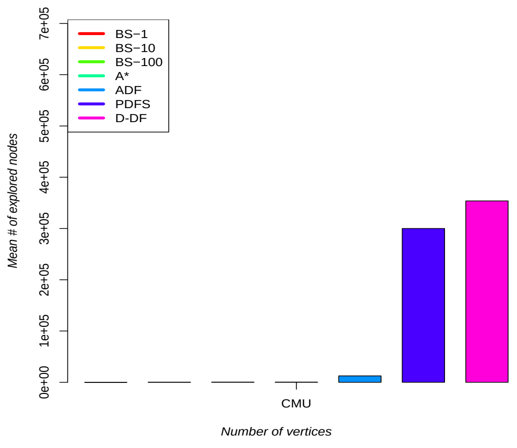
(c) Number of best found solutions



(d) Number of optimal solutions

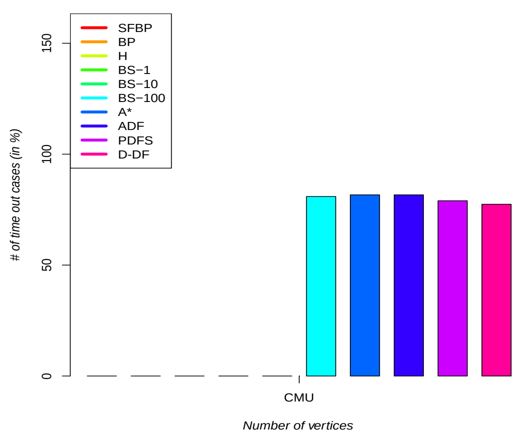


(e) Number of unfeasible solutions

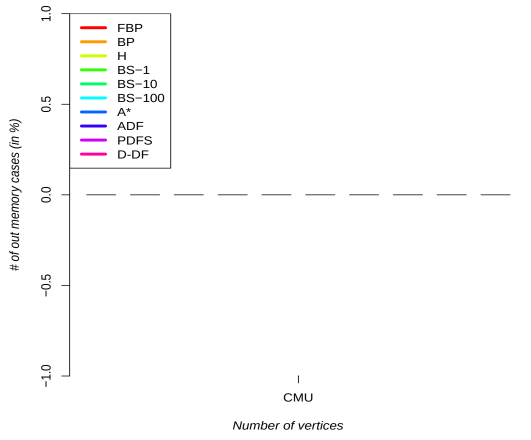


(f) Number of explored nodes

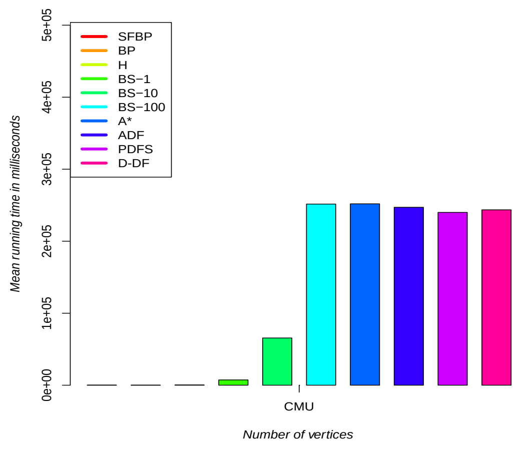
Figure C.10: Test under soft time constraints: Results on CMU



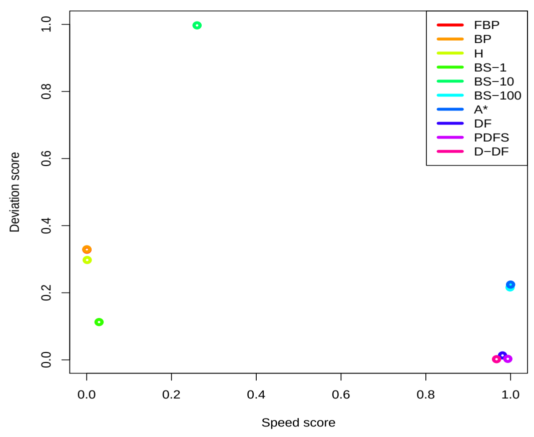
(a) Time-outs



(b) Out-of-memory



(c) Run time

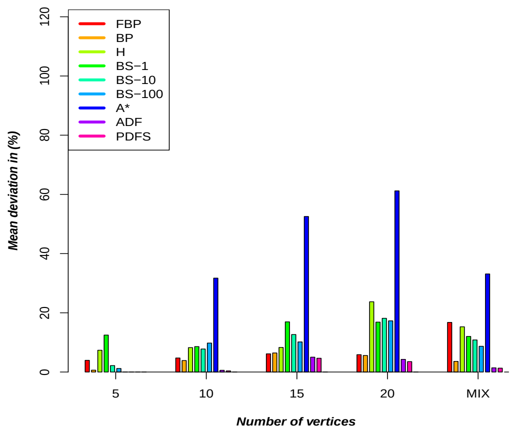


(d) Speed-deviation

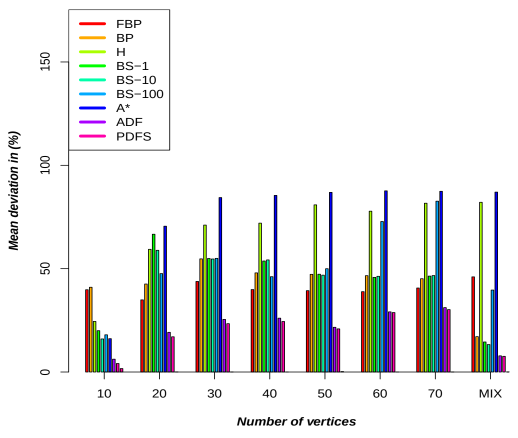
Figure C.11: Test under soft time constraints: Results on CMU

Appendix D

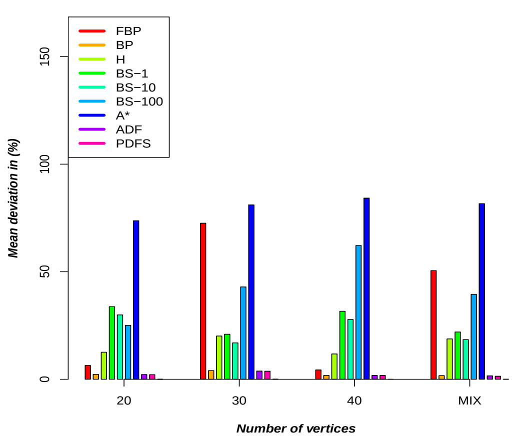
Tests under Hard Time Constraints



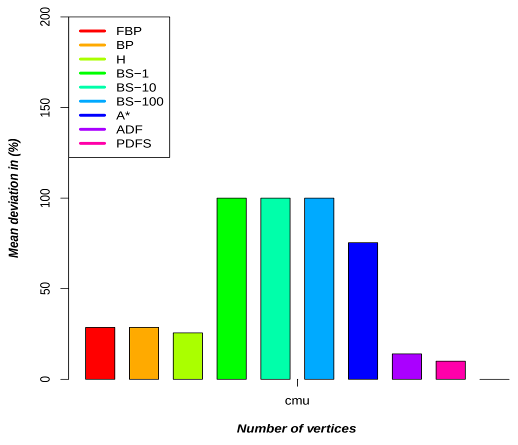
(a) GREC



(b) MUTA

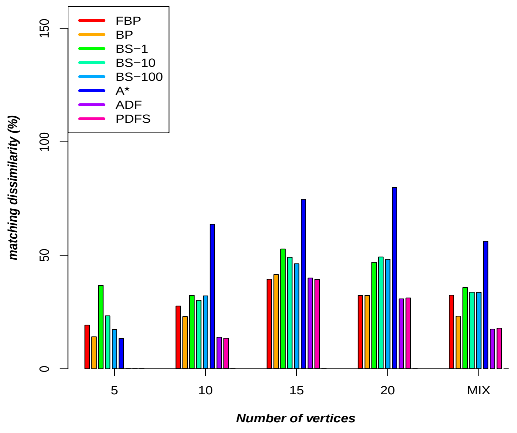


(c) Protein

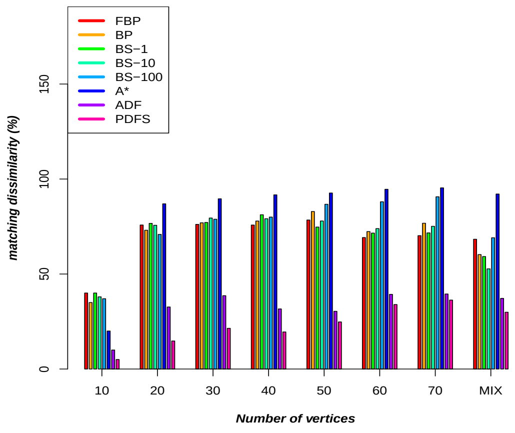


(d) CMU

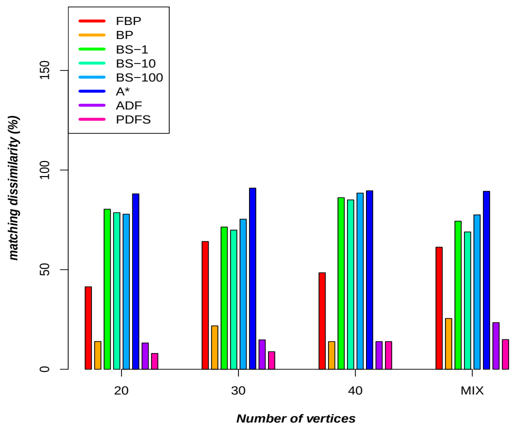
Figure D.1: Test under hard time constraints: Deviation



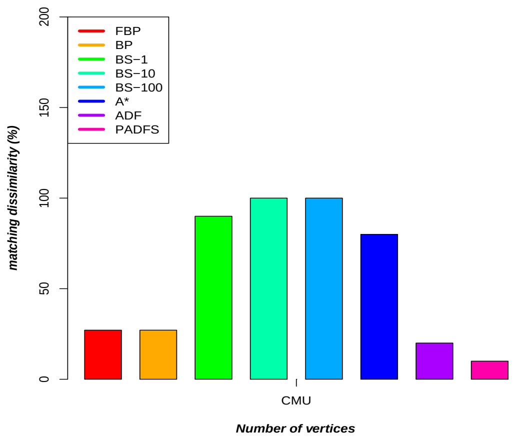
(a) GREC



(b) MUTA

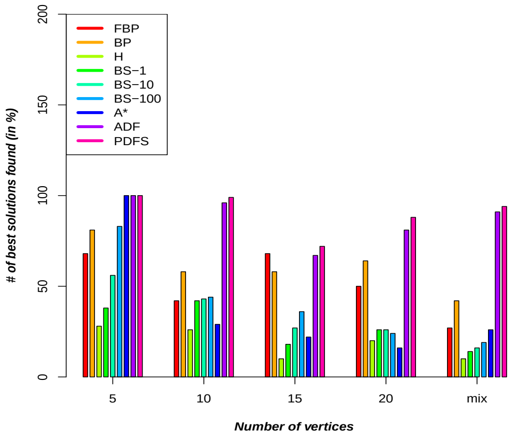


(c) Protein

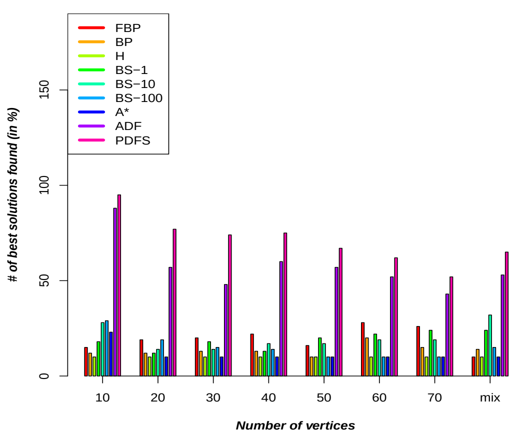


(d) CMU

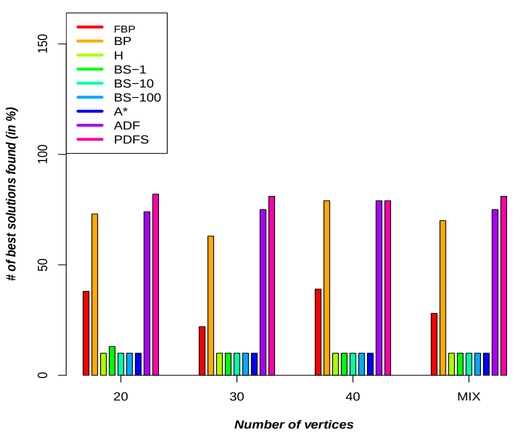
Figure D.2: Test under hard time constraints: Matching dissimilarity



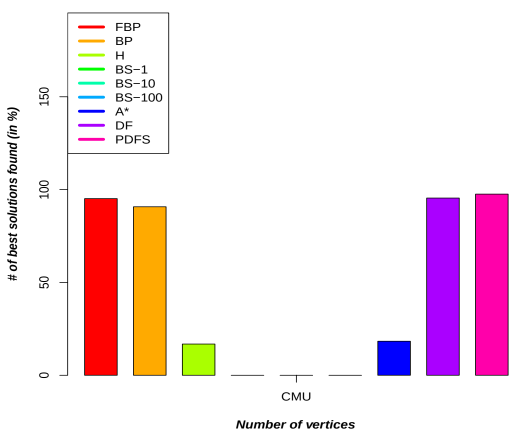
(a) GREC



(b) MUTA

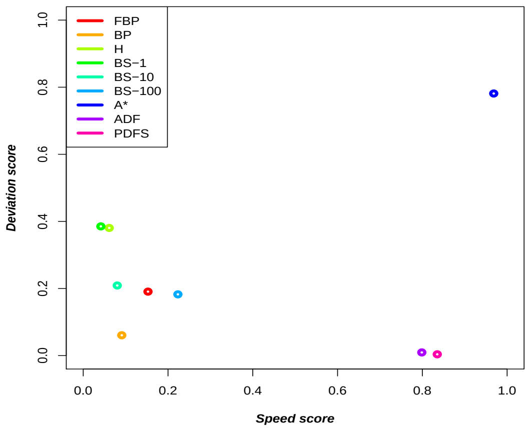


(c) Protein

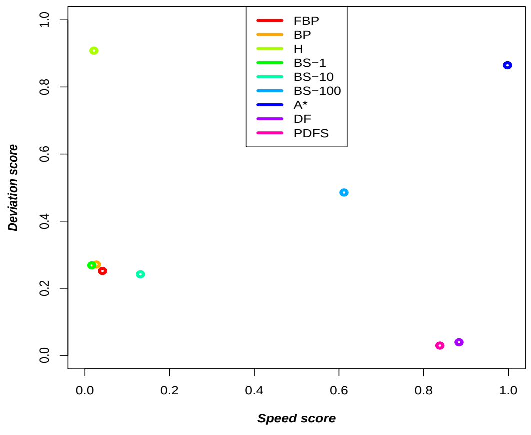


(d) CMU

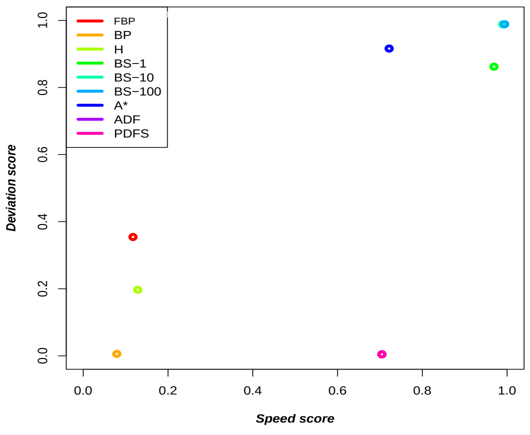
Figure D.3: Test under hard time constraints: Number of best found solutions



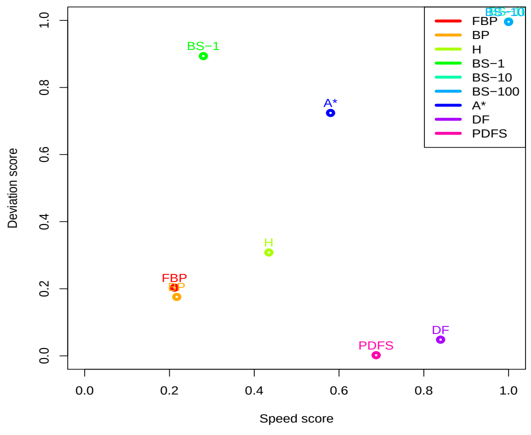
(a) GREC



(b) MUTA



(c) Protein



(d) CMU

Figure D.4: Test under hard time constraints: Time-deviation scores

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[146] 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.

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

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

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

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

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

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

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

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

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

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

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

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

List of Publications

Journal Paper

- Zeina Abu-Aisheh, Romain Raveaux and Jean-Yves Ramel: Anytime Graph Matching. (It has been submitted to the special issue of Pattern Recognition Letters on Advances in Graph-based Pattern Recognition (GBR) since November 30, 2015. ”Status: 2nd round - accepted under minor modifications”).

International Conference Contributions

- Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel and Patrick Martineau: An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems. ICPRAM (1) 2015: 271-278.
- Zeina Abu-Aisheh, Romain Raveaux and Jean-Yves Ramel: A Graph Database Repository and Performance Evaluation Metrics for Graph Edit Distance. GbRPR 2015: 138-147.
- Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel and Patrick Martineau: A Distributed Algorithm for Graph Edit Distance. GraphSM 2016.