UNIVERSIDADE FEDERAL FLUMINENSE

Leonardo Maricato Musmanno

Randomized metaheuristic-based algorithms for the generalized median graph problem

NITERÓI

2018

UNIVERSIDADE FEDERAL FLUMINENSE

Leonardo Maricato Musmanno

Randomized metaheuristic-based algorithms for the generalized median graph problem

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Doutor em Computação. Área de concentração: Algoritmos e Otimização

Orientador:
Celso Carneiro Ribeiro

NITERÓI

2018

Ficha catalográfica automática - SDC/BEE

M985r Musmanno, Leonardo Maricato
Randomized metaheuristic-based algorithms for the generalized median graph problem / Leonardo Maricato Musmanno ; Celso Carneiro Ribeiro, orientador. Niterói, 2018.
125 f. : il.

Tese (doutorado)-Universidade Federal Fluminense, Niterói, 2018.

DOI: http://dx.doi.org/10.22409/PGC.2018.d.00170516300

1. Grafo mediano generalizado. 2. Reconhecimento de padrões. 3. Produção intelectual. I. Título II. Ribeiro,Celso Carneiro, orientador. III. Universidade Federal Fluminense. Escola de Engenharia.

CDD -

Bibliotecária responsável: Fabiana Menezes Santos da Silva - CRB7/5274

Leonardo Maricato Musmanno

Randomized metaheuristic-based algorithms for the generalized median graph problem

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Doutor em Computação. Área de concentração: Algoritmos e Otimização

Aprovada em junho de 2018.

BANCA EXAMINADORA

Prof. Celso Carneiro Ribeiro - Orientador, IC-UFF

Profª. Isabel Rosseti, IC-UFF

Prof. José Viterbo Filho, IC-UFF

Profª. Luciana Salete Buriol, II-UFRGS

Prof. Rafael Augusto de Melo, IM-UFBA

Niterói

2018

Agradecimentos

Na realização do presente trabalho, contei com o apoio de várias pessoas às quais sou profundamente grato. Gostaria de deixar expresso os meus agradecimentos: ao orientador desta tese, Professor Celso Carneiro Ribeiro, pela orientação prestada, pelo seu incentivo, disponibilidade e apoio que sempre demonstrou. A convivência durante todo esse período foi decisiva para meu desenvolvimento, e para que pudesse alcançar o almejado objetivo de concluir o doutorado. Que o fim desta etapa possa ser o início de novos trabalhos conjuntos.

Ao meu irmão, Rafael Maricato Musmanno, pela sua fundamental ajuda em momentos importantes dessa caminhada. Seu inestimável apoio na arte da programação foi fundamental.

A todos os amigos e colegas que contribuíram, em especial à Philippe Leal, Tiago Santos (in memoriam) e Julliany Brandão. O apoio e as conversas que tivemos ao longo do tempo trouxeram mais leveza à essa jornada.

Não poderia esquecer de agradecer à minha família por todo o apoio, pela força e pelo carinho que sempre me prestaram ao longo de toda a minha vida acadêmica. À minha esposa por ter caminhado ao meu lado, pela sua paciência, compreensão e ajuda prestada durante a elaboração do presente trabalho. Além disso, me deu um incentivo a mais para a conclusão do trabalho, o nascimento do nosso Davi.

Agradeço também aos funcionários da UFF, em especial à Teresa e Hélio, pela sua importante ajuda durante todos esses anos.

Resumo

O conceito de similaridade entre objetos é fundamental na área de reconhecimento de padrões. Na chamada "abordagem estrutural", grafos são frequentemente utilizados para representar objetos. Assim sendo, é preciso definir um meio de medir a similaridade entre dois grafos. Uma das ferramentas mais utilizadas para realizar essa medição é a distância de edição, que consiste em medir a distância entre dois grafos de acordo com o grau de distorção necessário para transformar um grafo no outro. O grafo mediano generalizado de um conjunto de grafos S é aquele que minimiza a soma das distâncias dos grafos de S a ele e que melhor captura as informações desse conjunto de grafos, podendo ser considerado um representante deste conjunto. O conceito de grafo mediano já foi aplicado com sucesso em áreas como reconhecimento de símbolos gráficos, identificação biométrica e clusterização de imagens, entre outros. No entanto, computar o grafo mediano generalizado de um conjunto de grafos é uma tarefa complexa. Por si só, a versão de decisão do problema de cálculo da distância de edição entre dois grafos é NP-Completo. Algoritmos exatos lidam apenas com grafos de tamanho relativamente pequeno, sendo de pouca utilidade prática. Nesta tese são propostos três algoritmos aproximados para o problema, um deles baseado em uma estratégia gulosa e os outros dois baseados nas meta-heurísticas GRASP e BRKGA, além de dois novos resultados teóricos, um deles servindo de base para uma variação do algoritmo BRKGA. Os resultados obtidos indicam que os algoritmos podem ser utilizados para encontrar soluções aproximadas de boa qualidade em tempos computacionais razoáveis.

Palavras-chave: Reconhecimento de padrões, correspondência entre grafos, distância de edição, algoritmos gulosos, GRASP, BRKGA, grafo mediano.

Abstract

Structural approaches for pattern recognition frequently make use of graphs to represent objects. The concept of object similarity is of great importance in pattern recognition. Therefore, it is necessary to define a way to measure the similarity between two graphs. One of the most used tools to perform this similarity comparison is the graph edit distance, which consists of measuring the distance between two graphs by the amount of distortion necessary to transform one graph into the other. The generalized median graph of a set S of graphs is the graph that minimizes the sum of the distances to the graphs in S and it is the graph that best captures the information contained in S, and may be regarded as a representative of this set. The median graph concept has been succesfully applied in areas such as graphic symbol recognition, biometric identification and image clustering, among others. However, computing the generalized median graph of a set of graphs is a complex task. By itself, the decision version of the problem of computing the graph edit distance between two graphs is NP-Complete. Exact algorithms deal only with graphs of relatively small size, which makes them of not much use in practical situations. In this thesis three approximate algorithms for the generalized median graph problem are proposed, one of them based on a greedy strategy and the other two based on the GRASP and BRKGA metaheuristics. Also, two new theoretical results are presented, one of them serving as the basis for a variant of the BRKGA heuristic. The results obtained indicate that the algorithms can be used to find approximate solutions of good quality in reasonable computational times.

Keywords: Pattern recognition, graph matching, edit distance, greedy algorithms, GRASP, BRKGA, median graph.

List of Figures

List of Tables

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Two kinds of approaches are often used in pattern recognition problems. In the case of statistical approaches, objects are represented by vectors in $\mathbb{R}^n$ representing a set of measures. The advantages of using this type of representation come from the well defined mathematical operations in vector spaces (such as sum, product or distance between two vectors) and the efficiency with which it is possible to compute these operations. However, there are disadvantages in using this vector representation. It is not possible, for example, to represent relationships between parts of an object. Besides, objects that have different sizes and complexities must be represented by vectors of the same size.

Structural approaches for pattern recognition frequently make use of graphs to represent objects. They make it possible to represent relationships between different parts of the same object or pattern. The use of graphs also allows the representation of objects with a different number of items or parts by graphs with different numbers of vertices and edges. One possible disadvantage of this approach is the fact that there is no standard mathematical structure in the graph domain. For example, there is no standard way to sum or multiply two graphs.

One step towards the combination of these two approaches consists in using graphs to represent objects and associating attributes (labels) to their vertices and edges. Hence, instead of using the tradidional definition of a graph as a set of vertices and edges, in pattern recognition we work with the concept of a *labeled graph*, which is a quadruple $G = (V, E, \mu, \upsilon)$, where V is a set of vertices, E is the set of edges, μ is a function that associates labels to the vertices and υ is a function that associates labels to the edges [15].

The concept of similarity between objects plays a fundamental role in the area of pattern re-

cognition. Therefore, it is necessary to define a means of measuring similarity between graphs, that is, we must find a way to compare graphs and calculate their degree of similarity. This process of comparing graphs is called graph matching. Conte et al.[16] have noticed that graph matching techniques have been successfully applied to many areas, such as biology and biomedicine, biometric identification, document processing, and video analysis. As an example, in [35] a graph matching identification approach was applied for the retrieval of diatoms, a unicellular algae found in water and other places. The retrieval is based on the matching of labeled grid graphs (a regular, rectangular arrangement of nodes overlayed on an image) carrying texture information of the underlying diatom. Each node of the graph is labeled with texture features describing a rectangular sub-region of the object. Based on the grid graph representation, the problem of diatom identification can be formulated as one of labeled graph matching, where the goal is to find an optimal one-to-one correspondence between the nodes of an input graph and the nodes of a graph stored in an image database. Other examples of graph matching applications include 2D and 3D image analysis [65], document processing [64], biometric identification [24, 43, 45], image databases [9, 61], and video analysis [62]; see also [36] and [67].

The graph edit distance is one of the most well-known and used tools in graph matching. The basic idea of the edit distance is to allocate costs to edit operations (insertion and deletion of vertices and insertion and deletion of edges) necessary to transform one labeled graph into another. Given a set of objects, the concept of median is often used to indicate the element that best represents the set, that is, the one that best captures the information contained in the elements. The median graph (also called set median graph) is a widely used concept when one wants to find a representative of a set of graphs. Given a set S of graphs, the median graph $\hat{G}$ is the graph that has the smallest sum of distances (SOD) to the graphs in S, i.e:

$$\hat{G} = argmin_{G' \in S} \sum_{G \in S} d(G', G)$$

where $d(G', G)$ denotes the edit distance between G' and G. The median graph of S is therefore one of its members. In addition, the generalized median graph of S is defined as the graph that has the smallest sum of distances to the graphs in S, independently of belonging to S or not. The concept of median graph has a great potential for applications, with possible uses in classification problems or in any other situation where one wants to find a representative to a set of graphs [48, 49].

1.2 Goals

The main goal of this work is to develop heuristics for computing an approximate generalized median graph for a set of graphs. The few existing exact algorithms for the problem deal with graphs with at most 20 vertices. Another goal of this work is to develop heuristics that can deal with larger graphs in reasonable computational times, without losing the quality of the solutions. In order to achieve these goals, three heuristics were proposed. One of them is based on a greedy strategy and the other two are based on the GRASP and BRKGA meta-heuristics. This work also contributes with two theoretical results, one of them serving as the basis of a variant of the BRKGA heuristic.

1.3 Organization

In Chapter 2, the basic concepts and notations that will be used throughout the work are presented. The concepts of graph and subgraph are defined and a brief discussion of exact and inexact graph matching is presented. The graph edit distance is also introduced. It will be used as the main tool for computing the similarity between two graphs. Finally, the concept of median graph is presented along with some of its properties. The chapter also contains a discussion on the complexity of the problem. In Chapter 3, we present the greedy adaptive algorithm and the GRASP based heuristic. These two algorithms are based on a theoretical result, also presented in the chapter. This proposition works in two ways: it is used to speed up the computation of the distace between two graphs and it serves as the basis for the development of a greedy rule. The instances in which both heuristics will be tested are also presented in this chapter. A statistical comparison between the two algorithms is performed and the chapter concludes with an application to a classification task based on the median graph concept. Chapter 4 presents a theoretical proposition that gives a bound to the *SOD* of a graph, with the bound being based only on the number of nodes of the graph and on the number of graphs in the set. Chapter 5 presents the BRKGA based heuristics. The framework used for the implementation of the BRKGA is described, and a variation of this heuristic, based on the theoretical proposition presented in Chapter 4, is also presented. The results of a comparison between the variants of BRKGA and GRASP are shown in the final part of the chapter. In Chapter 6, the conclusions and possible future works are presented. Appendix A presents a theoretical proposition that shows how it is possible to determine the exact generalized median graph of set of graphs in certain special cases. In Appendix B, the instances used in this work are presented.

Chapter 2

Definitions and basic concepts

In this chapter, the basic concepts for the definition of the generalized median graph problem are described. In Section 2.1, we define the concept of labeled graphs. In order to understand the origin and the importance of the problem, graph matching concepts are explained in Section 2.2. Section 2.3 gives the formal definition of the median graph and of the generalized median graph of a set of graphs. Sections 2.4 and 2.5 explore algorithms for solving the maximum common subgraph problem and the minimum common supergraph problem of two graphs, which are used as the main building blocks of the heuristics proposed in the two next sections for the generalized median graph problem.

2.1 Labeled graphs

A graph $G = (V,E)$ is defined by a set V of nodes and a set E of edges (connecting pairs of vertices), with edges possibly having weights associated to them. This definition is not always adequate to represent objects from the real world. In order to make graphs to better represent real objects, more information should be associated to its nodes and edges. With this goal in mind, we associate labels to the nodes and edges of a graph.

Definition 1 (Labeled graph) *Given a finite attribute set $\mathcal{L}$, a labeled graph $G = (V,E,\mu,\upsilon)$ is a quadruple defined by a set V of vertices, a set $E \subseteq V \times V$ of edges, a function $\mu : V \to \mathcal{L}$ that associates an attribute value in $\mathcal{L}$ to each vertex in V, and a function $\upsilon : E \to \mathcal{L}$ that associates an attribute value in $\mathcal{L}$ to each edge in E.*

Unless otherwise stated, we refer to a labeled graph in the remainder of this text simply by a graph. There is no restriction on the nature of set $\mathcal{L}$. In most cases, $\mathcal{L} = \mathbb{R}^n$ or $\mathcal{L}$ is formed

by a discrete set of labels. If necessary, $\mathcal{L}$ may also include a special label ε to indicate that no specific label is assigned to a node or edge. If all elements of the graph are labeled with ε, the graph is called *unlabeled*. A weighted graph is a special case of a labeled graph, where all nodes are labeled with ε and each edge is labeled with a real number. In this work, only simple graphs will be considered.

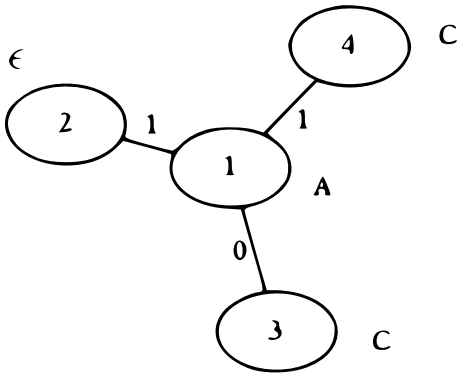


Figure 2.1: Labeled graph: nodes are identified by 1, 2, 3 and 4.

Figure 2.1 shows a labeled graph. In this case, $\mathcal{L} = \{0,1,A,B,C,\varepsilon\}$, $V = \{1,2,3,4\}$, $E = \{\{1,2\},\{1,3\},\{1,4\}\}$, $\mu(1) = A$, $\mu(2) = \varepsilon$, $\mu(3) = \mu(4) = C$, $\upsilon(\{1,2\}) = \upsilon(\{1,4\}) = 1$ e $\upsilon(\{1,3\}) = 0$.

2.2 Graph matching

In many fields, like chemistry or molecular biology, there are applications in which it is necessary to process images and locate regions in these images. When this processing is done automatically by a computer, without the help of a specialist, an efficient way to represent these images is through the use of graphs [8].

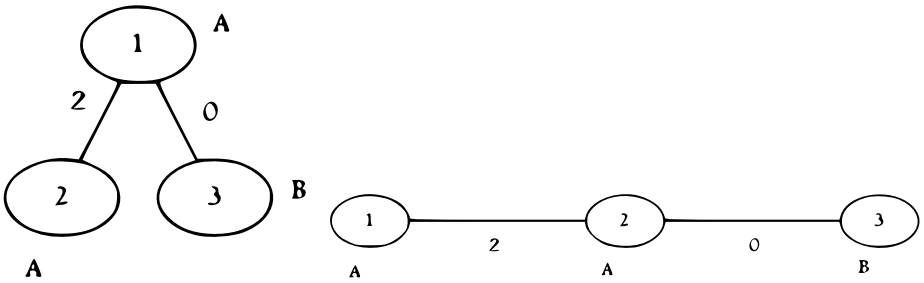
In these applications, frequently there are two images: a model and a test image. The idea is to compare these two images and verify the level of similarity between them, so that the test image can be classified correctly. Therefore, it is necessary to compare the graph representation of the model and test images. That is the main goal in graph matching: verify the similarity between the structures of two graphs. Depending on the reason for doing this matching, it is possible to make a distinction between exact graph matching and error-tolerant graph matching.

2.2.1 Exact graph matching

The goal in exact graph matching is to determine if two graphs, or parts of these two graphs, are identical in terms of their structure and labels [27]. Let $G_1 = (V_1, E_1, \mu_1, \upsilon_1)$ and $G_2 = (V_2, E_2, \mu_2, \upsilon_2)$ be two labeled graphs.

Definition 2 (Graph isomorphism) *G_1 and G_2 are said to be isomorphic if there is a bijection $f : V_1 \to V_2$ such that: (i) $\mu_1(u) = \mu_2(f(u))$, $\forall u \in V_1$; (ii) for each edge $e_1 = (u, v) \in E_1$, there exists an edge $e_2 = (f(u), f(v)) \in E_2$ with $\upsilon_1(e_1) = \upsilon_2(e_2)$; and (iii) for each edge $e_2 = (u, v) \in E_2$, there exists an edge $e_1 = (f^{-1}(u), f^{-1}(v)) \in E_1$ with $\upsilon_1(e_1) = \upsilon_2(e_2)$.*

Figure 2.2 shows two labeled graphs G and H that are isomorphic. Function $f : V_G \to V_H$ defines the isomorphism such that $f(1) = 2$, $f(2) = 1$ e $f(3) = 3$, where V_G and V_H represent, respectively, the node sets of G and H.



(a) Graph G (b) Graph H

Figure 2.2: Graph isomorphism.

Related to the graph isomorphism problem is the subgraph isomorphism problem, in which the goal is to verify if a graph is isomorphic to a part of another graph, the maximum common subgraph and the minimum common supergraph problems. These three problems will be formally defined and explained in Sections 2.4 and 2.5.

2.2.2 Error-tolerant graph matching

The methods used in exact graph matching have very rigorous conditions, which make them inadequate for many real world problems. In real world applications, when an object or image is represented by a graph, it is possible that a distortion be introduced in the representation of the object or image (due to some error in the process of acquiring the image, for example). Therefore, because of the possibility of the distortions, the same object may have different graph representations. In cases like this, the isomorphism concept would indicate that the objects are

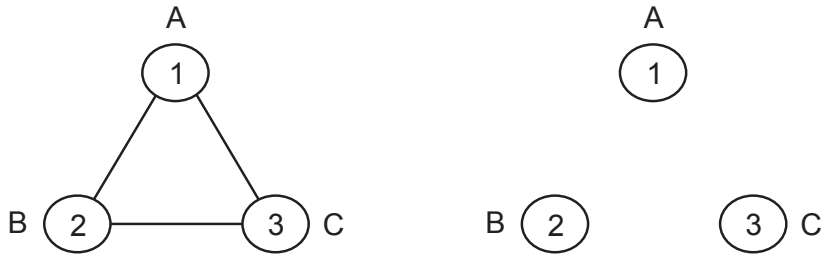
different, in spite of the great similarity between the two graphs. Therefore, the concept of exact matching is not adequate to this type of situation. Thus, it is necessary to introduce some error-tolerance in the graph matching process.

Error-tolerant graph matching focus on measuring the similarity between two graphs, instead of simply verifying wheter they have identical parts or not. It is necessary, then, to find a method to measure the dissimilarity between two graphs. One of the most used ways to compute the dissimilarity between two graphs is the graph edit distance.

The graph edit distance measures the dissimilarity between two graphs by the minimum amount of distortion required to transform one graph into the other [27]. The edit operations used to transform the graph are: insertion or deletion of nodes, insertion or deletion of edges and substitutions of nodes and edges. Given a pair of graphs G_1 and G_2, there is a sequence of edit operations (an edit path) $p(G1;G2) = (o_1;o_2;\ldots;o_m)$ that transforms G_1 into a graph that is isomorphic to G_2, where each o_i is an edit operation, $i = 1;\ldots;m$. Given two graphs G_1 and G_2, there might exist several edit paths that transform G_1 in G_2. The set composed of all these paths will be denoted by $P(G1;G2)$. In order to make a quantitative assessment of these paths, it is necessary to associate costs to the edit operations, i.e, it is necessary to define a function that associates costs to the operations of insertion, deletion and substitution of nodes and edges. The cost of the least costly path will be the distance between the two graphs. Formally, we have:

Definition 3 *Given two labeled graphs $G_1 = (V_1;E_1;\mu_1;\upsilon_1)$ and $G2 = (V_2;E_2;\mu_2;\upsilon_2)$, the edit distance between G_1 and G_2 is given by the sum of the costs of the edit operations of the least costly path that transforms G_1 in G_2.*

The computation of the graph edit distance between two graphs is ***NP***-hard [68, 69] and therefore exact algorithms to compute the edit distance can only be used for relatively small graphs. In this work, we use the graph edit distance to measure the similarity between two graphs, as proposed in [11] and used in [27]. Vertex insertions or deletions have a unit cost, while edge insertions or deletions have a null cost. The substitution of a vertex by another has a null cost if both have the same attribute, otherwise the substitution cost is assumed to be arbitrarily large. The same applies to the substitution cost of an edge by another. In [14] it was shown that, under this cost function, there is always an optimal path between two given graphs that involves only vertex and edge deletions, insertions, or substitutions with identical attributes. Figure 2.3 illustrates the computation of the graph edit distance between two labeled graphs G and H. In [52] an example of an approximate algorithm to compute this distance is presented.



(a) Labeled graph G (b) Labeled graph H

Figure 2.3: The following operations transform graph G into a graph that is isomorphic to H: delete vertex 1 from G (and remove the two edges adjacent to it), then reinsert vertex 1, next delete vertex 2 from G (and remove the edge still adjacent to it), and finally reinsert vertex 2. Deleting and reinserting vertex 1 has cost 2. The same applies to vertex 2. Therefore, the graph edit distance between G and H is $d(G,H) = 4$.

2.3 The generalized median graph problem

In this section, we formally introduce the concepts of median and generalized median graphs, followed by the formulation of the generalized median graph problem.

Definition 4 (Median graph) *Given a set $S = \{G_1, \ldots, G_n\}$ of labeled graphs defined over an alphabet $\mathcal{L}$, its median graph is a graph $\hat{G} = argmin\{\sum_{i=1}^{n} d(G, G_i) : G \in S\}$ that minimizes the sum of its distances to all graphs in S.*

The median graph $\hat{G}$ of S is necessarily one of its elements. If one seeks a representative graph that is not restricted to belonging to S, the following definition applies:

Definition 5 (Generalized median graph) *Given a set $S = \{G_1, \ldots, G_n\}$ of labeled graphs defined over an alphabet $\mathcal{L}$, its generalized median graph is any graph $\overline{G} = argmin\{\sum_{i=1}^{n} d(G, G_i) : G \in U\}$ that minimizes the sum of its distances to all graphs in S, where U is the set of all labeled graphs defined over the alphabet $\mathcal{L}$.*

Therefore, the generalized median graph $\overline{G}$ of S is any graph that minimizes the sum of its distances to the graphs in S, regardless of belonging to S or not.

The median graph can be computed in $O(n^2\delta)$, where n is the number of graphs in S and δ is the complexity of the computation of the distance function $d(.)$. Since the computation of the graph edit distance between two graphs is ***NP***-hard [68, 69], then the median graph problem considering the graph edit distance cannot be solved in polynomial time, unless ***P=NP***. Finding the median graph is ***NP***-hard even for strings [19].

The computation of the generalized median graph is even more time consuming, because it does not depend only on the complexity of the distance function $d(.)$, but also on the size of the search space U. State-of-the-art exact algorithms at the time of writing cannot deal with large graphs, with their application being restricted to small problems involving sets of graphs with no more than 25 vertices altogether. Earlier approximate approaches to find the generalized median graph include greedy [41] and genetic [44] algorithms. In [27] and [28] an exact approach for dealing with graphs with a total of up to 25 nodes was developed. In [29] Ferrer et al. proposed a new genetic algorithm for the case of a special class of graphs for which it is possible to compute the distance between any two graphs in polynomial time. This genetic algorithm can handle problem instances with up to a total of 1000 nodes. They also proposed exact and approximate approaches based on graph embeddings [30, 31]. Other approaches use the relationship between common-labeling and the median graph to compute and find bounds on the cost of the median graph [54].

The next two sections explore algorithms for solving the maximum common subgraph problem and the minimum common supergraph problem of two graphs, which are used as the main building blocks of the two heuristics proposed later in this work for the generalized median graph problem.

2.4 Maximum common subgraph problem

The concept of maximum common subgraph is central to this work. This section presents the main concepts related to it.

Definition 6 (Subgraph) *Let $G_1 = (V_1, E_1, \mu_1, \upsilon_1)$ and $G_2 = (V_2, E_2, \mu_2, \upsilon_2)$ be two labeled graphs. G_1 is said to be a subgraph of G_2 if and only if $V_1 \subseteq V_2$, $E_1 \subseteq E_2$, $\mu_1(u) = \mu_2(u)$, $\forall u \in V_1$, and $\upsilon_1(e) = \upsilon_2(e)$, $\forall e \in E_1$. In this case, we say that $G_1 \subseteq G_2$.*

When $E_1 = E_2 \cap (V_1 \times V_1)$, we say that G_1 is the induced subgraph of G_2 by V_1. Given a labeled graph $G = (V, E, \mu, \upsilon)$, a subset $V' \subseteq V$ uniquely determines an induced subgraph G' of G, usually represented by $G' = G(V')$. When G_1 is a subgraph of G_2, we say that $G_1 \subseteq G_2$.

Let $G_1 = (V_1, E_1, \mu_1, \upsilon_1)$ and $G_2 = (V_2, E_2, \mu_2, \upsilon_2)$ be two labeled graphs.

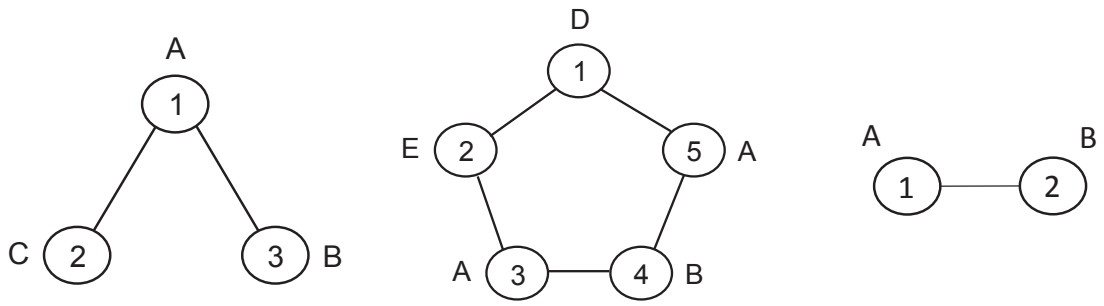
Definition 7 (Subgraph isomorphism) *There exists a subgraph isomorphism from G_1 to G_2 if there exists an induced subgraph $G \subseteq G_2$ and an injection $f : V_1 \to V_2$ defining a graph isomorphism from G_1 to G.*

Definition 8 (Common subgraph) *A labeled graph G is a common subgraph of G_1 and G_2 if and only if there is a subgraph isomorphism from G to G_1 and another from G to G_2.*

Given a graph $G = (V, E, \mu, \upsilon)$, the standard notation $|V|$ is used to represent the number of nodes of G. For ease of notation, we also use the symbol $\#(G)$ for the number of nodes of G.

Definition 9 (Maximum common induced subgraph) *A common induced subgraph G of G_1 and G_2 is maximum if all other common subgraphs of G_1 and G_2 have at most $\#(G)$ vertices.*

The decision version of the problem of finding a maximum common subgraph is proven to be ***NP***-complete [38] by a reduction from the maximum clique problem. Figure 2.4 illustrates two labeled graphs G_1 and G_2 and their maximum common subgraph. The maximum common subgraph of two graphs may be computed by a backtrack strategy [46]. Alternatively, one may seek a maximum clique in an auxiliary graph built from G_1 and G_2, which is then transformed into the maximum common subgraph [6, 22]. These three exact algorithms have the same time complexity $O((|V_2|+1)!/(|V_2|-|V_1|+1)!)$. They have been compared in [13, 17], with the numerical results indicating no clear advantage of one algorithm over the others.



(a) Labeled graph G_1 (b) Labeled graph G_2 (c) Maximum common subgraph G

Figure 2.4: Two labeled graphs G_1 and G_2 are displayed in (a) and (b), respectively. Their maximum common subgraph G is shown in (c). We denote by V_{G_1}, V_{G_2}, and V_G the vertex sets of G_1, G_2, and G, respectively. A subgraph isomorphism from G to G_1 is defined by $f : V_G \to V_{G_1}$ with $f(1) = 1$ and $f(2) = 3$. A subgraph isomorphism from G to G_2 is defined by $g : V_G \to V_{G_2}$ with $g(1) = 3$ and $g(2) = 4$.

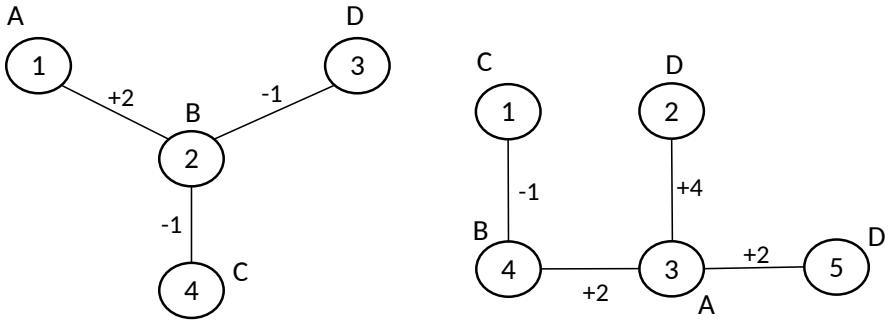
2.4.1 Algorithm for computing the maximum common induced subgraph

In the following, we use the algorithm of Durand-Pasari [22] to compute the maximum common induced subgraph of two graphs G_1 and G_2 and we denote its output by $\mathrm{MaxSub}(G_1, G_2)$. Given

the two graphs G_1 and G_2, the first step of the Durand-Pasari algorithm is the construction of an auxiliary (undirected) graph whose nodes correspond to pairs (n_1, n_2) of nodes of the two original graphs, where $n_1 \in V_1$, $n_2 \in V_2$, and $\mu_1(n_1) = \mu_2(n_2)$. Edges of the auxiliary graph represent the compatibility between pairs of nodes: the node corresponding to the pair (n_1, n_2) is connected to the node corresponding to the pair (m_1, m_2) if and only if edge (n_1, m_1) of G_1 has the same label of edge (n_2, m_2) of G_2. Each clique in the auxiliary graph corresponds to a common subgraph of G_1 and G_2 and vice versa. Therefore, the maximum common subgraph of G_1 and G_2 can be obtained by finding the maximum clique in the auxiliary graph. A common (but not necessarily maximum) subgraph of a set $S = \{G_1, \ldots, G_n\}$ of graphs may be obtained by the repeated pairwise application of the algorithm of Durand-Pasari to the graphs in S. In this case, we denote by MaxSub(S) the common subgraph so obtained.

To illustrate the functioning of the Durand-Pasari algorithm, consider graphs G_1 and G_2 shown in Figure 2.5. Its auxiliary graph is shown in Figure 2.6. The nodes in this graph are pairs (n_1, n_2), where $n_1 \in V_1$ and $n_2 \in V_2$ and $\mu_1(n_1) = \mu_2(n_2)$. For instance, pair $(1,3)$ is a node of this auxiliary graph, since node 1 belongs to V_1, node 3 belongs to V_2 and $\mu_1(1) = \mu_2(3) = A$. An edge exists between nodes $(1,3)$ and $(2,4)$ since edges $\{1,2\} \in E_1$ and $\{3,4\} \in E_2$ have the same label. Also, the Durand-Pasari algorithm considers that all non-existing edges are labeled with a null label ε. Therefore, in the auxiliary graph there is an edge between nodes $(3,2)$ and $(4,1)$, since $\{3,4\} \notin E_1$ and $\{2,1\} \notin E_2$. Every clique in this auxiliary graph corresponds to a common subgraph of G_1 and G_2, and the maximum clique corresponds to the maximum common subgraph of G_1 and G_2.

The maximum clique in this auxiliary graph is composed of nodes $(1,3)$,$(2,4)$ and $(4,1)$, as shown in Figure 2.7(a). To convert this clique into the correspondent maximum common subgraph, each node of the maximum clique of the auxiliary graph should be split in its coordinates. The first coordinate of each node will represent a node from G_1 and the second coordinate indicates a node from G_2. By selecting the first coordinates of each pair in the clique, we have that the maximum common subgraph of G_1 and G_2 is represented in G_1 by the subgraph of G_1 induced by the subset of nodes $\{1,2,4\} \in V_{G_1}$, as shown in Figure 2.7(b). In the same way, this maximum common subgraph is representend in G_2 by the subgraph of G_2 induced by the subset of nodes $\{3,4,1\} \in V_{G_2}$, as presented in Figure 2.7(c).



(a) Labeled graph G_1 (b) Labeled graph G_2

Figure 2.5: Graphs G_1 and G_2.

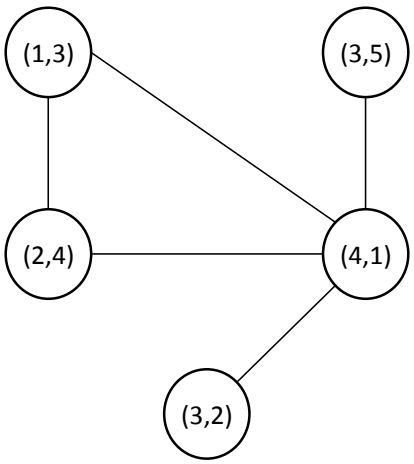
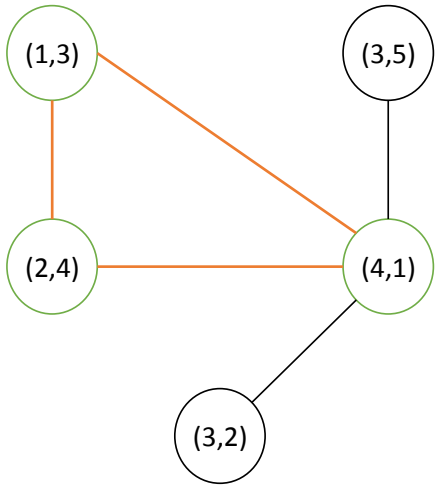
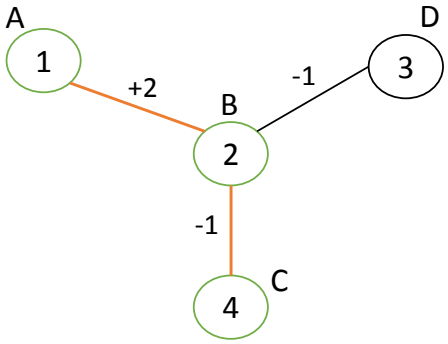


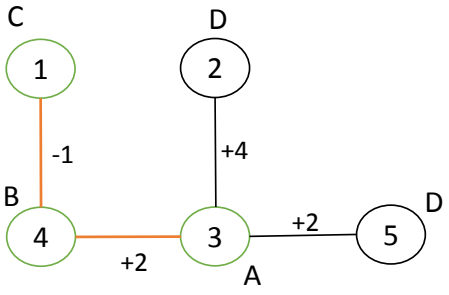
Figure 2.6: Auxiliary graph associated with graphs G_1 and G_2.



(a) Maximum clique in the auxiliary graph



(b)



(c)

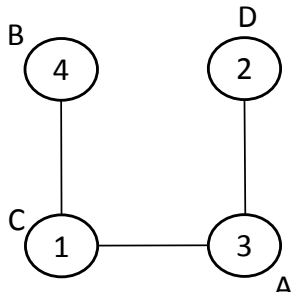
Figure 2.7: The maximum clique in the auxiliary graph corresponds to the maximum common subgraph of G_1 and G_2. This maximum common subgraph is represented in G_1 by the nodes 1,2 and 4 of V_{G_1}, as shown in (b) and by nodes 1,3 and 4 of V_{G_2} as shown in (c).

2.4.2 Graph edit distance and maximum common subgraph

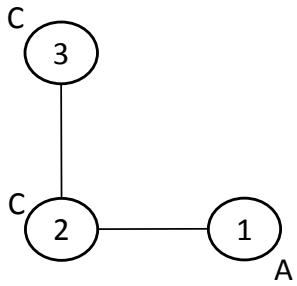
As presented in Section 2.2, as for the cost of the edit operations used in the graph edit distance, vertex insertions or deletions have a unit cost, while edge insertions or deletions have a null cost. The substitution of a vertex by another has a null cost if both have the same attribute, otherwise the substitution cost is assumed to be arbitrarily large. The same applies to the substitution cost of an edge by another. In [27] it was shown that the graph edit distance between two graphs G_1 and G_2 under the above operation costs can be computed as

$$d(G_1, G_2) = \#(G_1) + \#(G_2) - 2 \cdot \#(\text{MaxSub}(G_1, G_2)),$$

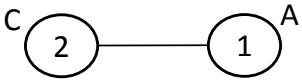
where $\#(\text{MaxSub}(G_1, G_2))$ denotes the number of vertices in the maximum common induced subgraph of G_1 and G_2 obtained by the algorithm of Durand-Pasari. This is the formula that will be used to compute all the distances between two graphs in this work. This formula illustrates the idea that, the more similar the graphs (the larger their maximum common induced subgraph is), the smaller their distance will be. As an example, consider graphs G and H in Figure 2.8. Graph G has four nodes and graph H has three nodes. The maximum common induced subgraph between G and H is the graph formed by a node with a the label A connected to another node with label C. Therefore, $\#(MaxSub(G,H)) = 2$. Thus, the distance between G and H is given by $d(G;H) = 4+3-2\times 2 = 3$. The same results can be obtained using the edit operations. The least cost path to transform graph G into H consists of deleting nodes 2 and 4 of V_G (together with the edges that are incident to them), inserting a node with a label C and inserting an edge connecting this new node to node 1 of V_G. The total cost of these operations is 3, the same result obtained using the formula.



(a) Graph G



(b) Graph H



(c) Maximum common induced subgraph of G and H

Figure 2.8: The subgraph isomorphism from $MaxSub(G,H)$ to G is given by function f : $V_{MaxSub} \rightarrow V_G$ such that $f(1) = 3$ and $f(2) = 1$. The subgraph isomorphism from $MaxSub(G,H)$ to H is given by function G such that $g(1) = 1$ and g(2) = 2. Any other common subgraph of G and H has less nodes than $MaxSub(G,H)$. Therefore, the distance between G and H is $d(G,H) = 4+3-2\times 2 = 3$, and the same result can be obtained through the edit operations.

2.5 Minimum common supergraph problem

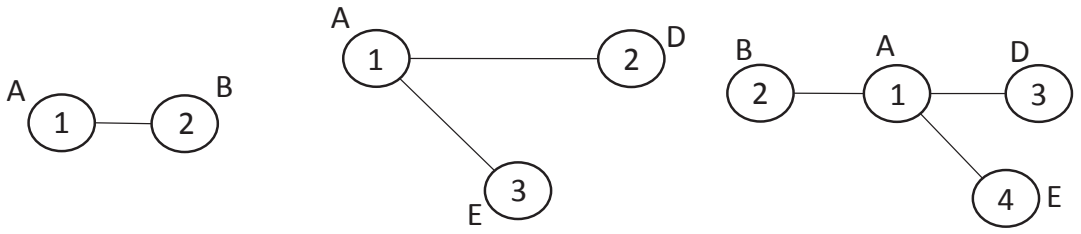
The concept of a minimum common supergraph of two or more graphs will play a fundamental role in this work. We start this section with the definition of a common supergraph of two

labeled graphs:

Definition 10 (Common supergraph) *A graph G is a common supergraph of G_1 and G_2 if and only if there is a subgraph isomorphism from G_1 to G and another from G_2 to G.*

Definition 11 (Minimum common supergraph) *A common supergraph G of G_1 and G_2 is minimum if all other common supergraphs of G_1 and G_2 have at least $\#(G)$ vertices.*

In [14] an exact algorithm to compute the minimum common supergraph of two graphs G_1 and G_2 was proposed. In the following, we denote the output of this algorithm by $\mathrm{MinSup}(G_1, G_2)$. In fact, in [14] it was shown that the minimum common supergraph problem can be solved by maximum common subgraph computations. Figure 2.9 illustrates two labeled graphs G_1 and G_2 and their minimum common supergraph. A common (but not necessarily minimum) supergraph of a set $S = \{G_1, \ldots, G_n\}$ of graphs may be obtained by the pairwise repeated application of the algorithm of Bunke [14] to the graphs in S. In this case, we denote by $\mathrm{MinSup}(S)$ the common supergraph so obtained. For more details on the computation of the minimum common supergraph, see [50].



(a) Labeled graph G_1 (b) Labeled graph G_2 (c) Minimum common supergraph

Figure 2.9: Two labeled graphs G_1 and G_2 are displayed in (a) and (b), respectively. Their minimum common supergraph G is shown in (c). We denote by V_{G_1}, V_{G_2}, and V_G the vertex sets of G_1, G_2, and G, respectively. A subgraph isomorphism from G_1 to G is defined by $f : V_{G_1} \to V_G$ with $f(1) = 1$ and $f(2) = 2$. A subgraph isomorphism from G_2 to G is defined by $g : V_{G_2} \to V_G$ with $g(1) = 1$, $g(2) = 3$, and $g(3) = 4$.

Let $S = \{G_1, \ldots, G_n\}$ be a set of labeled graphs defined over an alphabet $\mathcal{L}$. The sum of distances from a graph G to S is given by

$$\mathrm{SOD}(G, S) = \sum_{i=1}^{n} d(G, G_i),$$

where $d(G,G_i)$ is the graph edit distance between G and G_i, for $i = 1,\ldots,n$. Therefore, a generalized median graph of S is given by

$$\overline{G} = \text{argmin}\{\text{SOD}(G,S) : G \in U\},$$

where U is the set of all labeled graphs defined over the alphabet $\mathcal{L}$. The generalized median graph $\overline{G}$ of S satisfies

$$\#(\text{MaxSub}(S)) \leq \#(\overline{G}) \leq \#(\text{MinSup}(S)).$$

Furthermore, let G' be the graph induced in MinSup(S) by a subset V' of its vertices. Let G'' be any subgraph of G' with the same vertex set V'. In [27] it was demonstrated that $\text{SOD}(G',S) \leq \text{SOD}(G'',S)$. This result naturally suggests to consider the search space for the generalized median graph of S to be formed by all subgraphs that can be induced in MinSup(S) having no fewer vertices than MaxSub(S). Both the adaptive greedy algorithm and the GRASP heuristic proposed in the next sections will begin by computing MinSup(S) as their initial solution, followed by the evaluation of candidate solutions that will be subgraphs that can be induced in MinSup(S).

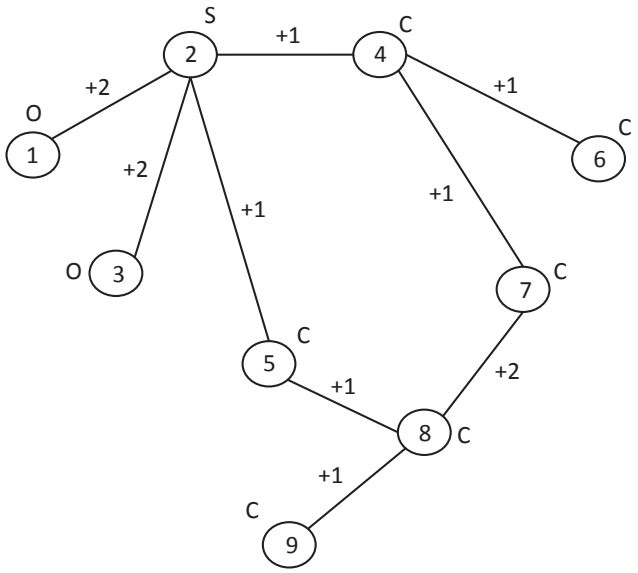
We illustrate the above definitions with one of the instances that will be used in the computational experiments that will be reported in Chapter 3. Instance i1.20 is composed of three graphs illustrated in Figure 2.10, i.e., $S = \{G_1, G_2, G_3\}$. Graph G_1 corresponds to the graph 2128 in the IAM Graph Database Repository [12, 55], G_2 corresponds to graph 6497, and G_3 to graph 13072. The total number of nodes in this instance is $\sum_{i=1}^{3} \#(G_i) = 20$.

Table 2.1 shows the distances between all pairs of graphs in S, as well as the sum of distances $\text{SOD}(G_i,S)$ from each graph G_i to S, for $i = 1,2,3$. Graph G_3 is the median graph of S, since it presents the smallest sum of distances to all other graphs in S.

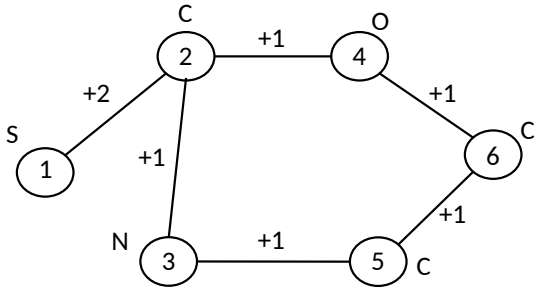
Table 2.1: Distances $d(G_i,G_j)$.

Graphs	G_1	G_2	G_3	$\sum_{j=1}^{3} d(G_i,G_j)$
G_1	0	9	8	17
G_2	9	0	7	16
G_3	8	7	0	15

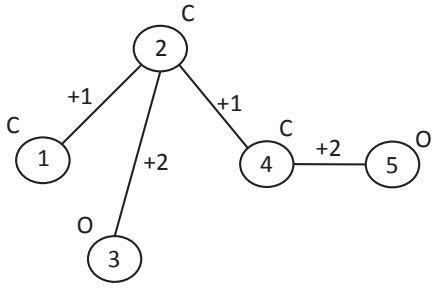
Figure 2.11 shows the maximum common subgraph MaxSub(S) of set S. It is a subgraph of G_1, G_2, and G_3, and there is no other common subgraph of them with a higher number of nodes.



(a) Graph G_1



(b) Graph G_2



(c) Graph G_3

Figure 2.10: Graphs G_1, G_2, and G_3 of set S for instance i1.20.

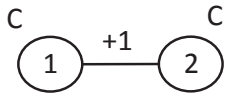


Figure 2.11: Maximum common subgraph of set S.

Figure 2.12 shows the minimum common supergraph MinSup(S). We observe that the size of the minimum common supergraph grows quickly with the number of nodes of the graphs in S.

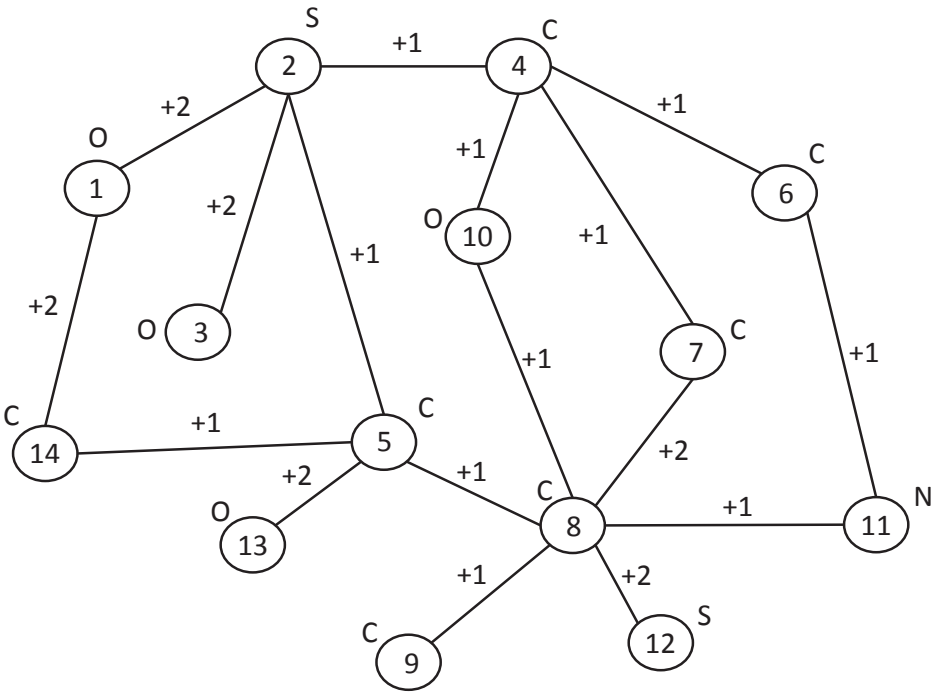


Figure 2.12: Minimum common supergraph of set S.

Since this graph is a supergraph of all three graphs in S, G_1, G_2 and G_3 are subgraphs of this graph. Figure 2.13 shows in blue the edges of graph G_2 as they appear in the minimum common supergraph and, in red, the edges of graph G_3.

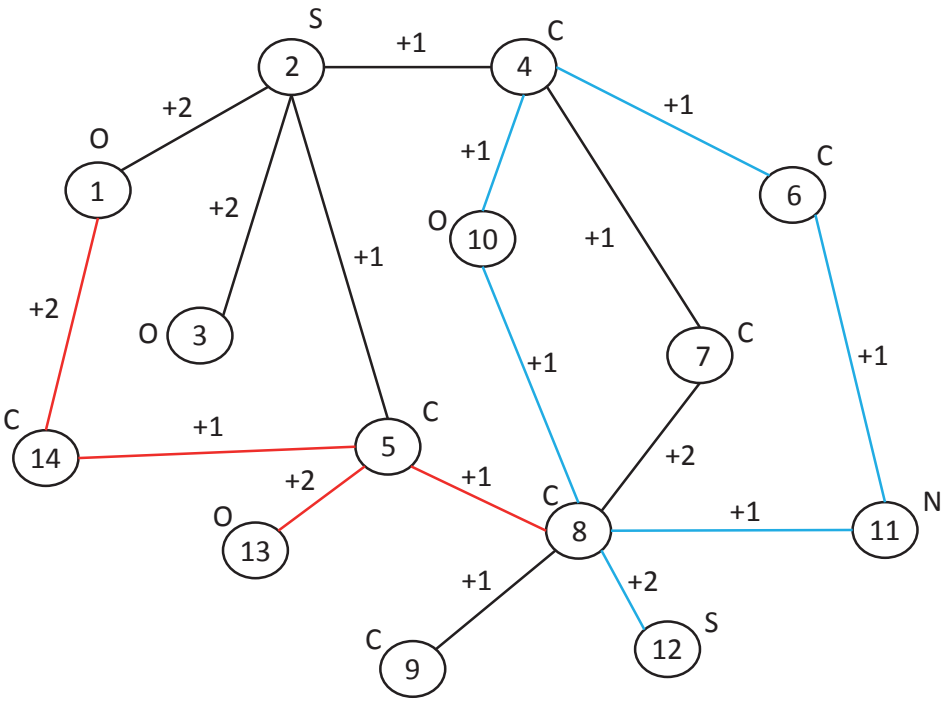


Figure 2.13: G_2 and G_3 as subgraphs of the minimum common supergraph.

The concepts presented in this chapter, especially the maximum common induced subgraph of two graphs and the minimum common supegraph of a set of graphs will have a fundamental role in the proposed heuristics.

2.6 Conclusions

In this chapter, the main concepts related to the median graph problem were presented. The formula that computes the distance between two graphs, obtained by assigning particular costs to edit operations, was also presented. The following chapter presents the first heuristics developed for the computation of approximate generalized median graphs.

Chapter 3

GRASP heuristic

In this chapter, we present two of the heuristics developed for finding an approximate generalized median graph of a set of graphs. An adaptive greedy algorithm is presented in Section 3.1, while a GRASP heuristic is described in Section 3.2. Computational experiments reporting experimental results indicating that the proposed heuristics can be used to obtain good approximate solutions for the generalized median graph problem in reasonable computation times are presented in Section 3.3. These results show that good approximations to the generalized median graph can be effectively computed by the heuristics proposed in this chapter, making it a better representation than the median graph to be used in a number of relevant pattern recognition applications. Section 3.4 evaluates the statistical significance of the results obtained by the heuristics. Section 3.5 illustrates an application of the generalized median graph related to a classification problem, whose numerical results support the conclusion that the generalized median graph is a good representative of graph sets for some pattern recognition or machine learning problems.

3.1 Greedy adaptive algorithm

The computation of the graph edit distance

$$d(G,H) = \#(G) + \#(H) - 2 \cdot \#(\text{MaxSub}(G,H))$$

between two graphs G and H is expensive, since it depends on the computation of their maximum common subgraph. Therefore, in the computation of the generalized median graph one is interested in avoiding maximum common subgraph computations whenever possible.

For the sake of explaining the evaluation of neighbors in the heuristics to be presented, we

assume that the distance $d(G,H)$ between G and H has been already computed. Let $G^{(-v)}$ be the graph obtained from G by removing any of its vertices v. Musmanno [50] has shown that if v is not a vertex of MaxSub(G,H), then $d(G^{(-v)},H) = d(G,H) - 1$. Otherwise, $d(G^{(-v)},H) = d(G,H) \pm 1$. In the last case, $d(G^{(-v)},H) = d(G,H) - 1$ if and only if G and H have another maximum common subgraph with the same number of vertices of MaxSub(G,H) that does not contain v.

Let MinSup(S) be the minimum common supergraph of the set of graphs $S = \{G_1,\ldots,G_n\}$. Since MinSup(S) is a supergraph of each G_i, then

$$\text{MaxSub}(\text{MinSup}(S),G_i) = G_i$$

for each $i = 1,\ldots,n$. Furthermore,

$$\text{SOD}(\text{MinSup}(S),S) = \sum_{i=1}^{n} d(\text{MinSup}(S),G_i).$$

Let MinSup$(S)^{(-v)}$ be the graph obtained by removing vertex v from MinSup(S) and $L^v = \{i = 1,\ldots,n : v \in \text{MaxSub}(G_i,\text{MinSup}(S))\}$. Assuming that the removal of vertex v increases the edit distance by 1 for each $G_i : i \in L^v$, we obtain the following estimate for SOD(MinSup$(S)^{(-v)},S$):

$$\overline{\text{SOD}}(\text{MinSup}(S)^{(-v)},S) =$$

$$= \text{SOD}(\text{MinSup}(S),S) - (n - |L^v|) + |L^v| =$$

$$= \text{SOD}(\text{MinSup}(S),S) - n + 2\cdot|L^v|.$$

A greedy algorithm for the generalized median graph problem is obtained by removing the vertices v of MinSup(S) one by one in the decreasing order of the estimates $\overline{\text{SOD}}(\text{MinSup}(S)^{(-v)},S)$. The best solution is updated whenever the removal of a vertex improves the incumbent.

A greedy adaptive algorithm can be derived from this greedy algorithm. In this case, every time a vertex is removed from the current solution, all estimates are recomputed from the new solution and the next vertex to be removed will be the one with the smallest updated estimate. The algorithm stops when all candidate vertices have been considered and examined for elimination. The pseudo-code for this algorithm is presented in Figure 3.1. In line 1, the minimum common supergraph of S is set as the CurrentSolution, and its *SOD* is set as the CurrentSOD in line 2. In line 3 the set Candidates is composed of all the nodes in CurrentSolution. Then, in line 4, for each vertex v in $MinSup(S)$, the estimate of the reduction on the *SOD* of $MinSup(S)$ when vertex v is removed, $\overline{SOD}(MinSup(S)^{(-v)},S)$, is computed. In lines 5 through 14, the

```
begin GREEDY-ADAPTIVE
1   Set CurrentSolution ← MinSup(S);
2   Set CurrentSOD ← SOD(MinSup(S), S);
3   Let Candidates be the vertex set of CurrentSolution;
4   Compute the estimate SOD(MinSup(S)^(−v), S) for each vertex v of MinSup(S);
5   while Candidates ≠ ∅ do;
6       u ← argmin{SOD(CurrentSolution^(−v), S) : v ∈ Candidates};
7       Let Solution be the graph obtained by removing vertex u from CurrentSolution;
8       if SOD(Solution, S) < SOD(CurrentSolution, S) then
9           CurrentSOD ← SOD(Solution, S);
10          CurrentSolution ← Solution;
11      end-if;
12      Candidates ← Candidates \ {u};
13      Update SOD(CurrentSolution^(−v), S) for each vertex v of CurrentSolution;
14  end-while;
15  return CurrentSolution;
end GREEDY-ADAPTIVE.
```

Figure 3.1: Pseudo-code of the greedy adaptive algorithm.

algorithm executes the loop that tentatively removes all nodes from Candidates. In line 6, the node that minimizes the estimate $\overline{SOD}(MinSup(S)^{(-v)}, S)$ is selected and stored in u. In line 7, Solution is the graph obtained by removing node u from CurrentSolution. In lines 8-11, CurrentSolution is updated to Solution if Solution has a better *SOD* than CurrentSolution. The set Candidates is updated with the removal of node u in line 12, and for each node in CurrentSolution the estimates $\overline{SOD}(CurrentSolution^{(-v)}, S)$ are computed in line 13. Finally, in line 15 the CurrentSolution is returned.

3.2 GRASP heuristic

GRASP (Greedy Randomized Adaptive Search Procedure) [25, 26] is a multi-start metaheuristic, in which each iteration consists of two phases: construction and local search. The construction phase builds a solution. We assume that if this solution is not feasible, then either a repair procedure is applied to achieve feasibility or a new attempt to build a feasible solution is made. Once a feasible solution is obtained, its neighborhood is investigated until a local minimum is found during the local search phase. The best overall solution is kept as the result. Literature surveys are presented in [33, 56, 57, 58, 59]. Extensive accounts of successful applications of GRASP are reported by Festa and Resende [32, 34], see also Nguyen et al.[53]. In some cases, GRASP can be combined with other metaheuristics or mixed integer programming approaches

```
begin GRASP(MaxIterations, Seed)
1   BestSOD ← ∞;
2   for k = 1, ..., MaxIterations do;
3       Solution ← GREEDY-RANDOMIZED-ADAPTIVE(Seed);
4       Solution ← LOCAL-SEARCH(Solution);
5       if SOD(Solution, S) < BestSOD then
6           BestSOD ← SOD(Solution, S);
7           BestSolution ← Solution;
8       end-if;
9   end-for;
10  return BestSolution;
end GRASP.
```

Figure 3.2: Pseudo-code of the GRASP heuristic.

(see e.g. [4], where GRASP is used together with ILS and simulated annealing to solve a vehicle routing problem). The pseudo-code in Figure 3.2 illustrates the main blocks of a GRASP procedure for minimization, in which *MaxIterations* iterations are performed and *Seed* is used as the initial seed for the pseudo-random number generator.

GRASP is an appropriate approach for tackling the generalized median graph problem because the greedy adaptive algorithm proposed in Section 3.1 can be straightforwardly randomized.

Figure 3.3 illustrates the pseudo-code of the greedy randomized adaptive algorithm used in the construction phase of the GRASP heuristic. Basically, this algorithm is an extension of the greedy adaptive algorithm in Figure 3.1, in which the node selected for elimination at each iteration is randomly chosen among those with the best SOD estimates, but not necessarily the best one. Let *RCL* be a restricted candidate list formed by all vertices v in the current solution for which $\overline{\text{SOD}}(\text{CurrentSolution}^{(-v)}, S) \in [\text{S}_{\min}, \text{S}_{\min} + \alpha \cdot (\text{S}^{max} - \text{S}_{min})]$, where $\text{S}_{\min} = \min \{\overline{\text{SOD}}(\text{CurrentSolution}^{(-v)}, S) : v \in \text{Candidates}\}$, $\text{S}_{\max} = \max\{\overline{\text{SOD}}(\text{CurrentSolution}^{(-v)}, S) : v \in \text{Candidates}\}$, and $\alpha \in [0,1]$ is a threshold parameter that controls the amounts of greediness and randomness in the algorithm. The case $\alpha = 0$ corresponds to a pure greedy algorithm, while $\alpha = 1$ is equivalent to a random construction. Experiments for tuning parameter α consisted of executing GRASP over 40 small to medium sized instances, using different values $\alpha = 0.05$, 0.1, 0.2, 0.3, 0.4, and 0.5. Since no significant difference was observed in the numerical results, in the next section we report numerical experiments performed with $\alpha = 0.1$.

The solutions generated by a greedy randomized construction are not necessarily optimal, even with respect to simple neighborhoods. A local search algorithm works iteratively by suc-

begin GREEDY-RANDOMIZED-ADAPTIVE
1 Set CurrentSolution $\leftarrow$ MinSup(S);
2 Set CurrentSOD $\leftarrow$ SOD(MinSup(S), S);
3 Let Candidates be the vertex set of CurrentSolution;
4 Compute the estimate $\overline{\text{SOD}}(\text{MinSup}(S)^{(-v)}, S)$ for each vertex v of MinSup(S);
5 **while** Candidates $\neq \emptyset$ **do**;
6 $RCL \leftarrow \{v \in \text{Candidates} : \text{S}_{\min} \leq \overline{\text{SOD}}(\text{CurrentSolution}^{(-v)}, S) \leq \text{S}_{\min} + +\alpha \cdot (\text{S}^{max} - \text{S}_{min})\}$;
7 Randomly select a vertex $u \in RCL$;
8 Let Solution be the graph obtained by removing vertex u from CurrentSolution;
9 **if** SOD(Solution, S) $<$ SOD(CurrentSolution, S) **then**
10 CurrentSOD $\leftarrow$ SOD(Solution, S);
11 CurrentSolution $\leftarrow$ Solution;
12 **end-if**;
13 Candidates $\leftarrow$ Candidates $\setminus \{u\}$;
14 Update $\overline{\text{SOD}}(\text{CurrentSolution}^{(-v)}, S)$ for each vertex v of CurrentSolution;
15 **end-while**;
16 **return** CurrentSolution;
end GREEDY-RANDOMIZED-ADAPTIVE.

Figure 3.3: Pseudo-code of the greedy randomized adaptive algorithm used in the construction phase.

cessively replacing the current solution by a better solution in its neighborhood. It terminates when no better solution is found in the neighborhood. Figure 3.4 illustrates the pseudo-code of a local search procedure. Each node belonging to MinSup(S), but not to the solution constructed in the first phase, is considered for insertion in the current solution. If the insertion of a new node improves the current solution, then it is inserted and the current solution is modified. Otherwise, another node is tested for insertion.

3.3 Experimental results

All algorithms described in the previous sections have been implemented in C++ and compiled with the compiler gcc (TDM-2 mingw31) 4.1.1. All computational experiments have been carried out on an Intel i5 2.8 GHz quadcore processor with 4 GB of RAM memory running under Windows 7 Home.

Test problems have been extracted from the AIDS group of graphs from the IAM Graph Database Repository [12]. The algorithms were tested on 100 randomly chosen instances, divided into ten test sets. Each test set contains ten instances of the same size, where the size of

```
begin LOCAL-SEARCH(Solution)
1   Set CurrentSolution ← Solution;
2   Set CurrentSOD ← SOD(Solution,S);
3   Let V' ← V_MinSup(S) − V_CurrentSolution, where V_G denotes the vertex set of a graph G;
4   for each v ∈ V' do;
5       if SOD(CurrentSolution^(+v), S) < CurrentSOD then
6           CurrentSOD ← SOD(CurrentSolution^(+v), S);
7           CurrentSolution ← CurrentSolution^(+v);
8           V' ← V_MinSup(S) − V_CurrentSolution;
9       end-if;
10  end for-each;
11  return CurrentSolution;
end LOCAL-SEARCH.
```

Figure 3.4: Pseudo-code of the local search phase.

an instance is characterized by the total number of vertices in the graphs within its input graph set S. We considered test sizes of 20, 40, 60, 80, 100, 120, 140, 160, 180 and 200 vertices altogether.

The numerical results obtained are illustrated in Tables 3.1 to 3.4. For each instance, we display its total number of vertices, the number of vertices $\#(\text{MinSup}(S))$ in $\text{MinSup}(S)$, the sum of distances $\text{SOD}(\text{MinSup}(S),S)$ from $\text{MinSup}(S)$ to S, the sum of distances $\text{SOD}(\hat{G},S)$ from the median graph $\hat{G}$ to S, the sum of distances from the generalized median graph $\overline{G}$ obtained by the adaptive greedy algorithm to S and the corresponding computation time in seconds, the sum of distances from the best generalized median graph obtained by the GRASP heuristic, the time taken by the GRASP heuristic to find the best solution and the iteration in which it was obtained, and, finally, the total computation time in seconds. The number of vertices in $\text{MinSup}(S)$ gives an upper bound to the number of vertices in the best solution (it also provides an upper bound to the number of estimates of the sum of distances that will have to be computed at each iteration of the adaptive greedy algorithm). Furthermore, $\text{SOD}(\text{MinSup}(S),S)$ and $\text{SOD}(\hat{G},S)$ give upper bounds to the sum of distances from the minimum cost generalized median graph to S.

The GRASP heuristic was run for 100 iterations for all instances. The solutions obtained by GRASP were at least as good as those found by the adaptive greedy algorithm for all test problems. Furthermore, GRASP found strictly better solutions than the adaptive greedy algorithm for 57 instances over all 100 test problems. The advantage of GRASP in terms of solution quality increases with the problem size: GRASP found strictly better solutions than the adaptive greedy algorithm for 35 out of the largest 50 instances with 120 to 200 vertices.

Table 3.5 displays, for each test set (formed by ten instances each, all of them with the same total number of vertices in the graphs within its input graph set S), the average reductions between the solution values $\mathrm{SOD}(\overline{G},S)$ obtained by the adaptive greedy algorithm and by the GRASP heuristic with respect to the sum of distances $\mathrm{SOD}(\mathrm{MinSup}(S),S)$ from $\mathrm{MinSup}(S)$ to S and to the sum of distances $\mathrm{SOD}(\hat{G},S)$ from the median graph $\hat{G}$ to S, showing by how much the proposed heuristics are able to improve, respectively, the initial solution and the upper bound given by the median graph.

Table 3.1: Results for the instances with 20, 40, and 60 vertices (100 GRASP iterations).

			Adaptive greedy			GRASP			
Instance	Vertices	#(MinSup(S))	SOD(MinSup(S),S)	SOD($\hat{G}$,S)	SOD($\overline{G}$,S)	Time (s)	Best SOD	Time to best (s)	Iteration to best	Time (s)
i01	20	14	22	15	12	0.24	12	0.30	1	40.49
i02	20	15	40	22	20	0.15	20	0.20	1	31.83
i03	20	18	52	22	20	0.26	20	0.36	1	34.36
i04	20	13	32	14	14	0.17	14	0.20	1	31.41
i05	20	11	24	10	10	0.18	10	0.21	1	33.77
i06	20	14	36	18	18	0.15	18	0.20	1	34.03
i07	20	11	24	16	16	0.16	16	0.20	1	22.43
i08	20	16	60	23	20	0.12	20	0.16	1	11.94
i09	20	16	60	24	20	0.12	20	0.15	1	12.54
i10	20	14	50	22	19	0.14	19	0.19	1	16.19
i01	40	22	92	38	36	0.81	34	5.56	4	132.10
i02	40	23	98	38	32	0.57	30	1.50	1	123.72
i03	40	23	98	40	38	0.78	38	0.89	1	129.37
i04	40	22	92	36	32	0.51	30	4.27	4	104.81
i05	40	22	92	36	32	0.49	30	2.15	2	100.24
i06	40	21	107	39	33	0.40	33	0.52	1	89.29
i07	40	19	93	38	34	0.35	34	0.48	1	74.75
i08	40	18	86	34	31	0.57	31	0.79	1	107.23
i09	40	24	152	50	40	0.40	40	0.53	1	85.67
i10	40	22	136	38	36	0.50	36	0.65	1	91.29
i01	60	30	150	64	49	2.85	48	8.98	2	389.06
i02	60	26	122	50	39	2.15	39	2.66	1	303.98
i03	60	28	164	48	48	2.34	44	10.91	3	342.56
i04	60	29	172	54	48	1.38	46	5.07	2	229.10
i05	60	20	200	46	45	0.63	45	0.83	1	171.80
i06	60	27	183	58	51	0.79	51	1.03	1	167.93
i07	60	24	180	52	48	1.13	48	1.44	1	203.97
i08	60	23	193	52	47	0.74	45	9.29	5	182.83
i09	60	22	182	50	48	0.73	47	1.87	1	172.52
i10	60	32	324	76	60	0.63	60	0.85	1	97.01

Table 3.2: Results for the instances with 80, 100, and 120 vertices (100 GRASP iterations).

			Adaptive greedy			GRASP			
Instance	Vertices	#(MinSup(S))	SOD(MinSup(S),S)	SOD($\hat{G}$,S)	SOD($\overline{G}$,S)	Time (s)	Best SOD	Time to best (s)	Iteration to best	Time (s)
i01	80	33	217	65	59	3.86	56	7.61	1	566.81
i02	80	32	240	70	58	2.79	58	3.50	1	396.11
i03	80	33	283	71	65	3.16	62	5.91	1	454.75
i04	80	32	272	63	55	2.26	55	2.88	1	369.62
i05	80	32	272	75	63	1.85	62	4.40	1	351.64
i06	80	35	305	78	65	3.77	63	13.91	2	641.80
i07	80	33	283	74	69	3.85	66	17.66	3	569.72
i08	80	33	283	74	64	2.46	60	4.90	1	447.64
i09	80	36	316	72	67	2.29	67	3.01	1	399.36
i10	80	35	305	72	68	2.52	67	4.85	1	415.14
i01	100	40	340	89	76	6.44	72	47.14	5	903.78
i02	100	38	356	86	78	3.97	72	14.11	2	639.07
i03	100	38	394	101	82	2.94	82	3.65	1	488.04
i04	100	38	394	96	85	6.10	83	57.96	7	812.46
i05	100	36	368	86	72	4.30	70	13.73	2	635.20
i06	100	38	394	80	70	2.98	70	3.99	1	531.55
i07	100	34	342	81	75	3.88	73	69.46	12	571.66
i08	100	37	418	88	76	3.44	74	18.72	3	575.84
i09	100	37	418	100	82	3.49	82	4.51	1	528.46
i10	100	38	432	100	86	5.16	86	6.70	1	654.59
i01	120	42	426	113	92	11.45	92	15.08	1	1552.70
i02	120	41	495	105	93	7.90	93	9.92	1	1132.39
i03	120	45	555	126	101	6.08	100	60.82	7	818.14
i04	120	41	495	102	95	7.89	94	12.03	1	968.01
i05	120	40	480	106	94	5.22	94	6.75	1	797.30
i06	120	41	536	124	100	4.14	96	390.59	58	674.28
i07	120	40	520	112	106	4.71	102	14.60	2	685.63
i08	120	37	472	102	90	4.30	86	9.32	1	722.92
i09	120	42	552	106	98	5.60	98	6.93	1	812.20
i10	120	35	440	98	86	4.54	84	16.44	2	714.82

Table 3.3: Results for the instances with 140, 160, and 180 vertices (100 GRASP iterations).

			Adaptive greedy			GRASP			
Instance	Vertices	#(MinSup(S))	SOD(MinSup(S),S)	SOD($\hat{G}$,S)	SOD($\overline{G}$,S)	Time (s)	Best SOD	Time to best (s)	Iteration to best	Time (s)
i01	140	50	710	130	117	12.15	110	22.93	1	1609.02
i02	140	44	652	122	110	9.75	104	31.50	2	1402.51
i03	140	43	677	132	124	4.92	114	20.01	2	859.21
i04	140	44	696	119	101	5.13	101	6.55	1	889.63
i05	140	38	582	112	108	4.25	104	499.43	62	806.95
i06	140	45	760	124	110	6.24	110	7.96	1	981.74
i07	140	49	889	151	124	4.91	124	6.65	1	849.57
i08	140	48	868	148	117	5.00	117	6.45	1	871.62
i09	140	47	847	135	118	5.88	115	11.54	1	987.45
i10	140	42	742	130	117	6.66	117	8.78	1	1051.93
i01	160	53	741	139	118	19.96	116	32.33	1	2841.25
i02	160	54	812	170	130	15.10	128	26.71	1	2037.39
i03	160	50	740	140	130	14.69	124	81.70	4	1919.15
i04	160	49	771	145	123	16.80	123	22.67	1	2153.52
i05	160	49	771	156	133	16.78	130	47.01	2	2164.30
i06	160	55	885	151	129	22.73	129	29.58	1	2620.99
i07	160	47	733	143	122	13.02	119	56.60	3	1740.34
i08	160	47	733	140	127	11.32	124	23.74	1	1852.80
i09	160	39	581	128	115	6.34	114	11.87	1	1102.13
i10	160	47	733	134	114	27.51	113	119.35	4	2742.63
i01	180	56	996	169	150	17.64	149	51.97	2	2231.33
i02	180	47	807	152	123	13.91	123	18.76	1	1899.15
i03	180	50	870	153	124	24.86	124	31.82	1	2811.30
i04	180	47	807	164	147	19.78	138	108.66	4	2637.01
i05	180	54	1008	162	138	19.20	134	2478.88	97	2549.65
i06	180	61	1162	174	152	13.79	150	25.82	1	1827.17
i07	180	56	1052	166	144	20.67	138	34.78	1	2892.28
i08	180	51	942	160	150	18.01	142	319.07	13	2398.33
i09	180	50	920	148	124	51.09	124	64.11	1	4825.27
i10	180	51	942	158	144	15.52	136	28.04	1	2200.90

Table 3.4: Results for the instances with 200 vertices (100 GRASP iterations).

				Adaptive greedy		GRASP			
Instance	Vertices	#(MinSup(S))	SOD(MinSup(S),S)	SOD($\hat{G}$,S)	SOD($\overline{G}$,S)	Time (s)	Best SOD	Time to best (s)	Iteration to best	Time (s)
i01	200	57	997	182	148	37.11	145	55.69	1	4187.89
i02	200	52	944	164	148	30.35	142	830.33	23	3594.42
i03	200	55	1010	166	144	29.18	140	48.40	1	3732.03
i04	200	59	1157	196	164	20.59	152	115.85	4	2722.80
i05	200	53	1019	172	150	19.36	142	34.28	1	2638.67
i06	200	55	1065	180	157	20.84	148	93.09	3	2762.97
i07	200	52	996	176	144	18.61	141	31.53	1	2523.99
i08	200	49	927	171	141	18.87	141	24.84	1	2503.30
i09	200	53	1019	183	148	18.17	147	31.58	1	2697.00
i10	200	54	1042	190	161	20.29	154	60.24	2	2631.77

Table 3.5: Average improvement in the sum of distances of the best solution found by each algorithm, with respect to the sum of distances from MinSup(S) and from the median graph $\hat{G}$ to S.

	Adaptive greedy		GRASP	
Vertices (total)	$\frac{SOD(\overline{G},S)}{SOD(MinSup(S),S)}$ (%)	$\frac{SOD(\overline{G},S)}{SOD(\hat{G},S)}$ (%)	$\frac{SOD(\overline{G},S)}{SOD(MinSup(S),S)}$ (%)	$\frac{SOD(\overline{G},S)}{SOD(\hat{G},S)}$ (%)
20	55.02	8.15	55.02	8.15
40	66.39	10.82	67.22	12.99
60	73.18	11.31	73.77	13.25
80	77.04	11.29	78.48	13.65
100	79.63	13.60	80.13	15.66
120	80.71	12.43	81.03	13.88
140	84.36	11.73	84.80	14.09
160	83.24	13.93	83.62	15.39
180	85.20	13.16	85.61	15.50
200	85.19	15.36	85.70	18.31
Average	77.00	12.18	77.54	14.09

Figure 3.5 displays the increase in the average sum of distances from the best solution to S found by each algorithm, with respect to the total number of vertices in each instance. It shows that GRASP becomes progressively better than the adaptive greedy algorithm as the problem size increases.

Figure 3.6 depicts the evolution of the best solution found along 100 GRASP iterations for instance i05.140 with a total of 140 vertices. It shows that solution quality improves with the number of iterations, i.e., with the running time. The larger is the number of iterations given to the GRASP heuristic, the better is the solution found. It also illustrates by how much the local search is able to improve the solution built by the greedy adaptive algorithm in the GRASP construction phase at each iteration. Considering all 100 instances, on average the local search phase was able to improve by approximately 1.2% the solutions built at the construction phase. Table 3.6 shows that the average relative improvement obtained by local search increases with the problem size.

We have also assessed the behavior of GRASP using the methodology proposed by [2] and the software distributed by the authors [3]. Two hundred independent runs of the heuristic have been performed for each algorithm. Each run was terminated when a solution with value less than or equal to a given target was found. Numerical results are illustrated for instance i05.140 with a total of 140 vertices, with the target value set at 106. The empirical probability distribution of the time observed to find a solution value less than or equal to the target is plotted

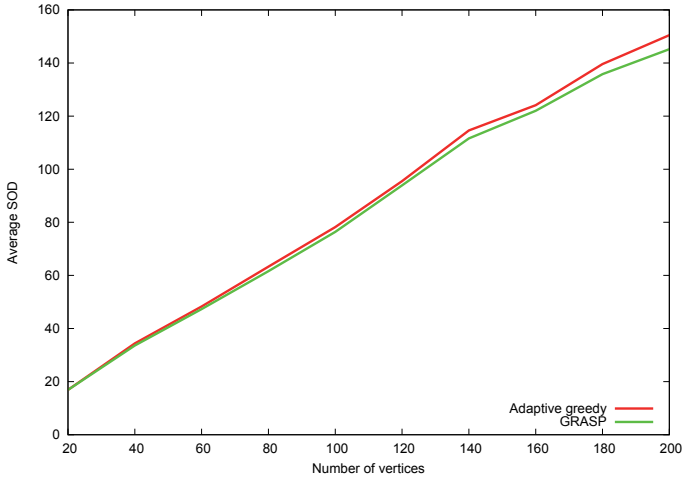


Figure 3.5: Average sum of distances from the best solution to S.

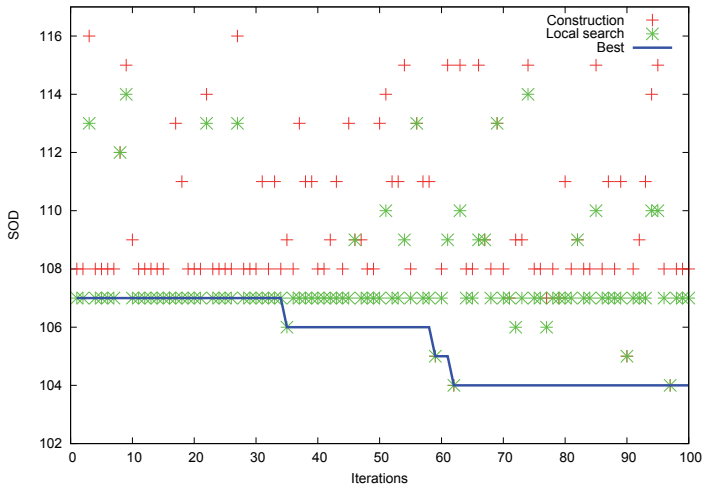


Figure 3.6: Evolution of the best solution found along 100 GRASP iterations for instance i05.140 with a total of 140 vertices.

in Figure 3.7. To plot the empirical distribution for each algorithm, we associate a probability $p_i = (i - \frac{1}{2})/200$ with the i-th smallest running time t_i and we plot the points $z_i = (t_i, p_i)$, for $i = 1, \ldots, 200$. It can be observed that the running times fit an exponential distribution.

3.4 Statistical evaluation: Nonparametric tests

Most statistical tests are based on the assumption that the random samples are taken from a population with a normal distribution. Traditionally, these procedures are called parametric methods, because they are based on a particular parametric family of distributions [47]. Nonparametric tests, also known as distribution-free methods, make no assumptions about the dis-

Table 3.6: Average improvement in percent by the local search phase.

Vertices	Improvement (%)
20	0.00
40	0.39
60	1.07
80	1.43
100	1.04
120	1.04
140	1.70
160	1.74
180	1.76
200	1.82
Average	1.20

tribution of the underlying population. In this section, we compare the results found by the two proposed heuristics using two non-parametric tests: the sign test and the Wilcoxon test.

3.4.1 Sign test

Let X_i and Y_i be, respectively, the solution values found by GRASP and by the greedy adaptive heuristic for each test problem numbered $i = 1,\ldots,100$. Since there are two populations to be compared, the sign test for paired samples will be used. Let $D_i = X_i - Y_i$ be the paired differences, for $i = 1,\ldots,100$. Testing if both populations have the same median can be done by testing if the median of their differences is null, i.e., if $\tilde{\mu}_D = 0$. A "+" sign is assigned to each positive difference, a "-" sign to each negative difference, and all ties are discarded. All paired differences are nonpositive, with 57 "-" signs, no "+" sign, and 43 ties. Since the ties are disconsidered, the sample consists of the 57 results and we test

$$H_0 : \tilde{\mu}_D = 0 \text{ and } H_1 : \tilde{\mu}_D < 0.$$

Since $n = 57 > 10$ and $p = 0.5$, the binomial distribution can be approximated by the normal distribution. The test statistic is given by $\min(57,0) = 0$. The null hypothesis H_0 can be rejected with a level of significance of 5% because

$$Z_0 = \frac{0 - (\frac{57}{2})}{\frac{\sqrt{57}}{2}} \cong -7.4$$

is less than the critical value -1.64.Therefore, we may say that the GRASP heuristic performs better with a level of significance of 5%.

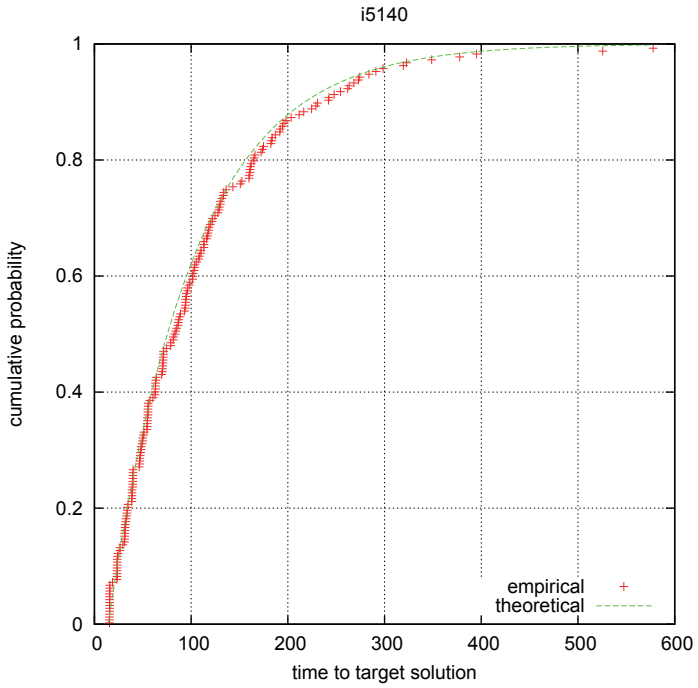


Figure 3.7: Runtime distribution from 200 runs of 100 GRASP iterations for instance i05.140 with a total of 140 vertices and target value set at 106 (the best known value is 104).

3.4.2 Wilcoxon signed-rank test

The previous test makes use only of the signs of the differences between pairs of observations. It does not take into account the magnitude of these differences. The Wilcoxon signed-rank test considers both the sign and the magnitude of these differences and also applies to the case of symmetric distributions.

As before, let X_i, Y_i, and $D_i = X_i - Y_i$ be, respectively, the solution value found by GRASP, the solution value found by the greedy adaptive heuristic, and the paired difference for each test problem numbered $i = 1, \ldots, 100$. The null hypothesis is $H_0 : \mu_X = \mu_Y$, which is equivalent to $H_0 : \mu_D = 0$. We initially consider the two-sided alternative $H_1 : \mu_X \neq \mu_Y$ (or, equivalently, $H_1 : \mu_D \neq 0$). To use the Wilcoxon signed-rank test, the differences are first ranked in ascending order of their absolute values, and then the ranks are given the signs of the differences. Ties are assigned average ranks. Let W^+ be the sum of the positive ranks and W^- be the absolute value of the sum of the negative ranks, and set $W = \min(W^+, W^-)$. If the observed value for this statistic is less than or equal to w^*_α, then the null hypothesis is rejected, where w^*_α is a critical value defined accordingly to the significance level α chosen for the experiment. For one-sided tests, if the alternative is $H_1 : \mu_D > 0$ (resp. $H_1 : \mu_D < 0$) then reject H_0 if $w^- \leq w^*_\alpha$ (resp. $w^+ \leq w^*_\alpha$).

In this case, the sum of the ranks corresponding to positive differences is $W^+ = 0$ and the sum of the ranks corresponding to negative differences is $W^- = 5.5 \times 10 + 18 \times 15 + 30 \times 9 + 39 \times 9 + 46 \times 5 + 49.5 \times 2 + 52 \times 3 + 54.5 \times 2 + 56 + 57 = 1653$. Then, we test:

$$H_0 : \tilde{\mu}_D = 0 \text{ and } H_1 : \tilde{\mu}_D < 0.$$

The test statistic is W_+, since we expect that GRASP performs better than the greedy adaptive heuristic. Since the size of the sample is large, a normal approximation can be used for this statistic [47]. Assuming H_0 is true, the normal approximation for W^+ has

$$\mu_{W^+} = \frac{n(n+1)}{4}, \sigma_{W^+} = \sqrt{\frac{n(n+1)(2n+1)}{24}} \text{ and } Z_0 = \frac{W_+ - \frac{n(n+1)}{4}}{\sqrt{\frac{n(n+1)(2n+1)}{24}}}.$$

For $n = 57$, $\mu_{W^+} = 826.5$ and $\sigma_{W^+} = 125.86$. Since

$$Z_0 = \frac{0 - 826.5}{125.86} \cong -6.58 < -1.64,$$

once again there is enough evidence to discard H_0 and we may say that the GRASP heuristic performs better with a level of significance of 5%.

3.5 Application to classification

The classification problem in machine learning consists in appointing the class that best fits to an input object, given a set of possible classes. Classification problems appear, e.g., in face detection (finding faces in images), spam filters (identifying email messages as spam or not-spam), medical diagnosis (diagnosing whether a patient suffers or not of some disease), weather prediction and others [5].

We have noticed before that the generalized median graph is the graph that best represents and summarizes the information provided by a set of graphs. Therefore, given a classification task where the objects are represented by graphs and divided in classes, it is reasonable to assume that each class can be represented by its generalized median graph. The goal of this section is to provide an experimental evaluation of this assumption and to illustrate that the algorithms proposed in this work may be appropriately used to solve some classification problems.

There are several approaches for solving classification problems, such as neural networks, Bayes classifiers, and decision trees, among others [21]. In the context of graph problems, nearest-neighbor classifiers are often used, mostly because of their simplicity, since they only require a way to measure dissimilarity between objects. Nearest-neighbor classifiers are super-

vised learning tasks based on a training set of patterns. In this training set, every pattern has a category label assigned to it. Given a test set composed of all patterns to be classified, each of its patterns is compared to all elements of the training set. The 1-nearest-neighbor classifier (1-NN) is defined by assigning a test pattern to the class of its most similar training pattern. The 1-NN classifier can be extended to consider not only the most similar pattern in the training set but, instead, the k closest patterns: in a k-NN classifier, the test pattern is assigned to the class that occurs most frequently among its k nearest or closest training patterns [27, 37]. Figure 3.8 shows an example of the application of a k-NN classifier.

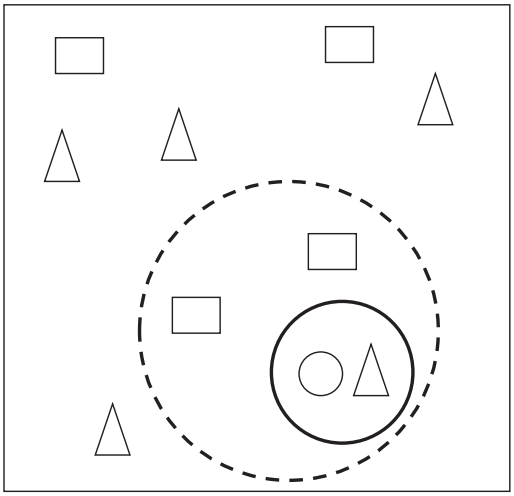


Figure 3.8: Example of k-NN classification: the test object (circle) must be classified either to the class of triangles or to the class of squares. If $k = 1$ (solid line circle), then it is assigned to the class formed by triangles. If $k = 3$ (dashed line circle), then it is assigned to the class formed by squares, since there are two squares against one triangle inside that circle.

The classification experiments reported next consisted of classifying some queries using two approaches: the k-NN classifier and generalized median graphs. The median graph approach amounts to computing an approximate generalized median graph of each class and comparing each query to these approximate generalized median graphs. The median graph approach presents the advantage that the number of comparisons is greatly reduced, since each query is compared only to a small number of graphs, while with k-NN the query is compared with every element of all classes. Figures 3.9 and 3.10 illustrate the two approaches in the case where the objects have to be classified in two classes (binary classification).

The instances used in this experiment were taken from the IAM Graph Database Repository [55]. This database consists of ten groups of graphs. Graphs in group AIDS used in this work represent molecules and are divided in two classes: active molecules and inactive molecules, depending on its relation with the AIDS virus. There are 2000 graphs in this database, divided in three sets: training set (250 graphs), test set (1500 graphs), and validation set (250

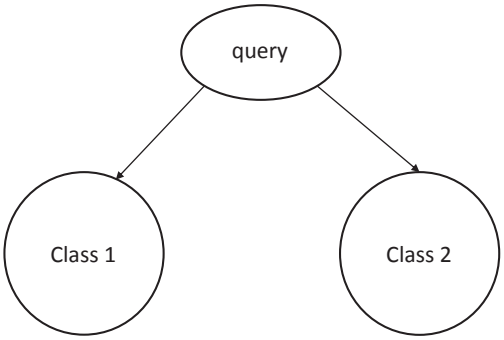


Figure 3.9: Classification using *k-NN* algorithm: query is compared with every element of classes 1 and 2.

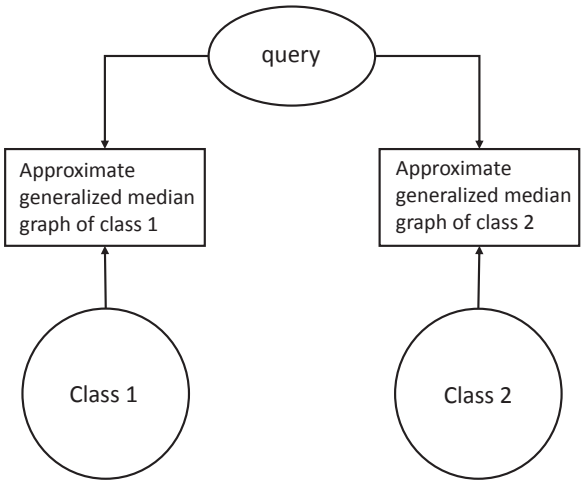


Figure 3.10: Classification using generalized median graph: query is compared only with classes' 1 and 2 median graphs.

graphs).

In order to evaluate the performance of the approximate generalized median graph for classifying objects, the following experiments were performed:

- Experiment A: 60 graphs (30 active molecules and 30 inactive molecules) were selected from the training set. Both classes had their approximate generalized median graphs computed by the GRASP heuristic. 190 graphs were selected from the test set and classified by algorithms 1-NN and 3-NN. Next, the same 190 queries have been classified using the approximate generalized median graph of each of the two classes (the distance between each query and the approximate generalized median graphs have been computed and the query was assigned to the class which showed the smallest distance).
- Experiment B: same as above, with the training set formed by 80 graphs (40 active molecules and 40 inactive molecules).

- Experiment C: same as above, with the training set formed by 100 graphs (50 active molecules and 50 inactive molecules).

The same 190 queries have been used in all experiments. The performance of the two approaches is addressed in terms of accuracy and computational time. The approximate generalized median graphs for each class have been computed from five randomly chosen graphs of each class. The process of randomly selecting five graphs from each class and computing their distances to the queries was repeated five times with different choices. The results in the forthcoming tables and figures refer to the average time and accuracy obtained over the five different choices.

It was observed in the experiments that the computational times of the *k-NN* algorithm grow quickly as the number of nodes increase. Table 3.7 displays the time consumed in computing the approximate generalized median graph of both classes in each experiment and the time performance of the *k-NN* algorithm for each experiment. It shows that classifying a query using the approximate generalized median graph requires smaller computation times, since only two distance computations have to be performed, between the query and the generalized median graph of each class. Table 3.8 shows the accuracy obtained by the methods in the experiments. The results in accuracy show that the generalized median graph approach is able to obtain an average level of accuracy of 97.33% in classificating the graphs. This result may indicate that the median graph can be effectively used when one wants to find a representative of a set of graphs.

Table 3.7: Classification times in seconds using the approximate generalized median graph, 1-NN, and 3-NN.

	Approximate generalized median graph			1-NN	3-NN
Experiment	Computation (s)	Classification (s)	Total (s)	(s)	(s)
A	687.49	4.43	691.92	415.18	420.09
B	820.52	6.08	826.60	1743.72	1744.51
C	1181.44	6.56	1188.00	8892.50	8894.72

These classification experiments comparing the use of the generalized median graph and the *k*-NN classifier have shown that the first is competitive with the latter and was even able to outperform it in terms of solution accuracy. In addition, the approach based on the use of the generalized median graph spends significantly smaller computation times, in particular when the size of the training set increases. Therefore, we may conclude that the generalized median graphs provided by the algorithms proposed in this Chapter provide useful information for classification problems.

Table 3.8: Accuracy in classification using the generalized median graph, 1-NN, and 3-NN.

Experiment	Approximate generalized median graph Accuracy	1-NN Accuracy	3-NN Accuracy
A	185.60/190 (97.68%)	183/190 (96,31%)	190/190 (100%)
B	189.60/190 (99.78%)	189/190 (99,47%)	190/190 (100%)
C	179.60/190 (94.52%)	182/190 (95,78%)	190/190 (100%)

3.6 Conclusions

We proposed two heuristics for computing generalized median graphs: an adaptive greedy algorithm and its extension to a GRASP heuristic. The GRASP heuristic obtained generalized median graphs that significantly improved the initial solutions and those provided by the median graphs. On average, the GRASP heuristic improved the sum of distances from the minimum common supergraph by 77.54 percent and the sum of distances from the median graph by 14.09 percent. The proposed adaptive greedy algorithm and GRASP heuristic made it possible to solve significantly larger problems than those solved in the literature to date. These results lead to the main conclusion of this Chapter. Good approximations to the generalized median graph can be effectively computed by the heuristics proposed in this chapter, making it a better representation than the median graph to be used in a number of relevant pattern recognition or machine learning applications, as illustrated by the experiments involving the graph based classification and the 1-NN classifier. The contributions and results in this chapter have already been published as reference [51].

Chapter 4

Bounds on the SOD of a graph

Section 4.1 presents the main theoretical result in this work, which gives a bound to the value of the *SOD* of a graph, based only on the number of nodes of the graph and on the number of graphs in the set. A consequence of this proposition, related to the reduction of the search space for the generalized median graph problem, is explored in Section 4.2. Section 4.3 shows how algorithms for the generalized median graph problem can benefit from this theoretical result.

4.1 Bounds on the SOD of a graph

In the search to find approximate generalized median graphs for a set S, the candidate solutions are always induced subgraphs of the minimum common supergraph of S. If $\#(MinSup(S)) = k$, then the number of possible induced subgraphs is 2^k. Thus, it would be interesting to find a criterion to discard some of these candidate subgraphs, and focus only on the most promising ones.

Proposition 1 determines a bound to the SOD of a candidate graph, depending only on its number of nodes and on the number of graphs in the set. All labeled graphs are taken from a universe set U, where U consists of all graphs that can be constructed from an alphabet $\mathcal{L}$ of labels.

Proposition 1 *Let $S = \{G_1, \ldots, G_n\} \subset U$ be a set of labeled graphs. Let $Cand \in U$. Then:*

$$SOD(Cand, S) \geq |\sum_{i=1}^{n} \#(G_i) - n \times \#(Cand)|.$$

Proof: Let $S = \{G_1, \ldots, G_n\}$ be a set of labeled graphs and let $Cand \in U$. By definition,

$MaxSub(G_i,Cand) \subseteq Cand$ and $MaxSub(G_i,Cand) \subseteq G_i$, for $1 \leq i \leq n$. Therefore, we have that:

$$\#(MaxSub(G_i,Cand)) \leq \#(Cand), 1 \leq i \leq n, \tag{4.1}$$

$$\#(MaxSub(G_i,Cand)) \leq \#(G_i), 1 \leq i \leq n. \tag{4.2}$$

Computing the distance between $Cand$ and the graphs in S we have:

$$\begin{cases} d(G_1,Cand) = \#(G_1) + \#(Cand) - 2 \times \#(MaxSub(G_1,Cand)) \\ d(G_2,Cand) = \#(G_2) + \#(Cand) - 2 \times \#(MaxSub(G_2,Cand)) \\ \vdots \\ d(G_n,Cand) = \#(G_n) + \#(Cand) - 2 \times \#(MaxSub(G_n,Cand)) \end{cases}$$

Therefore:

$$SOD(Cand,S) = \sum_{i=1}^{n} d(G_i,Cand) = \sum_{i=1}^{n} \#(G_i) + n \times \#(Cand) - 2 \times \sum_{i=1}^{n} \#(MaxSub(G_i,Cand))$$

Using inequality (4.1) in $SOD(Cand,S)$,

$$SOD(Cand,S) \geq \sum_{i=1}^{n} \#(G_i) + n \times \#(Cand) - 2 \times n \times \#(Cand)$$

and, therefore,

$$SOD(Cand,S) \geq \sum_{i=1}^{n} \#(G_i) - n \times \#(Cand). \tag{4.3}$$

Using inequality (4.2) in $SOD(Cand,S)$,

$$SOD(Cand,S) \geq \sum_{i=1}^{n} \#(G_i) + n \times \#(Cand) - 2 \times \sum_{i=1}^{n} \#(G_i)$$

and,

$$SOD(Cand,S) \geq n \times \#(Cand) - \sum_{i=1}^{n} \#(G_i). \tag{4.4}$$

Finally, from (4.3) and (4.4),

$$SOD(Cand,S) \geq |\sum_{i=1}^{n} \#(G_i) - n \times \#(Cand)|. \blacksquare$$

4.2 Reduction of the search space

The heuristics already implemented (adaptive greedy and GRASP) have the goal of computing an approximate generalized median graph of a set of labeled graphs S, i.e, find a graph with a small SOD to the graphs of S. Given a candidate graph $Cand \in U$, the previous proposition gives a bound to $SOD(Cand,S)$ based only on its number $\#(Cand)$ of nodes.

It is possible to use Proposition 1 to reduce the search space for the generalized median graph of a set of graphs. In order to understand how this reduction works, consider instance i5.20, which consists of graphs 153, 718, 4901 and 14734 from the *IAM GraphDatabase Repository* (in this case, $n = 4$ and $\sum_{i=1}^{4} \#(G_i) = 20$). The best candidate found by the adaptive greedy heuristic was a graph G such that $SOD(G,S) = 10$. When executing any of the heuristics, if we are interested only in graphs whose SOD is less than or equal to $SOD(G,S)$, it is not necessary to evaluate any graphs $Cand$ such that $\#(Cand) = 0, 1$ or 2, because:

- If #(Cand) = 0, then $SOD(Cand) \geq |20 - 4 \times 0| = 20$.
- If #(Cand) = 1, then $SOD(Cand) \geq |20 - 4 \times 1| = 16$.
- If #(Cand) = 2, then $SOD(Cand) \geq |20 - 4 \times 2| = 12$.

Therefore, it is not necessary to examine any graphs such that $\#(Cand) = 0, 1$ or 2, since its SOD will be necessarily greater than that result found by the adaptive greedy heuristic. In the same way, graphs that have 8 or more nodes need not be evaluated, since:

- If #(Cand) = 8, then $SOD(Cand,S) \geq |20 - 4 \times 8| = 12$.
- If #(Cand) = 9, then $SOD(Cand,S) \geq |20 - 4 \times 9| = 16$, and so on.

As mentioned earlier, all candidates to be a generalized median graph are induced subgraphs of the minimum common supergraph($MinSup(S)$). If $\#(MinSup(S)) = k$, then the number of induced subgraphs of $MinSup(S)$ is 2^k. In the case of instance i5.20, the minimum common

supergraph is a graph with 11 nodes. Therefore, the original search space consisted of all induced subgraphs of $MinSup(S)$ with a number of nodes less than or equal to 11. Hence, there were $2^{11} = 2048$ induced subgraphs of the minimum common supergraph that could be tested as approximate generalized median graphs for the instance. Using this proposition and the bound given by the adaptive greedy heuristic, we were able to reduce the search space to induced subgraphs with the number of nodes ranging in the interval [3,7]. The induced subgraphs of $MinSup(S)$ with 0, 1, 2, 8, 9, 10 and 11 nodes can be excluded from the search space. The number of these subgraphs is given by:

$$\binom{11}{0} + \binom{11}{1} + \binom{11}{2} + \binom{11}{8} + \binom{11}{9} + \binom{11}{10} + \binom{11}{11} = 299$$

where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is the binomial coefficient. Therefore, for instance i5.20 the reduction in the number of candidates, i.e, the reduction in the search space was of $\frac{299}{2048} = 14.59\%$.

The same reasoning applied to instance i5.20 can be used in the general case: let $S = \{G_1, \ldots, G_n\}$ be a set of labeled graphs for which we want to find a generalized median graph. Suppose that we have already obtained a graph G such that $SOD(G) = L > 0$. By Proposition 1, we know that for any graph $Cand$, $SOD(Cand, S)$ is $|\sum_{i=1}^{n} \#(G_i) - n \times \#(Cand)|$ at best. If we are only interested in graphs $Cand$ such that $SOD(Cand, S) \leq L$ then we can exclude any graph $Cand$ such that:

$$|\sum_{i=1}^{n} \#(G_i) - n \times \#(Cand)| > L.$$

We have:

$$|\sum_{i=1}^{n} \#(G_i) - n \times \#(Cand)| > L \Leftrightarrow \begin{cases} \sum_{i=1}^{n} \#(G_i) - n \times \#(Cand) > L & (*) \\ \text{or} \\ \sum_{i=1}^{n} \#(G_i) - n \times \#(Cand) < -L & (**) \end{cases}$$

Solving inequalities $(*)$ and $(**)$ we have:

$$|\sum_{i=1}^{n} \#(G_i) - n \times \#(Cand)| > L \Leftrightarrow \begin{cases} \#(Cand) < \dfrac{\sum_{i=1}^{n} \#(G_i) - L}{n} \\ \text{or} \\ \#(Cand) > \dfrac{\sum_{i=1}^{n} \#(G_i) + L}{n} \end{cases}$$

Therefore, if $\#(Cand) < \frac{\sum_{i=1}^{n} \#(G_i) - L}{n}$ or $\#(Cand) > \frac{\sum_{i=1}^{n} \#(G_i) + L}{n}$ then we know for sure that $SOD(Cand, S) > L$ and there is no need to evaluate this graph. Thus, we are only interested in graphs that have a number of nodes in the interval $\left[\frac{\sum_{i=1}^{n} \#(G_i) - L}{n}, \frac{\sum_{i=1}^{n} \#(G_i) + L}{n}\right]$. Since the number of nodes is a natural number, we should actually consider only the interval:

$$\left[\left\lceil \frac{\sum_{i=1}^{n} \#(G_i) - L}{n} \right\rceil, \left\lfloor \frac{\sum_{i=1}^{n} \#(G_i) + L}{n} \right\rfloor\right].$$

Also, we cannot have a negative number of nodes and we are not interested in graphs larger than $MinSup(S)$. Thus, more precisely, the only subgraphs that should be considered are the ones that have their number of nodes ranging in the interval:

$$\left[max\left\{0, \left\lceil \frac{\sum_{i=1}^{n} \#(G_i) - L}{n} \right\rceil\right\}, min\left\{\#(MinSup(S)), \left\lfloor \frac{\sum_{i=1}^{n} \#(G_i) + L}{n} \right\rfloor\right\}\right].$$

Tables 4.1 to 4.5 show the reduction in the search space that can be obtained using Proposition 1 for all 100 instances executed by the heuristics. For each instance, the limits $\frac{\sum_{i=1}^{n} \#(G_i) - L}{n}$ and $\frac{\sum_{i=1}^{n} \#(G_i) + L}{n}$ are shown, where L is the SOD of the approximate generalized median graph found by the adaptive greedy heuristic. The original search interval $[0, \#(MinSup(S))]$ and the reduced search interval $\left[\left\lceil \frac{\sum_{i=1}^{n} \#(G_i) - L}{n} \right\rceil, \left\lfloor \frac{\sum_{i=1}^{n} \#(G_i) + L}{n} \right\rfloor\right]$ are also shown. The last column indicates by how much the search space decreased in percent.

Table 4.1: Reduction in the search interval for instances with sizes 20 and 40.

Instance	$\frac{\sum \#(G_i) - L}{n}$	$\frac{\sum \#(G_i) + L}{n}$	Original search interval	Reduced search interval	Reduction(%)
i1.20	2.67	10.67	[0,14]	[3,10]	3.51
i2.20	0.00	10.00	[0,15]	[0,10]	5.92
i3.20	0.00	10.00	[0,18]	[0,10]	24.03
i4.20	1.50	8.50	[0,13]	[2,8]	13.51
i5.20	2.50	7.50	[0,11]	[3,7]	14.59
i6.20	0.50	9.50	[0,14]	[1,9]	8.98
i7.20	1.00	9.00	[0,11]	[1,9]	0.63
i8.20	0.00	8.00	[0,16]	[0,8]	40.18
i9.20	0.00	8.00	[0,16]	[0,8]	40.18
i10.20	0.20	7.80	[0,14]	[1,7]	39.53
i1.40	0.67	12.67	[0,22]	[1,12]	26.17
i2.40	1.33	12.00	[0,23]	[2,12]	33.88
i3.40	0.33	13.00	[0,23]	[0,13]	20.24
i4.40	1.33	12.00	[0,22]	[2,12]	26.17
i5.40	1.33	12.00	[0,22]	[2,12]	26.17
i6.40	1.00	10.43	[0,21]	[1,10]	50.00
i7.40	0.86	10.57	[0,19]	[1,10]	32.38
i8.40	1.29	10.14	[0,18]	[2,10]	24.04
i9.40	0.00	10.00	[0,24]	[0,10]	72.93
i10.40	0.50	9.50	[0,22]	[1,9]	73.82

Table 4.2: Reduction in the search interval for instances with sizes 60 and 80.

Instance	$\frac{\sum \#(G_i) - L}{n}$	$\frac{\sum \#(G_i) + L}{n}$	Original search interval	Reduced search interval	Reduction(%)
i1.60	1.57	15.57	[0,30]	[2,15]	42.77
i2.60	3.00	14.14	[0,26]	[3,14]	27.86
i3.60	1.50	13.50	[0,28]	[2,13]	57.47
i4.60	1.50	13.50	[0,29]	[2,13]	64.44
i5.60	1.15	8.08	[0,20]	[2,8]	74.82
i6.60	1.00	12.33	[0,27]	[1,12]	64.94
i7.60	1.20	10.80	[0,24]	[2,10]	72.93
i8.60	1.18	9.73	[0,23]	[2,9]	79.75
i9.60	1.09	9.82	[0,22]	[2,9]	73.82
i10.60	0.00	10.00	[0,32]	[0,10]	97.49
i1.80	2.33	15.44	[0,33]	[3,15]	63.58
i2.80	2.20	13.80	[0,32]	[3,13]	81.14
i3.80	1.36	13.18	[0,33]	[2,13]	85.18
i4.80	2.27	12.27	[0,32]	[3,12]	89.23
i5.80	1.55	13.00	[0,32]	[2,13]	81.14
i6.80	1.36	13.18	[0,35]	[2,13]	91.22
i7.80	1.00	13.55	[0,33]	[1,13]	85.18
i8.80	1.45	13.09	[0,33]	[2,13]	85.18
i9.80	1.18	13.36	[0,36]	[2,13]	93.37
i10.80	1.09	13.45	[0,35]	[2,13]	91.22

Table 4.3: Reduction in the search interval for instances with sizes 100 and 120.

Instance	$\frac{\sum \#(G_i) - L}{n}$	$\frac{\sum \#(G_i) + L}{n}$	Original search interval	Reduced search interval	Reduction(%)
i1.100	2.18	16.00	[0,40]	[3,16]	86.59
i2.100	1.83	14.83	[0,38]	[2,14]	92.83
i3.100	1.38	14.00	[0,38]	[2,14]	92.83
i4.100	1.15	14.23	[0,38]	[2,14]	92.83
i5.100	2.15	13.23	[0,36]	[3,13]	93.37
i6.100	2.31	13.08	[0,38]	[3,13]	96.35
i7.100	1.92	13.46	[0,34]	[2,13]	88.52
i8.100	1.71	12.57	[0,37]	[2,12]	97.64
i9.100	1.29	13.00	[0,37]	[2,13]	95.05
i10.100	1.00	13.29	[0,38]	[1,13]	96.35
i1.120	2.15	16.31	[0,42]	[3,16]	91.79
i2.120	1.80	14.20	[0,41]	[2,14]	97.02
i3.120	1.27	14.73	[0,45]	[2,14]	99.19
i4.120	1.67	14.33	[0,41]	[2,14]	97.02
i5.120	1.73	14.27	[0,40]	[2,14]	95.96
i6.120	1.25	13.75	[0,41]	[2,13]	98.62
i7.120	0.88	14.12	[0,40]	[1,14]	95.96
i8.120	1.88	13.12	[0,37]	[2,13]	95.05
i9.120	1.38	13.62	[0,42]	[2,13]	99.02
i10.120	2.12	12.88	[0,35]	[3,12]	95.52

Table 4.4: Reduction in the search interval for instances with sizes 140 and 160.

Instance	$\frac{\sum \#(G_i) - L}{n}$	$\frac{\sum \#(G_i) + L}{n}$	Original search interval	Reduced search interval	Reduction(%)
i1.140	1.35	15.12	[0,50]	[2,15]	99.66
i2.140	1.67	13.89	[0,44]	[2,13]	99.52
i3.140	0.84	13.89	[0,43]	[1,13]	99.31
i4.140	2.05	12.68	[0,44]	[3,12]	99.81
i5.140	1.68	13.05	[0,38]	[2,13]	96.35
i6.140	1.50	12.50	[0,45]	[2,12]	99.87
i7.140	0.76	12.57	[0,49]	[1,12]	99.97
i8.140	1.10	12.24	[0,48]	[2,12]	99.96
i9.140	1.05	12.29	[0,47]	[2,12]	99.94
i10.140	1.10	12.24	[0,42]	[2,12]	99.60
i1.160	2.47	16.35	[0,53]	[3,16]	99.72
i2.160	1.67	16.11	[0,54]	[2,16]	99.80
i3.160	1.67	16.11	[0,50]	[2,16]	99.23
i4.160	1.95	14.89	[0,49]	[2,14]	99.80
i5.160	1.42	15.42	[0,49]	[2,15]	99.53
i6.160	1.63	15.21	[0,55]	[2,15]	99.94
i7.160	2.00	14.84	[0,47]	[2,14]	99.60
i8.160	1.74	15.11	[0,47]	[2,15]	99.06
i9.160	2.37	14.47	[0,39]	[3,14]	94.59
i10.160	2.42	14.42	[0,47]	[3,14]	99.60

Table 4.5: Reduction in the search interval for instances with sizes 180 and 200.

Instance	$\frac{\sum \#(G_i) - L}{n}$	$\frac{\sum \#(G_i) + L}{n}$	Original search interval	Reduced search interval	Reduction(%)
i1.180	1.43	15.71	[0,56]	[2,15]	99.96
i2.180	2.71	14.43	[0,47]	[3,14]	99.60
i3.180	2.67	14.48	[0,50]	[3,14]	99.86
i4.180	1.57	15.57	[0,47]	[2,15]	99.06
i5.180	1.91	14.45	[0,54]	[2,14]	99.97
i6.180	1.27	15.09	[0,61]	[2,15]	99.99
i7.180	1.64	14.73	[0,56]	[2,14]	99.98
i8.180	1.36	15.00	[0,51]	[2,15]	99.76
i9.180	2.55	13.82	[0,50]	[3,13]	99.95
i10.180	1.64	14.73	[0,51]	[2,14]	99.91
i1.200	2.48	16.57	[0,57]	[3,16]	99.93
i2.200	2.36	15.82	[0,52]	[3,15]	99.84
i3.200	2.55	15.64	[0,55]	[3,15]	99.94
i4.200	1.57	15.83	[0,59]	[2,15]	99.98
i5.200	2.17	15.22	[0,53]	[3,15]	99.89
i6.200	1.87	15.52	[0,55]	[2,15]	99.94
i7.200	2.43	14.96	[0,52]	[3,14]	99.94
i8.200	2.57	14.83	[0,49]	[3,14]	99.80
i9.200	2.26	15.13	[0,53]	[3,15]	99.89
i10.200	1.70	15.70	[0,54]	[2,15]	99.92

The results shown in the previous tables indicate a significant reduction in the search intervals, and consequently, in the search space. This reduction mostly eliminates the subgraphs with larger sizes, and it becomes more relevant if we consider how difficult it is to compute the distance between two graphs as they become larger.

4.3 Application to the heuristics

The results presented in this chapter can be used to improve heuristics designed to compute an approximate generalized median graph of a set of graphs. More specifically, the result presented in Section 4.2 can be used to reduce the number of graphs to be evaluated, since it allows the heuristic to discard graphs that have a *SOD* larger than a bound *L*.

An implementation of the GRASP heuristic using these results was tested, but no changes in the results were observed. One possible explanation for this is in the characteristics of the implemented GRASP heuristic. In each iteration, the constructive phase of GRASP uses a greedy strategy to tentatively remove nodes from the solution. This phase mainly determines the number of nodes of the solution, with the local search phase contributing with small changes on the number of nodes of the solution. Demanding that the number of nodes of the solutions fall in a pre-determined range, as Proposition 1 does, does not seem to work well, since the phases of the algorithm already efficiently compute the appropriate number of nodes of the solutions.

In Chapter 5, a BRKGA algorithm will be presented. In a BRKGA algorithm the initial population is chosen completely at random. In this case, this heuristic is likely to benefit from a reduction in the search space, since this reduction will probably allow for a better initial population (and for better mutant individuals). A variant of the BRKGA algorithm, based on the results presented in this chapter, will also be presented in Chapter 5.

Other metaheuristic-based algorithms may also utilize the results of this chapter. For instance, in a Tabu Search or in a Simulated Annealing heuristic, the reduced search interval can be used to restrict the number of nodes of the initial solution of the heuristic. In the case of a matheuristic algorithm, the reduced search interval can be incorporated in the mathematical model through a constraint on the number of nodes in the solution.

Chapter 5

Biased random-key genetic algorithms

In this chapter, BRKGA-based heuristics for the generalized median graph problem are presented. Section 5.1 shows the basic definitions of genetic algorithms. Section 5.2 presents the BRKGA heuristic and in Sections 5.3 and 5.4 two variants of the BRKGA heuristic are presented. Section 5.5 shows the numerical results and a comparison between the BRKGA heuristics and GRASP.

5.1 Genetic algorithms

Genetic algorithms were first proposed in 1975 by Holland [42], based on Darwin's theory of evolution. In the basic terminology of genetic algorithms, an individual c is an array of n components. Each component c_i, for $i = 1, ..., n$ is called a gene and its value is called an allele. The individuals are associated with possible solutions of a problem. An evaluation function is applied for each individual and returns the fitness value, i.e, the capacity that this individual has to solve the problem.

A genetic algorithm (GA) evolves a set of individuals that compose a population P through a certain number of generations. At every generation a new population is created, using the genetic operators of crossover and mutation. In a crossover, individuals of the current population are combined to produce new individuals to the next generation. The mutation operator randomly modifies one or more genes of a certain number of individuals. The algorithm is repeated until some stopping criterion is reached. The pseudo-code of a typical GA is presented in Figure 5.1. In line 1 population P is initialized, and its evolution happens in lines 2 through 8. The fitness of all individuals of the population are computed in line 3. In line 4, the parents of that generation are selected and the crossover operator is applied on them. The mutant indi-

```
begin Genetic algorithm
1   Initialize initial population P;
2   while stopping criterion not reached ;
3       Compute the fitness value of every individual of P;
4       Select parents and apply crossover operator;
5       Select mutants and apply mutation operator;
6       Update population;
7   end-while;
8   return best individual in the population;
end Genetic algorithm.
```

Figure 5.1: Pseudo-code of the genetic algorithm.

viduals are selected in line 5 and the operation of mutation is applied on them. This procedure is executed until some stopping criterion is reached. Finally, line 8 returns the best individual in the population.

Genetic algorithms with random keys, or random-key genetic algorithms (denoted by RKGA), were first introduced by Bean [7] for combinatorial optimization problems whose solutions may be represented by permutation vectors. In a RKGA, the individuals are represented by an array of real numbers in the interval [0,1). Each element of the array is called a key and is randomly generated in the initial population.The population is partioned in two subsets, one composed by the most fit individuals of the population, called the elite set, and the other one composed by the remaining individuals, called the non-elite set. A deterministic algorithm called the decoder maps each array of random-keys in a solution to the optimization problem. The cost of this solution is used as the fitness value.

RKGA uses the parameterized uniform crossover of Spears and Dejong [63] to combine two randomly selected individuals of the population. Let n be the number of genes in the individuals. Given two individuals c_1 and c_2, randomly selected in the population, p_a is the probability that a descendent individual c_{new} inherits an allele from c_1. This descendent c_{new} is generated in the following manner: for $i = 1,\ldots,n$, the ith allele $c_{new}(i)$ inherits the ith allele of c_1 with a probability p_a or from individual c_2 with a probability $1 - p_a$.

Figure 5.2 illustrates this crossover process of two individuals with four genes each. In this example, we have $p_a = 0.7$. A real number is randomly generated in the interval [0,1). If the number generated is less than 0.7, then the descendant inherits the allele from the first individual, otherwise it inherits the allele from the second individual. In this example, the descendant inherited the first, third and fourth genes from the first individual, making it more similar to it than to the second individual.

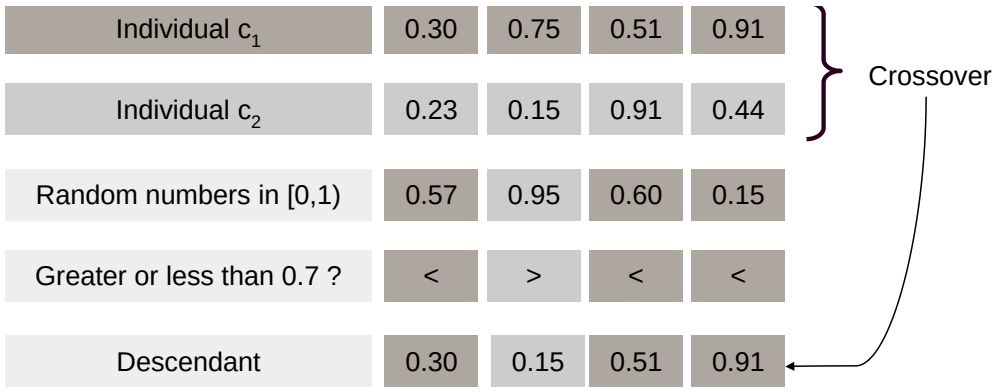


Figure 5.2: Parameterized uniform crossover.

Biased random-key genetic algorithms - BRKGA - first appeared in 2002 [23], and the main difference from RKGA is in the way that the individuals are selected for the crossover operation. In a BRKGA, in the crossover operation one individual is randomly selected from the elite-set and the other is selected from the non-elite set. In the RKGA, as mentioned earlier, the two individuals are randomly selected from the population. This improvement is sufficient to make BRKGA outperform RKGA. In both algorithms, the individuals can be selected more than once for mating in the same generation.

A BRKGA heuristic basically evolves a population of random-key vectors through a number of generations. Figure 5.3 illustrates the transition between two BRKGA generations. The left side of the figure represents the current population, partitioned in two subsets: TOP and REST, where TOP refers to elite individuals and REST refers to the remaining individuals. The size of the population is $|TOP| + |REST|$. The individuals are sorted by their fitness values. The set TOP contains the fittest individuals of the population. The set REST is formed by two subsets: MID and BOT, the subset BOT being formed by the worst individuals of the population. The population of the new generation is created in the following manner: the individuals from the set TOP are copied without modifications to the next generation. A number of $|BOT|$ mutant individuals are randomly generated and |MID| = |REST| - |BOT| individuals are created by crossover between a randomly selected individual from TOP and another from REST. Observe that an elite individual of the previous generation may not belong to the elite set in the current generation.

BRKGAs also use the uniform parameterized crossover of Spears and DeJong [63]. Table 5.1 shows the intervals for the values of the BRKGA parameters, as recommended in [39]. Other parameters are problem-specific: representing a solution of the problem, decodifying a chromossome, and stopping criterion.

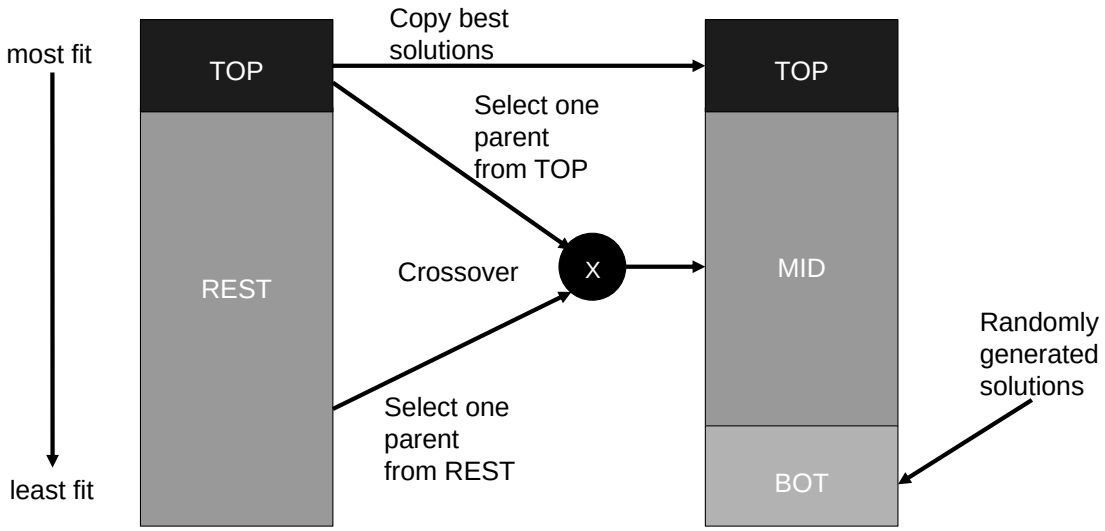


Figure 5.3: Population evolution between consecutive generations of a BRKGA

Table 5.1: Recommended value for the parameters

Parameter	Description	Recommended value
P	Population size	$\|P\| = a \cdot n$, where $a \geq 1$ is a constant and n is the size of the individual
$\|TOP\|$	Size of elite population	$0.10 \cdot \|P\| \leq \|TOP\| \leq 0.25 \cdot \|P\|$
$\|BOT\|$	Size of mutant population	$0.05 \cdot \|P\| \leq \|BOT\| \leq 0.30 \cdot \|P\|$
p_a	Probability of inheriting an elite allele	$0.5 \leq p_a \leq 0.8$

The pseudo-code of a BRKGA is presented in Figure 5.4. In line 1 the population is initialized. The evolution of this population occurs in lines 2-9. In line 3 all the individuals are decoded and their fitness values are computed. In line 4 the population is sorted in non-decreasing order with respect to the fitness values. Population P is then partioned in two subsets: TOP and $REST$. Subset TOP contains the fittest individuals of the population, while subset $REST$ is composed by the remaining individuals. In line 6 the next population is initialized with the individuals from the TOP set of the current population. In line 7, $|BOT|$ mutant individuals are randomly generated for the next population. In line 8, $|P| - |TOP| - |BOT|$ individuals are created by a parameterized uniform crossover for the next population, where one of the individuals is selected from the TOP set and the other one from the $REST$ set. This procedure is executed until a stopping criterion is reached. This criterion may be, for example, the number of evolved populations, and the quality of the best found solution, among others. Finally, in line 10 the best solution is returned.

begin Biased random key genetic algorithm
1 Initialize initial population P;
2 **while** stopping criterion not reached ;
3 Compute the fitness value of every individual of P;
4 Sort population P in non-decreasing order of the fitness values;
5 Partition P in two subsets: TOP and $REST$;
6 Copy individuals from set TOP of the current population to the next population;
7 Randomly generate $|BOT|$ mutant individuals for the next population;
8 Generate $|P| - |TOP| - |BOT|$ individuals by uniform parameterized crossover for the next population, selecting one individual from TOP and the other one from $REST$;
9 **end-while**;
10 **return** best individual in the population;
end Biased random key genetic algorithm.

Figure 5.4: Pseudo-code of the biased random-key genetic algorithm.

BRKGA heuristics have been succesfully applied to many optimization problems. For example, BRKGA was compared in [40] with six standard genetic algorithms for job-shop scheduling (GA [18], GLS1 and GLS2 [1], P-GA, SBGA(40), SBGA(60) [20]). On the 12 test instances where BRKGA was compared with GA, an average reduction in cost of 2.02% was observed. On the 37 and 35 test instances where it was compared, respectively, with GLS1 and GLS2, average reductions of 3.79% and 0.58% were observed. In the comparison with P-GA, on 20 test instances, the reduction in the cost of the solutions was of 0.48%. In the comparison with SBGA(40) and SBGA(60) on 42 test instances, the respective average solution cost reductions were of 1.27% and 1.01% [39]. In [10], BRKGA was applied to an unconstrained multi-round divisible load scheduling problem, with the computational experiments showing that the makespans obtained by the proposed heuristic improved upon those obtained by the best algorithm in the literature by 11.68%, on average.

5.2 BRKGA for the generalized median graph problem

The implementation of biased random-key genetic algorithms for the generalized median graph problem made use of the C++ library brkgaAPI developed by Toso and Resende [66], which is a framework for the development of biased random-key genetic algorithms. It can also be used in parallel architectures running OpenMP. The instantiation of the framework shown in Figure 5.5 to some specific optimization problem requires exclusively the development of a class implementing the decoder for this problem. This is the only problem-dependent part of the tool. The decoding process used for the generalized median graph problem is explained in

details in subsection 5.2.1. According to Gonçalves et al. [39], the BRKGA framework requires the following parameters: (a) the population size ($p = |TOP| + |REST|$); (b) the fraction p_e of the population corresponding to the elite set TOP; (c) the fraction p_m of the population corresponding to the mutant set $BOTTOM$; (d) the probability *rhoe* that the offspring inherits each of its keys from the best fit of the two parents; and (e) the number k of generations without improvement in the best solution until a restart is performed. Whenever a restart occurs, the full population is randomly generated from scratch as for the first generation. The tuning of these parameters is explained in Section 5.5.

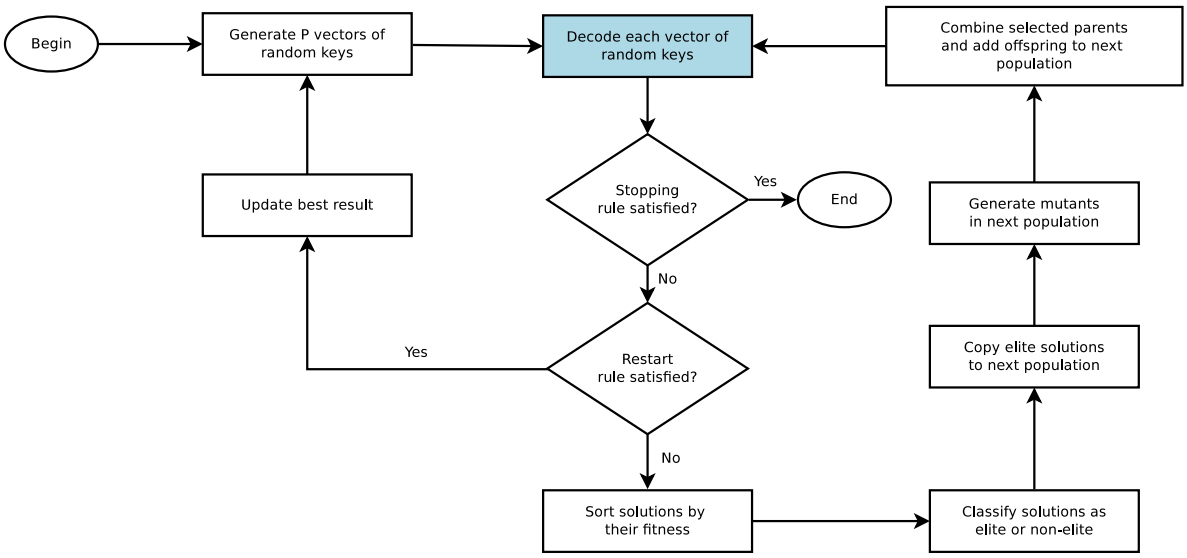


Figure 5.5: BRKGA framework.

5.2.1 Decoder

In a BRKGA, chromossomes represent solutions to the problem in hand. Each chromossome is composed of random-keys (real numbers in the range [0,1)) and is decoded by an algorithm (the decoder) that receives the keys and builds a solution to the problem. In our implementation of the BRKGA for the generalized median graph problem, the chromossomes will have size $\#(MinSup(S)) + 1$, where S is the set for which we want to compute an approximate generalized median graph. Our decoder transforms each chromossome into an induced subgraph of the minimum common supergraph $MinSup(S)$ of set S.

The decoding process that transforms each chromossome into an induced subgraph of $MinSup(S)$ is applied in two steps: in the first step, each random-key of a chromossome c is transformed into an integer number. In the second step, the chromossome obtained in the first step is transformed into an induced subgraph of $MinSup(S)$ (this is accomplished by transforming each gene of the chromossome in a node of $MinSup(S)$).

In the first step of the decodification, for each gene $c[i]$, $i = 0, \ldots, \#(MinSup(S))$, we set:

$$c[i] \leftarrow \lfloor c[i] \times 10^t \rfloor$$

.

Since, originally, each $c[i] \in [0,1)$, the new value of $c[i]$ is an integer in the interval $[0, 10^t - 1]$. As mentioned, in the second step of the decoder each gene $c[i]$ will represent a node from $MinSup(S)$. Therefore, determining the appropriate value for t depends on $\#(MinSup(S))$. In all test instances considered in this work, $\#(MinSup(S)) < 70$. Therefore, the computed integers need to be at least in the range $[0,99]$, since this allows for the representation of up to 100 nodes of the minimum common supergraph of any instance. Thus, in our implementation, it is sufficient to use $t = 2$. We observe that larger values of t might be needed for larger instances where the minimum common supergraph has a larger number of nodes.

In the second step of the decoding phase, the chromossome (now composed of integer values) will be transformed into an induced subgraph of $MinSup(S)$. The decodification of the integer in the first position of c, $c[0]$, will indicate the number of nodes in that solution. It is necessary that this number of nodes be between 0 and $\#(MinSup(S))$, since this chromossome represents an induced subgraph of $MinSup(S)$. In order to find a number that falls in that range, the remainder of the division of $c[0]$ by $\#(MinSup(S)) + 1$ is computed. This remainder indicates how many positions of the chromossome will be considered for decodification.

From the second position on, the decoding process consists of computing the remainder of the division of the integer stored in this position, $c[i]$, by $\#(MinSup(S))$. Observe that this value is an integer from 0 to $\#(MinSup(S)) - 1$. The nodes from $MinSup(S)$ are also labelled from 0 to $\#(MinSup(S)) - 1$. The node from $MinSup(S)$ with the same label as the remainder is present in the subgraph. In case there is a collision, i.e, when a node previously selected is selected again, the decoder searchs for the next node of $MinSup(S)$ still unselected.

As an example, consider S such that $\#(MinSup(\mathrm{S})) = 7$, as in Figure 5.6. Figure 5.7 shows a chromossome with its random-keys, and Figure 5.8 shows the chromossome after the first step of the decodification process, which transforms each random-key in an integer by multiplying it by 10^2 and taking the floor of this value. Figure 5.9 shows the chromossome after the second step of the decodification. The size of the chromossome consists of 7+1 =8 genes (as mentioned, the extra gene is needed since it will represent the number of nodes in the chromossome).

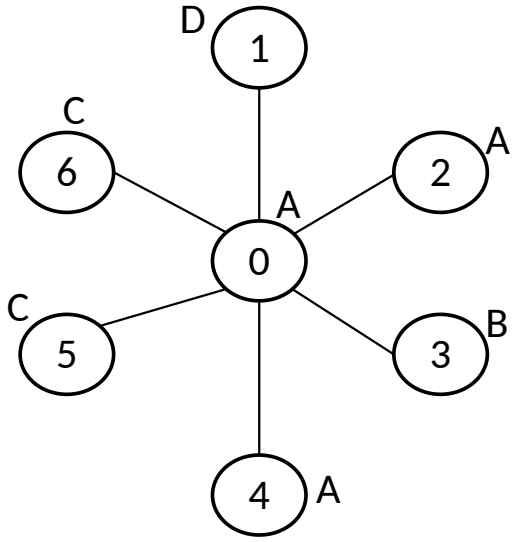


Figure 5.6: Minimum common supergraph of set S. The decoder will convert each chromossome into an induced subgraph of this graph.

0.1345...	0.0823...	0.1034...	0.0612...	0.5327...	0.4211...	0.0268...	0.2589...

Figure 5.7: Chromossome with 7+1 = 8 genes. The random-keys are real numbers in the interval $[0,1)$.

13	8	10	6	53	42	2	25

Figure 5.8: First step in the decoding phase: all random-keys are transformed into integer numbers by $c[i] \leftarrow \lfloor c[i] \times 10^2 \rfloor$.

5	1	3	6	4	0	2	25

Figure 5.9: Decoded chromossome. The first gene indicates the number of nodes of the induced subgraph, and the next five positions indicate the nodes of $MinSup(S)$ that will be in the subgraph. The last two positions are ignored.

The decodification of the first gene is made by dividing its value, 13, by $\#(MinSup(S)) + 1 = 7 + 1 = 8$ and computing the remainder of the division. Since $13 = 8 \times 1 + 5$, the remainder of this division is 5. The decodification of this first gene is 5, and this chromossome will represent an induced subgraph of MinSup(S) with five nodes. These five nodes will be obtained

by the decodification of the next five positions of the chromossome (positions 2 to 6 of the chromossome).

The decodification of the genes in positions 2 to 6 is done by computing the remainder of the division of the integer stored in the gene by $\#(MinSup(S)) = 7$. Then:

- Position 2: $8 = 7 \times 1 + 1 \rightarrow$ node 1 from $MinSup(S)$ belongs to subgraph.
- Position 3: $10 = 7 \times 1 + 3 \rightarrow$ node 3 from $MinSup(S)$ belongs to subgraph.
- Position 4: $6 = 7 \times 0 + 6 \rightarrow$ node 6 from $MinSup(S)$ belongs to subgraph.
- Position 5: $53 = 7 \times 7 + 4 \rightarrow$ node 4 from $MinSup(S)$ belongs to subgraph.
- Position 6: $42 = 7 \times 6 + 0 \rightarrow$ node 0 from $MinSup(S)$ belongs to subgraph.

Positions 7 and 8 of the chromossome are ignored (because the first gene indicates that the subgraph is composed of only five nodes). Therefore, the decodification of this chromossome corresponds to the subgraph of MinSup(S) with nodes 0, 1, 3, 4 and 6, as represented in Figure 5.10. To obtain the fitness of this solution, the distance between this graph and all the graphs from set S are computed.

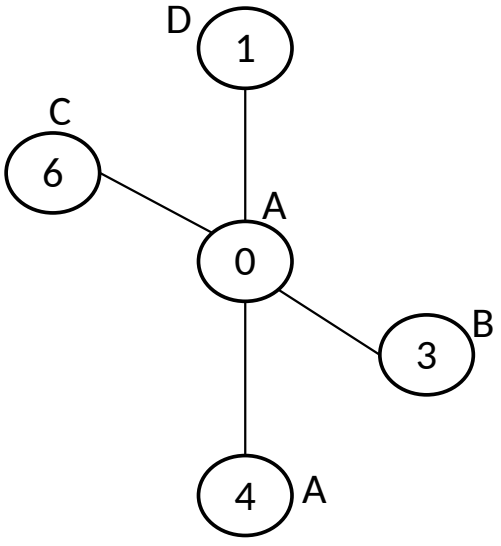


Figure 5.10: The decodification of the chromossome results in an induced subgraph of $MinSup(S)$ with nodes 0, 1, 3, 4 and 6.

Figure 5.11 shows the pseudo-code of the decoder of the BRKGA algorithm. It takes as input the minimum common supergraph of set S, a random-key chromossome $c[0,\ldots,n]$, and a

natural number t. The decoded chromossome is represented by $d[0,\ldots,n]$. In line 1, the size of the minimum common supergraph is stored in n. In lines 2-4, the random-key chromossome is transformed in a chromossome with integer values. The remainder of the division of the first gene, $c[0]$, by $n+1$ is stored in m, in line 5, and this value m is stored in the first position of the decoded chromosomme, $d[0]$, in line 6. In lines 7-9 occurs the decodification of the other positions of the chromossome. In line 8, the remainder of the division of $c[i]$ by n is stored in $d[i]$, for $i = 1,\ldots,m$. In line 10, graph G is set as empty. In lines 11-15, the induced subgraph defined by the decoded chromossome is assembled and stored in G (the insertion of node v and its adjacent edges in G is indicated by $G \cup \{v\}$, as shown line in 14). In line 16, G is returned.

```
begin Decoder-BRKGA (MinSup(S), c[0...n], t)
1   n ← #(MinSup(S));
2   for i = 0,...,n
3       c[i] ← ⌊c[i] × 10^t⌋;
4   end-for;
5   m ← remainder of the division of c[0] by n+1;
6   d[0] ← m;
7   for i = 1,...,m
8       d[i] ← remainder of the division of c[i] by n;
9   end-for
10  G ← ∅;
11  for i = 1,...,m
12      v ← d[i];
13      select node v from MinSup(S);
14      G ← G ∪ {v};
15  end-for;
16  return G;
end Decoder-BRKGA.
```

Figure 5.11: Pseudo-code of Decoder-BRKGA.

5.3 Bounded BRKGA

In this section we describe the Bounded BRKGA variant of the BRKGA heuristic.

The Bounded BRKGA heuristic uses Proposition 1 from Chapter 4 to reduce the search

space and to speed up the execution of the heuristic. In comparison to the BRKGA heuristic, Bounded BRKGA has one additional parameter: the bound L, which can be obtained by any heuristic previously executed. Bounded BRKGA will only consider candidate graphs G such that $SOD(G,S) \leq L$.

5.3.1 Decoder

The decoder of the Bounded BRKGA is a slight modification of the BRKGA decoder. Chromossomes still have size $\#(MinSup(S)) + 1$ and the decodification of all genes is not modified, except for the first one. The first gene of the chromossome represents the number of nodes of the subgraph, while in BRKGA's decoder it could assume any value between 0 and $\#(MinSup)$. In Bounded BRKGA, the bound L reduces the search space and the chromossomes can only be decoded into subgraphs that have a number of nodes in the interval $[min\_node, max\_node]$. In order to find a number of nodes that falls in this range, the decodification of the first gene consists in computing the remainder of the division of the first random-key (after it has been transformed into an integer) by $(max\_node - min\_node + 1)$ and to add min_node to this remainder. This guarantees that the subgraph obtained after decodification has a number of nodes in the interval $[min\_node, max\_node]$.

Figure 5.12 shows the pseudo-code of the decoder of Bounded BRKGA. It takes as input the minimum common supergraph of set S, a random-key chromossome $c[0,\ldots,n]$ and two positive numbers L and t. The decoded chromossome is represented by $d[0,\ldots,n]$. In line 1 the size of the minimum common supergraph is stored in n. In lines 2-4, the random-key chromossome is transformed in a chromossome with integer values. In lines 5 and 6, the bound L is used to find the minimum and maximum number of nodes of the graphs in the populations, which are stored in min_node and max_node, respectively. In line 7, the remainder of the division of $c[0]$ by $(max\_node - min\_node + 1)$ is stored in m. In line 8, the first position of the decoded chromossome, $d[0]$, receives the value $m + min\_node$, which represents the number of nodes in the subgraph (observe that the value stored in $d[0]$ is an integer in the interval $[min\_node, max\_node]$). In line 10, the remainder of the division of $c[i]$ by n is stored in $d[i]$, for $i = 1,\ldots,m + min\_node$. In lines 12-16, the induced subgraph defined by the decoded chromossome is assembled and stored in G. Finally, in line 17, the graph G is returned.

begin Decoder-Bounded BRKGA ($MinSup(S)$, $c[0\ldots n]$, L, t)
1 $n \leftarrow \#(MinSup(S))$;
2 **for** $i = 0,\ldots,n$
3 $\quad c[i] \leftarrow \lfloor c[i] \times 10^t \rfloor$;
4 **end-for**;
5 $min\_node \leftarrow \left\lceil \frac{\sum_{i=1}^{n} \#(G_i) - L}{n} \right\rceil$;
6 $max\_node \leftarrow \left\lfloor \frac{\sum_{i=1}^{n} \#(G_i) + L}{n} \right\rfloor$;
7 $m \leftarrow$ remainder of the division of $c[0]$ by $(max\_node - min\_node + 1)$;
8 $d[0] \leftarrow m + min\_node$;
9 **for** $i = 1,\ldots,m + min\_node$
10 $\quad d[i] \leftarrow$ remainder of the division of $c[i]$ by n;
11 **end-for**
12 **for** $i = 1,\ldots,m + min\_node$
13 $\quad v \leftarrow d[i]$;
14 $\quad$ select node v from $MinSup(S)$;
15 $\quad G \leftarrow G \cup \{v\}$;
16 **end-for**;
17 **return** G;
end Decoder-Bounded BRKGA.

Figure 5.12: Pseudo-code of Decoder-Bounded BRKGA.

5.4 Bounded BRKGA with local search

A bounded BRKGA with a local search phase (denoted Bounded BRKGA + LS) was also implemented. At each generation, the k best solutions of the population are selected and local search phase is applied to each of them. This local search phase consists of two heuristics, denoted by LS_1 and LS_2, executed sequentially, each being a local search itself. In LS_1, the neighbours of the incumbent solution are all the graphs that can be obtained by the insertion of one node to the incumbent. The local optimum found in this first phase will serve as the incumbent for the second phase of the local search. In LS_2, the neighbours of the incumbent solution are all the graphs that can be obtained by removing one node from the incumbent. The graph returned in this second phase is the solution of the local search phase. The original k best solutions in generation i are substituted by the k graphs found in the local search.

Figure 5.13 shows the pseudo-code of the local search. It takes as input an array composed of the k best chromossomes $(c_1, c_2, \ldots, c_k)$ of a population. Lines 1-4 execute the loop that will replace the decodification of chromossomes $(c_1, c_2, \ldots, c_k)$ for the graphs $(G_1, G_2, \ldots, G_k)$ in the next population. In line 2, *Incumb* receives the graph obtained by the decodification of c_i. This graph is used as input for LS_1, and the result of the local search is stored in graph G, as indicated in line 3. In line 4, graph G is used as input for LS_2, and the result of this local search is stored in G_i. In line 6, the array of graphs G_i is returned.

Figure 5.14 shows the pseudo-code for LS_1. This algorithm executes while it finds a neighbour of G that improves upon the fitness of G. The boolean variable *improve_fitness* is used to control this condition, as it is set to *true* in line 1. The **while** loop in lines 2-13 is executed while *improve_fitness* is true. In line 3, the set $Neighbor(G)$ is formed by the induced subgraphs that can be obtained by inserting in G a node that is not currently in G. In line 4, *improve_fitness* is set to false. The loop in lines 5-12 that searches for a neighbor that improves upon the incumbent is executed while the neighborhood of the incumbent is not empty and *improve_fitness* is false, as indicated in line 5. In line 6, a neighbour G' is selected and if its fitness is better than that of G, then G is updated to G', *improve_fitness* is set to true and the loop breaks, as shown in lines 7-10. In line 11, graph G' is removed from the neighborhood of G. Finally, in line 14 graph G is returned. Observe that the variable *improve_fitness* is set to false at the end of the loop in lines 5-12 only if no neighbours of an incumbent solution improve upon the incumbent. This guarantees that the solution returned by the algorithm is a local optimum with respect to this neighbourhood.

Figure 5.15 shows the pseudo-code for LS_2, where the neighbours of a graph G are the graphs obtained by removing one node from G, as indicated in line 3. Except for this, LS_2 functions exactly like LS_1.

begin Local search $(c_1, c_2, \ldots, c_k)$
1 **for** $i = 1 \ldots k$
2 $Incumb \leftarrow$ Decoder-Bounded BRKGA($MinSup(S)$, c_i, L, t);
3 $G \leftarrow LS_1(Incumb)$;
4 $G_i \leftarrow LS_2(G)$;
5 **end-for**;
6 **return** $(G_1, G_2, \ldots, G_k)$;
end Local search.

Figure 5.13: Pseudo-code of local search of Bounded BRKGA + LS.

```
begin LS1(G)
1   improve_fitness ← true;
2   while(improve_fitness)
3       Neighbor(G) ← {G∪{v}|v ∈ V_MinSup(S) − V_G};
4       improve_fitness ← false;
5       while (Neighbor(G) ≠ ∅ and !(improve_fitness))
6           select G' ∈ Neighbor(G);
7           if(fitness(G') < fitness(G))
8               G ← G';
9               improve_fitness ← true;
10          end-if;
11          Neighbor(G) ← Neighbor(G) − {G'};
12      end-while;
13  end-while;
14  return G;
end LS1.
```

Figure 5.14: Pseudo-code of LS_1.

```
begin LS2(G)
1   improve_fitness ← true;
2   while(improve_fitness)
3       Neighbor(G) ← {G − {v}|v ∈ V_G};
4       improve_fitness ← false;
5       while (Neighbor(G) ≠ ∅ and !(improve_fitness))
6           select G' ∈ Neighbor(G);
7           if(fitness(G') < fitness(G))
8               G ← G';
9               improve_fitness ← true;
10          end-if;
11          Neighbor(G) ← Neighbor(G) − {G'};
12      end-while;
13  end-while;
14  return G;
end LS2.
```

Figure 5.15: Pseudo-code of LS_2.

5.5 Computational experiments

In this section, we address the effectiveness of the heuristics based on biased random-key genetic algorithms. We compare the results obtained with the proposed BRKGA heuristics with those obtained by the GRASP heuristic. All BRKGA heuristics used the restart strategy if 50 generations had gone by without improvement of the best value found. The BRKGA heuristics

were implemented in C++ with the GNU GCC compiler. The experiments have been performed on a Dell Studio i3-3240M with a 3.40GHz CPU with 4 GB of RAM under the operating system Windows Home 7 Basic.

5.5.1 Tuning

The BRKGA heuristics have three main parameteres: p_e, the size of the elite population to be copied to the next generation, p_m, the size of the population to be replaced by mutants, and *rhoe*, the probability that the offspring inherits an allele from the elite parent.

In order to extract the most of these heuristics, it is necessary to find the best combination of these three parameters. In the following sections, these three parameters will be analysed so that a good combination of them can be found. The method for tuning these parameters was to choose one of the parameters at a time and assign different values to this parameter, while the other two are fixed. Once the best value was found for this parameter, it will be subsequently used in the tuning of the other two.

For this tuning, 20 instances were chosen, with sizes ranging from 80 to 100 vertices in total. Ten runs of each instance were performed for each value of the parameter in test, each with a different seed, resulting in a total of $10 \times 20 = 200$ executions of the BRKGA heuristic. In all tuning experiments, the BRKGA heuristic was run for 50 generations.

In order to compare the perfomance of the different parameterizations of BRKGA, the following criteria were used:

- **#Best**: This criterion indicates how many times each parameterization found the best solution.
- **Total average time to best**: This criterion gives the average time each parameterization took to find the best value.
- **Total average deviation**: For each instance and parameterization, **Average deviation** indicates the average relative deviation from the best solution found (considering the ten runs of all parameterizations), in percent. The **Total average deviation** is the average value of the **Average deviation** over all instances.
- **Score**: Given an instance and a parametrization, NScore gives the number of parameterizations that found better solutions than that parameterization. In case of ties, all parameterizations which have tied receive the same score, equal to the number of parameterizations strictly better than all of them. The computation of NScore was performed using the

average SOD of the ten runs of each parameterization. **Score** is the sum of the NScore values over all instances in the experiment, for each parametrization. Thus, lower values of Score correspond to better parameterizations.

5.5.1.1 Size of the elite population - p_e

Three values were tested for the p_e parameter: 0.2, 0.5 and 0.7. The other two parameters, p_m and *rhoe* were fixed at 0.1 and 0.7, respectively. Table 5.2 shows the ten runs of BRKGA for instances i1.80 and i2.80. Table 5.3 shows a summary of the results for the BRKGA heuristic for these three values of parameter p_e. From Table 5.3 it is possible to see that the best values for parameter p_e are 0.5 and 0.7. Since the value 0.5 allows for more diversity in the populations, it is preferred in detriment to the value 0.7.

The best values for the other two parameters are obtained analogously.

	$p_e = 0.2$			$p_e = 0.5$			$p_e = 0.7$		
i1.80	Best SOD	Time (s)	Time to Best (s)	Best SOD	Time (s)	Time to Best (s)	Best SOD	Time (s)	Time to Best (s)
ex.1	57	76.5	13.3	59	61.5	56.0	57	50.9	24.6
ex.2	57	76.4	7.2	57	54.7	23.4	57	44.4	23.1
ex.3	57	77.0	19.0	57	58.7	6.9	57	46.5	20.6
ex.4	57	77.0	48.9	58	60.7	17.6	57	44.0	38.4
ex.5	57	81.7	81.4	57	56.1	9.2	57	45.3	18.1
ex.6	57	75.3	9.0	57	58.4	8.8	57	47.3	19.3
ex.7	57	76.7	11.6	57	54.3	28.7	57	42.6	40.0
ex.8	57	78.2	7.6	56	61.6	60.9	57	45.9	19.0
ex.9	57	77.0	8.3	57	59.0	3.8	57	42.9	10.5
ex.10	57	78.0	8.2	57	54.8	9.3	57	44.0	25.3
Average SOD	57.0	77.38	21.45	57.2	57.98	22.46	57.0	45.38	23.89
Average deviation	1.8			2.1			1.8		
i2.80	Best SOD	Time (s)	Time to Best (s)	Best SOD	Time (s)	Time to Best (s)	Best SOD	Time (s)	Time to Best (s)
ex.1	58	67.0	6.4	58	46.8	5.0	58	33.1	15.4
ex.2	58	67.8	6.6	58	46.8	8.8	58	33.5	10.9
ex.3	58	65.0	6.8	58	45.5	10.7	58	33.3	9.9
ex.4	58	65.5	5.1	58	45.8	9.5	58	33.2	15.2
ex.5	58	68.4	4.8	58	46.4	3.7	58	32.3	15.1
ex.6	58	65.1	5.3	58	45.4	7.2	58	32.6	14.6
ex.7	58	66.0	6.0	58	46.2	8.9	58	31.9	10.7
ex.8	58	65.5	4.5	58	45.7	6.0	58	34.3	10.1
ex.9	58	68.2	3.5	58	45.2	9.0	58	32.6	12.5
ex.10	58	65.9	4.8	58	44.6	8.4	58	33.6	6.0
Average SOD	58.0	66.44	5.38	58.0	45.84	7.72	58.0	33.04	12.04
Average deviation	0.0			0.0			0.0		

Table 5.2: Tuning of parameter: $p_e = 0.2, 0.5$ and 0.7. Parameters p_m and *rhoe* were fixed at 0.1 and 0.7, respectively.

	$p_e = 0.2$	$p_e = 0.5$	$p_e = 0.7$
#Best	126	134	139
Total Average time to best	12.9	12.7	14.1
Total average deviation	16.3	15.1	15.6
Score	17	14	13

Table 5.3: Summary of the criteria for tuning parameter p_e.

5.5.1.2 Fraction of the population to be replaced by mutants - p_m

Three values were considered for the analysis of this parameter: 0.1, 0.2 and 0.3. Since the results in the previous section indicated that the value $p_e = 0.5$ found the best results, it was fixed in the tuning of the other parameters. Parameter *rhoe* remained at its previous value of 0.7.

Table 5.4 shows the ten runs of BRKGA for instances i1.80 and i2.80. Table 5.5 shows a summary of the results for the BRKGA heuristic for the three values of parameter p_m. From Table 5.5 it is possible to see that the best value for parameter p_m is 0.2.

	$p_m = 0.1$			$p_m = 0.2$			$p_m = 0.3$		
i1.80	Best SOD	Time (s)	Time to Best (s)	Best SOD	Time (s)	Time to Best (s)	Best SOD	Time (s)	Time to Best (s)
ex.1	59	59.5	54.4	57	74.8	29.2	57	93.7	68.0
ex.2	57	54.9	23.4	57	73.6	12.8	57	85.6	37.0
ex.3	57	58.8	6.9	57	74.0	8.8	57	87.7	14.0
ex.4	58	60.8	17.7	57	78.7	15.9	57	89.7	40.8
ex.5	57	56.4	9.3	57	74.0	35.2	57	88.8	19.1
ex.6	57	58.7	8.8	57	74.2	11.8	57	84.9	18.0
ex.7	57	54.6	28.8	57	71.5	23.8	57	84.9	12.9
ex.8	56	61.0	60.3	57	77.3	21.0	57	93.1	29.1
ex.9	57	59.4	3.9	57	77.4	3.9	57	86.5	3.9
ex.10	57	55.0	9.4	57	73.6	6.2	57	88.7	30.8
Average SOD	57.2	57.91	22.29	57.0	74.91	16.86	57.0	88.36	27.36
Average deviation	2.1			1.8			1.8		
i2.80	Best SOD	Time (s)	Time to Best (s)	Best SOD	Time (s)	Time to Best (s)	Best SOD	Time (s)	Time to Best (s)
ex.1	58	47.0	5.0	58	54.6	5.5	58	61.5	6.6
ex.2	58	49.6	8.5	58	54.8	8.9	58	62.6	13.4
ex.3	58	49.5	12.7	58	53.0	7.5	58	61.8	11.5
ex.4	58	46.2	9.7	58	54.7	5.6	58	61.5	9.0
ex.5	58	47.1	3.7	58	56.0	9.7	58	62.4	15.0
ex.6	58	46.4	7.4	58	55.5	15.1	58	61.1	13.0
ex.7	58	49.5	9.8	58	53.4	6.2	58	61.9	13.5
ex.8	58	48.8	6.1	58	55.1	7.5	58	61.9	6.6
ex.9	58	49.2	10.4	58	57.1	11.4	58	61.5	5.9
ex.10	58	45.0	8.5	58	55.3	8.0	58	61.4	7.1
Average SOD	58.0	47.83	8.18	58.0	54.95	8.54	58.0	61.76	10.16
Average deviation	0.0			0.0			0.0		

Table 5.4: Tuning of parameter: $p_m = 0.1, 0.2$ and 0.3. Parameters p_e and *rhoe* were fixed at 0.5 and 0.7, respectively.

	$p_m = 0.1$	$p_m = 0.2$	$p_m = 0.3$
#Best	134	142	140
Total average time to best	12.8	17.1	20.9
Total average deviation	15.1	12.3	13.7
Score	15	9	9

Table 5.5: Summary of the criteria for the mutation parameter.

5.5.1.3 Probability of inheriting an allele from a parent - *rhoe*

For the analysis of parameter *rhoe*, the values of the other two parameters have already been determined to be $p_e = 0.5$ and $p_m = 0.2$. To evaluate the best value for *rhoe*, three values will be considered for this parameter: 0.5, 0.7 and 0.9. Table 5.6 shows the ten runs of BRKGA for instances i1.80 and i2.80. Table 5.7 shows a summary of the results for the BRKGA heuristic for these three values of parameter *rhoe*. From Table 5.7 it is possible to see that the best value for parameter *rhoe* is 0.5.

Therefore, according with these tuning experiments, the best configuration of these parameters is: $p_e = 0.5$, $p_m = 0.2$ and $rhoe = 0.5$. also executed and compared to the previously found combination.

	rhoe = 0.5			*rhoe* = 0.7			*rhoe* = 0.9		
i1.80	Best SOD	Time (s)	Time to Best (s)	Best SOD	Time (s)	Time to Best (s)	Best SOD	Time (s)	Time to Best (s)
ex.1	57	84.6	51.8	57	74.5	28.9	59	74.7	28.7
ex.2	57	82.9	36.5	57	73.5	12.8	57	77.6	22.8
ex.3	57	80.1	9.2	57	73.9	8.8	57	74.6	8.8
ex.4	57	86.3	28.5	57	78.4	15.9	56	84.0	32.1
ex.5	57	82.0	5.6	57	73.6	34.9	57	80.3	15.5
ex.6	57	79.4	46.2	57	73.7	11.7	57	77.5	27.0
ex.7	57	78.7	22.1	57	71.4	23.8	57	77.1	25.7
ex.8	57	85.8	36.5	57	77.0	20.9	57	80.5	19.8
ex.9	57	82.9	19.8	57	77.3	3.9	57	78.2	13.2
ex.10	57	80.0	26.5	57	73.4	6.2	57	82.2	15.0
Average SOD	57.0	82.27	28.27	57.0	74.67	16.78	57.1	78.67	20.86
Average deviation	1.8			1.8			2.0		
i2.80	Best SOD	Time (s)	Time to Best (s)	Best SOD	Time (s)	Time to Best (s)	Best SOD	Time (s)	Time to Best (s)
ex.1	58	57.4	14.6	58	54.4	5.4	58	71.3	8.5
ex.2	58	59.2	5.6	58	54.6	8.9	58	66.1	7.7
ex.3	58	56.8	8.3	58	52.8	7.5	58	60.5	6.8
ex.4	58	57.8	18.2	58	54.4	5.5	58	67.8	11.3
ex.5	58	58.7	4.6	58	55.7	9.7	58	66.5	15.8
ex.6	58	57.7	14.1	58	55.4	14.9	58	64.8	20.0
ex.7	58	56.4	9.0	58	53.2	6.1	58	65.7	9.9
ex.8	58	58.8	15.9	58	54.6	7.4	58	60.3	9.4
ex.9	58	60.4	24.6	58	56.9	11.4	58	65.3	10.9
ex.10	58	58.0	12.1	58	54.9	8.0	58	59.6	8.7
Average SOD	58.0	58.12	12.7	58.0	54.69	8.48	58.0	64.79	10.9
Average deviation	0.0			0.0			0.0		

Table 5.6: Tuning of parameter: *rhoe* = 0.5, 0.7 and 0.9. Parameters p_e and p_m were fixed at 0.5 and 0.2, respectively.

	$rhoe = 0.5$	$rhoe = 0.7$	$rhoe = 0.9$
Best	150	142	110
Total average time to best	21.2	17.0	15.7
Total average deviation	11.1	12.3	21.3
Score	3	8	30

Table 5.7: Summary of the criteria for tuning parameter *rhoe*.

5.5.2 Experiments

The main idea in a BRKGA heuristic is to start with completely random chromossomes, and make these chromossomes improve as the generations pass. In our first experiment, we test if BRKGA is effectivelly learning along the execution of the generations. This is done by comparing a BRKGA against a purely random algorithm. Figure 5.16 shows the distributions of the objective function values of the 100-element population of a BRKGA and the repeated generation of sets of 100 random solutions for instance i3.200. The random solutions are generated with the same code using the BRKGA parameters $p = 101$, $p_e = 1$, and $p_m = 100$. This way, the mutants are the random solutions, the best solution is saved in the elite set, and no crossover is ever done. This figure shows not only that BRKGA executes faster than the random heuristic, but also finds an overall better solution. The frequency distributions of the fitness of the chromossomes and the descriptive statistics for both heuristics are shown, respectively, on Figures 5.17 and 5.18 and on Table 5.8. It is possible to see that the random heuristic has a much worse performance, both in terms of fitness and time.

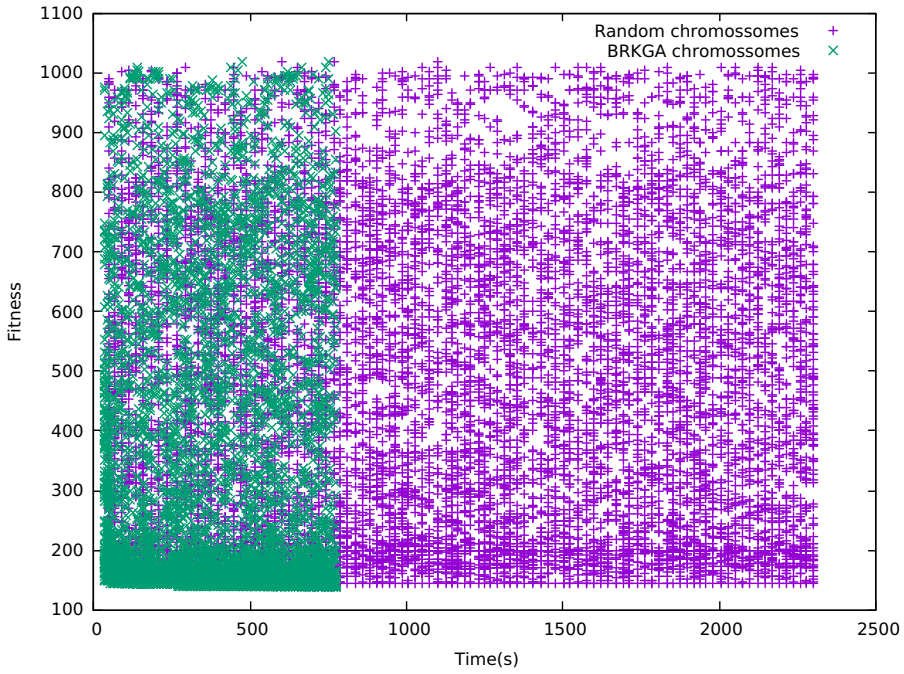


Figure 5.16: Comparing BRKGA with a random multistart heuristic on instance i3.200

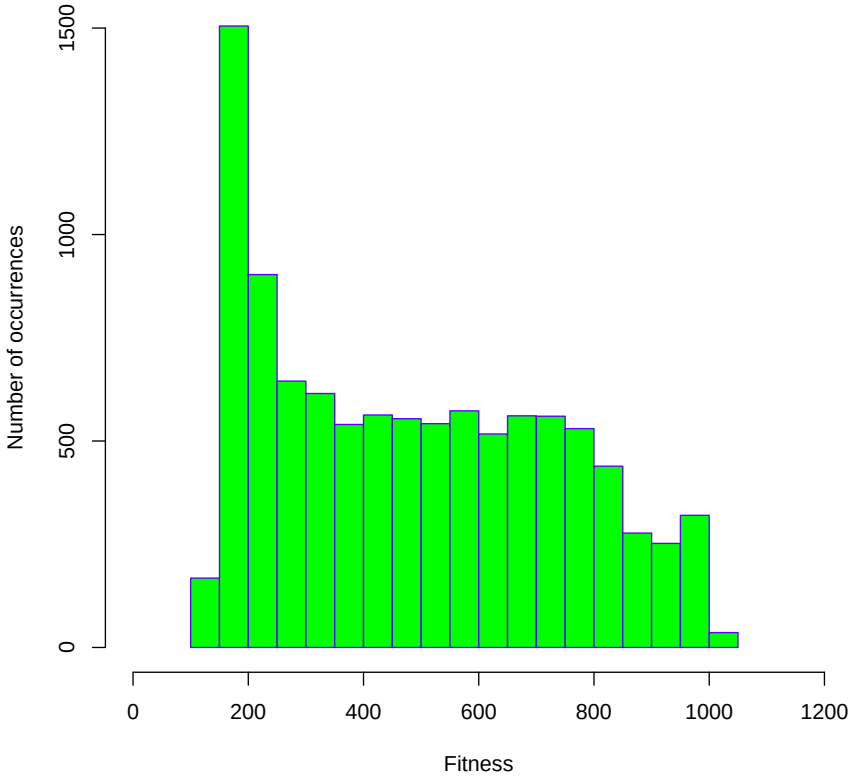


Figure 5.17: Frequency distribution of the fitness of the chromossomes for the random heuristic.

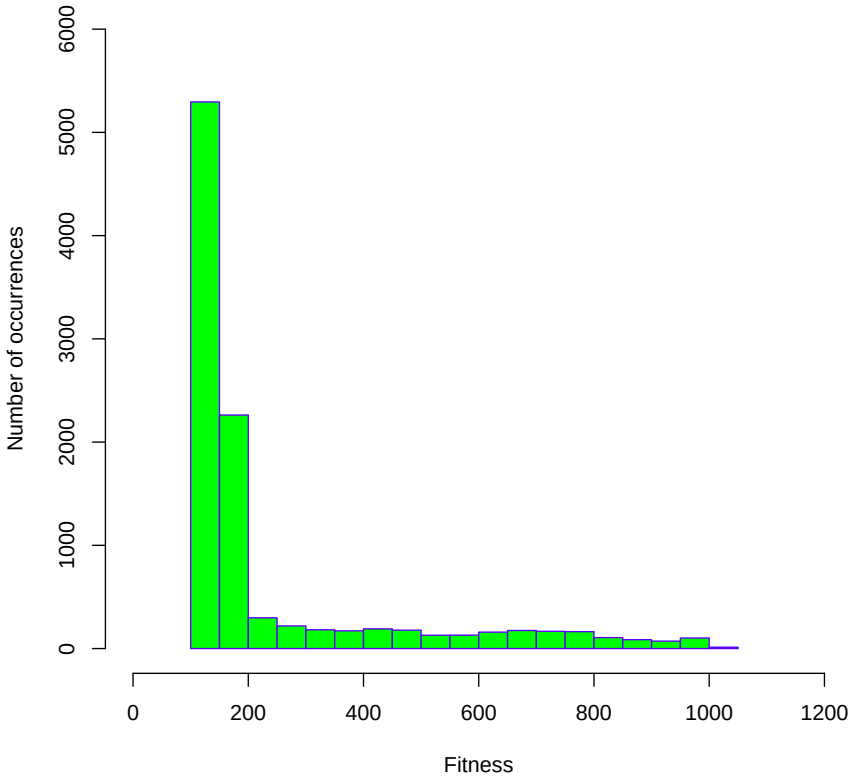


Figure 5.18: Frequency distribution of the fitness of the chromossomes for BRKGA.

Statistic	Random heuristic	BRKGA
Mean	485.74	248.41
Mode	200	144
Median	461	146
Best fitness	144	140

Table 5.8: Descriptive statistics for random heuristic and BRKGA

Table 5.9: Results for the instances with 140 and 160 vertices

	BRKGA		Bounded BRKGA		Bounded BRKGA + LS	
Instance	Time	Best SOD	Time	Best SOD	Time	Best SOD
i1.140	815.023	110	238.602	110	579.104	110
i2.140	633.346	104	249.023	104	504.723	104
i3.140	422.745	114	217.511	114	440.404	114
i4.140	442.931	101	256.324	101	478.874	101
i5.140	391.795	104	264.686	104	477.126	104
i6.140	498.64	110	243.298	110	453.478	110
i7.140	431.715	124	175.625	124	365.353	124
i8.140	459.514	117	218.759	117	432.043	117
i9.140	500.339	115	234.765	115	452.463	115
i10.140	513.943	120	240.49	117	436.894	117
i1.160	1587.787	117	286.603	117	657.4	116
i2.160	850.373	128	262.938	128	531.384	128
i3.160	1084.233	124	285.465	124	561.757	124
i4.160	1214.509	123	283.078	123	605.858	123
i5.160	1375.658	130	270.848	130	516.564	130
i6.160	1540.472	129	266.994	129	531.399	129
i7.160	895.036	120	293.218	120	570.135	119
i8.160	784.619	124	268.086	125	560.321	124
i9.160	510.932	114	297.633	114	559.479	114
i10.160	2293.001	113	301.579	113	581.288	113

In the second experiment we compare the three BRKGA heuristics: BRKGA, Bounded BRKGA and Bounded BRKGA with local search. The parameters for Bounded BRKGA and Bounded BRKGA + LS were set to the same values as in BRKGA, i.e, $p_e = 0.5$, $p_m = 0.2$, and $rhoe = 0.5$. All three heuristics were executed for 100 generations, with 200 individuals per generations and restarts after 50 generations without improvement. Bounded BRKGA and Bounded BRKGA + LS used the best value found by the adaptive greedy heuristic to limit the size of the subgraphs in the populations. For Bounded BRKGA + LS, the best two individuals were selected for local search in each generation. Tables 5.9 and 5.10 show the running times and the best values found by each heuristic for instances with sizes 140 to 200 vertices.

These tables show that BRKGA finds the best value in 94 of the 100 instances, Bounded BRKGA finds the best value in 90 and Bounded BRKGA + LS finds the best values in all 100

Table 5.10: Results for the instances with 180 and 200 vertices

	BRKGA		Bounded BRKGA		Bounded BRKGA + LS	
Instance	Time	Best SOD	Time	Best SOD	Time	Best SOD
i1.180	971.257	149	293.951	149	589.665	149
i2.180	959.043	123	329.971	123	629.82	123
i3.180	1730.199	124	346.539	124	703.468	124
i4.180	1387.326	139	343.715	139	680.239	138
i5.180	1293.367	134	316.571	134	675.216	134
i6.180	945.455	150	276.401	150	627.995	150
i7.180	1202.06	142	286.37	142	746.009	138
i8.180	1202.887	142	305.651	142	618.619	142
i9.180	3566.01	124	353.029	124	700.161	124
i10.180	1030.319	136	313.686	136	671.363	136
i1.200	1806.514	144	396.162	145	891.682	144
i2.200	1744.723	142	379.892	142	833.821	142
i3.200	1492.782	140	393.292	140	882.603	140
i4.200	1036.575	152	324.745	155	789.736	152
i5.200	1219.813	142	358.317	143	863.696	142
i6.200	1181.312	148	341.796	148	739.269	148
i7.200	1088.227	141	355.805	141	807.956	141
i8.200	1483.859	141	367.459	141	723.592	141
i9.200	1186.226	147	357.599	147	778.488	147
i10.200	1166.336	154	340.627	154	693.25	154

instances. Figure 5.19 shows the average execution times for the three heuristics. The bounded versions have a much smaller average execution times, when compared to BRKGA, because of the reduction in the search space. The local phase in the BRKGA with local search heuristic makes its average execution time larger than the Bounded BRKGA.

In the next experiment, BRKGA, Bounded BRKGA and Bounded BRKGA + LS were executed for a fixed execution time equal to the time taken by BRKGA to perform 100 generations. Tables 5.11 and 5.12 show the results for the instances with sizes 140 to 200. It was observed in this experiment that BRKGA finds the best result in 94 out of the 100 instances, Bounded BRKGA finds the best value in 93 and Bounded BRKGA + LS finds the best values in all 100 instances.

Tables 5.13 to 5.16 compare the results of the Bounded BRKGA + LS with the GRASP heuristic, presented in Chapter 3. Both heuristics were executed for the same 100 instances previously mentioned. Each instance was executed three times, for a fixed execution time equal to the time taken by GRASP to perform 100 iterations. These tables show that there were 98 ties out of the 100 instances, GRASP found a better solution than Bounded BRKGA + LS in one instance and the Bounded BRKGA + LS found a better solution in one instance.

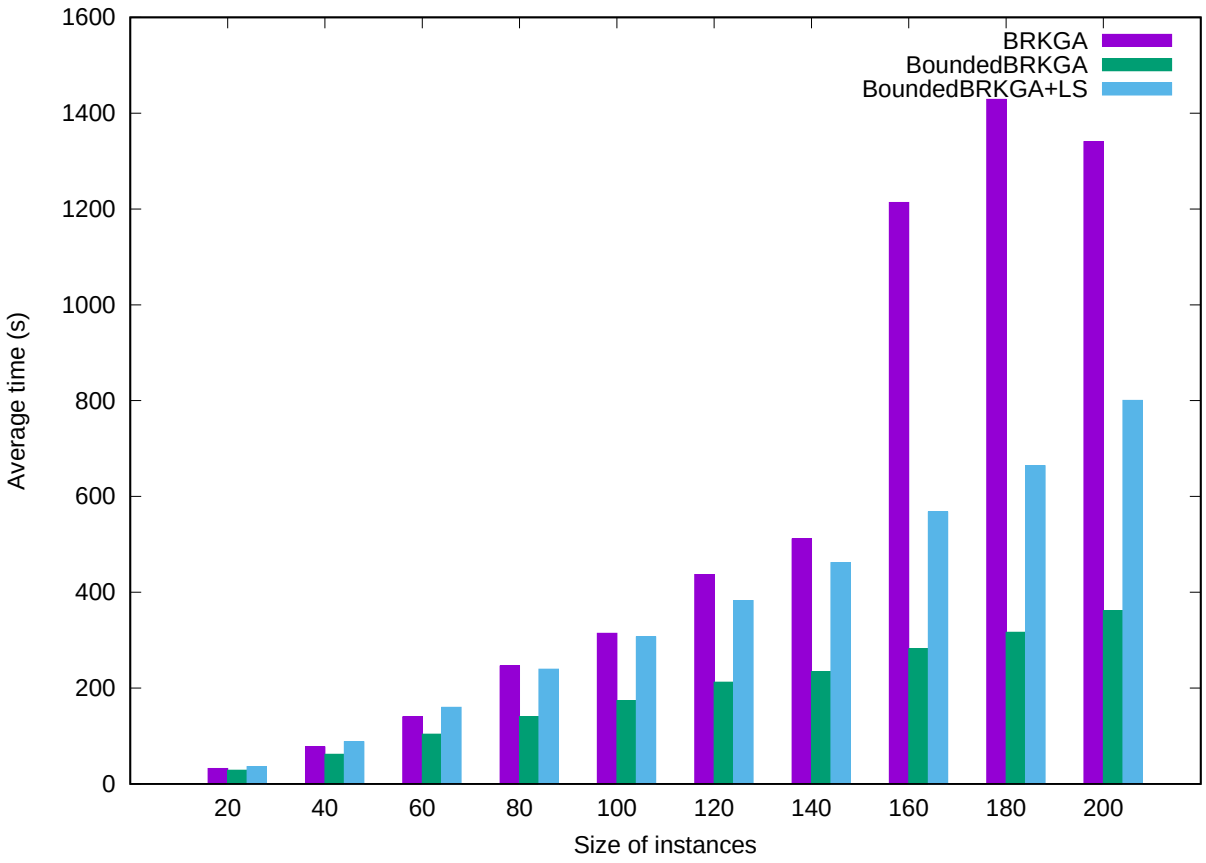


Figure 5.19: Average execution times for the BRKGA heuristics.

Table 5.11: Results for the instances with 140 and 160 vertices.

		BRKGA	Bounded BRKGA	Bounded BRKGA + LS
Instance	Execution time	Best SOD	Best SOD	Best SOD
i1.140	815.023	110	110	110
i2.140	633.346	104	104	104
i3.140	422.745	114	114	114
i4.140	442.931	101	101	101
i5.140	391.795	104	104	104
i6.140	498.64	110	110	110
i7.140	431.715	124	124	124
i8.140	459.514	117	117	117
i9.140	500.339	115	115	115
i10.140	513.943	120	117	117
i1.160	1587.787	117	116	116
i2.160	850.373	128	128	128
i3.160	1084.233	124	124	124
i4.160	1214.509	123	123	123
i5.160	1375.658	130	130	130
i6.160	1540.472	129	129	129
i7.160	895.036	120	120	119
i8.160	784.619	124	125	124
i9.160	510.932	114	114	114
i10.160	2293.001	113	113	113

Table 5.12: Results for the instances with 180 and 200 vertices.

		BRKGA	Bounded BRKGA	Bounded BRKGA + LS
Instance	Execution time	Best SOD	Best SOD	Best SOD
i1.180	971.257	149	149	149
i2.180	959.043	123	123	123
i3.180	1730.199	124	124	124
i4.180	1387.326	139	139	138
i5.180	1293.367	134	134	134
i6.180	945.455	150	150	150
i7.180	1202.06	142	142	138
i8.180	1202.887	142	142	142
i9.180	3566.01	124	124	124
i10.180	1030.319	136	136	136
i1.200	1806.514	144	145	144
i2.200	1744.723	142	142	142
i3.200	1492.782	140	140	140
i4.200	1036.575	152	152	152
i5.200	1219.813	142	142	142
i6.200	1181.312	148	148	148
i7.200	1088.227	141	141	141
i8.200	1483.859	141	141	141
i9.200	1186.226	147	147	147
i10.200	1166.336	154	154	154

Table 5.13: Results for the instances with 20, 40, and 60 vertices.

						GRASP	Bounded BRKGA + LS
Instance	Vertices	#(MinSup(S))	SOD(MinSup(S),S)	SOD($\hat{G}$,S)	Execution time (s)	Average *SOD*	Average *SOD*
i01	20	14	22	15	27.00	12	12
i02	20	15	40	22	21.02	20	20
i03	20	18	52	22	20.24	20	20
i04	20	13	32	14	18.79	14	14
i05	20	11	24	10	20.21	10	10
i06	20	14	36	18	20.51	18	18
i07	20	11	24	16	13.63	16	16
i08	20	16	60	23	7.25	20	20
i09	20	16	60	24	7.59	20	20
i10	20	14	50	22	9.67	19	19
i01	40	22	92	38	73.57	34	34
i02	40	23	98	38	73.10	30	30
i03	40	23	98	40	74.88	38	38
i04	40	22	92	36	63.52	30	30
i05	40	22	92	36	59.84	30	30
i06	40	21	107	39	53.83	33	33
i07	40	19	93	38	44.42	34	34
i08	40	18	86	34	62.90	31	31
i09	40	24	152	50	50.38	40	40
i10	40	22	136	38	52.41	36	36
i01	60	30	150	64	210.33	48	48
i02	60	26	122	50	166.98	39	39
i03	60	28	164	48	186.79	44	44
i04	60	29	172	54	132.08	46	46
i05	60	20	200	46	102.91	45	45
i06	60	27	183	58	99.38	51	51
i07	60	24	180	52	117.31	48	48
i08	60	23	193	52	109.85	45	45
i09	60	22	182	50	101.33	47	47
i10	60	32	324	76	58.68	60	60

Table 5.14: Results for the instances with 80, 100, and 120 vertices.

						GRASP	Bounded BRKGA + LS
Instance	Vertices	#(MinSup(S))	SOD(MinSup(S),S)	SOD($\hat{G}$,S)	Execution time (s)	Average *SOD*	Average *SOD*
i01	80	33	217	65	339.51	56	56
i02	80	32	240	70	248.15	58	58
i03	80	33	283	71	291.28	62	62
i04	80	32	272	63	234.25	55	55
i05	80	32	272	75	231.82	62	62
i06	80	35	305	78	442.27	63	63
i07	80	33	283	74	327.90	66	66
i08	80	33	283	74	258.60	60	60
i09	80	36	316	72	218.75	67	67
i10	80	35	305	72	231.91	67	67
i01	100	40	340	89	491.72	72	72
i02	100	38	356	86	353.41	72	72
i03	100	38	394	101	280.22	82	82
i04	100	38	394	96	429.17	83	83
i05	100	36	368	86	352.09	70	70
i06	100	38	394	80	301.72	70	70
i07	100	34	342	81	310.02	73	73
i08	100	37	418	88	318.86	74	74
i09	100	37	418	100	290.95	82	82
i10	100	38	432	100	347.50	86	86
i01	120	42	426	113	792.77	92	92
i02	120	41	495	105	606.06	93	93
i03	120	45	555	126	458.67	100	100
i04	120	41	495	102	518.70	94	94
i05	120	40	480	106	437.78	94	94
i06	120	41	536	124	377.31	98	96
i07	120	40	520	112	382.51	102	102
i08	120	37	472	102	403.83	86	86
i09	120	42	552	106	442.24	98	98
i10	120	35	440	98	397.72	84	84

Table 5.15: Results for the instances with 140, 160, and 180 vertices.

						GRASP	Bounded BRKGA + LS
Instance	Vertices	#(MinSup(S))	SOD(MinSup(S),S)	SOD($\hat{G}$,S)	Execution time (s)	Average *SOD*	Average *SOD*
i01	140	50	710	130	862.80	110	110
i02	140	44	652	122	775.08	104	104
i03	140	43	677	132	479.07	114	114
i04	140	44	696	119	500.63	101	101
i05	140	38	582	112	455.70	104	104
i06	140	45	760	124	540.36	110	110
i07	140	49	889	151	465.92	124	124
i08	140	48	868	148	491.49	117	117
i09	140	47	847	135	541.22	115	115
i10	140	42	742	130	572.49	117	117
i01	160	53	741	139	1886.52	116	116
i02	160	54	812	170	1063.23	128	128
i03	160	50	740	140	1043.84	124	124
i04	160	49	771	145	1230.10	123	123
i05	160	49	771	156	1186.72	130	130
i06	160	55	885	151	1424.64	129	129
i07	160	47	733	143	901.651	119	119
i08	160	47	733	140	1101.49	124	124
i09	160	39	581	128	733.728	114	114
i10	160	47	733	134	1590.29	113	113
i01	180	56	996	169	1159.99	149	149
i02	180	47	807	152	1048.73	123	123
i03	180	50	870	153	1465.40	124	124
i04	180	47	807	164	1343.27	138	138.3
i05	180	54	1008	162	1331.76	134	134
i06	180	61	1162	174	995.87	150	150
i07	180	56	1052	166	1458.79	138	138
i08	180	51	942	160	1272.06	142	142
i09	180	50	920	148	3121.15	124	124
i10	180	51	942	158	1151.44	136	136

Table 5.16: Results for the instances with 200 vertices.

					GRASP	Bounded BRKGA + LS
Instance	Vertices	#(MinSup(S))	SOD(MinSup(S),S)	SOD($\hat{G}$,S)	Execution time (s)	Average *SOD*	Average *SOD*
i01	200	57	997	182	2143.07	144.3	144.3
i02	200	52	944	164	1875.64	142	142
i03	200	55	1010	166	1925.82	140	140
i04	200	59	1157	196	1413.61	152	152
i05	200	53	1019	172	1378.07	142	142
i06	200	55	1065	180	1434.84	148	148
i07	200	52	996	176	1313.66	141	141
i08	200	49	927	171	1322.63	141	141
i09	200	53	1019	183	1402.86	147	147
i10	200	54	1042	190	1373.83	154	154

In the next experiment we assess the behavior of both the Bounded BRKGA + LS heuristic and the GRASP heuristic using time-to-target plots. Two hundred independent runs have been performed for each algorithm. Each run was terminated when a solution with value less than or equal to a given target was found. The perl program *tttplots-compare* [60], developed to compare time-to-target plots or general runtime distribution for measured CPU times of any two heuristics based on stochastic local search, was used to compare the results obtained by Bounded BRKGA + LS and GRASP. Given two algorithms A_1 and A_2, *tttplots-compare* gives the probability that algorithm A_1 finds a solution at least as good as a given target value in a smaller computation time than A_2, for the case where the runtimes of the two algorithms follow any general runtime distribution. The continuous random variable denoted by X_1 (resp. X_2) represents the time needed by algorithm A_1 (resp. A_2) to find a solution at least as good as a given target value. The probability that GRASP finds a solution in a smaller computational time than Bounded BRKGA + LS is shown in Tables 5.17 and 5.18, for 50 instances, with sizes of 100 to 180 vertices. These tables show that, out of the 50 instances, GRASP is more likely to find the target faster than Bounded BRKGA + LS in 40 instances (80% of the cases). Figures 5.20 to 5.24 show the superimposed runtime distributions of Bounded BRKGA + LS and GRASP for five of these instances.

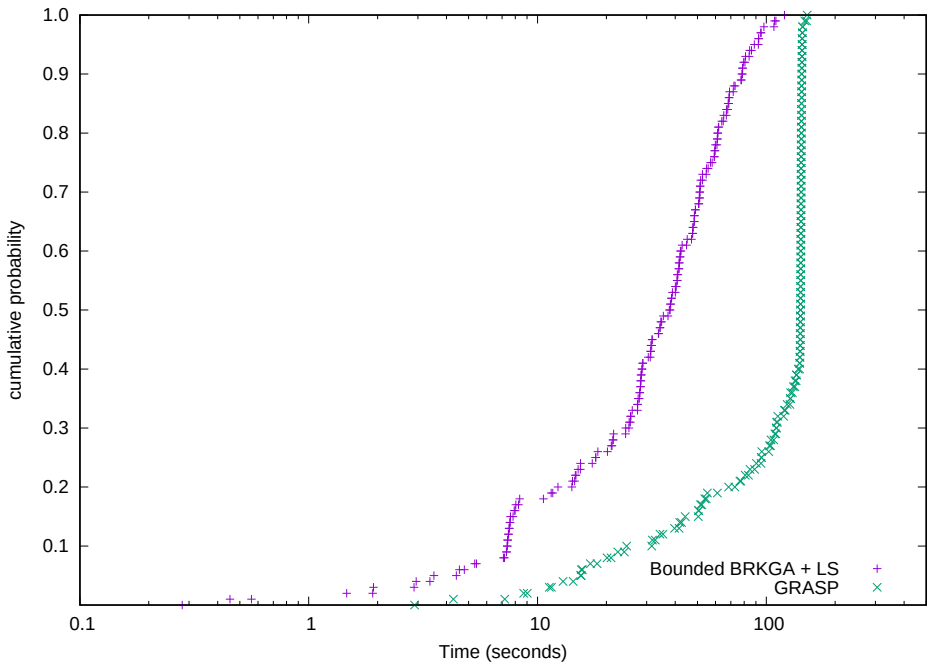


Figure 5.20: Runtime distribution from 200 runs of Bounded BRKGA + LS and GRASP for instance i6.120 with a total of 120 vertices and a target value set at 98 (best known value is 96). For this instance, $Pr(X_1 \leq X_2) = 0.12$.

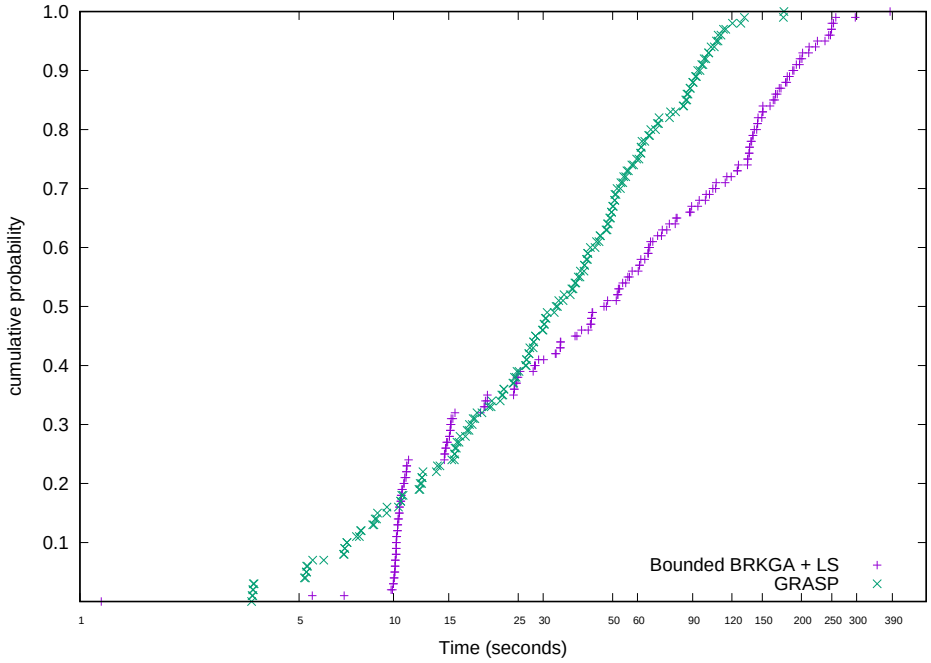


Figure 5.21: Runtime distribution from 200 runs of Bounded BRKGA + LS and GRASP for instance i5.140 with a total of 140 vertices and a target value set at 104 (best known value is 104). For this instance, $Pr(X_1 \leq X_2) = 0.59$.

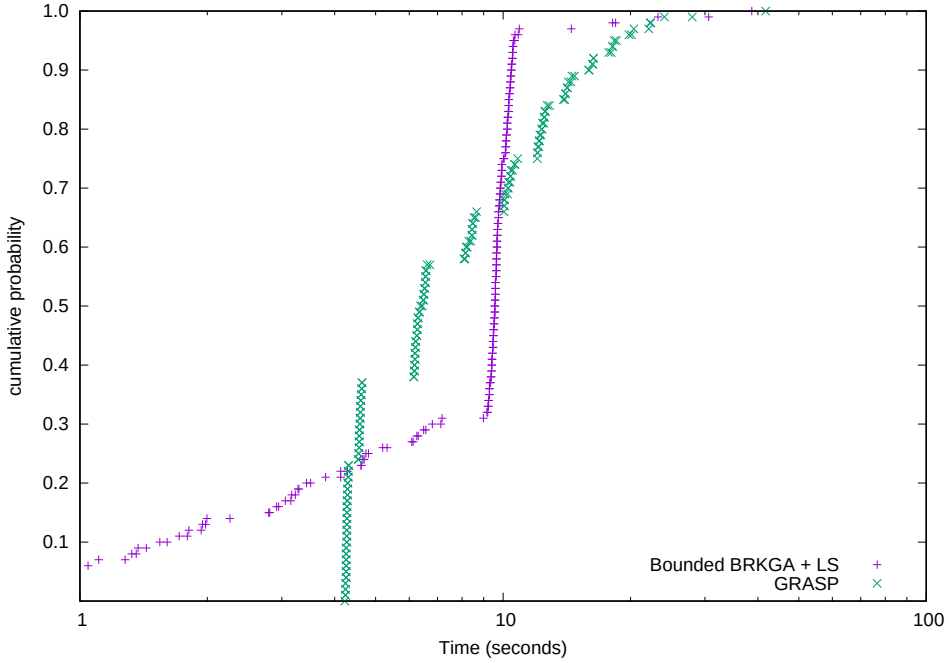


Figure 5.22: Runtime distribution from 200 runs of Bounded BRKGA + LS and GRASP for instance i9.140 with a total of 140 vertices and a target value set at 115 (best known value is 115). For this instance, $Pr(X_1 \leq X_2) = 0.51$.

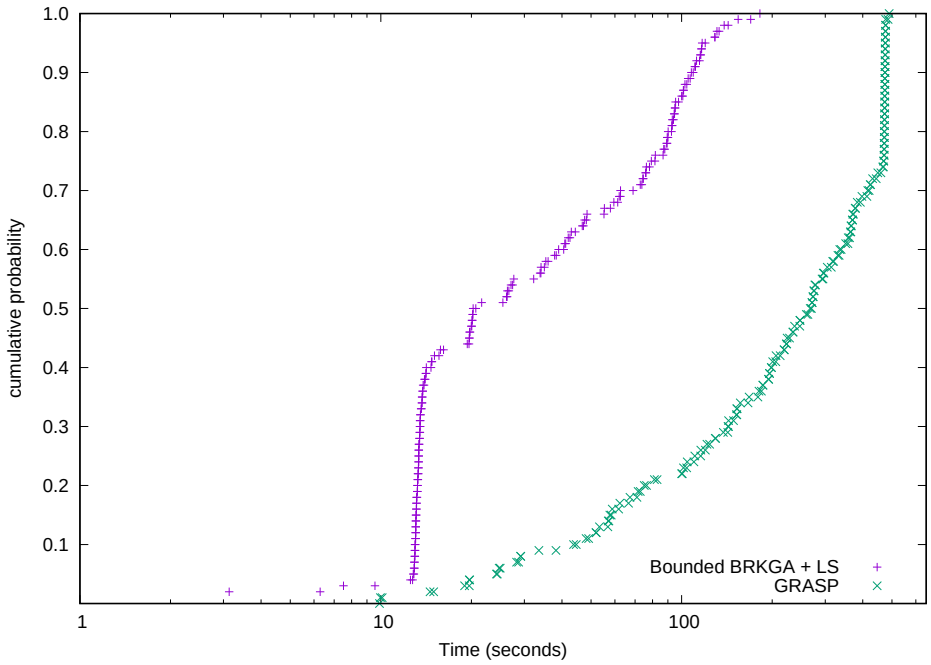


Figure 5.23: Runtime distribution from 200 runs of Bounded BRKGA + LS and GRASP for instance i5.180 with a total of 180 vertices and a target value set at 134 (best known value is 134). For this instance, $Pr(X_1 \leq X_2) = 0.09$.

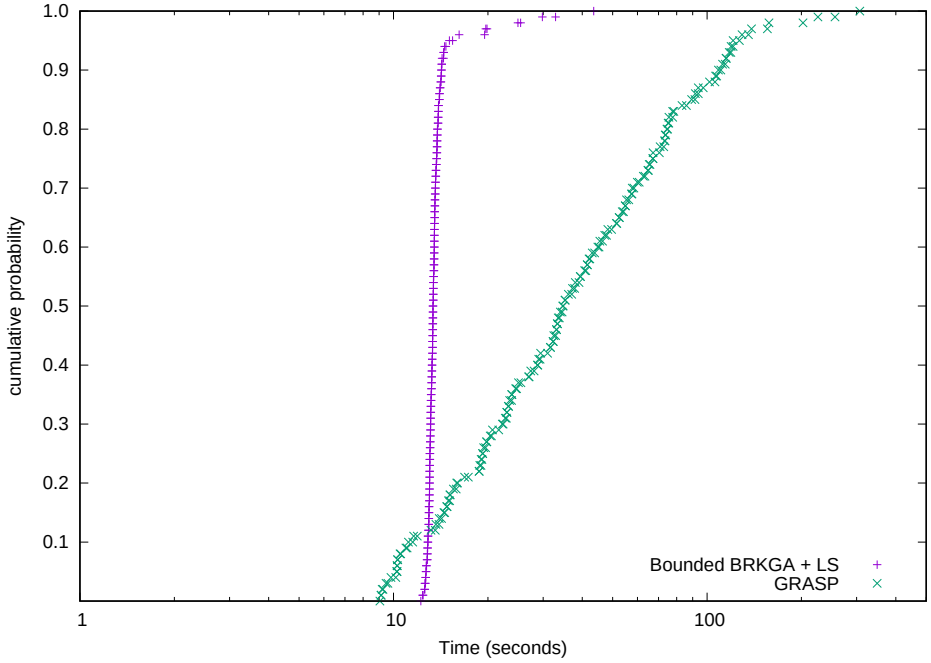


Figure 5.24: Runtime distribution from 200 runs of Bounded BRKGA + LS and GRASP for instance i8.180 with a total of 180 vertices and a target value set at 142 (best known value is 142). For this instance, $Pr(X_1 \leq X_2) = 0.13$.

5.6 Conclusions

In this chapter three heuristics for the generalized median graph were proposed: a heuristic based on the BRKGA metaheuristic, a variant of the BRKGA heuristic, called Bounded BRKGA, that uses a theoretical result to reduce the search space of the problem, and Bounded BRKGA with a local search phase inserted between any two consecutive generations. These heuristics were compared among themselves, showing that Bounded BRKGA and Bounded BRKGA + LS provided the best results in terms of execution time, and Bounded BRKGA + LS provided the best solutions among the three heuristics. Bounded BRKGA + LS was then compared with GRASP, with both heuristics presenting equivalent results in terms of solution quality. In the comparison using time-to-target plots, GRASP showed to be more likely to find target values in less computational times than Bounded BRKGA + LS in 80% of the tested instances. The theoretical result that served as the basis for Bounded BRKGA was able to significantly reduce the search space of the algorithm, with a strong effect in reducing the execution times of the heuristic.

Table 5.17: Probabilities that GRASP finds a solution at least as good as the target value in a smaller computational time than Bounded BRKGA + LS (instances with sizes 100 to 120).

Instance	Target value	$Pr(X_1 \leq X_2)$
i1.100	72	0.55
i2.100	72	0.98
i3.100	82	0.98
i4.100	83	0.93
i5.100	70	0.79
i6.100	70	0.98
i7.100	73	0.14
i8.100	74	0.83
i9.100	82	0.85
i10.100	86	0.64
i1.120	92	0.98
i2.120	93	0.94
i3.120	100	0.42
i4.120	94	0.74
i5.120	94	0.75
i6.120	96	0.12
i7.120	102	0.13
i8.120	86	0.82
i9.120	98	0.94
i10.120	84	0.67

Table 5.18: Probabilities that GRASP finds a solution at least as good as the target value in a smaller computational time than Bounded BRKGA + LS (instances with sizes 140 to 180).

Instance	Target value	$Pr(X_1 \leq X_2)$
i1.140	110	0.81
i2.140	104	0.84
i3.140	114	0.87
i4.140	101	0.76
i5.140	104	0.59
i6.140	110	0.54
i7.140	124	0.40
i8.140	117	0.72
i9.140	115	0.51
i10.140	117	0.97
i1.160	116	0.43
i2.160	128	0.73
i3.160	124	0.37
i4.160	123	0.98
i5.160	130	0.85
i6.160	129	0.09
i7.160	119	0.94
i8.160	124	0.99
i9.160	114	0.96
i10.160	113	0.52
i1.180	149	0.59
i2.180	123	0.92
i3.180	124	0.92
i4.180	138	0.99
i5.180	134	0.09
i6.180	150	0.77
i7.180	138	0.88
i8.180	142	0.13
i9.180	124	0.99
i10.180	136	0.75

Chapter 6

Concluding remarks

In this work, five heuristics for the generalized median graph problem were presented. One of them was based on a greedy strategy, another on the GRASP metaheuristic, and the others on the BRKGA metaheuristic. The BRKGA heuristics consisted of a pure BRKGA, a variant called Bounded BRKGA, and the Bounded BRKGA with a local search. The instances used in the computational experiments consisted of molecules related to the AIDS virus. The heuristics were executed in larger instances than the ones used in exact algorithms, and the results showed that the approximate generalized median graphs computed by the heuristics were of good quality, being able to be used in an application involving a classification task. In the comparison with the set median graph, the graphs obtained by the heuristics presented a superior quality. In the comparison of the five heuristics, GRASP and the Bounded BRKGA with local search heuristics were both superior to the greedy heuristic, and both presented similar results in terms of solution quality.

Two theoretical results were also presented in this work. The first one gives a bound to the sum of distances of a graph, and a practical use of this result is also presented: in conjunction with a previously computed solution, it is used to eliminate candidate graphs of poor quality from the search space of Bounded BRKGA and Bounded BRKGA with local search. Usually, these graphs of poor quality have a large number of nodes. As a consequence of eliminating these large sized graphs, Bounded BRKGA and Bounded BRKGA with local search presented smaller computational times when compared with BRKGA.

The second theoretical result, presented in Appendix A, shows that the empty graph G_e is the generalized median graph of a set S, if the graphs in S satisfy a certain property.

As future work, BRKGA and its variants can be improved in terms of efficiency. As mentioned, the GRASP heuristic uses a technique to avoid recomputing the distances between two

graphs whenever possible. The inclusion of this technique in BRKGA can possibly speed up the computation of the distances and consequently lessen the execution times. Data mining techniques may also be used to improve BRKGA and its variants. One possible idea is to extract patterns that appear frequently in good solutions and use them in post-optimization processes (for example, in a path-relinking procedure).

The results obtained by the first theoretical result can be further explored. More specifically, it can be used in other metaheuristic algorithms, such as tabu search and simulated annealing, to possibly reduce the search spaces and the computation times. The search space reduction can also be implemented in matheuristic algorithms, where the conditions on the number of nodes in the solutions are obtained by incorporating a constraint in the mathematical model.

Still as future work, approximative algorithms for the generalized median graph problem can be researched.

References

[1] AARTS, E. H. L.; VAN LAARHOVEN, P. J. M.; LENSTRA, J. K.; ULDER, N. L. J. A computational study of local search algorithms for job shop scheduling. *ORSA Journal on Computing 6* (1994), 118–125.

[2] AIEX, R. M.; RESENDE, M.; RIBEIRO, C. Probability distribution of solution time in GRASP: An experimental investigation. *Journal of Heuristics 8* (2002), 343–373.

[3] AIEX, R. M.; RESENDE, M.; RIBEIRO, C. TTTPLOTS: A perl program to create time-to-target plots. *Optimization Letters 1* (2007), 355–366.

[4] ALLAHYARI, S.; SALARI, M.; VIGO, D. A hybrid metaheuristic algorithm for the multi-depot covering tour vehicle routing problem. *European Journal of Operational Research 242* (2015), 756–768.

[5] ALPAYDIN, E. *Introduction to Machine Learning*, 2nd ed. The MIT Press, 2010.

[6] BALAS, E.; YU, C. S. Finding a maximum clique in an arbitrary graph. *SIAM Journal on Computing 15* (1986), 1054–1068.

[7] BEAN, J. C. Genetic algorithms and random keys for sequencing and optimization. *INFORMS Journal on Computing 6* (1994), 154–160.

[8] BENGOETXEA, E. *Inexact Graph Matching Using Estimation of Distribution Algorithms*. Tese de Doutorado, Ecole Nationale Supérieure des Télécommunications, Paris, 2002.

[9] BERRETTI, S.; BIMBO, A. D.; VICARIO, E. Efficient matching and indexing of graph models in content-based retrieval. *IEEE Transactions Pattern Analysis Machine Intelligence 23* (2001), 1089–1105.

[10] BRANDÃO, J. S.; NORONHA, T. F.; RESENDE, M.; RIBEIRO, C. A biased random-key genetic algorithm for scheduling heterogeneous multi-round systems. *International Transactions in Operational Research 24* (2017), 1061–1077.

[11] BUNKE, H. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters 18* (1997), 689–694.

[12] BUNKE, H. Graph representation for intelligent information processing - Fundamentals and algorithms for classification and clustering, 2011. Online reference available at http://cvpr-ss-2010.cecs.anu.edu.au/pdfs/HorstBunke.pdf, last visited on March 12, 2018.

[13] BUNKE, H.; FOGGIA, P.; GUIDOBALDI, C.; SANSONE, C.; VENTO, M. A comparison of algorithms for maximum common subgraph on randomly connected graphs. *Lecture Notes in Computer Science 2396* (2002), 123–132.

[14] BUNKE, H.; JIANG, X.; KANDEL, A. On the minimum common supergraph of two graphs. *Computing 65* (2000), 13–25.

[15] BUNKE, H.; RIESEN, K. Towards the unification of structural and statistical pattern recognition. *Pattern Recognition Letters 33* (2012), 811–825.

[16] CONTE, D.; FOGGIA, P.; SANSONE, C.; VENTO, M. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence 18* (2004), 265–298.

[17] CONTE, D.; FOGGIA, P.; VENTO, M. 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* (2007), 99–143.

[18] CROCE, F. D.; TADEI, R.; VOLTA, G. A genetic algorithm for the job shop problem. *Computers Operations Research 22* (1995), 15 – 24.

[19] DE LA HIGUERA, C.; CASACUBERTA, F. Topology of strings: Median string is NP-complete. *Theoretical Computer Science 230* (2000), 39–48.

[20] DORNDORF, U.; PESCH, E. Evolution based learning in a job shop scheduling environment. *Computers Operations Research 22* (1995), 25 – 40.

[21] DUDA, R. O.; HART, P. E.; STORK, D. G. *Pattern Classification*, 2nd ed. Wiley, New York, 2000.

[22] DURAND, P.; PASARI, R.; BAKER, J. W.; TSAI, C.-C. An efficient algorithm for similarity analysis of molecules. *Internet Journal of Chemistry 2*, 17 (1999). Online reference available at http://www.cs.kent.edu/∼jbaker/paper, last visited on March 12, 2018.

[23] ERICSSON, M.; RESENDE, M.; PARDALOS, P. M. A genetic algorithm for the weight setting problem in OSPF routing. *Journal of Combinatorial Optimization 6* (2002), 299–333.

[24] FAN, K. C.; LIU, C. W.; WANG, Y. K. A fuzzy bipartite weighted graph matching approach to fingerprint verification. In *Proceedings of the 1998 IEEE International Conference on Systems, Man, and Cybernetics* (San Diego, 1998), vol. 5, IEEE, pp. 4363–4368.

[25] FEO, T.; RESENDE, M. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters 8* (1989), 67–71.

[26] FEO, T.; RESENDE, M. Greedy randomized adaptive search procedures. *Journal of Global Optimization 6* (1995), 109–133.

[27] FERRER, M. *Theory and Algorithms on the Median Graph - Application to Graph-based Classification and Clustering*. Tese de Doutorado, Universitat Autonoma de Barcelona, Belaterra, 2008.

[28] FERRER, M.; VALVENY, E.; SERRATOSA, F. Median graph: A new exact algorithm using a distance based on the maximum common subgraph. *Pattern Recognition Letters 30* (2009), 579–588.

[29] FERRER, M.; VALVENY, E.; SERRATOSA, F. Median graphs: A genetic approach based on new theoretical properties. *Pattern Recognition 42* (2009), 2003–2012.

[30] FERRER, M.; VALVENY, E.; SERRATOSA, F.; RIESEN, K.; BUNKE, H. An approximate algorithm for median graph computation using graph embedding. In *19th International Conference on Pattern Recognition* (Tampa, 2008), vol. 2, pp. 1–4.

[31] FERRER, M.; VALVENY, E.; SERRATOSA, F.; RIESEN, K.; BUNKE, H. Generalized median graph computation by means of graph embedding in vector spaces. *Pattern Recognition 43* (2010), 1642–1655.

[32] FESTA, P.; RESENDE, M. GRASP: An annotated bibliography. In *Essays and Surveys in Metaheuristics*, C. Ribeiro and P. Hansen, Eds. Kluwer, 2002, pp. 325–367.

[33] FESTA, P.; RESENDE, M. An annotated bibliography of GRASP, Part I: Algorithms. *International Transactions in Operational Research 16* (2009), 1–24.

[34] FESTA, P.; RESENDE, M. An annotated bibliography of GRASP, Part II: Applications. *International Transactions in Operational Research 16* (2009), 131–172.

[35] FISCHER, S.; GILOMEN, K.; BUNKE, H. Identification of diatoms by grid graph matching. In *Proceedings of the Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition* (London, 2002), Springer, pp. 94–103.

[36] FOGGIA, P.; PERCANNELLA, G.; VENTO, M. Graph matching and learning in pattern recognition in the last 10 years. *International Journal of Pattern Recognition and Artificial Intelligence 28* (2014), 1450001–1–1450001–40.

[37] FUKUNAGA, K. *Introduction to Statistical Pattern Recognition*, 2nd ed. Academic Press, San Diego, 1990.

[38] GAREY, M.; JOHNSON, D. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, 1979.

[39] GONÇALVES, J. F.; RESENDE, M. Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics 17* (2011), 487–525.

[40] GONÇALVES, J. F.; DE MAGALHÃES MENDES, J. J.; RESENDE, M. A hybrid genetic algorithm for the job shop scheduling problem. *European Journal of Operational Research 167* (2005), 77 – 95.

[41] HLAOUI, A.; WANG, S. A new median graph algorithm. *Lecture Notes in Computer Science 2726* (2003), 225–234.

[42] HOLLAND, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, 1992.

[43] HONG, P.; WANG, R.; HUANG, T. Learning patterns from images by combining soft decisions and hard decisions. In *Proceedings of the Conference on Computer Vision and Pattern Recognition* (Hilton Head, 2000), vol. 1, IEEE Computer Society, pp. 79–83.

[44] JIANG, X.; MUNGER, A.; BUNKE, H. On median graphs: Properties, algorithms, and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence 23* (2001), 1144–1151.

[45] LADES, M.; VORBRUGGEN, J. C.; BUHMANN, J.; LANGE, J.; VON DER MALSBURG, C.; P., R.; WURZ; KONEN, W. Distortion invariant object recognition in the dynamic link architecture. *IEEE Transactions on Computers 42* (1993), 300–311.

[46] MCGREGOR, J. J. Backtrack search algorithms and the maximal common subgraph problem. *Software Practice and Experience 12* (1982), 23–34.

[47] MONTGOMERY, D. C.; RUNGER, G. C. *Applied Statistics and Probability for Engineers*. John Wiley and Sons, 2003.

[48] MUKHERJEE, L.; SINGH, V.; PENG, J.; XU, J.; ZEITZ, M. J.; BEREZNEY, R. Generalized median graphs: Theory and applications. In *Proceedings of the IEEE 11th International Conference on Computer Vision* (Buffalo, 2007), IEEE, pp. 1–8.

[49] MUKHERJEE, L.; SINGH, V.; PENG, J.; XU, J.; ZEITZ, M. J.; BEREZNEY, R. Generalized median graphs and applications. *Journal of Combinatorial Optimization 17* (2009), 21–44.

[50] MUSMANNO, L. Approximate algorithms for the generalized median graph problem (in Portuguese). Master's thesis, Universidade Federal Fluminense, Niterói, 2013.

[51] MUSMANNO, L. M.; RIBEIRO, C. Heuristics for the generalized median graph problem. *European Journal of Operational Research 254* (2016), 371–384.

[52] NEUHAUS, M.; RIESEN, K.; BUNKE., H. Fast suboptimal algorithms for the computation of graph edit distance. *Lecture Notes in Computer Science 4109* (2006), 163–172.

[53] NGUYEN, V.-P.; PRINS, C.; PRODHON, C. Solving the two-echelon location routing problem by a GRASP reinforced by a learning process and path relinking. *European Journal of Operational Research 216* (2012), 113–126.

[54] REBAGLIATI, N.; SOLÉ-RIBALTA, A.; PELILLO, M.; SERRATOSA, F. On the relation between the common labelling and the median graph. In *Proceedings of the 2012 Joint IAPR International Conference on Structural, Syntactic, and Statistical Pattern Recognition* (2012), Springer, pp. 107–115.

[55] RESEARCH GROUP ON COMPUTER VISION AND ARTIFICIAL INTELLIGENCE. IAM graph database repository, 2011. online reference at http://www.iam.unibe.ch/fki/databases/iam-graph-database, last visited on March 12, 2018.

[56] RESENDE, M.; RIBEIRO, C. Greedy randomized adaptive search procedures. In *Handbook of Metaheuristics*, F. Glover and G. Kochenberger, Eds. Kluwer, 2002, pp. 219–249.

[57] RESENDE, M.; RIBEIRO, C. GRASP with path-relinking: Recent advances and applications. In *Metaheuristics: Progress as Real Problem Solvers*, T. Ibaraki, K. Nonobe, and M. Yagiura, Eds. Springer, 2005, pp. 29–63.

[58] RESENDE, M.; RIBEIRO, C. Greedy randomized adaptive search procedures: Advances, hybridizations, and applications. In *Handbook of Metaheuristics*, M. Gendreau and J.-Y. Potvin, Eds., 2nd ed. Springer, 2010, pp. 283–319.

[59] RESENDE, M.; RIBEIRO, C. GRASP: Greedy Randomized Adaptive Search Procedures. In *Search Methodologies*, E. Burke and G. Kendall, Eds., 2nd ed. Springer, 2014, ch. 11, pp. 285–310.

[60] RIBEIRO, C.; ROSSETI, I.; VALLEJOS, R. On the use of run time distributions to evaluate and compare stochastic local search algorithms. In *Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics* (Berlin, Heidelberg, 2009), T. Stützle, M. Birattari, and H. H. Hoos, Eds., Springer Berlin Heidelberg, pp. 16–30.

[61] SERRATOSA, F.; CORTÉS, X.; SOLÉ-RIBALTA, A. Component retrieval based on a database of graphs for hand-written electronic-scheme digitalisation. *Expert Systems with Applications 40* (2013), 2493–2502.

[62] SHEARER, K.; BUNKE, H.; VENKATESH, S. Video indexing and similarity retrieval by largest common subgraph detection using decision trees. *Pattern Recognition 34* (2001), 1075–1091.

[63] SPEARS, V. M.; JONG, K. A. D. On the virtues of parameterized uniform crossover. In *In Proceedings of the Fourth International Conference on Genetic Algorithms* (1991), pp. 230–236.

[64] SUGANTHAN, P. N.; YAN, H. Recognition of handprinted chinese characters by constrained graph matching. *Image and Vision Computing 16* (1998), 191 – 201.

[65] TORSELLO, A.; HANCOCK, E. R. Computing approximate tree edit-distance using relaxation labeling. *Pattern Recognition Letters 24* (2003), 1089–1097.

[66] TOSO, R. F.; RESENDE, M. A c++ application programming interface for biased random-key genetic algorithms. *Optimization Methods and Software 30* (2015), 81–93.

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

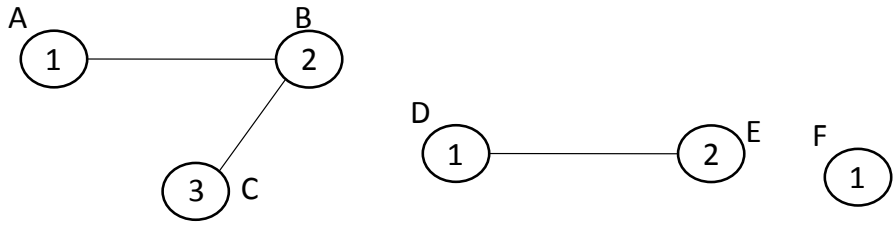
[68] VOIGT, K. Semi-automatic matching of heterogeneous model-based specifications. *Lecture Notes in Informatics P-160* (2010), 537–542. Online reference at http://subs.emis.de/LNI/Proceedings/Proceedings160/article5479.html, last visited on March 12, 2018.

[69] ZENG, Z.; TUNG, A. K. H.; WANG, J.; FENG, J.; ZHOU, L. Comparing stars: On approximating graph edit distance. *Proceedings of the VLDB Endowment 2* (2009), 25–36.

APPENDIX A – Computing the exact generalized median graph - Special case

The computation of the generalized median graph of a set S of graphs is a difficult task. This appendix illustrates a special situation where it is possible to find the generalized median graph of a set of graphs. More specifically, it was shown in [44] that $min\{SOD(G_e,S),SOD(G_u,S)\}$ is an upper bound for the SOD of the generalized median graph of S, where G_e is the empty graph and G_u is the union graph of the graphs in S. It will be proved that, in some particular cases, G_e is the actual generalized median graph of the set.

Consider a set $S = \{G_1, G_2, G_3\}$, as shown in Figure A.1.



(a) Graph G_1 (b) Graph G_2 (c) Graph G_3

Figure A.1: Graphs in set S

In this case, $\#(MaxSub(G_i,G_j)) = 0, \forall i,j, i \neq j, i,j = 1,2,3$. The goal of this section is to show that, in cases like this, where $\#(MaxSub(G_i,G_j)) = 0$ for all pairs of graphs G_i and G_j in the set, the empty graph G_e is the graph that presents the smallest sum of distances (SOD) to the graphs in set S, i.e, G_e is the generalized median graph of set S.

We first compute $SOD(G_e,S) = d(G_e,G_1) + d(G_e,G_2) + d(G_e,G_3)$:

- $d(G_e,G_1) = \#(G_e) + \#(G_1) - 2 \times \#(MaxSub(G_e,G_1)) = 3 + 0 - 2 \times 0 = 3$
- $d(G_e,G_2) = \#(G_e) + \#(G_2) - 2 \times \#(MaxSub(G_e,G_2)) = 2 + 0 - 2 \times 0 = 2$

- $d(G_e, G_3) = \#(G_e) + \#(G_3) - 2 \times \#(MaxSub(G_e, G_3)) = 1 + 0 - 2 \times 0 = 1$

Hence, $SOD(G_e, S) = \#(G_1) + \#(G_2) + \#(G_3) = 3 + 2 + 1 = 6$.

In the attempt to find a candidate graph that presents a smaller *SOD*, one idea is to take combinations (union) of subgraphs of G_1, G_2 and G_3. The expectation is that these subgraphs possibly result in smaller distances. Consider as our first candidate the graph $Cand_1$, shown in Figure A.2 (a). This graph consists of an induced subgraph $G'_1 \subset G_1$, with $\#(G'_1) = 2$, and an induced subgraph $G'_2 \subset G_2$, with $\#(G'_2) = 1$. The *SOD* of this candidate is:

- $d(Cand_1, G_1) = \#(Cand_1) + \#(G_1) - 2 \times \#(MaxSub(Cand_1, G_1)) = 3 + 3 - 2 \times 2 = 2$
- $d(Cand_1, G_2) = \#(Cand_1) + \#(G_2) - 2 \times \#(MaxSub(Cand_1, G_2)) = 2 + 3 - 2 \times 1 = 3$
- $d(Cand_1, G_3) = \#(Cand_1) + \#(G_3) - 2 \times \#(MaxSub(Cand_1, G_3)) = 1 + 3 - 2 \times 0 = 4$

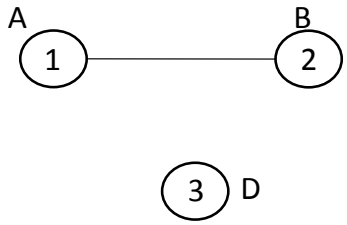
Thus we have $SOD(Cand_1, S) = 9$. If, instead, we had taken as a candidate the graph $Cand_2$ shown in Figure A.2(b), the *SOD* would be even larger, since now the distance $d(Cand, G_1)$ increased, because the subgraph of G_1 is not and induced subgraph of G_1, since the edge connecting the nodes with labels *A* and *B* is missing.

Connecting the subgraphs with edges makes no difference, as the graph $Cand_3$ in Figure A.2(c) shows. This graph in Figure A.2(c) has the same *SOD* as the graph shown in Figure A.2(a). The inserted edge (B, D) does not modify the distances, since it makes no change in the maximum common subgraphs. Inserting into the candidate a graph that is not a subgraph of any of the graphs G_i, $i = 1, 2, 3$, can only further increase the *SOD*. As an example, the graph $Cand_4$ in Figure A.2(d) presents nodes with labels *X* and *Z*, not present in any of the graphs in set *S*. Computing the *SOD* of $Cand_4$ we have:

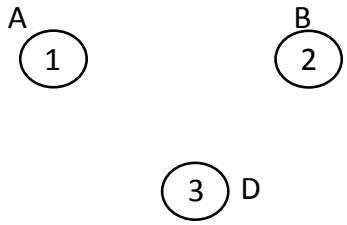
- $d(Cand_4, G_1) = \#(Cand_4) + \#(G_1) - 2 \times \#(MaxSub(Cand_4, G_1)) = 3 + 5 - 2 \times 2 = 4$
- $d(Cand_4, G_2) = \#(Cand_4) + \#(G_2) - 2 \times \#(MaxSub(Cand_4, G_2)) = 2 + 5 - 2 \times 1 = 5$
- $d(Cand_4, G_3) = \#(Cand_4) + \#(G_3) - 2 \times \#(MaxSub(Cand_4, G_3)) = 1 + 5 - 2 \times 0 = 6$

Hence, $SOD(Cand_4, S) = d(Cand_4, G_1) + d(Cand_4, G_2) + d(Cand_4, G_3) = 15$, and thus it is a worse representative to set *S* than the graph presented in Figure A.2(a).

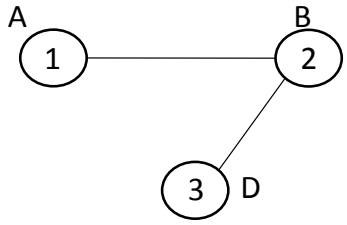
Therefore, in the search for a better representative for set *S*, the best approach is to construct a candidate by taking the union $\bigcup_{i=1}^{3} G'_i$, where each G'_i is an induced subgraph of G_i, $i = 1, 2, 3$.



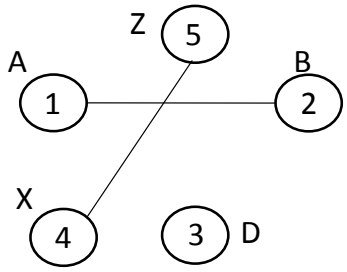
(a) Graph $Cand_1$ consists of induced subgraphs of G_1 and G_2.



(b) Graph $Cand_2$: without edge $(1,2)$, G_1's subgraph is not induced.



(c) Graph $Cand_3$ consists of the same nodes and edges of $Cand_1$, with the inclusion of edge $(2,3)$.



(d) Graph $Cand_4$: inserting nodes with labels not present in S.

Figure A.2: Possibilities in the construction of a candidate.

However, we will show that not even this approach can improve on the empty graph. Take as a candidate $Cand$ the union $\bigcup_{i=1}^{3} G'_i$, where each G'_i is an induced subgraph of G_i, $i = 1,2,3$, with $\#(G'_1) = m_1$, $\#(G'_2) = m_2$ and $\#(G'_3) = m_3$. Computing the distances from this candidate to the

graphs in S we have:

$$\begin{aligned} d(G_1, Cand) &= \#(G_1) + \#(Cand) - 2 \times \#(MaxSub(Cand, G_1)) = \\ &= \#(G_1) + (m_1 + m_2 + m_3) - 2 \times m_1 = \\ &= \#(G_1) + (m_2 + m_3 - m_1) \end{aligned}$$

$$\begin{aligned} d(G_2, Cand) &= \#(G_2) + \#(Cand) - 2 \times \#(MaxSub(Cand, G_2)) = \\ &= \#(G_2) + (m_1 + m_2 + m_3) - 2 \times m_2 = \\ &= \#(G_2) + (m_1 + m_3 - m_2) \end{aligned}$$

$$\begin{aligned} d(G_3, Cand) &= \#(G_3) + \#(Cand) - 2 \times \#(MaxSub(Cand, G_3)) = \\ &= \#(G_3) + (m_1 + m_2 + m_3) - 2 \times m_3 = \\ &= \#(G_3) + (m_1 + m_2 - m_3) \end{aligned}$$

Hence, this candidate's *SOD* is equal to:

$$\begin{aligned} SOD(Cand, S) &= d(Cand, G_1) + d(Cand, G_2) + d(Cand, G_3) = \\ &= \#(G_1) + \#(G_2) + \#(G_3) + (m_2 + m_3 - m_1) + (m_1 + m_3 - m_2) + (m_1 + m_2 - m_3) = \\ &= \#(G_1) + \#(G_2) + \#(G_3) + (m_1 + m_2 + m_3) = \\ &= SOD(G_e) + (m_1 + m_2 + m_3) \end{aligned}$$

Thus, we have $SOD(Cand, S) \geq SOD(G_e, S)$, and the equality holds only if $m_1 = m_2 = m_3 = 0$, i.e, if the candidate is the empty graph itself. Therefore, the empty graph G_e is the generalized median graph of this set.

The next property formalizes this result, showing that, in sets like set S above, where $\#(MaxSub(G_i, G_j)) = 0$, $i, j = 1, \ldots, n$, $i \neq j$, G_e is the exact generalized median graph of the set.

Proposition 2 *Let $S = \{G_1, \ldots, G_n\}$, with $n \geq 2$, be a set of labeled graphs, where $\#(MaxSub(G_i, G_j)) = 0$, $i, j = 1, \ldots, n$, $i \neq j$. Then, the empty graph is the generalized median graph of set S.*

Proof: Let $S = \{G_1, \ldots, G_n\}$ with $n \geq 2$, be a set of labeled graphs such that $\#(MaxSub(G_i, G_j)) = 0$, $i, j = 1, \ldots, m$, $i \neq j$. Initially, let's compute the *SOD* of the empty graph G_e :

$$d(G_e, G_1) = \#(G_e) + \#(G_1) - 2 \times \#(MaxSub(G_e, G_1)) = 0 + \#(G_1) - 2 \times 0 = \#(G_1)$$

$$d(G_e, G_2) = \#(G_e) + \#(G_2) - 2 \times \#(MaxSub(G_e, G_2)) = 0 + \#(G_2) - 2 \times 0 = \#(G_2)$$

$$\vdots$$

$$d(G_e, G_n) = \#(G_e) + \#(G_n) - 2 \times \#(MaxSub(G_e, G_n)) = 0 + \#(G_n) - 2 \times 0 = \#(G_n)$$

Therefore, we have that $SOD(G_e, S) = \sum_{i=1}^{n} \#(G_i)$.

As explained previously, the best attempt to find a better representative than the empty graph is to take the union of induced subgraphs of the graphs from the set. Thus, let's consider as our candidate the graph $\bigcup_{i=1}^{n} G'_i$, where each G'_i is an induced subgraph of G_i and $\#(G'_i) = m_i$, $1 \leq i \leq n$. Computing the distances from this graph to the graphs of S we have:

$$\begin{aligned} d(G_1, \bigcup_{i=1}^{n} G'_i) &= \#(G_1) + \#(\bigcup_{i=1}^{n} G'_i) - 2 \times \#(MaxSub(\bigcup_{i=1}^{n} G'_i, G_1)) = \\ &= \#(G_1) + \sum_{i=1}^{m} m_i - 2 \times m_1 \end{aligned}$$

$$\begin{aligned} d(G_2, \bigcup_{i=1}^{n} G'_i) &= \#(G_2) + \#(\bigcup_{i=1}^{n} G'_i) - 2 \times \#(MaxSub(\bigcup_{i=1}^{n} G'_i, G_2)) = \\ &= \#(G_2) + \sum_{i=1}^{m} m_i - 2 \times m_2 \end{aligned}$$

$$\vdots$$

$$\begin{aligned} d(G_n, \bigcup_{i=1}^{n} G'_i) &= \#(G_m) + \#(\bigcup_{i=1}^{n} G'_i) - 2 \times \#(MaxSub(\bigcup_{i=1}^{n} G'_i, G_n)) = \\ &= \#(G_n) + \sum_{i=1}^{n} m_i - 2 \times m_n \end{aligned}$$

Therefore, the *SOD* of this candidate is:

$$\begin{aligned} SOD(\bigcup_{i=1}^{n} G'_i) &= d(G_1, \bigcup_{i=1}^{n} G'_i) + d(G_2, \bigcup_{i=1}^{n} G'_i) + \ldots + d(G_n, \bigcup_{i=1}^{n} G'_i) = \\ &= \sum_{i=1}^{n} \#(G_i) + n \times \sum_{i=1}^{n} m_i - 2 \times (m_1 + m_2 + \ldots + m_n) = \\ &= \sum_{i=1}^{n} \#(G_i) + (n-2) \times (m_1 + m_2 + \ldots + m_n) = \\ &= SOD(G_e) + (n-2) \times (m_1 + m_2 + \ldots + m_n) \end{aligned}$$

Since $n \geq 2$, we have $SOD(G_e, S) \leq SOD(\bigcup_{i=1}^{n} G'_i, S)$. It follows that the empty graph is a generalized median graph of set S. ■

In the more general case where the set S is such that $\#(MaxSub(S)) > 0$, it can be shown that $SOD(MaxSub(S), S) < SOD(G_e, S)$.

APPENDIX B – Instances

The instances used in this work were taken from the *AIDS* group of the *IAM Graph Database Repository*.

Table B.1: Instances with number of nodes equal to 20

Instance	Graphs
i1.20	2128, 6497, 13072
i2.20	153, 973, 4493, 8117
i3.20	4901, 6075, 6614, 8117
i4.20	813, 1541, 8117, 22446
i5.20	153, 718, 4901, 14734
i6.20	41, 262, 646, 5461
i7.20	646, 718, 1540, 21840
i8.20	153, 2325, 5727, 7910, 8117
i9.20	41, 2325, 5727, 6073, 8117
i10.20	914, 5727, 6075, 8117, 13072

Table B.2: Instances with number of nodes equal to 40

Instance	Graphs
i1.40	8117, 6075, 1540, 4446, 37148, 31114
i2.40	8117, 1540, 165, 1079, 23533, 29886
i3.40	153, 5727, 7910, 31574, 2205, 12741
i4.40	153, 13072, 1540, 7796, 36308, 36531
i5.40	6075, 1540, 973, 607, 6801, 1561
i6.40	8117, 13072, 7910, 1540, 21840, 2803, 3181
i7.40	41, 718, 5461, 14787, 607, 2199, 10567
i8.40	5461, 718, 13072, 7910, 1540, 770, 25925
i9.40	8117, 5727, 153, 1540, 7910, 348, 7411, 5497
i10.40	8117, 153, 22446, 4901, 150, 41, 90, 2528

Table B.3: Instances with number of nodes equal to 60

Instance	Graphs
i1.60	8117, 5727, 4142, 28329, 14689, 21760, 2454
i2.60	5461, 31, 22, 6266, 5048, 9939, 41363
i3.60	22446, 4901, 6073, 646, 486, 1322, 1812, 18026
i4.60	13072, 7910, 11016, 4446, 10567, 5067, 1026, 12427
i5.60	8117, 153, 4901, 22446, 646, 718, 150, 6073, 5461, 7910, 5268, 9901, 21825
i6.60	153, 5727, 718, 1137, 1744, 2632, 231, 3442, 7513
i7.60	8117, 41, 5461, 150, 1540, 426, 107, 4257, 2291, 11563
i8.60	153, 4901, 22446, 718, 19440, 11016, 973, 31, 97, 6497, 2728
i9.60	718, 150, 6073, 4901, 22446, 973, 31, 6497, 11216, 1079, 4464
i10.60	8117, 5727, 153, 6073, 6075, 13072, 1540, 1541, 426, 348, 14912, 5067

Table B.4: Instances with number of nodes equal to 80

Instance	Graphs
i1.80	7910, 13072, 1540, 1089, 28915, 975, 11095, 23414, 16664
i2.80	153, 11216, 56, 7411, 21840, 34256, 986, 38951, 23129, 6191
i3.80	8117, 153, 7910, 1540, 13072, 4250, 1486, 25316, 6081, 32791, 11958
i4.80	8117, 1540, 18105, 7834, 97, 7575, 6262, 16370, 10338, 14562, 38971
i5.80	8117, 13072, 348, 21840, 261, 7899, 3344, 1259, 30821, 38, 2029
i6.80	5727, 153, 5461, 22446, 18105, 19440, 6239, 14286, 10498, 16999, 1200
i7.80	5727, 6073, 22446, 13072, 7910, 11216, 5581, 28437, 8478, 16013, 23125
i8.80	153, 914, 426, 246, 7834, 3488, 31, 20691, 21829, 25951, 373
i9.80	6073, 5461, 718, 150, 6075, 1540, 40760, 1154, 32167, 12203, 17167
i10.80	6073, 22446, 7910, 1540, 14930, 9901, 486, 22331, 5057, 1481, 2648

Table B.5: Instances with number of nodes equal to 100

Instance	Graphs
i1.100	6075, 1540, 11216, 1079, 2291, 36894, 28336, 3058, 1640, 11958, 2075
i2.100	8117, 607, 3344, 31574, 1323, 2694, 35059, 20681, 1861, 5811, 3056, 1422
i3.100	8117, 5727, 13072, 1540, 7910, 7796, 21825, 9834, 1974, 24418, 24420, 133, 22695
i4.100	153, 5727, 646, 718, 1540, 97, 426, 23558, 25929, 16703, 14029, 2817, 1455
i5.100	153, 5727, 22446, 1540, 214, 7899, 266, 18907, 6614, 2818, 27893, 10646, 34221
i6.100	6075, 22446, 1540, 973, 19440, 983, 4257, 2051, 30003, 11064, 3010, 427, 3616
i7.100	150, 7910, 13072, 1540, 214, 803, 7281, 6801, 1079, 1498, 1152, 1917, 22696
i8.100	8117, 5727, 153, 1540, 13072, 11216, 1079, 214, 15698, 2029, 6643, 1422, 17570, 1014
i9.100	8117, 153, 41, 2097, 14930, 7281, 8652, 13480, 1119, 403, 4446, 3182, 26336, 2445
i10.100	5727, 153, 6075, 4901, 6073, 13072, 1540, 1885, 3182, 17256, 10936, 31998, 10858, 18928

Table B.6: Instances with number of nodes equal to 120

Instance	Graphs
i1.120	8117, 261, 12326, 813, 8960, 11425, 2356, 2238, 25358, 37328, 700, 17958, 18682
i2.120	8117, 6073, 646, 718, 11016, 13480, 8652, 2097, 5015, 10320, 21831, 1322, 23600, 10706, 14511
i3.120	8117, 6073, 12431, 214, 90, 41423, 14974, 11041, 4257, 3762, 20702, 7870, 696, 24628, 16694
i4.120	153, 41, 718, 646, 2097, 262, 3759, 21825, 34210, 9009, 28389, 10674, 16402, 11788, 2222
i5.120	22446, 718, 6075, 41, 646, 15698, 22448, 252, 38639, 38974, 30112, 14313, 11369, 15841, 8331
i6.120	8117, 153, 5727, 6073, 34878, 2803, 18931, 31569, 3442, 1089, 1622, 1694, 1259, 1391, 2488, 1085
i7.120	8117, 4901, 6073, 150, 1540, 348, 261, 22448, 1573, 377, 1017, 9335, 1640, 20685, 14920, 1315
i8.120	153, 13072, 7910, 1540, 5330, 1137, 6801, 347, 6757, 35332, 27861, 20691, 1732, 52, 36308, 537
i9.120	22446, 646, 5461, 41, 18105, 2856, 6801, 262, 12116, 3182, 22767, 14899, 1854, 38982, 10982, 1979
i10.120	6073, 150, 5461, 1540, 11216, 13480, 1079, 5268, 7575, 1573, 8652, 1974, 7078, 8165, 25930, 11215

Table B.7: Instances with number of nodes equal to 140

Instance	Graphs
i1.140	153, 5727, 22446, 973, 14787, 7834, 2694, 2688, 1316, 25935, 2263, 10320, 7870, 16381, 29598, 16392, 2099
i2.140	5727, 150, 1540, 13072, 7910, 14787, 914, 90, 102, 34412, 2128, 2728, 1437, 2275, 20708, 13307, 27162, 18330
i3.140	153, 5727, 6073, 646, 1540, 7910, 56, 12431, 347, 3940, 45, 20702, 36307, 10962, 42366, 13468, 15283, 23496, 34258
i4.140	153, 1540, 7910, 973, 3488, 19440, 246, 97, 7834, 983, 214, 2418, 4464, 6036, 32867, 9642, 2967, 337, 18868
i5.140	6073, 150, 22446, 1540, 6497, 56, 1541, 973, 348, 4250, 2291, 28389, 3761, 10151, 20702, 38312, 16370, 23527, 35328
i6.140	153, 5727, 41, 6073, 646, 13072, 6497, 56, 914, 607, 11216, 12326, 377, 22636, 5307, 13195, 10301, 11426, 14337, 1822
i7.140	8117, 153, 5727, 6075, 646, 13072, 7910, 1540, 6153, 1573, 7411, 90, 635, 770, 1026, 22331, 6104, 243, 23612, 10979, 11215
i8.140	8117, 5727, 6073, 718, 6075, 11216, 607, 56, 97, 246, 12116, 948, 16598, 486, 27584, 12326, 6104, 39570, 14899, 118, 20682
i9.140	8117, 5727, 6073, 13072, 7910, 14787, 97, 56, 1541, 19440, 6497, 11016, 3488, 90, 12116, 2028, 3940, 3578, 28597, 2742, 258
i10.140	8117, 41, 6073, 6075, 4901, 150, 5461, 718, 7910, 13072, 1540, 7834, 10567, 39570, 6787, 27870, 25530, 3578, 27735, 22214, 10245

Table B.8: Instances with number of nodes equal to 160

Instance	Graphs
i1.160	1540, 246, 5330, 813, 31574, 3759, 2025, 45, 6266, 34256, 2408, 1738, 2709, 16013, 11819, 1570, 26420
i2.160	8117, 153, 5727, 9600, 4493, 2144, 1769, 28915, 27870, 6036, 5811, 34945, 2151, 18018, 13418, 865, 12459, 16999
i3.160	41, 150, 11216, 97, 426, 14787, 3344, 14931, 8130, 4479, 10247, 2078, 23035, 11606, 1455, 10674, 16260, 23984
i4.160	8117, 646, 5461, 13072, 7796, 335, 27584, 8960, 27321, 1437, 6002, 2013, 2728, 1981, 11036, 1722, 1561, 1315, 25360
i5.160	8117, 6075, 7910, 6497, 914, 7411, 7575, 5330, 36116, 4257, 9834, 10094, 21875, 5934, 10498, 8104, 15286, 19619, 14338
i6.160	8117, 7910, 1540, 13072, 107, 973, 11216, 261, 1970, 20478, 3940, 2630, 2244, 16616, 19619, 18037, 18040, 18637, 12761
i7.160	5727, 153, 2856, 9901, 6104, 27584, 5067, 22636, 1744, 20680, 2487, 8394, 1585, 3762, 28597, 14227, 1178, 12340, 11819
i8.160	22446, 6073, 6075, 13072, 1540, 2856, 1431, 1137, 3759, 4493, 35405, 7767, 34224, 16872, 12494, 12360, 26420, 25929, 14920
i9.160	718, 5461, 22448, 2739, 5330, 7899, 4250, 486, 1704, 1440, 52, 20478, 5595, 3363, 22, 7711, 15286, 12283, 16758
i10.160	646, 7834, 11016, 348, 426, 14787, 1541, 18105, 19265, 811, 6266, 6261, 463, 2968, 35329, 1246, 1776, 27970, 12360

Table B.9: Instances with number of nodes equal to 180

Instance	Graphs
i1.180	8117, 7910, 1540, 14787, 3488, 90, 6801, 214, 15698, 30, 23612, 21734, 25256, 18018, 8582, 23614, 37, 15590, 39692, 14944, 1765
i2.180	8117, 7910, 1540, 2856, 7125, 14974, 10301, 3363, 30, 6787, 3064, 18878, 1641, 5581, 33085, 38312, 42366, 34218, 18364, 1210, 12313
i3.180	5727, 13072, 7411, 1079, 2739, 1885, 347, 7796, 15698, 6104, 2694, 62, 2632, 991, 857, 6204, 22695, 27162, 230, 21866, 10180
i4.180	6073, 6075, 41, 4901, 7910, 973, 56, 33085, 4151, 6614, 857, 34218, 2818, 1861, 1889, 34416, 35264, 11390, 13888, 1994, 12744
i5.180	8117, 5727, 5461, 13072, 1540, 7834, 1454, 34878, 23493, 30809, 69, 13239, 12262, 3010, 1481, 2263, 8894, 21332, 34846, 1051, 38, 7444
i6.180	8117, 153, 7910, 13072, 246, 348, 18105, 11016, 11216, 56, 2630, 9600, 30, 6964, 3895, 32873, 19353, 133, 10670, 26693, 11841, 116
i7.180	5727, 153, 6075, 4901, 5461, 41, 6073, 1885, 2487, 20680, 6396, 28389, 2052, 24934, 9915, 6119, 1496, 19622, 19595, 16716, 23802, 18954
i8.180	5727, 153, 4901, 41, 1540, 7910, 13072, 973, 3759, 37660, 21825, 4962, 4550, 21044, 3892, 4897, 36531, 1854, 10963, 19353, 2175, 14338
i9.180	22446, 6073, 5268, 2856, 137, 90, 7575, 15698, 7411, 3759, 14973, 486, 2418, 1089, 28915, 36307, 475, 8394, 2275, 10979, 10670, 10256
i10.180	1540, 7910, 246, 56, 973, 3488, 14930, 7899, 3344, 7796, 2097, 261, 1431, 7125, 335, 34217, 10858, 19085, 11777, 14406, 21594, 28074

Table B.10: Instances with number of nodes equal to 200

Instance	Graphs
i1.200	5727, 7575, 403, 770, 12326, 859, 813, 24830, 2803, 12116, 32363, 8914, 36894, 15841, 17532, 15429, 25958, 11845, 13816, 1994, 21873
i2.200	153, 646, 718, 6073, 11216, 2688, 16598, 12116, 20693, 3578, 25935, 36652, 23524, 29598, 2010, 16381, 13314, 19619, 16713, 10972, 18868, 28164
i3.200	13072, 19440, 11216, 983, 1079, 5268, 137, 3344, 90, 1454, 1320, 36534, 5934, 19098, 749, 13418, 1404, 8582, 27252, 18868, 21760, 11788
i4.200	8117, 153, 5461, 6075, 7796, 4250, 2097, 7281, 9367, 5321, 79, 8572, 35271, 4151, 16871, 37077, 19098, 342, 32167, 16610, 35192, 10982, 1205
i5.200	8117, 4901, 41, 646, 6497, 97, 348, 107, 32363, 5651, 11423, 20681, 1889, 9103, 2715, 3056, 8201, 35328, 29783, 15429, 12360, 11563, 11575
i6.200	8117, 5461, 13072, 7910, 14930, 10099, 771, 488, 5497, 27893, 10109, 39964, 11423, 42366, 2205, 2275, 20566, 10320, 18907, 29785, 5580, 13314, 15592
i7.200	8117, 1540, 7910, 12431, 214, 10122, 62, 12116, 172, 2414, 2144, 13917, 14312, 907, 3719, 118, 4344, 10534, 2130, 12688, 21831, 26358, 215
i8.200	153, 5727, 4901, 41, 214, 1431, 1573, 10946, 20706, 986, 9103, 30014, 643, 451, 36660, 33714, 2185, 41681, 13418, 7319, 11791, 35329, 10936
i9.200	5727, 153, 13072, 7910, 19440, 165, 348, 8806, 2418, 30, 28389, 29586, 9915, 9346, 32791, 30113, 3616, 1918, 537, 8478, 35507, 6191, 13314
i10.200	5727, 646, 7910, 2688, 6104, 15415, 6266, 34217, 35059, 30, 359, 10102, 4143, 11617, 7280, 1048, 941, 46, 4430, 12535, 20140, 28344, 1165