UNIVERSIDADE FEDERAL FLUMINENSE

JORGE REYNALDO MORENO RAMÍREZ

HEURISTIC AND EXACT APPROACHES FOR SOME COMBINATORIAL OPTIMIZATION PROBLEMS ON GRAPHS

NITERÓI

2019

UNIVERSIDADE FEDERAL FLUMINENSE

JORGE REYNALDO MORENO RAMÍREZ

HEURISTIC AND EXACT APPROACHES FOR SOME COMBINATORIAL OPTIMIZATION PROBLEMS ON GRAPHS

Proposal of a Ph.D. thesis submitted to the Post-Graduation Program in Computing of the Universidade Federal Fluminense, as part of the requirements required to obtain a Ph.D. in Computer Science. Concentration area: ALGORITHMS AND OPTIMIZATION

Tutor:
SIMONE DE LIMA MARTINS

Co-tutor:
YURI ABITBOL DE MENEZES FROTA

NITERÓI

2019

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

R173h Ramírez, Jorge Reynaldo Moreno
Heuristic and exact approaches for some combinatorial optimization problems on graphs. / Jorge Reynaldo Moreno Ramírez ; Simone De Lima Martins, orientadora ; Yuri Abitbol de Menezes Frota, coorientador. Niterói, 2019.
95 f. : il.

Tese (doutorado)-Universidade Federal Fluminense, Niterói, 2019.

DOI: http://dx.doi.org/10.22409/PGC.2019.d.06335968762

1. Heurística. 2. Otimização combinatória (Computação). 3. Teoria dos grafos. 4. Programação linear. 5. Produção intelectual. I. De Lima Martins, Simone, orientadora. II. Abitbol de Menezes Frota, Yuri, coorientador. III. Universidade Federal Fluminense. Instituto de Computação. IV. Título.

CDD -

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

JORGE REYNALDO MORENO RAMÍREZ

HEURISTIC AND EXACT APPROACHES FOR SOME COMBINATORIAL OPTIMIZATION PROBLEMS ON GRAPHS

Thesis presented to the Computing Graduate program of the Universidade Federal Fluminense in fulfillment of the requirements for the Ph.D. degree. Topic area: Algorithms and Optimization

Approved in July of 2019 by:

Prof. D.Sc. Simone de Lima Martins - Advisor
IC/UFF

Prof. D.Sc.Yuri Abitbol de Menezes Frota - Co-Advisor
IC/UFF

Prof. D.Sc. Fábio Protti
IC/UFF

Prof. Uéverton dos Santos Souza
IC/UFF

Prof. PhD. Cid Carvalho de Souza
UNICAMP

Prof. D.Sc. Luidi Gelabert Simonetti
UFRJ

Niterói
2019

Dedicatoria: A minha familia.

Acknowledgment

Agradeço a minha família, por seu apoio incondicional.

Agradeço aos meus orientadores Simone Martins e Yuri Frota pela sua compreensão e confiança.

Agradeço aos meus amigos por sempre me encorajarem.

Resumo

Nesta tese foram estudados os problemas *The Minimum d-Branch Vertices*, *The Rainbow Cycle Cover* e *The Rainbow Spanning Forest*. O problema The Minimum d-Branch Vertices possui aplicações no desenho de redes ópticas, visando a construção de uma rede com o menor número de replicadores de sinal. Por sua vez, The Rainbow Cycle Cover e The Rainbow Spanning Forest são usados no estudo de redes sociais e de transporte. Esses dois últimos problemas pertencem à classe de problemas sobre grafos com cores nas arestas e visam a obtenção de componentes em que cada cor aparece no máximo uma vez. Diferentes estratégias foram desenvolvidas para resolver esses problemas, como a redução do tamanho das instâncias baseada em propriedades dos grafos, o uso de estruturas de dados eficientes e a implementação de heurísticas. Dentro das contribuições deste trabalho está a definição do conceito de co-classe, que foi usado para o pré-processamento das instâncias, assim como para a definição de inequações válidas. Além disso, foram implementados diferentes níveis de colaboração entre métodos heurísticos e exatos. Como resultado, foram desenvolvidas novas formulações matemáticas e heurísticas que resultaram ser mais efetivas do que as encontradas na literatura consultada.

Palavras-chave: Metaheurística, ramificação e poda, matheurística, cortes, otimização combinatória.

Abstract

In this thesis The Minimum d-Branch Vertices, The Rainbow Cycle Cover and The Rainbow Spanning Forest problems were studied. The Minimum d-Branch Vertices problem has applications in the design of optical networks, aiming at the design of a network with the least number of signal replicators. Meanwhile, The Rainbow Cycle Cover and The Rainbow Spanning Forest problems are used to study social and transportation networks. These last two problems belong to the class of problems on edge-colored graphs whose objective is obtaining components in which each color appears at maximum once. Different strategies were developed to solve these problems, such as the reduction of the size of the instances based on properties of the graphs, the use of efficient data structures and the implementation of heuristic methods. Among the contributions of this work is the definition of the concept of co-class, which was used for preprocessing the instances as well as for the definition of valid inequalities. Further, different levels of collaboration between heuristic and exact methods were implemented. As a result, new mathematical formulations and heuristics were developed which are more effective than those found in the reviewed literature.

Key-words: Metaheuristic, branch and cut, matheuristic, cuts, combinatorial optimization.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Several computational problems and everyday phenomena are successfully modeled by graphs. These problems appear in many different scenarios, such as computer networks, vehicle routing, social networks, parallel computing, among others. Some of these problems are solvable in polynomial time, such as to determine the minimum number of nodes that should fail in a computer network to disconnect it (*Vertex Connectivity Problem* [49]) and to find the shortest distance between two points in a city (*Shortest Path Problem* [14]).

Although many problems on graphs are solvable in polynomial time and are considered easy problems, there are many problems that belong to the class of computationally hard problems. It is unknown if these problems may be solvable in polynomial time and the complete exploration of the solution space depends exponentially on the size of the analyzed instance. This issue has aroused great interest in the scientific community due to the challenge of developing effective strategies for finding good quality solutions in reasonable time.

An example of a computationally hard problem is to find the optimum set of routes to be performed by a fleet of vehicles to satisfy the demand of a given set of customers (*Vehicle Routing Problem* [13]). Another example is to determine which is the smallest number of colors needed to color the cities on a map in such a way that two adjacent cities do not have the same color (*Graph Coloring Problem* [16]).

Algorithms that perform an intelligent search in the solution space are used to find exact solutions for hard problems. The Branch-and-Bound (B&B) algorithm [23] is one of the most used algorithms with this feature. It performs a systematic enumeration of candidate solutions, discarding in the process subsets of candidates by using upper and lower bounds estimated for the value of the objective function. Using the same idea and

incorporating cutting plans, the Branch-and-Cut algorithm(B&C) [37] is also one of the algorithms most commonly used to solve combinatorial optimization problems.

Depending on the size of the instance under analysis, the above algorithms may require unreasonable computational time. To solve this issue, different heuristics and metaheuristics were developed. Metaheuristics are algorithms (usually non-deterministic) that produce good quality solutions in reasonable times. Unlike the exact methods, metaheuristics do not guarantee to find the global optimum of a given problem. Among the most used metaheuristics are *Greedy Randomized Adaptive Search Procedures* (GRASP) [41], *Genetic Algorithms* (GA)[42], *Iterated Local Search* (ILS) [27] and *Tabu Search* (TS) [52].

In the last years, different strategies have been developed that combine exact and heuristics algorithms [4, 5]. These strategies may be separated in two groups: *Collaborative Combination* and *Integrated Combination* [40]. In Collaborative Combination, heuristic and exact methods exchange data, but are executed individually. For example, a heuristic can be used to create a set of good solutions and patterns may be extracted from this set. These patterns may be used to fix some variables in the mathematical model of the problem, obtaining a solution that generally improves that obtained by the heuristic [38]. In Integrated Combination, one method can be embedded as a component within the other. For example, a metaheuristic could explore neighboring solutions of a solution, using an exact algorithm to find the best neighbor [53]. Another strategy is to use heuristics to determine cutting planes for algorithms such as Branch-and-Cut.

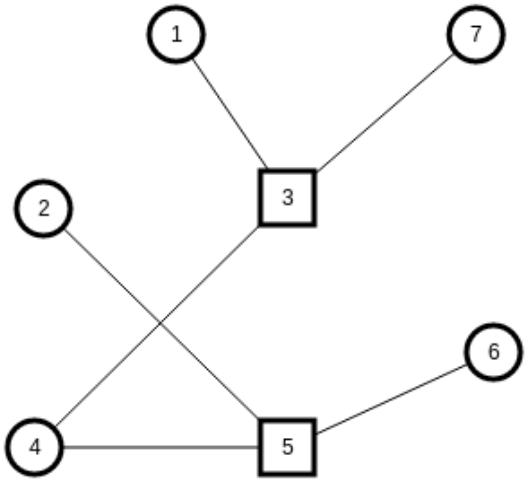
In this work, exact and heuristic approaches are proposed for combinatorial optimization problems on graphs. The selected problems are *The Minimum d-Branch Vertices* [17], *The Rainbow Cycle Cover* [47] and *The Rainbow Spanning Forest* [7].

1.1 Motivation

The Minimum Branch Vertices problem was introduced by Gargano et al. [17] to help in the design of optical networks. Recently, this problem was generalized by Merabet et al. [33]. Basically, if a network node connects with d or more nodes ($d \geq 2$), then it is necessary to introduce a switch to replicate the signal. These switches represent an additional cost in the design of the network, and each switch has a capacity of replicating several times the original signal. For that reason, designing a network with the least number of these replicators becomes a task of great importance. Figure 1.1 shows the

design of an optical network with two signal replicator switches (vertices 3 and 5). The results obtained in Merabet et al. [33] show that as the parameter d increases, the instances are more difficult to solve. However, the model developed by them does not explore the advantages of the preprocessing used in Melo et al. [32]. In addition, there is no heuristic procedure for this problem when $d > 2$. All these situations make this problem very attractive to be studied.

Figure 1.1: Optical network with two signal replicators switches.



Meanwhile, the *Rainbow Cycle Cover* and the *Rainbow Spanning Forest* problems belong to the set of problems on edge-colored graphs. Practical applications in social networks, transport systems, route designs and computer networks may be modeled as edge colored graphs problems. In the context of computer networks, the colors of the edges can specify a certain security protocol. Finding multicolored routes would bring as a benefit a strengthening in the level of security of the network by applying different security protocols on transmitting data.

The Minimum 2-Branch Vertices and the Rainbow Spanning Forest problems were recently studied by Silvestri et al. in [48] and Carrabs et al. [7] respectively. In these works, exact and heuristics algorithms were proposed for those problems, but no methods were investigated to decompose and reduce the instances. Also, no procedures integrating exact and heuristic methods were found to solve these problems. In addition, the Rainbow Cycle Cover problem is very difficult to solve, but no heuristic method was found to solve it in the literature.

Therefore, we decided to propose new mathematical models for each of these problems, as well as heuristic and hybrid methods in order to obtain better quality solution using

less computational time.

1.2 Objectives

The general objective of this thesis is to develop strategies based on graph properties and the use of different levels of collaboration between exact and heuristic methods to obtain good quality solutions on the selected problems. To achieve this objective the following activities were developed:

- Analysis of the selected problems and the most effective methods found in the literature to solve them.
- Development of new mathematical formulations for these problems based on integer linear programming, as well as valid inequalities to improve the performance of the developed exact methods.
- Development of effective heuristics for the selected problems.
- Use of different levels of collaboration between heuristics and the exact algorithms.
- Validation of the performance of the proposed methods, making comparisons with the main works already developed to solve these problems.
- Dissemination of the results in scientific events and journals related to the subject of the thesis.

1.3 Thesis outline

The remainder of this thesis is structured as follows:

- Chapters 2 presents a new model for *The Minimum d-Branch Vertices* problem. Cutting planes based on the graph structure are determined, as well as a scheme to decompose the problem into less complex subproblems. In addition, an ILS heuristic for the RCC problem is presented.
- Chapters 3 presents algorithms to solve exactly and approximately the *Rainbow Cycle Cover* problem. A branch and cut algorithm is developed for this problem and new cutting planes based on the graph structure are determined, as well as

a scheme to decompose the problem into less complex subproblems. Also, an ILS matheuristic is presented in this chapter.

- Chapter 4 presents a new model for the *Rainbow Spanning Forest* problem based on the non-trivial connected components. Moreover, a GRASP heuristic is presented in this chapter.

- Chapter 5 presents the conclusions of this work, as well as some directions for future research.

Chapter 2

The Minimum d-branch Vertices Problem

Optimization problems related to finding a spanning tree of an undirected graph have been extensively studied in the literature [1], [9], [11], [29], [32]. The criterion for choosing this tree depends on the particular problem and may be associated with properties of the vertices, edges, or both. The *Minimum Branch Vertices* (MBV) problem is associated to the degree of vertices. The goal of this problem is to find a spanning tree with the lowest number of vertices of degree greater than 2. This problem was introduced by Gargano et al. [17] to help the design of optical networks. In Cerrulli et al. [11], a mixed integer modeling and three heuristics for this problem were developed. Other approaches in the literature, besides proposing mathematical models, presented heuristics or metaheuristics [50] to solve this problem.

In Silva et al. [45] a heuristic was developed based on an exchange of edges, in which edges of higher weight are replaced by edges of lower weight. The weights of the edges were determined by the degrees of the end vertices of the edge. A refinement of this heuristic was proposed by Silva et al. [46]. Other two heuristics were proposed by Carrabas et al. [9], and tested on a wide set of instances. In Marín [29], a new model was presented, as well as a heuristic with the best average results for the instances used in Carrabas et al. [9].

Silvestri et al. [48] developed a hybrid formulation containing undirected and directed variables. This formulation was solved by a branch-and-cut algorithm, improving the results obtained by Marín [29]. Finally, Melo et al. [32] proposed an effective constructive heuristic, which takes into consideration the problem structure in order to obtain good feasible solutions. Also, a decomposition approach based on bridges and cut vertices of the graph was developed, reducing the size of the subproblems to solve.

Recently, a generalization of the MBV problem was proposed by Merabet et al. [33]. This problem uses the concept of k-branch, which is a vertex with degree strictly greater than $k+2$. The value of k is considered as a tolerance parameter for the design of optical networks, since if a light signal is splitted into k copies, the signal power of one copy will be reduced with, at least, a factor of $1/k$ of the original signal power. The k-Minimum Branch Vertices problem (k-MBV) consists in searching for a spanning tree with the minimum number of k-branch vertices. Merabet et al. [33] proved that this problem is NP-hard whatever the value of k. Also, an ILP based on a single flow formulation was developed and applied on sparse graphs for different values of the parameter k.

To simplify the notation, we introduce a parameter $d = k + 2$ and call a node in the graph with degree strictly greater than d, ($d \geq 2$) of a d-branch vertex. According to this definition the d-MBV problem is defined as:

Problem 1 (d-MBV problem) *Given an undirected graph $G = (V, E)$ with $n = |V|$ vertices, the d-Minimum Branch Vertices (d-MBV) problem consists in finding a spanning tree of G with the minimum number of vertices with degree greater than a fixed integer value d, ($d \geq 2$).*

The remainder of this chapter is organized as follows: Section 2.1 presents a mathematical formulation for this problem and the strategies developed to solve it using exact methods. Section 2.2 describes the proposed heuristic algorithm. Section 2.3 presents the results obtained for the heuristic and exact method. Finally, the conclusions are discussed in Section 2.4.

2.1 Mathematical Formulation

2.1.1 Miller-Tucker-Zemlin based formulation

Given an undirected graph $G = (V, E)$, where V denotes the set of vertices ($|V| = n$) and E the set of edges, a neighborhood in G of vertex $v \in V$ is defined as $N_G(v) = \{u \in V \mid \{u, v\} \in E\}$ where $d_G(v) = |N_G(v)|$ is denoted degree of vertex v. Similarly, the neighborhood of a subset $S \subseteq V$ is defined as $N_G(S) = \bigcup_{u \in S} N_G(u)$.

A usual way to model the MBV problem is through formulations based on creating an arborescence with root $r \in V$ and the use of a set of arcs A, such that for all $\{i, j\} \in E$, then $(i, j), (j, i) \in A$. The main difference between these formulations is how to avoid

cycles.

The most interesting instances for the d-MBV problem are those on sparse graphs, which have a greater chance of having d-branch vertices [9, 33]. In Leggieri et al. [25] the **Miller-Tucker-Zemlin** (MTZ) constraints were applied with success over sparse graphs with up to 1000 vertices. So, in this work, we use MTZ constraints to avoid cycles. The idea to solve the problem is finding an arborescence with vertex source r.

We define variable y_i equal to 1 if the node $i \in V$ is a d-branch vertex, otherwise it is equal to 0. We also define variable x_{ij}, which will take the value 1 if and only if arc $(i, j) \in A$ belongs to the solution. Variable z_i represents the level of vertex $i \in V$ on the arborescence (the root is at level 0). These variables are used to prevent cycles (e.g. if arc x_{ij} belongs to the solution then $z_j > z_i$). We propose the following formulation for the d-MBV problem.

$$\min \sum_{i=1}^{n} y_i \tag{2.1}$$

Subject to:

$$\sum_{j \in V:(j,i) \in A} x_{ji} = 1, \qquad \forall i \in V, i \neq r \tag{2.2}$$

$$\sum_{(i,j) \in A} x_{ij} = n - 1, \tag{2.3}$$

$$x_{ij} + x_{ji} \leq 1, \qquad \forall \{i, j\} \in E \tag{2.4}$$

$$\sum_{j \in V, j \neq r:(i,j) \in A} x_{ij} \leq (d_G(i) - d)y_i + d - 1, \qquad \forall i \in V, i \neq r \tag{2.5}$$

$$\sum_{j \in V:(r,j) \in A} x_{rj} \leq (d_G(r) - d)y_r + d \tag{2.6}$$

$$\sum_{(j,r) \in A} x_{jr} = 0 \tag{2.7}$$

$$z_r = 0 \tag{2.8}$$

$$z_j \geq z_i + x_{ij}n + x_{ji}(n - 2) - (n - 1), \qquad \forall (i, j) \in A, j \neq r \tag{2.9}$$

$$y_i \in \{0, 1\}, \qquad \forall i \in V \tag{2.10}$$

$$x_{ij} \in \{0, 1\}, \qquad \forall (i, j) \in A \tag{2.11}$$

$$z_i \in \mathbb{Z}, z_i \in [0, n - 1], \qquad \forall i \in V \tag{2.12}$$

The objective function (2.1) requires minimizing the number of d-branch vertices in the

tree. Constraints (2.2) indicate that there must be exactly one edge entering each vertex, with the exception of the root vertex. Constraints (2.3) forces that the arborescence contains exactly $n-1$ edges. Constraints (2.4) require that there is at most one arc between any pair of vertices. Moreover, constraints (2.5) ensure that a vertex (different from the root) is d-branch if more than $d-1$ arcs leave it, while constraints (2.6) assure that the root vertex is d-branch if at least $d+1$ arcs leave it. Constraints (2.7) impose that no arcs enter the root vertex. Constraints (2.8) define that the level of the root vertex is 0, and Constraints (2.9) determine that the level of each vertex j is greater than the level of the vertex i, if the arc (i,j) belongs to the arborescence. Finally, constraints (2.10) to constraints (2.12) ensure the integrality requirements on the variables.

2.1.2 Graph decomposition

A graph decomposition approach was developed for the MBV problem by Melo et al. [32] and also recently by Landete et al. [24]. The basic idea is to analyze the problem by decomposing the graph into subgraphs that are easier to solve, and then recombine the solutions of these subgraphs to generate a solution to the problem. To implement this decomposition, they detect and analyze bridges and articulation points (or cut vertices) of the graph.

2.1.2.1 Decomposition based on bridges

A bridge in a graph G is an edge that when removed from the graph increases the number of connected components of G. Finding the set B_G of bridges in the original graph G allows to characterize some of its vertices as obligatorily d-branch vertices. Let $\beta(v)$ be the number of bridges incident on the vertex v, then we define the following sets:

- $O_D = \{u \in V | \, d_G(u) > d \wedge \beta(u) \geq d\}$
- $N_D = \{u \in V | \, d_G(u) \leq d\}$

Note that set O_D only contains vertices that will be d-branch in the optimal solution. If the number of adjacent bridges to vertex v is greater than d, then the vertex v has to be d-branch in the optimal solution. Moreover, suppose that vertex v has exactly d adjacent bridges. In this case, if these bridges were removed, the vertex v would belong to a component consisting of more than one vertex ($d_G(v) > d$) and therefore, for any

spanning tree on this component, it would be necessary at least one edge incident to vertex v. As a consequence v have at least $d+1$ incident edges and therefore has to be a d-branch. On the other hand, the set N_D contains the vertices that cannot be d-branch in any solution, since none of these vertices have enough incident edges to become a d-branch.

As in Melo et al. [32], a decomposition approach can be applied to solve smaller subgraphs using the previous formulation and combining the solutions. The idea is to eliminate each bridge and in its place incorporate the parameter $l(v)$ associated with each vertex that indicates the number of bridges incident to vertex v. If k is the number of resultant connected components, then the optimal solution of the problem is $f = \sum_{i=1}^{k} f_k$, where f_k is the optimal value obtained for the k-th connected component $G_k = (V_k, E_k)$, with $|V_k| = n_k$. The formulation for the d-MBV problem in G_k is presented below.

$$\min \sum_{i=1}^{n_k} y_i \tag{2.13}$$

Subject to:
(2.2)-(2.4), (2.9)-(2.12); (replacing n by n_k)

$$\sum_{j \in V, j \neq r:(i,j) \in A} x_{ij} + l(i) \leq (d_{G_k}(i) + l(i) - d)y_i + (d-1), \qquad \forall i \in V, i \neq r \tag{2.14}$$

$$\sum_{j \in V:(r,j) \in A} x_{rj} + l(r) \leq (d_{G_k}(r) + l(r) - d)y_r + d \tag{2.15}$$

$$y_i = 1, \qquad \forall i \in O_D \tag{2.16}$$

$$y_i = 0, \qquad \forall i \in N_D \tag{2.17}$$

All bridges must be in any spanning tree of the graph. Since every vertex v has associated the value $l(v)$ (i.e. the number of incident bridges), constraints (2.14) -(2.15) are sufficient to determine whether or not a vertex will be a d-branch vertex in the solution. Note that, except for the root, all the vertices have an incident arc. Therefore, if the number of arcs coming out of the vertex plus the number of adjacent bridges (obligatory edges) is greater than or equal to d, the vertex will be d-branch. For the root the same principle applies: the root vertex will be d-branch if the number of arcs leaving it plus the number of bridges adjacent to it is greater than or equal to $d+1$. Finally, constraints

(2.16) and (2.17) are related to the preprocessing phase and are used to help speed up the problem solving process.

2.1.2.2 Decomposition based on articulation point

Melo et al. [32] showed that it is possible to further decompose the graph considering the articulation points (cutting vertices) which increase the number of connected components of the graph when removed.

Let $G_k = (V_k, E_k)$ be one of the connected graphs obtained after removing all bridges as explained before such that each vertex $v \in V_k$ has associated the number $l(v)$ of adjacent bridges. Let $c_{G_k}(v)$ be the number of connected components by eliminating v of G_k and suppose $c_{G_k}(v) > 1$. Note that the only way to connect these $c_{G_k}(v)$ components is using an edge incident to v. Therefore, $c_{G_k}(v)$ represents a number of obligatory incident edges to v. Since $l(v)$ also represents a number of obligatory incident edges to v, then the vertex v will be a d-branch if $c_{G_k}(v) + l(v) > d$.

The Algorithm 2.1 named SolveGraph was developed to solve the d-MBV problem for the subgraphs G' obtained by eliminating the bridges from G. It receives as input the subgraph $G' = (V', E')$, the values $l(v')$ associated to each vertex $v' \in V'$ and the set $Cut_D = \{v' \in V' \,|\, c_{G'}(v') > 1 \wedge c_{G'}(v') + l(v') > d\}$ which contains all cut vertices that will necessarily be d-branch vertices in the optimal solution. The procedure SolveModel in line 3 is used to solve the proposed model presented in the Section 2.1.2.1 for the subcomponents G_k that are subgraphs of G' after removing all cut vertices $v \in Cut_D$. In each step, the method delete a vertex $v' \in Cut_D$ (line 5) and create $c_{G'}$ connected components by duplicating vertex v' in each component $k = 1, \ldots, c_{G'}(v')$ (lines 10 and 11). The neighborhood of the new vertex (denoted as v'_k) is defined as the set of vertices to which v' is adjacent in the component k (line 12). Furthermore, the value $l(v'_k)$ will not necessarily hold the same value after this process, since $l(v'_k)$ is incremented by the number of edges incident to v' that do not belong to the component k. Note that vertex v'_k will be classified as a d-branch vertex in each of the connected components, so the value $c_{G'}(v') - 1$ must be subtracted from the number of vertices of the solution in G' (line 13).

Algorithm 2.1: SolveGraph

Input: A graph without bridges $G' = (V', E')$ and the value $l(v')$ associated to each vertex $v' \in V'$ and the set Cut_D.
Output: The optimal solution for the d-MBV problem

```
s ← 0
if Cut_D = ∅ then
    s ← SolveModel(G')
else
    Choose any v' ∈ Cut_D and delete it from G'
    Obtain the connected components G_k = (V_k, E_k) k = 1, ..., c_G'(v')
    Let Cut_D^(k) = Cut_D ∩ V_k,  k = 1, ..., c_G'(v')
    for k ← 1 to c_G'(v') do
        /* create a copy of vertex v' in each component k */
        V_k ← V_k ∪ {v'_k}
        E_k := E_k ∪ {{u', v'_k} | u' ∈ V_k ∧ {u', v'_k} ∈ E'}
        l(v'_k) ← l(v'_k) + d_G'(v') − d_G_k(v'_k)
        s ← s + SolveGraph(G_k, Cut_D^(k))
    s ← s − c_G'(v') + 1
return s
```

2.1.3 Analyzing the 2-cocycles

Let $G = (V, E)$ be a connected graph. A 2-cocycle (also know as 2-edge-cut) is defined as a set of two edges $\{e, f\} \subseteq E$ such that $(V, E \setminus \{e, f\})$ is not connected, while $(V, E \setminus \{e\})$ and $(V, E \setminus \{f\})$ are connected. So, at least one of the edges of a 2-cocyle has to belong to any feasible solution. Considering $e = \{u, v\}$ and $f = \{w, t\}$ the following constraints may be added [29, 48]:

$$x_{uv} + x_{vu} + x_{wt} + x_{tw} \geq 1, \qquad \forall \{e, f\} \in C \tag{2.18}$$

where C denotes the set of 2-cocycles of G.

In this context, we define a *co-class* $H \subseteq E$ as a set of edges, with $\mid H \mid \geq 2$, such that any pair of edges in H is a 2-edge-cut for G. Notice that by this definition, if an edge of a co-class is eliminated from the graph, the other edges of the co-class in the resulting graph would become bridges. Furthermore, we denote C^C as the family of all maximal co-classes in G, where a co-class is maximal if it is not included in any other co-class.

Note that each edge that belongs to a co-class C_e also belongs to a 2-cocycle structure with each other edge from C_e. Therefore, any two edges from C_e cannot be outside the d-MBV solution at the same time (i.e. the solution would be a disconnected graph). For

this reason, the following stronger constraints are used in the model instead those used by Marín [29] and Silvestri et al. [48]:

$$\sum_{\{u,v\}\in H} (x_{uv} + x_{vu}) \geq |H| - 1, \qquad \forall H \in C^C \tag{2.19}$$

As an example, consider the graph of Figure 2.1. As shown in Figure 2.2, the set of maximal co-classes is $C^C = \{C_1, C_2, C_3, C_4\}$, where $C_1 = \{\{1,2\},\{1,12\},\{11,12\}\}$, $C_2 = \{\{3,4\},\{4,5\}\}$, $C_3 = \{\{7,8\},\{8,9\}\}$ and $C_4 = \{\{2,3\},\{6,7\},\{9,10\},\{10,11\}\}$.

Figure 2.1: A connected graph.

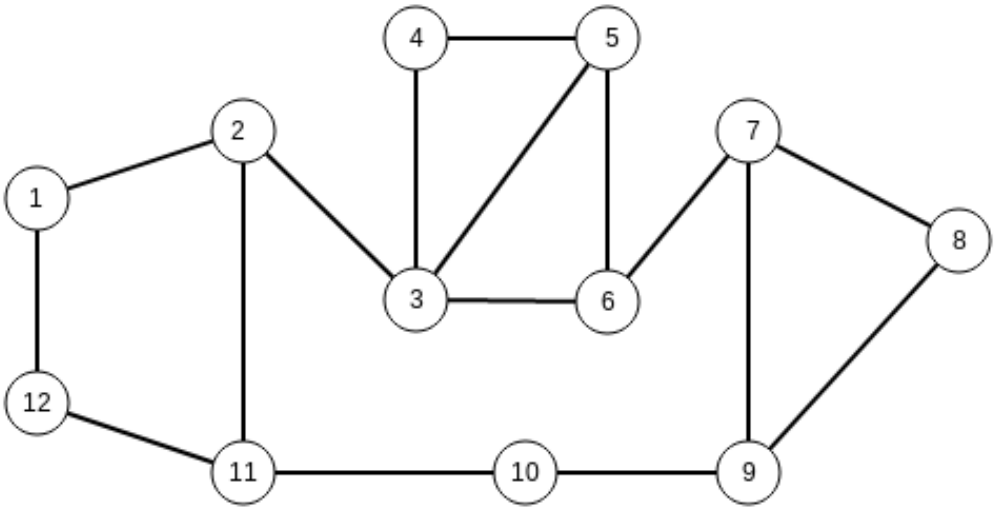
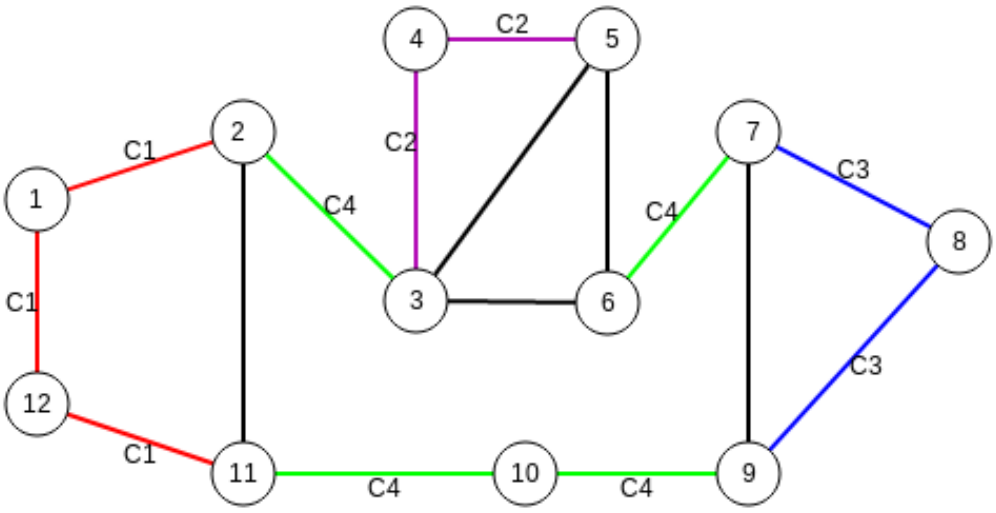


Figure 2.2: A connected graph with 4 maximal co-classes.



We use the same method used in Marín [29] for the detection of 2-cocycles. The method consists of removing a non-bridge edge and applying the bridge detection algorithm proposed by Schmidt [43]. Note that if an edge belongs to a calculated co-class, it is not necessary to perform the procedure to find the associated 2-cocycles, decreasing the required computational time needed.

2.1.4 Other Valid Inequalities

In addition to the constraints presented for the $d-$MBV model, the following inequalities are useful to strengthen the formulation.

$$\sum_{j \in V, j \neq r:(i,j) \in A} x_{ij} + l(i) \geq d\, y_i, \quad \forall i \in V, i \neq r \tag{2.20}$$

$$\sum_{j \in V:(r,j) \in A} x_{rj} + l(r) \geq d\, y_r + 1 \tag{2.21}$$

Basically, those constraints indicate that if a vertex v is a d-branch vertex, then the number of arcs leaving v must be equal or greater than d ($d+1$ in case v is the root). Furthermore, a new family of valid inequalities were developed to the d-MBV problem by extending the valid inequalities proposed in Silvestri et al. [48] for the 2-MBV problem.

Proposition 1 *For all $v \in V \setminus \{r\}$, $S \subset N_G^+(v)$ with $|S| \geq d - l(v)$ and $l(v) < d$:*

$$\sum_{(v,u) \in S} x_{vu} + l(v) + 1 \leq (|S| + l(v) + 1 - d) y_v + d \tag{2.22}$$

where $N_G^+(v) = \{u \in V \mid (v,u) \in A\}$.

Proof: Let G be the preprocessed graph. For any subset $S \subset N_G^+(v)$, if the sum of the arcs leaving v (i.e. $\sum_{(v,u) \in S} x_{vu}$) with the number of adjacent leaves (i.e. $l(v)$) and the incident arc on v (i.e. $+1$) exceeds the value of d in the solution, then vertex v has to be a d-branch. □

The separation of inequalities in Proposition 1 is pretty straight forward. Let $(\overline{x},\overline{y})$ be a feasible solution for the linear programming relaxation and $\overline{x}_v \subseteq \overline{x}$ be the set of all values of variables $\overline{x}_{vu}$, $\forall u \in N_G^+(v)$. For each vertice $v \setminus \{r\}$ where $d_G(v) > d > l(v)$, we first sort in descending order the values of $\overline{x}_v$. Then, search for the minimal k for the set $S_v^k = \{u \in N_G^+(v) \mid \overline{x}_{vu}$ is one of the first k elements of $\overline{x}_v\}$ where

$$\sum_{u \in S_v^k} \overline{x}_{vu} + l(v) + 1 > (k + l(v) + 1 - d)\overline{y}_v + d$$

if we find such inequality, then the following cut is added to the formulation

$$\sum_{u \in S_v^k} \overline{x}_{vu} + l(v) + 1 > (k + l(v) + 1 - d)\overline{y}_v + d$$

2.2 ILS Heuristic for the d-MBV problem

The metaheuristic Iterated Local Search (ILS) [27] has been used in several optimization problems and has obtained good quality results [3],[12],[54]. The pseudocode for the metaheuristic ILS is presented in Algorithm 2.2.

Algorithm 2.2: ITERATED LOCAL SEARCH

Input: The graph G.

Output: A valid solution T.

```
1 x_0 ← Initial_Solution(G)
2 x* ← Local_Search(G)
3 repeat
4 |   x' ← Perturbation(x*)
5 |   x*' ← Local_Search(x')
6 |   x* ← AcceptanceCriterion(x*, x*', history)
7 until termination condition met
8 return x*
```

First, an initial solution is generated for the problem (line 1) and a local search is applied in this solution to improve the quality of the constructed solution (line 2). Between lines 3 and 7, iterations are performed until a stopping criterion is reached. In each iteration, a perturbation in the current solution is done trying to escape of local optimum and then a local search is performed. In line 6, a criterion is used to decide if the current solution will be replaced by the new generated solution.

We propose an ILS heuristic aiming to provide a good quality solution to be used as an upper bound for the previous proposed formulation. The metaheuristic will be applied to each of the connected subgraphs $G' = (V', E')$ resulting from the decomposition process (Section 2.1.2), where G' is a graph without bridges. The solution (upper bound) for the original graph G will be achieved by merging the solutions obtained for each subgraph, using Algorithm 2.1, replacing method SolveModel(G') (line 13) with the metaheuristic ILS(G').

Procedures for generating the initial solution, performing local search and perturba-

tion were developed. The acceptance criterion updates the current solution if the solution obtained in the local search has less number of d-branch vertices than the best solution found in previous iterations. The termination condition is met when perturbation can no longer be applied. More details are provided in Section 2.2.3.

2.2.1 Building an initial solution

A heuristic based on the selection of edges of the undirected graph G that are not already in the tree was developed to generate an initial solution.

The following weights w_{sOD} and w_{aOD} of an edge $\{u, v\}$ are defined as:

$$w_{sOD} = d_G(u) + l(u) + d_G(v) + l(v) - f_D(u) \cdot n - f_D(v) \cdot n$$

and

$$w_{aOD} = d_G(u) + l(u) + d_G(v) + l(v) + f_D(u) \cdot n + f_D(v) \cdot n$$

respectively. Moreover, the function $f_D(v)$ defines if a vertex v belongs to the set O_D and is defined as:

$$f_D(v) = \begin{cases} 1 & \text{if } v \in O_D \\ 0 & \text{if } v \notin O_D \end{cases}$$

Algorithm 2.3 contains the pseudocode of the strategy used for the initial construction of the solution. A tree T is initialized with all vertices of G without any edges. In this algorithm, the set of arcs A of the graph (as defined in Subsection 2.1) is explored to select the edges that will be in T. Between lines 3 and 6, the arc (u, v) with associated edge $\{u, v\}$ of minimum w_{sOD} value is selected if **(i)** vertex u has no incident edges in T, **(ii)** vertex v is incident to T and **(iii)** the sum of the degree of v in T with its associated $l(v)$ value is different from d. For this selected arc, the associated edge will be added in T. This added edge will not create neither cycles nor new d-branch vertices in T. Also, selecting arcs with minimum w_{sOD} values for the associated edges prioritizes those arcs whose vertices are d-branch vertices and have a small degree in the graph G. In this way, the obligatory vertices will have more added edges.

Between lines 7 and 9, arcs are selected according to the following ordered criteria. First, in criterion (a) an arc (u, v) is selected if vertex u has a degree less than d and if vertex v is a d-branch vertex, so inserting the associated edge will not generate new d-branch vertices in T. This criterion prioritizes arcs whose vertices have to be d-branch vertices in the final solution and have large degree in G. If no vertices are found in criterion

Algorithm 2.3: INITIAL SOLUTION

Input: A graph $G = (V, E)$ without bridges, the value $l(v)$ associated to each vertex, the set O_D and the set of arcs A such that for all $\{i, j\} \in E$, then $(i, j), (j, i) \in A$.

Output: A spanning tree T of the graph

$T \leftarrow (V, \emptyset)$
$m \leftarrow 0$
repeat
  Find the arc $(u, v) \in A$ such that $d_T(u) = 0$ and $d_T(v) + l(v) \neq d$ and whose associated edge $\{u, v\}$ has minimum w_{sOD} value; then, add the edge $\{u, v\}$ in T.
  $m \leftarrow m + 1$
until *There is no arc that satisfies this condition*
repeat
  Consider criteria (a)-(f), whose priorities are in descending order ((a) has the highest priority). Find an arc (u, v) in A with associated edge $\{u, v\}$, such that $T \cup \{u, v\}$ is a tree and no other arc satisfies a higher priority criterion.
    (a) arc (u, v) has maximum value $d_G(v) + l(v) + n \cdot f_D(v)$ such that $d_T(u) + l(u) < d$, $d_T(v) + l(v) > d$.
    (b) edge $\{u, v\}$ has maximum w_{aOD} in T, and arc (u, v) with $d_T(u) + l(u) > d$, $d_T(v) + l(v) > d$.
    (c) edge $\{u, v\}$ has minimum w_{sOD} in T and arc (u, v) with $d_T(u) + l(u) < d$, $d_T(v) + l(v) < d$.
    (d) edge $\{u, v\}$ has maximum w_{aOD} in T and arc (u, v) with $d_T(u) + l(u) > d$, $d_T(v) + l(v) = d$.
    (e) edge $\{u, v\}$ has maximum w_{aOD} in T and arc (u, v) with $d_T(u) + l(u) < d$, $d_T(v) + l(v) = d$.
    (f) edge $\{u, v\}$ has maximum w_{aOD} in T and arc (u, v) with $d_T(u) + l(u) = d$, $d_T(v) + l(v) = d$.
  Add $\{u, v\}$ in T
  $m \leftarrow m + 1$
until $m < |V| - 1$
return T

a, then an arc with maximum w_{aOD} value in T for the associated edge and whose both vertices are already d-branch vertices is selected (criterion (b)), so that the number of d-branch vertices is not incremented. This criterion also prioritizes arcs whose vertices have to be d-branch vertices and have large degree. Again, if no vertices are found in criterion (b), an arc is selected if both vertices are not already d-branch vertices in T in such a way that when an edge is inserted in T none of them turns to be a d-branch vertex (criterion (c)). This criterion prioritizes arcs whose vertices have small degrees, letting the vertices with larger degrees to be analyzed later. Moreover, criteria (d), (e) and (f) choose arcs whose associated edges will create new d-branch vertices, (one when using

(d) and (e) and two when using (f)). These criteria select arcs whose vertices have large degrees, so when a vertex turns to be a d-branch vertex, there is a chance that new edges will be chosen incident to this vertex in the next selections. The edge associated to the arc selected in line 8 will be inserted in the tree T.

2.2.2 Local Search

There is no guarantee that the initial solution returns a locally optimal solution with respect to some neighborhood. Therefore, the tree obtained by Algorithm 2.3 may be improved by the local search procedure described in Algorithm 2.4. In line 1, the best current solution T_{best} is initialized with the solution T_{curr}, and the procedure FirstBest_Neighbor (line 4), described in Algorithm 2.5, is executed while there is improvement in the current solution.

Algorithm 2.4: LOCAL SEARCH

Input: The current solution T_{curr}.
Output: The best solution T_{best}.

```
1 T_best ← T_curr
2 repeat
3 |   improvement ← 0
4 |   T ←FirstBest_Neighbor(G,T_best, O_D)
5 |   if T ≠ T_best then
6 |   |   T_best ← T
7 |   |   improvement ← 1
8 until improvement = 0
9 return T_best
```

The FirstBest_Neighbor method (Algorithm 2.5) looks for a neighbor solution (i.e. a solution obtained by swapping some edges) with less d-branch vertices than the current solution. The aim of this procedure is to remove edges from a d-branch vertex until its degree is equal or less than d. In line 3 the list of candidate vertices DB is initialized with vertices that are not obligatory and whose degree is greater than d. The d-branch vertices in DB with smaller degree should be easier to process, so DB is sorted in ascending order related to the degree of vertices in T. Then, for each $v \in DB$, each one of its neighbors u is analyzed. In lines 5 to 17, the procedure tries to find another edge (different from the one that connects u and v in T) that does not create new d-branch vertices. If a new tree is obtained with fewer d-branch vertices the procedure returns it in line 16. Otherwise, vertex v is inserted in the list L_D of d-branch vertices.

Algorithm 2.6 shows the procedure that tries to find an edge different from $\{u, v\}$ that

Algorithm 2.5: FIRSTBEST_NEIGHBOR

Input: A graph $G = (V, E)$ without bridges, the current solution T_{curr} and the set O_D of obligatory d-branch vertices.
Output: The solution T.

$T \leftarrow T_{curr}$
$L_D \leftarrow O_D$
$DB \leftarrow \{v \in V \setminus O_D \mid d_T(v) + l(v) > d\}$
Sort DB in ascending order by value $(d_T(v) + l(v))$
foreach $v \in DB$ **do**
  foreach $u \in N_T(v)$ **do**
    Remove the edge $\{u, v\}$ from T
    $e \leftarrow$Find_Edge(u, v, T, L_D)
    if $e \neq \emptyset$ **then**
      Insert the edge e in T
    else
      Insert the edge $\{u, v\}$ in T
    `/* v or u are d−branch vertices no more */`
    if $d_T(v) + l(v) \leq d$ ***or*** $((d_T(u) + l(u) \leq d) \wedge (u \in DB))$ **then**
      return T
  $L_D \leftarrow L_D \cup \{v\}$
return T

Algorithm 2.6: FIND EDGE

Input: Vertices u and v, a forest T, where $T_u = (V_u, E_u)$ and $T_v = (V_v, E_v)$ are the connected subtrees containing u and v respectively, and the set L_D of d-branch vertices.
Output: An edge $e \neq \{u, v\}$ between T_u and T_v or $\emptyset$ if not exists.

$U_1 \leftarrow L_D \cap V_u$
$U_2 \leftarrow \{w \in V_u \mid d_T(w) + l(w) < d\}$
$U_3 \leftarrow \{w \in V_u \setminus L_D \mid d_T(w) + l(w) > d\}$

$V_1 \leftarrow L_D \cap V_v$
$V_2 \leftarrow \{w \in V_v \mid d_T(w) + l(w) < d\}$
$V_3 \leftarrow \{w \in V_v \setminus L_D \mid d_T(w) + l(w) > d\}$

$e \leftarrow \{u', v'\}$ such that $\{u', v'\} \neq \{u, v\}$, $u' \in U_i$, $v' \in V_j$ and (i, j) is lexicographically smaller than any other valid pair $(i, j \in \{1, 2, 3\})$.
return e

connects the subtrees T_u and T_v so that T has fewer d-branch vertices. Subtree T_u is the subtree obtained from T when $\{u, v\}$ is removed and contains vertex u, while subtree T_v is the subtree that contains vertex v. The following sets of vertices are defined for each

subtree T_u and T_v :

- U_1,V_1: contain vertices that are obligatory d-branch vertices or have already been processed by Algorithm 2.5 and are considered d-branch vertices.
- U_2,V_2: contain vertices that are not d-branch vertices and if an edge incident to them is inserted, they do not turn to be d-branch vertices.
- U_3,V_3: contain vertices that are d-branch vertices but are not obligatory d-branch vertices or have not yet been considered d-branch vertices by the Algorithm 2.5.

The search of a new edge is performed by looking for edges $\{u', v'\}$ (or $\{v', u'\}$) such that $u' \in U_i, v' \in V_j$ with $i, j \in \{1, 2, 3\}$, in the following order:

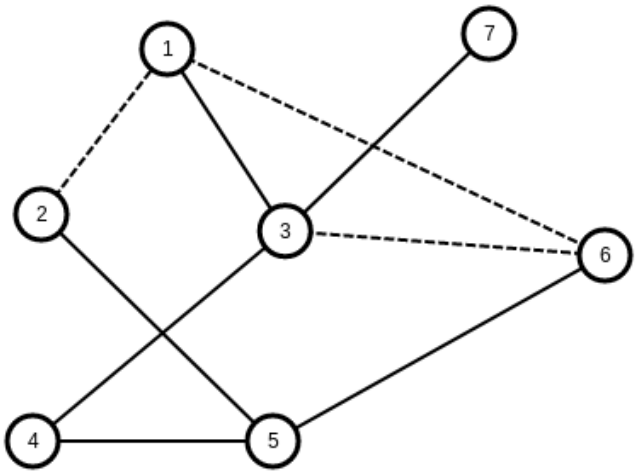
1. $u' \in U_1$ and $v' \in V_1$ and the edge $\{u', v'\}$ has the smallest value $d_G(u') + d_G(v') - n \cdot f_D(u') - n \cdot f_D(v')$. This prioritizes edges whose vertices are obligatory d-branch vertices and have small degree.
2. $u' \in U_1$ and $v' \in V_2$ and the edge $\{u', v'\}$ has the smallest value $d_G(u') - n \cdot f_D(u') + d_G(v')$. In this case, one of the vertices of the edge is a obligatory d-branch and the other vertex has a small degree in the graph. The aim of this criterion is to choose an edge that has one obligatory d-branch vertex and the other one has small degree so that a vertex with small degree is chosen to be part of the solution.
3. $u' \in U_1$ and $v' \in V_3$ and the edge $\{u', v'\}$ has the smallest value $d_G(u') - n \cdot f_D(u') - d_T(v')$. In this case, the objective is to look for an edge with a obligatory d-branch and a non-obligatory d-branch with a large degree, which probably is a d-branch vertex in the optimal solution.
4. $u' \in U_2$ and $v' \in V_2$ and the edge $\{u', v'\}$ has the smallest value $d_G(u') + d_G(v')$. The vertices of these edges are not d-branch vertices and will not turn to be d-branch vertices if an edge incident to them is inserted. The objective is to choose vertices with small degrees.
5. $u' \in U_2$ and $v' \in V_3$ and the edge $\{u', v'\}$ has the smallest value $d_G(u') - d_T(v')$. The objective is to find an edge with one vertex (non d-branch) with a small degree in the graph, and the other vertex is a non-obligatory d-branch vertex with large degree in the tree.

6. $u' \in U_3$ and $v' \in V_3$ and the edge $\{u', v'\}$ has the highest value $d_T(v') + d_T(u')$. The objective is to find an edge whose vertices are non-obligatory d-branch vertices with large degree. This type of vertices are probably in the optimal solution.

In all described cases, the inserted edge will not generate a tree T with new d-branch vertices.

Consider the graph of figure 2.3. In this graph, the edges with broken lines represent edges that belong to the original graph but are not in the solution tree. Working with d = 2, the tree T in figure 2.3 has two branch vertices. When applying the local search on this tree, the tree T' showed in figure 2.4 can be obtained through the following operations: (1) remove the edge $\{1, 3\}$ from the branch vertex v_3 and add the edge $\{1, 2\}$ in the vertex leaf v_2; (2) remove the edge $\{5, 6\}$ from the branch vertex v_5 and add the edge $\{1, 6\}$ in the vertex leaf v_1.

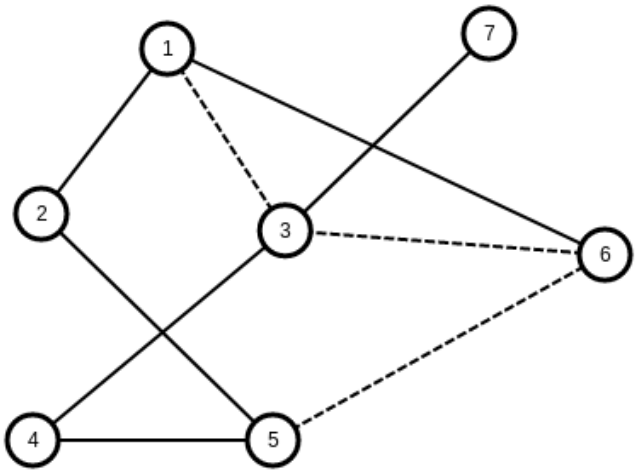
Figure 2.3: A spanning tree with two branch vertices (v_3 and v_5)



2.2.3 Perturbation

The perturbation method should enable the algorithm to escape from local optima and provide diversification to the ILS. The method attempts to replace an edge $\{u, v\}$ of T, which has at least one non-obligatory d-branch vertex, by another edge $\{u', v'\} \neq \{u, v\}$, with $u' \in T_u$ and $v' \in T_v$, creating another d-branch vertex (u' or v'). Algorithm 2.7 presents the pseudocode of the implemented perturbation movement.

Figure 2.4: A spanning tree after the local search without branch vertices



Algorithm 2.7: PERTURBATION

Input: A graph $G = (V, E)$ without bridges, the current solution T_{curr} and the set O_D of obligatory d-branch vertices.

Output: The solution T.

```
1  T ← T_curr
2  DB ← {v ∈ V \ O_D | d_T(v) + l(v) > d}
3  Sort DB in ascending order by value (d_G(v) + l(v))
4  foreach v ∈ DB do
5      foreach u ∈ N_T(v) do
6          Remove the edge {u, v} from T
7          Let {u', v'} = argmin_{i,j}{d_G(i) + d_G(j) | i ∈ T_u,
             j ∈ T_v such d_T(i) = d or d_T(j) = d, but not both } and {u', v'} is not
             forbidden
8          if {u', v'} ≠ ∅ then
9              Insert the edge {u', v'} in T and mark this edge as forbidden.
10             return T
11         else
12             Insert the edge {u, v} in T
13 return T
```

Each vertex v is analyzed according to the value $d_G(v) + l(v)$ in ascending order. Among all possible edges to be added, the chosen edge $\{u', v'\}$ that will reconnect the tree and create (exactly) one d-branch vertex (i.e. $d_T(u') = d$ or $d_T(v') = d$), should be the one that minimizes:

$$\{u', v'\} = argmin_{\{i,j\}}\{d_G(i) + d_G(j)\}$$

This criterion selects edges whose vertices have small degree in G. In this way, when inserting the new edge, the other vertices with larger degree in G will have more chance to "steal" edges from the d-branch vertices during the local search.

The inserted edges are marked as forbidden and cannot be manipulated by the perturbation process until the method finds a better solution. This rule was established with the purpose of not creating cycles of moves by adding and deleting the same edge without having an improvement over the best value found.

The ILS method halts when is not possible to find any unmarked edges to proceed with the perturbation.

2.3 Computational Experiments

To validate the effectiveness of the proposed method, several computational experiments were performed (results are available online [1]). First, experiments were executed with the most used instances in the literature for the 2-MBV problem: 500 instances proposed by Carrabas et al. [9], which contain sparse graphs with different densities, and 21 instances proposed by Silva et al. [46], which are based on graphs that have a Hamiltonian path and therefore it is possible to find spanning trees without branch vertices. Second, we conducted experiments to investigate the impact of different values of d in a set of random instances. The experiments were developed on an Intel (R) Core i5-4460S CPU @ 2.90GHz, with 6 Mb of cache and 8 Gb of RAM using Linux and all methods were programmed in C++ language using the gcc compiler.

2.3.1 Heuristic Results

2-MBV results:

Tables 2.1 and 2.2 show the results obtained by the developed heuristic for the group of instances proposed by Carrabas et al. [9] classified as Medium Instances (Table 2.1) and Large Instances (Table 2.2) for the MBV problem ($d = 2$). These instances correspond to very sparse graphs and they were also studied in Melo et al. [32], but the authors only presented the results of a subset of instances. In addition, the results presented in Melo et al. [32] did not improve those obtained by Marín [29] in that group of instances and for that reason were not considered in Tables 2.1 and 2.2.

[1] `www.ic.uff.br/~yuri/files/dMBV.zip`

Each row represents a group of 25 graphs in Table 2.1 and a group of 5 graphs in Table 2.2. The first two columns represent the number of vertices and the average number of edges of each group. Columns 3 and 4 represent the number of vertices and edges respectively (after decomposition). The **opt** column shows the optimal value of each group of instances. The column **ubM** presents the results obtained by the heuristic developed by Marín [29] and the column **gapM** shows the gap obtained by his heuristic in relation to optimum value. The last three columns show the value obtained by the **ILS** heuristic, the **gap** in relation to the optimum value, and the average **time** to process the group of instances in seconds. The gap shows the percentage difference of the value obtained by the heuristic method in relation to the optimum value, using the following equation: $gap = \frac{heur-opt}{opt} \times 100$.

Table 2.1: Results of the heuristic for Medium Instances for the MBV problem ($d = 2$)

n'	**m'**	n_p	m_p	**opt**	**ubM**	**gapM**	**ILS**	**gap**	**time**
20	41.8	18.1	39.9	0.8	0.8	0.0	0.8	**0.0**	0.00
40	70.8	33.4	64.2	2.8	2.9	3.6	3.0	7.1	0.00
60	95.0	46.0	81.0	6.3	6.6	4.8	6.7	6.3	0.00
80	119.8	58.1	97.8	9.2	9.5	3.3	9.6	4.3	0.00
100	144.0	69.0	112.9	13.3	13.9	4.5	13.8	3.8	0.00
120	168.8	80.0	128.7	17.5	18.0	2.9	18.2	4.0	0.01
140	193.0	92.1	145.0	20.9	21.8	4.3	21.6	**3.3**	0.01
160	217.8	103.1	160.8	25.0	25.8	3.2	25.9	3.6	0.02
180	242.0	112.9	174.8	29.1	30.3	4.1	30.2	**3.8**	0.02
200	266.8	122.4	189.2	32.6	33.8	3.7	33.8	**3.7**	0.02
250	321.0	145.4	216.4	44.6	46.0	3.1	45.7	**2.5**	0.03
300	380.0	164.2	244.2	57.4	59.0	2.8	58.7	**2.3**	0.05
350	434.8	188.7	273.3	68.6	70.3	2.5	70.1	**2.2**	0.08
400	489.0	204.3	293.2	81.8	83.8	2.4	83.5	**2.1**	0.11
450	548.0	228.9	326.7	93.4	95.7	2.5	95.3	**2.0**	0.15
500	602.8	243.7	346.4	106.7	109.4	2.5	108.6	**1.8**	0.19

Marín [29] presented the best result achieved after executing 100 times each instance using his proposed heuristic (**ubM**). The total execution times for Medium and Large instances were 239 seconds and 264 seconds respectively leading to a total of 503 seconds using an Intel Core 2 Quad CPU Q9300, 2.50GHz × 4, with 3 Gb of RAM memory and running on Linux. The heuristic developed in this work consumed 17.5 seconds and 28.7 seconds to solve the Medium and Large instances respectively with a total time of 46.2 seconds.

Table 2.2: Results of the heuristic for Large Instances for the MBV problem ($d = 2$)

n'	**m'**	n_p	m_p	**opt**	**ubM**	**gapM**	**ILS**	**gap**	**time**
600	637	106.4	143.4	183.8	184.0	0.1	185.0	0.7	0.0
600	674	162.6	236.6	167.2	168.8	1.0	168.6	**0.8**	0.1
600	712	205.6	317.6	150.6	154.8	2.8	153.0	**1.6**	0.1
600	749	236.6	385.6	138.8	144.0	3.7	140.4	**1.2**	0.1
600	787	266.4	453.2	125.8	132.8	5.6	128.8	**2.4**	0.2
700	740	123.2	163.2	214.4	215.0	0.3	215.4	0.5	0.0
700	780	181.6	261.6	198.0	199.6	0.8	199.8	0.9	0.1
700	821	229.8	350.8	180.0	184.0	2.2	182.4	**1.3**	0.2
700	861	263.4	424.4	164.0	169.2	3.2	167.4	**2.1**	0.2
700	902	296.8	498.8	154.2	161.8	4.9	157.0	**1.8**	0.2
800	843	133.2	176.2	245.6	246.6	0.4	246.6	**0.4**	0.0
800	886	200.6	286.6	227.6	229.6	0.9	229.8	1.0	0.1
800	930	253.4	383.4	208.4	213.0	2.2	211.4	**1.4**	0.1
800	973	294.2	467.2	194.2	200.2	3.1	197.2	**1.5**	0.3
800	1017	331.8	548.8	176.2	184.0	4.4	179.6	**1.9**	0.4
900	944	143.6	187.6	279.6	280.6	0.4	280.6	**0.4**	0.1
900	989	214.4	303.4	259.2	261.6	0.9	261.0	**0.7**	0.2
900	1034	267.0	401.0	240.6	245.4	2.0	243.6	**1.2**	0.3
900	1079	316.4	495.4	223.2	229.8	3.0	226.6	**1.5**	0.5
900	1124	352.4	576.4	206.0	214.8	4.3	209.4	**1.7**	0.4
1000	1047	150.4	197.4	312.0	313.4	0.4	313.2	**0.4**	0.1
1000	1095	233.0	328.0	290.0	292.4	0.8	292.2	**0.8**	0.3
1000	1143	295.0	438.0	271.2	275.8	1.7	275.0	**1.4**	0.4
1000	1191	342.4	533.4	251.0	257.8	2.7	254.8	**1.5**	0.5
1000	1239	390.2	629.2	235.2	244.6	4.0	238.6	**1.4**	0.9

In these tables (2.1 and 2.2) we can see that the gap value decreases as the instance size is increased, which indicates that the proposed heuristic maintains a small absolute difference with respect to the optimum. On the other hand, as the instances become more complex, the proposed heuristic begins to perform better than the one proposed by Marín [29]. For the 25 Large instances, the ILS heuristic obtained 19 better results and 2 ties, while Marín heuristic obtained 4 better values. For the 41 groups of instances, the proposed heuristic obtained strictly better results in 65.8 % of the instances and equal results in 9.7 % of the instances. The average gap for the heuristic proposed in Marín [29] was 2.6, while the average gap obtained by the ILS heuristic was 2.0.

Table 2.3 shows the results obtained for 21 instances proposed in Silva et al. [46], also for the MBV problem ($d = 2$), where each line represents a graph. The first three

Table 2.3: Heuristics results for *dimacs*, *stein* and *tcp* instances for the MBV problem ($d = 2$)

Instance	**n**	**m**	**ubM**	**ubM-time**	**MPE**	**ILS**	**ILS-time**
le450_15a	450	5714	0	1.5	0	**0**	0.2
le450_15b	450	5734	0	0.7	0	**0**	0.2
le450_15c	450	9803	0	7.2	0	**0**	0.7
le450_15d	450	9757	0	2.5	0	**0**	0.8
le450_25a	450	8168	0	0.4	0	**0**	0.2
le450_25b	450	8169	0	5.5	0	**0**	0.2
le450_25c	450	16680	0	0.4	0	**0**	0.8
le450_25d	450	16750	0	0.4	1	**0**	0.8
le450_5a	450	8260	0	0.2	0	**0**	0.1
le450_5b	450	8263	0	0.2	0	**0**	0.1
le450_5c	450	17343	0	0.5	0	**0**	0.3
le450_5d	450	17425	0	0.4	0	**0**	0.3
steind11	1000	5000	4	45.1	0	**0**	0.3
steind12	1000	5000	4	47.8	1	**0**	0.3
steind13	1000	5000	4	45.0	0	**0**	0.2
steind14	1000	5000	4	42.0	1	**0**	0.3
steind15	1000	5000	4	48.7	1	**0**	0.3
alb1000	1000	1998	9	84.0	16	**1**	1.0
alb2000	2000	3996	19	697.0	28	**2**	7.8
alb3000a	3000	5999	29	2467.0	43	**4**	35.5
alb4000	4000	7997	39	8783.0	58	**4**	67.5

columns of the tables present the file name, the number of vertices n and the number of edges m of the graph. Unfortunately, the decomposition process does not bring any benefits to these instances, so, we omit columns n_p and m_p. Furthermore, columns 4, 6 and 7 show, respectively, the results for these instances obtained by the heuristics of Marín [29] (**ubM**), Melo et al. [32] (**MPE**) and the proposed heuristic in this work (**ILS**). Finally, columns 5 and 8 show the times (in seconds) used by the heuristics **ubM** and **ILS** respectively. In the case of the heuristic **MPE**, the times used to obtain these results are not presented by the authors.

The proposed ILS heuristic presents superior performance in time and quality of the results. The optimal value was reached in all instances of the first two groups (*dimacs* and *stein*). For *tcp* instances, the heuristic obtained better values in much less time than required by other heuristics. The value **ubM** is the minimum value obtained in 100

executions, while heuristics **MEP** and **ILS** were executed only once.

***d*-MBV results:**

In Merabet et al. [33] the authors presented results for the d-MBV problem over a set of random instances with different values of d. Their instances have d-branch vertices in the optimal solution for high values of d. Unfortunately, we were not able to use these instances to compare with our method because they were not available. The authors informed that the set of instances was disposed after the experimental analysis, but they provided the generator used by them to create sparse graphs. So, we used this generator to create new instances which are similar to the instances used by them.

As in Merabet et al. [33] we consider 9 values for the number of vertices $|V| \in \{50, 100, 200, 300, 400, 500, 600, 700, 800\}$ and, for the number of edges m, we used the same equation:

$$m = \left\lfloor |V| - 1 + i \times 1.5 \times \left\lceil \sqrt{|V|} \right\rceil \right\rfloor$$

with $i \in \{1, 2, 3\}$. We have generated 30 instances for each pair $(|V|, i)$ and performed experiments for several values of d. Tables 2.4 and 2.5 show the results obtained for values of $d \in \{2, 3, 4, 5\}$ and $d \in \{6, 7, 8, 9\}$. Each row represents a group of 30 graphs. First column indicates the number of vertices of the group. Columns **ILS** and **time** show the upper bound and computational time obtained by the proposed heuristic. Column **gap** shows the gap relative to the optimum. It can be observed that the number of d-branch vertices decreases rapidly while increasing the d value. Our heuristic again shows a good performance, presenting a very small gap relative to the optimum within low computational time. As shown in Tables 2.4 and 2.5, our heuristic yielded the optimal value several times, mainly for $d \geq 4$.

2.3.2 Exact results

2-MBV results:

Tables 2.6 and 2.7 show the computational time required to obtain exact solutions for the MBV problem ($d = 2$). These optimal values were obtained by solving the model developed in Section 2.1 using Constraints (2.20) and (2.21). If the obtained solution was not an integer solution then a search for violated Constraints (2.22) was performed, and the violated cuts were added to the model.

The first two columns represent the number of vertices and the average number of

Table 2.4: Heuristic results for $d \in \{2, 3, 4, 5\}$ on random instances

	$i = 1$			$i = 2$			$i = 3$		
$\|V\|$	**ILS**	**time**	**gap**	**ILS**	**time**	**gap**	**ILS**	**time**	**gap**
				$d = 2$					
50	6.60	0.000	0.5	3.80	0.001	10.7	2.13	2.134	25.5
100	17.17	0.001	0.6	12.43	0.004	4.5	8.87	0.006	12.2
200	38.00	0.005	0.9	28.93	0.018	3.0	22.30	0.032	8.1
300	60.07	0.011	0.4	47.87	0.045	1.6	37.97	0.089	4.4
400	82.47	0.017	0.2	68.80	0.073	1.2	56.97	0.163	2.6
500	105.20	0.025	0.2	87.73	0.108	1.2	74.93	0.325	2.7
600	128.23	0.039	0.2	107.93	0.188	0.9	93.67	0.428	2.2
700	151.00	0.044	0.2	130.43	0.232	0.7	111.23	0.630	2.0
800	174.77	0.056	0.0	151.47	0.347	0.6	131.23	0.879	1.8
				$d = 3$					
50	1.03	0.000	0.0	0.23	0.000	0.0	0.10	0.100	0.0
100	4.77	0.001	1.4	1.37	0.000	0.0	0.37	0.000	0.0
200	11.47	0.002	2.1	5.30	0.002	1.3	1.53	0.001	0.0
300	20.50	0.005	0.7	10.30	0.008	1.3	5.03	0.004	2.7
400	29.87	0.010	0.8	18.27	0.020	1.9	9.87	0.011	3.1
500	38.60	0.015	0.4	25.40	0.034	2.1	13.87	0.024	4.0
600	50.17	0.020	0.5	32.03	0.061	1.4	19.70	0.046	2.1
700	60.43	0.031	0.3	39.67	0.084	1.5	24.97	0.078	2.3
800	69.67	0.038	0.3	47.43	0.139	1.6	32.63	0.154	3.2
				$d = 4$					
50	0.13	0.000	0.0	0.03	0.000	0.0	0.00	0.000	0.0
100	1.07	0.000	0.0	0.10	0.000	0.0	0.07	0.000	0.0
200	2.57	0.000	0.0	0.43	0.001	0.0	0.10	0.001	0.0
300	5.47	0.001	0.0	1.37	0.001	0.0	0.47	0.002	0.0
400	8.93	0.003	0.8	3.07	0.002	0.0	1.77	0.002	0.0
500	11.93	0.004	0.8	4.80	0.003	0.0	1.83	0.003	0.0
600	15.80	0.006	0.2	6.73	0.004	0.0	2.93	0.005	0.0
700	19.57	0.010	0.3	9.23	0.007	0.0	4.17	0.006	0.0
800	24.57	0.013	0.7	10.17	0.009	1.3	5.33	0.007	0.0
				$d = 5$					
50	0.00	0.000	0.0	0.00	0.000	0.0	0.00	0.000	0.0
100	0.20	0.000	0.0	0.00	0.000	0.0	0.00	0.000	0.0
200	0.47	0.000	0.0	0.03	0.000	0.0	0.03	0.000	0.0
300	1.40	0.000	0.0	0.23	0.000	0.0	0.10	0.000	0.0
400	1.90	0.000	0.0	0.60	0.000	0.0	0.27	0.000	0.0
500	3.60	0.000	0.9	0.93	0.000	0.0	0.30	0.000	0.0
600	3.87	0.000	0.0	1.77	0.000	0.0	0.23	0.000	0.0
700	4.83	0.000	0.0	1.93	0.000	0.0	0.80	0.000	0.0
800	6.77	0.000	0.0	1.87	0.000	0.0	1.10	0.000	0.0

Table 2.5: Heuristic results for $d \in \{6, 7, 8, 9\}$ on random instances

	$i = 1$			$i = 2$			$i = 3$		
$\|V\|$	**ILS**	**time**	**gap**	**ILS**	**time**	**gap**	**ILS**	**time**	**gap**
				$d = 6$					
50	0.00	0.000	0.0	0.00	0.000	0.0	0.00	0.000	0.0
100	0.00	0.000	0.0	0.00	0.000	0.0	0.00	0.000	0.0
200	0.07	0.001	0.0	0.00	0.001	0.0	0.00	0.001	0.0
300	0.33	0.001	0.0	0.00	0.001	0.0	0.00	0.001	0.0
400	0.57	0.001	0.0	0.03	0.001	0.0	0.00	0.002	0.0
500	0.90	0.002	0.0	0.17	0.002	0.0	0.03	0.003	0.0
600	1.00	0.002	0.0	0.53	0.003	0.0	0.07	0.003	0.0
700	1.27	0.003	0.0	0.30	0.004	0.0	0.17	0.004	0.0
800	1.60	0.003	0.0	0.40	0.004	0.0	0.17	0.006	0.0
				$d = 7$					
50	0.00	0.000	0.0	0.00	0.000	0.0	0.00	0.000	0.0
100	0.00	0.000	0.0	0.00	0.000	0.0	0.00	0.000	0.0
200	0.03	0.000	0.0	0.00	0.001	0.0	0.00	0.001	0.0
300	0.07	0.001	0.0	0.00	0.001	0.0	0.00	0.001	0.0
400	0.10	0.001	0.0	0.00	0.001	0.0	0.00	0.002	0.0
500	0.10	0.001	0.0	0.00	0.002	0.0	0.00	0.003	0.0
600	0.20	0.002	0.0	0.10	0.003	0.0	0.00	0.003	0.0
700	0.17	0.003	0.0	0.07	0.003	0.0	0.03	0.004	0.0
800	0.37	0.003	0.0	0.07	0.004	0.0	0.03	0.005	0.0
				$d = 8$					
50	0.00	0.000	0.0	0.00	0.000	0.0	0.00	0.000	0.0
100	0.00	0.000	0.0	0.00	0.000	0.0	0.00	0.000	0.0
200	0.00	0.000	0.0	0.00	0.001	0.0	0.00	0.001	0.0
300	0.03	0.001	0.0	0.00	0.001	0.0	0.00	0.001	0.0
400	0.03	0.001	0.0	0.00	0.001	0.0	0.00	0.002	0.0
500	0.03	0.001	0.0	0.00	0.002	0.0	0.00	0.002	0.0
600	0.00	0.002	0.0	0.00	0.003	0.0	0.00	0.003	0.0
700	0.03	0.002	0.0	0.00	0.003	0.0	0.00	0.004	0.0
800	0.00	0.003	0.0	0.03	0.004	0.0	0.00	0.005	0.0
				$d = 9$					
50	0.00	0.000	0.0	0.00	0.000	0.0	0.00	0.000	0.0
100	0.00	0.000	0.0	0.00	0.000	0.0	0.00	0.000	0.0
200	0.00	0.000	0.0	0.00	0.000	0.0	0.00	0.000	0.0
300	0.00	0.000	0.0	0.00	0.000	0.0	0.00	0.001	0.0
400	0.00	0.001	0.0	0.00	0.001	0.0	0.00	0.002	0.0
500	0.00	0.001	0.0	0.00	0.002	0.0	0.00	0.002	0.0
600	0.00	0.002	0.0	0.00	0.002	0.0	0.00	0.003	0.0
700	0.00	0.002	0.0	0.00	0.003	0.0	0.00	0.004	0.0
800	0.00	0.003	0.0	0.00	0.003	0.0	0.00	0.005	0.0

edges for each group of instances. Columns 3 and 4 show the average number of co-classes and optimal value for each group. Columns 5 and 6 correspond to the times in seconds to find the optimal value by the methods proposed in Marín [29] and Silvestri et al. [48] respectively. Silvestri et al. [48] used one Intel Xeon X5675 running at 3.07 GHz with 96 GB of RAM and a 64-bit Linux operating system with IBM ILOG CPLEX 12.5. Melo et al. [32] method had a poor performance in this group of instance so we chose to compare our results only with Marín [29] and Silvestri et al. [48].

Column 7 shows the execution time obtained by the proposed model in this work using the IBM ILOG CPLEX 12.6 solver. We use the CPLEX default settings and configure it to run over a single thread of execution and a time limit of one hour. The value reported in this column includes the time used to execute all preprocessing operations described in Section 2.1 and the heuristic described in Section 2.2 to define an upper bound. For all instances, the preprocessing time was very close to zero seconds. Finally, the last two columns contain the number of the user's cuts (constraints 2.19) and nodes.

Martinez et al. [31] showed that the time to solve a based Miller-Tucker-Zemlin formulation may be influenced by the vertex that is chosen as the arborescence root.The root selection usually depends on the addressed problem. For example, Akgun et al. [1] proposed a methodology to select the root node in their Miller-Tucker-Zemlin-based formulation for the Min-degree Constrained Minimum Spanning Tree problem. In this problem, the edges have weights and their criterion was to select a vertex with a small sum of the weights of its incident edges.

In our context, we developed the following criterion to select the root node:

$$r = argmax_{u \in V}\{d_G(u) - (d+1) \cdot l(u) + n \cdot f_D(u) + \sum_{v \in N_G(u)} d_G(v) \cdot (1 + f_D(v))\}$$

The first term $(d_G(u) - (d+1) \cdot l(u) + n \cdot f_D(u))$ considers the particular characteristics of vertex u. Vertices that are obligatorily d-branch vertices ($f_D(u) = 1$), with large degree and small $l(u)$ value (adjacent leaves) have a greater chance of being chosen as root. The second term of the expression $(\sum_{v \in N_G(u)} d_G(v) \cdot (1 + f_D(v)))$ benefits those vertices that are in a dense zone of the graph and with adjacent vertices that are obligatorily d -branch vertices. We believe that a good root will be in a dense zone of the graph, reducing the height of the resulting arborescence.

The results show the effectiveness of the proposed exact method. All groups of instances in Table 2.6 were solved faster than the previous exact methods. The results

Table 2.6: Exact results for Medium Instances for the MBV problem ($d = 2$)

n'	m'	co-class	opt	timeM	timeS	time	cuts	nodes
20	41.8	2.4	0.8	0.0	0.0	**0.0**	0.4	0.0
40	70.8	7.5	2.8	0.1	0.1	**0.0**	3.0	2.7
60	95.0	12.2	6.3	1.4	0.5	**0.1**	14.7	17.9
80	119.8	16.4	9.2	2.2	0.7	**0.1**	18.0	4.5
100	144.0	20.2	13.3	2.9	1.0	**0.2**	23.4	23.4
120	168.8	24.7	17.5	3.7	1.1	**0.4**	28.8	28.0
140	193.0	28.7	20.9	4.9	2.0	**0.6**	35.4	71.4
160	217.8	31.9	25.0	6.1	1.9	**0.7**	38.0	36.4
180	242.0	35.6	29.1	6.8	2.5	**1.0**	43.0	86.8
200	266.8	38.4	32.6	7.8	3.1	**0.8**	43.1	41.4
250	321.0	46.1	44.6	11.3	3.1	**1.3**	48.1	101.8
300	380.0	51.8	57.4	13.6	4.2	**1.5**	51.9	79.7
350	434.8	60.1	68.6	20.3	6.9	**3.1**	68.7	221.7
400	489.0	64.9	81.8	24.2	9.1	**2.5**	67.3	181.8
450	548.0	72.3	93.4	29.7	9.5	**3.9**	70.1	335.3
500	602.8	79.0	106.7	35.3	9.8	**3.3**	70.0	206.2

Table 2.7: Exact results for Large Instances for the MBV problem ($d = 2$)

n'	m'	co-class	opt	timeM	timeS	time	cuts	nodes
600	637	37.6	183.8	5.1	3.2	**0.3**	30.0	18.2
600	674	52.8	167.2	10.9	8.7	**1.4**	45.6	94.0
600	712	56.8	150.6	19.9	10.3	**1.9**	64.4	30.0
600	749	50.0	138.8	26.7	17.6	**2.3**	70.2	85.2
600	787	47.8	125.8	39.2	16.2	**3.9**	62.8	197.4
700	740	42.0	214.4	6.5	8.7	**0.5**	37.0	20.0
700	780	58.8	198.0	16.8	11.0	**2.0**	57.8	170.0
700	821	64.6	180.0	37.8	12.5	20.0	74.8	1591.2
700	861	58.4	164.0	39.7	17.4	**7.5**	75.4	904.8
700	902	59.2	154.2	57.8	14.7	**7.5**	93.2	812.0
800	843	45.8	245.6	6.7	10.3	**0.4**	39.4	7.6
800	886	67.4	227.6	19.1	11.2	**1.8**	62.8	178.0
800	930	74.4	208.4	85.0	22.7	**5.8**	90.6	365.4
800	973	74.0	194.2	65.6	48.8	**9.8**	88.0	695.8
800	1017	69.4	176.2	167.2	37.1	**11.3**	103.8	568.6
900	944	51.0	279.6	10.0	12.6	**0.8**	44.6	46.0
900	989	69.8	259.2	23.4	66.2	**2.8**	63.2	268.8
900	1034	79.0	240.6	44.5	30.2	**19.4**	92.6	1121.2
900	1079	86.0	223.2	94.0	90.5	**9.0**	105	406.8
900	1124	77.2	206.0	81.2	30.7	**9.1**	94.6	322.0
1000	1047	52.4	312.0	10.6	26.2	**1.1**	42.2	62.6
1000	1095	78.8	290.0	71.1	17.0	**4.0**	86.0	172.0
1000	1143	91.4	271.2	112.4	57.1	**7.8**	99.4	458.0
1000	1191	91.6	251.0	150.2	75.4	**16.1**	109.4	1138.6
1000	1239	96.4	235.2	642.9	62.6	**31.0**	110.0	1920.8

presented in Table 2.7 shows that for all instances, excepting the group of graphs with 700 vertices and 821 edges, the optimal values were reached using the proposed formulation in shorter times than previous works. We believe this behavior was obtained due to the preprocessing method, which made it possible to obtain graphs with smaller dimensions, and also allowed to fix the value of several variables by using Constraints (2.16) and (2.17), resulting in a lighter formulation.

Table 2.8: Exact results for *tcp* instances for the MBV problem ($d = 2$)

Instance	**n**	**m**	**opt**	**timeMelo**	**time**	**cuts**	**nodes**
alb1000	1000	1998	0	30.13	284.3	0	875
alb2000	2000	3996	0	180.18	1856.7	0	1094
alb3000a	3000	5999	0	74.70	3600.0	0	40
alb4000	4000	7997	0	1778.87	3600.0	0	0

Table 2.8 shows the results only for the *tcp* instances, since instances *dimacs* and *stein* were exactly solved by the ILS (the method found a 0 bound). As we stated before, the decomposition process does not bring any benefits to these instances because none of them have cut vertices that split the graph into three or more components and only instance le450_15b has two bridges. For this reason, two (out of four) of these instances were not solved by the proposed exact method into the time limit. On the other hand, Melo et al. [32] method seems to perform particularly well in these instances. As we can see in Table 2.8, their method could solve the 4 instances into the time limit with a faster machine (Intel Core i7-4790K (4.00GHz) CPU and 16GB of RAM).

***d*-MBV results:**

Results of the exact approach for $d \in \{2, 3, 4, 5, 6, 7, 8, 9\}$ are presented in Tables 2.9 and 2.10.

In Merabet et al. [33] the results obtained by using an exact method with their formulation are shown for different values of d. Their results have shown that the problem becomes more difficult to solve as the parameter d increases. So, we have tested our approach to solve similar instances using exact methods and we came to a different conclusion.

We notice that instances with greater value of d are "easier" to solve because less computational time has been spent to solve them. We believe that the inclusion of the constraints (2.16) and (2.17) in our formulation is the main reason for this difference. With the increment of the parameter d it is easier to classify a vertex as non obligatory

Table 2.9: Exact results for values of $d \in \{2, 3, 4\}$ on random instances

	$i = 1$			$i = 2$			$i = 3$		
$\|V\|$	**opt**	**time**	**nodes**	**opt**	**time**	**nodes**	**opt**	**time**	**nodes**
				$d = 2$					
50	6.57	0.0	0.1	3.43	0.0	3.8	1.70	1.8	20.7
100	17.07	0.0	1.4	11.90	0.1	3.0	7.90	0.3	31.5
200	37.67	0.0	1.5	28.10	0.1	1.3	20.63	0.7	31.1
300	59.83	0.1	16.3	47.10	0.3	46.5	36.37	0.8	68.9
400	82.33	0.1	0.0	68.00	0.2	0.4	55.53	1.0	34.3
500	105.00	0.1	3.1	86.70	0.4	51.5	72.93	1.5	108.0
600	127.97	0.1	4.8	107.00	0.5	10.5	91.63	1.4	6.5
700	150.70	0.1	0.0	129.53	0.5	11.3	109.10	2.6	114.5
800	174.70	0.1	0.0	150.57	0.8	30.3	128.93	2.8	401.2
				$d = 3$					
50	1.03	0.0	13.0	0.23	0.0	0.6	0.10	0.1	2.8
100	4.70	0.0	5.7	1.37	0.1	255.6	0.37	0.0	2.5
200	11.23	0.1	52.3	5.23	0.6	806.9	1.53	0.1	30.3
300	20.37	0.1	2.3	10.17	0.5	294.6	4.90	0.2	35.1
400	29.63	0.1	0.0	17.93	0.2	24.3	9.57	0.2	16.5
500	38.43	0.1	6.6	24.87	0.2	11.0	13.33	0.5	33.3
600	49.90	0.1	2.5	31.60	0.5	24.1	19.30	0.6	37.1
700	60.27	0.1	3.6	39.07	0.5	33.5	24.40	0.9	58.8
800	69.47	0.2	5.2	46.67	0.7	27.9	31.63	6.6	1503.1
				$d = 4$					
50	0.13	0.0	0.1	0.03	0.0	0.3	0.00	0.0	0.0
100	1.07	0.0	0.0	0.10	0.0	1.0	0.07	0.0	1.7
200	2.57	0.0	0.9	0.43	0.0	4.7	0.10	0.1	6.4
300	5.47	0.0	3.3	1.37	0.1	10.8	0.47	0.1	13.4
400	8.87	0.1	47.5	3.07	0.1	4.5	1.77	0.2	24.6
500	11.83	0.1	1.3	4.80	0.2	38.5	1.83	0.2	10.6
600	15.77	0.4	302.1	6.73	0.2	38.0	2.93	0.3	26.6
700	19.50	0.1	6.6	9.23	0.3	40.4	4.17	0.3	22.8
800	24.40	0.3	104.3	10.03	0.3	38.0	5.33	0.4	35.8

d-branch. On the other hand, it is also more difficult to classify a vertex as obligatory d-branch.

2.3.3 Analyzing the impact of the inequalities

To investigate how the inequalities used affect the performance of the exact method, the result of the relaxation of the model in the root node was analyzed under several circumstances. Tables 2.11 and 2.12 show the results of these experiments.

The first column shows the result of the relaxation when executing the model without

Table 2.10: Exact results for values of $d \in \{5, 6, 7, 8, 9\}$ on random instances

	$i = 1$			$i = 2$			$i = 3$		
$\|V\|$	**opt**	**time**	**nodes**	**opt**	**time**	**nodes**	**opt**	**time**	**nodes**
				$d = 5$					
50	0.00	0.0	0.0	0.00	0.0	0.0	0.00	0.0	0.0
100	0.20	0.0	0.0	0.00	0.0	1.1	0.00	0.0	0.2
200	0.47	0.0	0.7	0.03	0.0	0.0	0.03	0.0	1.9
300	1.40	0.0	2.5	0.23	0.0	3.3	0.10	0.1	6.3
400	1.90	0.0	0.7	0.60	0.1	9.2	0.27	0.1	10.8
500	3.57	0.1	5.1	0.93	0.1	2.1	0.30	0.2	8.8
600	3.87	0.2	211.3	1.77	0.1	5.8	0.23	0.2	10.6
700	4.83	0.1	37.3	1.93	0.2	42.0	0.80	0.3	42.9
800	6.77	0.1	9.7	1.87	0.2	27.2	1.10	0.3	34.2
				$d = 6$					
50	0.00	0.0	0.0	0.00	0.0	1.2	0.00	0.0	0.1
100	0.00	0.0	0.0	0.00	0.0	0.1	0.00	0.0	1.5
200	0.07	0.0	0.0	0.00	0.0	0.1	0.00	0.0	0.0
300	0.33	0.0	3.6	0.00	0.0	2.5	0.00	0.1	4.3
400	0.57	0.0	5.3	0.03	0.1	4.4	0.00	0.1	2.3
500	0.90	0.0	10.3	0.17	0.1	1.2	0.03	0.2	10.8
600	1.00	0.0	1.9	0.53	0.1	26.8	0.07	0.2	2.8
700	1.27	0.1	6.6	0.30	0.1	9.2	0.17	0.2	3.7
800	1.60	0.1	4.7	0.40	0.2	17.5	0.17	0.3	16.4
				$d = 7$					
50	0.00	0.0	0.0	0.00	0.0	0.0	0.00	0.0	0.0
100	0.00	0.0	0.0	0.00	0.0	0.0	0.00	0.0	0.8
200	0.03	0.0	0.0	0.00	0.0	0.0	0.00	0.0	0.3
300	0.07	0.0	1.4	0.00	0.0	2.7	0.00	0.1	2.2
400	0.10	0.0	0.9	0.00	0.0	0.2	0.00	0.1	1.0
500	0.10	0.0	1.4	0.00	0.1	1.1	0.00	0.1	2.2
600	0.20	0.0	4.3	0.10	0.1	2.6	0.00	0.2	6.1
700	0.17	0.0	2.6	0.07	0.1	2.3	0.03	0.2	5.5
800	0.37	0.1	1.1	0.07	0.1	1.6	0.03	0.2	3.3
				$d = 8$					
50	0.00	0.0	0.0	0.00	0.0	0.0	0.00	0.0	0.0
100	0.00	0.0	0.0	0.00	0.0	0.0	0.00	0.0	0.0
200	0.00	0.0	0.0	0.00	0.0	0.0	0.00	0.0	0.0
300	0.03	0.0	0.0	0.00	0.0	0.0	0.00	0.1	0.1
400	0.03	0.0	0.8	0.00	0.0	0.7	0.00	0.1	1.0
500	0.03	0.0	0.6	0.00	0.1	2.3	0.00	0.1	4.4
600	0.00	0.0	0.3	0.00	0.1	2.0	0.00	0.1	0.4
700	0.03	0.0	0.0	0.00	0.1	0.4	0.00	0.2	3.2
800	0.00	0.1	0.5	0.03	0.1	4.7	0.00	0.2	2.8
				$d = 9$					
50	0.00	0.0	0.0	0.00	0.0	0.0	0.00	0.0	0.0
100	0.00	0.0	0.0	0.00	0.0	0.0	0.00	0.0	0.0
200	0.00	0.0	0.0	0.00	0.0	0.0	0.00	0.0	0.0
300	0.00	0.0	0.0	0.00	0.0	0.0	0.00	0.0	0.0
400	0.00	0.0	0.0	0.00	0.0	0.0	0.00	0.1	0.0
500	0.00	0.0	1.0	0.00	0.0	3.0	0.00	0.1	0.9
600	0.00	0.0	0.3	0.00	0.1	1.0	0.00	0.1	4.6
700	0.00	0.0	1.0	0.00	0.1	0.7	0.00	0.1	1.2
800	0.00	0.0	0.6	0.00	0.1	0.5	0.00	0.2	1.6

Table 2.11: Impact of inequalities on relaxation in medium instances

R	**R(2.16)**	**R(2.17)**	**R(2.19)**	**R(2.21)**	**R(2.22)**
0.386	0.386	0.575	0.386	0.386	0.386
1.676	1.676	2.153	1.705	1.676	1.676
4.184	4.184	5.127	4.194	4.184	4.184
6.564	6.564	7.783	6.573	6.564	6.564
9.900	9.900	11.726	9.912	9.900	9.900
13.160	13.160	15.584	13.160	13.160	13.160
15.910	15.910	18.620	15.910	15.910	15.910
19.221	19.221	22.706	19.228	19.221	19.221
22.820	22.820	26.347	22.820	22.820	22.820
25.726	25.726	29.981	25.733	25.726	25.726
36.352	36.352	41.675	36.352	36.352	36.352
46.832	46.832	53.681	46.832	46.832	46.832
56.842	56.842	63.877	56.852	56.842	56.842
69.021	69.021	77.271	69.021	69.021	69.021
78.584	78.584	88.241	78.593	78.584	78.584
90.264	90.264	101.686	90.264	90.264	90.264

Table 2.12: Impact of inequalities on relaxation in large instances

R	**R(2.16)**	**R(2.17)**	**R(2.19)**	**R(2.21)**	**R(2.22)**
172.850	172.850	180.333	172.850	172.850	172.850
150.483	150.483	163.756	150.483	150.483	150.483
130.987	130.987	147.210	130.987	130.987	130.987
115.492	115.492	136.070	115.492	115.492	115.492
101.147	101.147	123.905	101.147	101.147	101.147
203.400	203.400	211.017	203.400	203.400	203.400
178.567	178.567	193.633	178.567	178.567	178.567
157.333	157.333	175.653	157.333	157.333	157.333
138.386	138.386	160.772	138.386	138.386	138.386
124.450	124.450	151.135	124.450	124.450	124.450
233.667	233.667	242.017	233.667	233.667	233.667
207.250	207.250	223.393	207.250	207.250	207.250
182.937	182.937	204.230	182.937	182.937	182.937
164.637	164.637	189.850	164.637	164.637	164.637
145.033	145.033	172.346	145.033	145.033	145.033
266.733	266.733	275.083	266.733	266.733	266.733
238.634	238.634	253.900	238.634	238.634	238.634
215.117	215.117	235.620	215.117	215.117	215.117
192.087	192.087	217.920	192.087	192.087	192.087
172.357	172.357	202.178	172.357	172.357	172.357
297.933	297.933	307.483	297.933	297.933	297.933
267.167	267.167	282.950	267.167	267.167	267.167
240.883	240.883	265.287	240.883	240.883	240.883
217.630	217.630	244.905	217.630	217.630	217.630
196.340	196.340	230.267	196.340	196.340	196.340

any of the restrictions $\{2.16, 2.17, 2.19, 2.21, 2.22\}$ (denoted as ILP_{basic}). The next 4 columns show the R relaxation value of the ILP_{basic} model by adding only the valid inequality in parentheses.

These results highlight that inequality (2.17) and, to a lesser extent, the inequality (2.19) impact the performance of the model. Interesting is the fact that the model is stronger when it establishes which vertices are not branch vertices than when it establishes which vertices are obligatory. This would explain why instances with large values of d were solved in less time. The instances used in these experiments are sparse graphs, so when values of d are large, many vertices can be detected as not being branch vertices in the solution, because they do not have the sufficient degree for this. Inequalities related to co-classes (2.19) also impact the performance of the model as evidenced in Table 2.11. For larger instances, at least in the root node, it was not seen to have a positive impact, but in general performance, the model with co-classes is faster than without them.

2.4 Conclusions

In this chapter, methods were developed for obtaining exact and heuristic solutions for the d-MBV problem. A decomposition scheme was applied based on bridges and articulation points of the graph. In the computational experiments, the average number of vertices in relation to the original graph was reduced in 42,3% and the average number of edges was reduced in 52,7%.

An ILS heuristic was developed and obtained good quality results for different d values. Particularly, for the $2-$MBV problem, the heuristic provided better results in 65.8% of the instances and equal results in 9.7% for the instances of Carrabas et al. [9]. For the instances used in Silva et al. [46] the heuristic obtained better results in all instances which previous works have not reached the optimal value.

A based Miller-Tucker-Zemlin formulation and some new valid inequalities were proposed for the problem. The computational results show the effectiveness of the proposed method, since 97,6% of the instances of Carrabas et al. [9] were solved faster than previous works (for the $2-$MBV problem).

As verified by the experiments carried out, the analysis of the structure of the graph, as well as the use of concepts such as co-classes had the greatest impact on the effectiveness of the proposed method, both in the decomposition process and in the creation of new equations for the model. Moreover, the experiments on the analyzed instances

created based on [33] have shown that the number of solved instances increases and the computational time decrease as the d value rises, since the number of non-obligatorily vertices is increased.

Chapter 3

The Rainbow Cycle Cover

An edge-colored graph is a graph $G = (V, E)$, with a set of labels or colors L and a color function $f : E \to L$ which assigns a color from L to each edge of E. A large number of applications can be modeled using edge-colored graphs, being useful in situations that need to differentiate the types of connections between the vertices [19, 20, 21, 22].

For example, suppose a company wants to do a product promotion tour going through a few places in a city. When passing between two places, the area being crossed may be labeled by a major characteristic of its population (young, elderly, women, etc.). The places may be represented as vertices of a graph, the connection between the places as edges and these characteristics as colors of the edges. If the objective of the tour is to reach a wide range of possible customers, the problem may be defined as finding a multicolored cycle with the largest possible size. Multicolored cycles are also called rainbow cycles [2].

A rainbow cycle is a cycle with all its edges of different colors. Single vertices are considered trivial rainbow cycles. A rainbow cover for the graph G is defined as a disjoint collection of rainbow cycles, which means that each vertex can only belong to exactly one rainbow cycle. According to these definitions, the RCC problem is defined as:

Problem 2 (RCC problem) *Given an undirected graph G, the Rainbow Cycle Cover (RCC) problem consists of finding the minimum number of disjoint rainbow cycles covering G.*

Figure 3.1 shows an example of an edge-colored graph with 4 possible colors (labels) for the edges. A feasible solution for the RCC problem on this graph is shown in Figure 3.2. In this case there are two non-trivial rainbow cycles and one trivial rainbow cycle.

Moreover, the RCC problem has been proved to be NP-hard in [26]. In [55] it is

Figure 3.1: Edge-colored graph with 4 possible colors (labels)

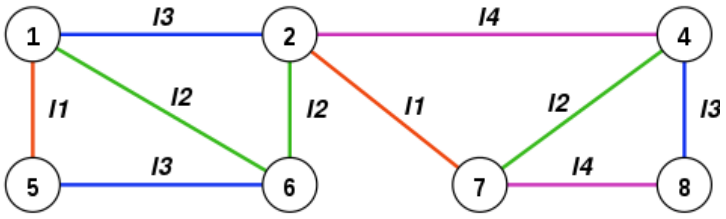
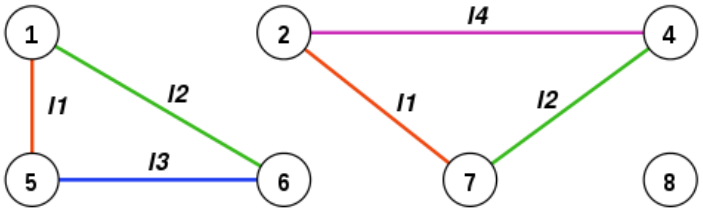


Figure 3.2: A feasible solution for the graph



shown that it is still NP-hard even if the graph does not have cycles of size 3. Recently, in [47] an integer mathematical formulation was presented for this problem and tested in different instances.

In this chapter we present a modification of the mathematical formulation presented in [47], as well as a preprocessing procedure to reduce the size of instances for the RCC problem. Moreover, we define a new family of valid cuts based on properties of edge-colored graphs.

3.1 Mathematical Model

Given an undirected graph $G = (V, E)$, where V denotes the set of vertices ($|V| = n$) and E the set of edges, we define by $\delta(v)$ the set of edges incident to a vertex v in G, where $d(v) = |\delta(v)|$ denotes the degree of vertex v. We also define $E_l = \{e \in E \mid f(e) = l\}$ as the set of edges associated with a color $l \in L$. Finally, R denotes the family of all non-trivial rainbow cycles of the graph G, where a non-trivial rainbow cycle $H \in R$ is defined by its edge set, i.e., $H \subseteq E$.

As observed in [47] the maximum number of non-trivial rainbow cycles in a solution for a graph with n vertices is bounded by the value $\bar{c} = \lfloor \frac{n}{3} \rfloor$, that occurs when all rainbow cycles have the minimum length (3 edges). In [47], the authors proposed an integer formulation, defining a set of indices $I = \{0, \ldots, \bar{c} - 1\}$ that represents possible non-trivial cycles in a solution. The binary variables used in their model are defined as follows:

y_v^c: 1, if and only if the vertex v belongs to the non-trivial cycle c.

x_e^c: 1, if and only if the edge e belongs to the non-trivial cycle c.

γ_c: 1, if and only if c is a non-trivial cycle of the solution.

And the proposed formulation:

$$\min \sum_{c \in I} \gamma_c + \sum_{v \in V} M \left(1 - \sum_{c \in I} y_v^c\right) \tag{3.1}$$

subject to:

$$\sum_{v \in V} y_v^c \leq |L| \gamma_c, \qquad c \in I \tag{3.2}$$

$$\sum_{v \in V} y_v^c \geq 3\gamma_c, \qquad c \in I \tag{3.3}$$

$$\sum_{c=0}^{\overline{c}-1} y_v^c \leq 1, \qquad v \in V \tag{3.4}$$

$$\sum_{e \in \delta(v)} x_e^c = 2y_v^c, \qquad v \in V, c \in I \tag{3.5}$$

$$x_e^c \leq y_v^c, \qquad v \in V, e \in \delta(v), c \in I \tag{3.6}$$

$$\sum_{e \in \delta(S)} x_e^c \geq 2(y_v^c + y_u^c - 1), \qquad S \subset V,\ \{v,u\} \in \{S, V \setminus S\},\ c \in I \tag{3.7}$$

$$\sum_{e \in E_l} x_e^c \leq 1, \qquad l \in L,\ c \in I \tag{3.8}$$

$$\gamma_{c+1} \leq \gamma_c, \qquad c \in \{0, \ldots, \overline{c} - 2\} \tag{3.9}$$

$$\sum_{c=v+1}^{\overline{c}} y_v^c = 0, \qquad v \in V : v < \overline{c} \tag{3.10}$$

$$y_v^c \leq \sum_{w<v} y_w^{c-1}, \qquad v \in V \setminus \{0\},\ c \in \{2, \ldots, \overline{c} - 1\} \tag{3.11}$$

$$y_v^c \in \{0,1\}, \qquad v \in V,\ c \in I \tag{3.12}$$

$$x_e^c \in \{0,1\}, \qquad e \in E,\ c \in I \tag{3.13}$$

$$\gamma_c \in \{0,1\}, \qquad c \in I \tag{3.14}$$

where $\delta(S) = \{\{u,v\} \in E \mid u \in S \text{ and } v \in V \backslash S\}$.

The objective function (3.1) aims to minimize the number of rainbow cycles. The first part minimizes the sum of variables γ_c, that is, the number of non-trivial cycles. The second part tries to force the vertices to belong to a cycle, giving a sufficient large weight M to each trivial cycle (i.e. isolated vertex).

Constraints (3.2) express that the number of vertices belonging to a non-trivial cycle

cannot be greater than the number of colors. Constraints (3.3) ensure that a non-trivial cycle must have at least three vertices. Constraints (3.4) guarantee that a vertex will belong to at most one non-trivial cycle. Constraints (3.5) ensure that a vertex belonging to a non-trivial cycle must have exactly two adjacent edges. Constraints (3.6) impose that if a vertex does not belong to a non-trivial cycle c, none of its adjacent edges can belong to that cycle. One can notice that, due to constraints (3.5), constraints (3.6) are redundant in the formulation. However in [47], the authors used (3.6) as valid cuts in the branch-and-cut algorithm. Moreover, constraints (3.7) prevent two different non-trivial cycles to use the same variable γ_c. Constraints (3.8) impose that in each non-trivial cycle there may be at most one edge with color l.

Constraints (3.9)-(3.11) are not necessary for the model, but their inclusion helps to eliminate symmetric solutions. Constraints (3.9) ensure that a variable for the non-trivial cycle γ_c is not used if the variable γ_{c-1} is not used. Constraints (3.10) indicate that a vertex v, with index lower than $\overline{c}$, cannot belong to a non-trivial cycle with index c, such that $c > v$. Finally (3.11) impose that a vertex v can only belong to a non-trivial cycle with index c if at least one vertex w, where $w < v$, belongs to the non-trivial cycle with index $c - 1$.

In [47], the authors presented the formulation (3.1)-(3.14) and chose to set $M = 2\overline{c}$ in the second part of the objective function (3.1). They claimed that this strategy forces the vertices of the graph to belong to a non-trivial cycle whenever possible and, consequently, solve the RCC problem. Actually, with $M = 2\overline{c}$, the formulation presented in [47] aims to cover the graph with disjoint rainbow cycles that minimizes the number of trivial cycles, breaking ties in favor of covers with minimum number of non-trivial cycles. As a consequence of this choice, the solution obtained in [47] does not always correspond to the solution of the RCC problem. To illustrate this point let us consider the graph of Figure 3.3.

Note that, while the optimal solution for the RCC is the one illustrated in Figure 3.4a with 3 cycles, the optimal solution obtained by solving the mathematical model proposed in [47] with

$$M = 2\overline{c} = 2\lfloor\frac{n}{3}\rfloor = 2\lfloor\frac{10}{3}\rfloor = 6$$

is the one depicted in Figure 3.4b with 4 cycles. In fact, only when $M = 1$, the formulation (3.1)-(3.14) can be used to solve the RCC problem.

Our contribution here are twofold: first, in order to differentiate the two problems, we define the Trivial Cycle RCC (TC-RCC) problem as the problem of finding a rainbow

Figure 3.3: Edge-colored graph with 8 colors

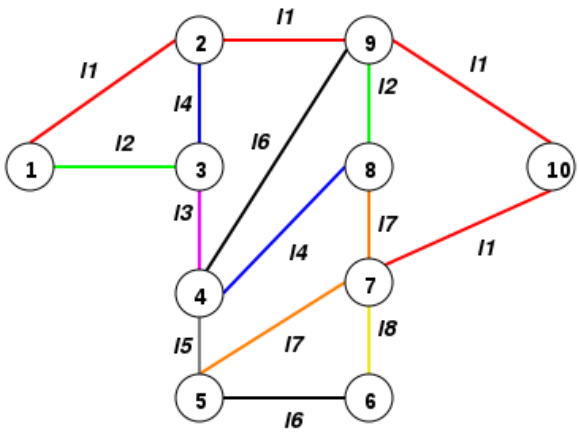
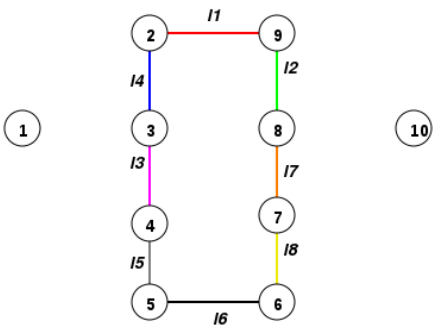
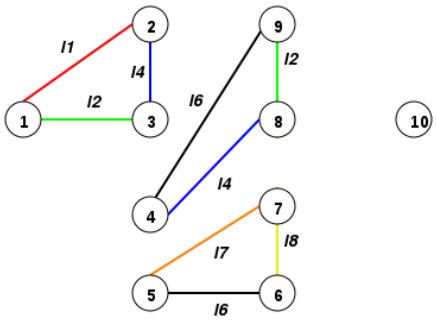


Figure 3.4: (a) Optimal solution for the RCC problem. (b) Optimal solution by solving the model proposed in [47] with $M = 2\bar{c}$.

(a) A non-trivial cycle and two trivial cycles (isolated vertices).



(b) Three non-trivial cycles and one trivial cycle (isolated vertex).



cover for the graph with minimum number of trivial cycles (isolated vertices) and, as a secondary objective, to minimize the number of non-trivial cycles. Next, we propose a slight modification in the set of constraints of the original formulation presented for the TC-RCC problem. We replace constraints (3.7) by constraints (3.15) in order to prevent two different non-trivial cycles to use the same variable γ_c.

$$\sum_{e \in H} x_e^c + x_f^c \leq |H|, H \in R,\ f \in E \setminus H,\ c \in I \tag{3.15}$$

Besides that, in [47], the authors claimed that in the objective function (3.1), the first

summation that minimizes the number of non-trivial cycles is less then or equal to $\overline{c}$ in the worst case. Thanks to this observation a value equal to $2\overline{c}$ is given to the weight M in their experiments. Actually, one can notice that any value $M \geq \overline{c}$ is sufficient to prioritize the minimization of the trivial cycles. Therefore, we refer to the original and modified formulation for the TC-RCC problem (when $M \geq \overline{c}$) as TC-RCC^{IP} and TC-RCC$^{IP}_{MOD}$, respectively. Similarly, we refer to the modified formulation for the RCC problem (when $M = 1$) as RCC$^{IP}_{MOD}$.

3.1.1 Preprocessing and cutting

In [47] some properties of multicolored graphs were presented which were added as constraints (cuts) to the problem. Following similar ideas, we design new constraints in order to strengthen the formulation.

Let's first define $\zeta(v)$ as the number of different colors incident to vertex $v \in V$. In addition, let $\zeta(v) \oplus \zeta(v)$ be the number of different colors incident to the vertices u and v. Note that, if $\zeta(v) < 2$, the vertex v will be an isolated vertex (i.e. a trivial cycle) and it can be removed from the original graph in a preprocessing phase. A similar reasoning can be applied with edges. If an edge $\{u, v\} \in E$ belongs to a rainbow cycle, then the edges connected to u and v must have different colors. This leads to the conclusion that if $\zeta(u) \oplus \zeta(v) \leq 2$, then $\{u, v\}$ cannot belong to any rainbow cycle and can also be removed. These properties were first noted in [47] and used as formulation constraints. Following their lead, we also consider removing bridges of the graph, as they do not belong to any cycle.

Furthermore, as in Chapter 2, the co-classes will have an application in the process of dimension reduction. We notice that edges that belong to the same co-class have an interesting property: if one edge of a co-class belongs to a rainbow cycle, then all other edges of this co-class have to be in this same cycle. This is easy to see since any pair of edges of the co-class is a 2-edge-cut in the graph. Therefore, if a co-class has at least two edges with the same color, then none of its edges will be in any solution and the set of edges can be eliminated. Otherwise, we include the following constraint for each *feasible* co-class found:

$$\sum_{e \in H} x^c_e = x^c_f \cdot |H|, \qquad H \in C^C,\ c \in I,\ \text{for any } f \in H. \tag{3.16}$$

Notice that constraints (3.16) ensure that if an edge f, that represents co-class H, belongs

to cycle c, then all edges of H must be in the same cycle. Otherwise none of the edges of the co-class will be in the solution. Preprocessing and reducing the instances of a problem can decrease the resolution time [32, 34]. For this reason Algorithm 3.1 was developed to process the original graph $G = (V, E)$.

Algorithm 3.1: REDUCE_GRAPH

```
Input: A graph G
Output: A reduced graph Ḡ.
change ← True
repeat
    change ←False
    if there are vertices such that ζ(v) < 2 then
        remove these vertices from G
        change ←True
    else
        if there are edges {u, v} such that ζ(u) ⊕ ζ(v) ≤ 2 then
            remove these edges from G
            change ←True
        else
            if there are bridges {u, v} then
                remove these edges from G
                change ←True
            else
                C^C = FIND_COCLASS(G)
                foreach co-class H ∈ C^C do
                    if H has repeated colors then
                        remove all edges of H from G
                        change ←True
                    else
                        include constraints (3.16), based on co-class H
until change =False
return Ḡ
```

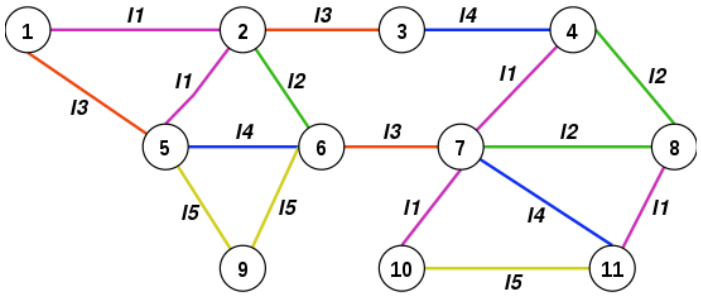
After applying the preprocessing procedure described in Algorithm 3.1, we could end up with a disconnected graph. Let $\overline{G}_1, \overline{G}_2, ..., \overline{G}_k$ be the set of graphs induced by the connected components of the reduced graph $\overline{G}$. Since two connected components do not share any edges, and consequently, any cycles, we can express the objective function of the problem as:

$$\sum_{i=1}^{k} z_k + |V_D|$$

where V_D is the set of removed vertices from G and z_k is the optimal solution obtained from

solving the problem in graph G_k. In order to illustrate the reduction process presented in Algorithm 3.1, we consider the graph shown in Figure 3.5.

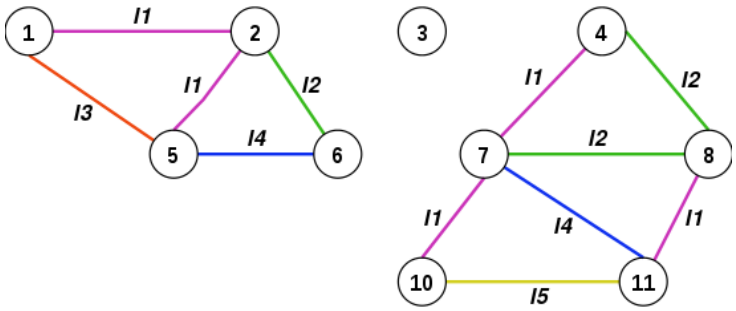
Figure 3.5: Edge-colored graph



Note that, since vertex 9 has only one incident color, the reduction process shown in Algorithm 3.1 removes this vertex (lines 4 - 6). The resulting graph does not contain edges $\{u, v\}$ such that $\zeta(v) \oplus \zeta(v) \leq 2$ or bridges, so only the co-classes are analyzed (lines 16 - 20).

The co-classes of the graph after removing vertex 9 are: $C_1 = \{\{1,2\},\{1,5\}\}$, $C_2 = \{\{2,3\},\{3,4\},\{6,7\}\}$ and $C_3 = \{\{7,10\},\{10,11\}\}$. The edges {2,3} and {6,7} of the co-class C_2 have the same color l_3(red), so all edges of C_2 are removed from the graph resulting in the graph shown in Figure 3.6.

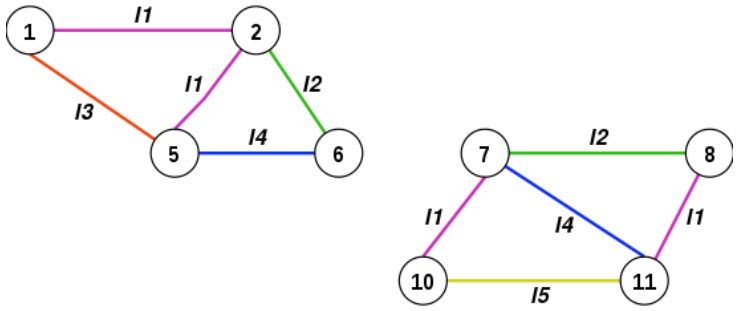
Figure 3.6: Graph after removing vertex 9 and all edges in the co-class C_2



Now, vertex 3 became isolated and it is removed from the graph (lines 4 - 6). Furthermore, edge $\{4,8\}$ is also removed from the graph because $\zeta(4) \oplus \zeta(8) \leq 2$ (lines 8 - 10). Vertex 4 is removed from the resulting graph because it has only one incident color (lines 4 - 6) resulting in the graph of Figure 3.7.

After the reduction process, we have to solve the problem in two graphs with 4 vertices and 5 edges each, instead of one with 11 vertices and 16 edges. As a final step, the method will determine the following co-classes in the processed graph: $C_1 = \{\{1,2\},\{1,5\}\}$,

Figure 3.7: Graph after the reduction process



$C_2 = \{\{2,6\},\{5,6\}\}$, $C_3 = \{\{7,10\},\{10,11\}\}$ and $C_4 = \{\{7,8\},\{8,11\}\}$, including the following constraints in the formulation:

$$\begin{aligned} x^c_{\{1,2\}} + x^c_{\{1,5\}} &= 2x^c_{\{1,2\}}, \qquad c \in I \\ x^c_{\{2,6\}} + x^c_{\{5,6\}} &= 2x^c_{\{2,6\}}, \qquad c \in I \\ x^c_{\{7,10\}} + x^c_{\{10,11\}} &= 2x^c_{\{7,10\}}, \qquad c \in I \\ x^c_{\{7,8\}} + x^c_{\{8,11\}} &= 2x^c_{\{7,8\}}, \qquad c \in I \end{aligned}$$

with $\mid I \mid = \overline{c} = \lfloor \frac{8}{3} \rfloor = 2$.

We finish this section by presenting an approach to find the co-classes of a graph, based on the algorithm for bridges detection proposed in [44].

Algorithm 3.2: FIND_COCLASS

Input: A graph $G = (V, E)$ without bridges.
Output: The co-class family C^C.

```
1  C^C ← ∅
2  foreach e ∈ E do
3      mark[e] ← FALSE
4  foreach e ∈ E do
5      if mark[e] = FALSE then
6          mark[e] ← TRUE
7          β_G ← the set of all bridges in G\e (presented in [44])
8          if β_G ≠ ∅ then
9              foreach h ∈ β_G do
10                 mark[h] ← TRUE
11             β_G ← β_G ∪ {e}
12             C^C ← C^C ∪ β_G
13 return C^C
```

The method, presented in Algorithm 3.2, receives as input a graph without bridges and repeatedly removes an edge from the graph and determines which edges become bridges

by this process. The new bridges and the eliminated edge form a maximal co-class, so they are marked as analyzed to avoid finding the same co-class again. This algorithm guarantees to find the set of maximal co-classes, because it processes all the edges of the graph and does not include twice the same co-class in C^C. Since the algorithm proposed in [44] has complexity $O(|V|+|E|)$, then Algorithm 3.2 has complexity $O(|E|(|V|+|E|))$.

3.2 Matheuristic for the RCC problem

The RCC problem becomes quickly more difficult as the size of the instances increases. In this sense, developing heuristic methods arises as a necessity. Analyzing the structure of the problem we can see that from a given solution, finding a neighbor solution of better quality can be an extremely complicated and expensive task. In the last years, different strategies have been developed that combine exact and heuristics algorithms. In this context, matheuristics [28, 30, 15] have attracted the attention of the scientific community. Matheuristics may use some features of the mathematical models developed for the problems to customize heuristics to solve these problems or use heuristics to improve time effectiveness of mathematical programming techniques. In this section, we develop a matheuristic for the RCC problem based on the Iterated Local Search (Algorithm 2.2) and the ILP model (Section 3.1), in order to quickly obtain solutions with good quality.

The developed matheuristics is based on ILS metaheuristic ([27]), so this section will show the methods used for the Initial Solution, Local Search and the Perturbation. In our implementation, the best solution found is updated every time a better solution is obtained in the local search. Moreover, we use as stop condition that a maximum of 10 iterations be performed, as we want to quickly get a solution.

3.2.1 Initial Solution

The mathematical formulation for the RCC (Section 3.1) problem works well for small graphs, but as the size of the graphs increases, the computational time to solve the problem increases very fast. One of the causes is the large number of variables and restrictions of this model.

Let $ILP^{(q)}(G)$ be the procedure that solves the formulation for the RCC in the graph G, restricting to q the number of non-trivial cycles. Algorithm 3.3 shows the pseudocode used to construct an initial solution for the RCC problem.

Algorithm 3.3: INITIAL SOLUTION

Input: The graph G.
Output: A rainbow cycle cover R.

1 $R \leftarrow \emptyset$
2 $R' \leftarrow ILP^{(2)}(G)$
3 $N \leftarrow$ non-trivial cycles in R'
4 $T \leftarrow$ trivial cycles in R'
5 **while** $N \neq \emptyset$ **do**
6 $\quad R \leftarrow R \cup N$
7 $\quad H \leftarrow$ induced graph by T
8 $\quad R' \leftarrow ILP^{(2)}(H)$
9 $\quad N \leftarrow$ non-trivial cycles in R'
10 $\quad T \leftarrow$ trivial cycles in R'
11 $R \leftarrow R \cup T$
12 **return** R

Basically, Algorithm 3.3 solves the mathematical formulation by restricting the maximum number of non-trivial cycles to two. Each non-trivial cycle found will be part of the solution under construction. Then, the algorithm solves again the restricted mathematical formulation for the induced graph on the set of isolated vertices. This process is repeated until the solution for the induced graph does not contain any non-trivial cycle. In that case, all the vertices will be added to the solution as trivial cycles (line 11). When fixing the maximum number of non-trivial cycles for the formulation, the number of the variables and the restrictions are considerably reduced.

3.2.2 Local Search

Once an initial solution is obtained, a local search is applied (Algorithm 3.4). A neighbor solution of a current solution is obtained as follows. First, any two non-trivial cycles are removed and their vertices are added to the set of isolated vertices. The other non-trivial cycles are fixed. Then, $ILP^{(q)}(G)$ is applied over the set of isolated vertices limiting the maximum number of non-trivial cycles to three. Then, the neighbor is obtained joining the solution obtained by solving the formulation $ILP^{(3)}$ over the set of isolated vertices with the previous fixed non-trivial cycles.

Instead of analyzing all the possible neighbors obtained by removing two non-trivial cycles, we decided to analyze NN neighbors for a given solution. We set NN as two times the amount of non-trivial cycles. The reason is to limit to a linear exploration of the search space related to the number of non-trivial cycles and not to a quadratic one,

as it would be if exploring all the neighbors obtained by removing each two non-trivial cycles.

Moreover, we explored this neighborhood using the first improvement strategy that stops the local search as soon as the first neighbor that improves the current solution is found.

Algorithm 3.4: *Local_Search*

Input: A rainbow cycle cover R.
Output: A rainbow cycle cover R^*.

```
1  nN ← 0
2  R* ← R
3  while nN < NN do
4      nN ← nN + 1
5      N ← non-trivial cycles in R*
6      T ← trivial cycles in R*
7      C, C' ← any two cycles in N
8      N ← N \ {C ∪ C'}
9      T' ← all vertices in the cycles C and C'
10     H ← induced graph by T ∪ T'
11     R' ← ILP^(3)(H)
12     if |N| + |R'| < |R*| then
13         R* ← N ∪ R'
14         nN ← 0
15 return R*
```

3.2.3 Perturbation

The goal of the perturbation in the ILS metaheuristics is to escape from local optima and to introduce diversity in the search space. To achieve this, $\alpha|N|$ ($\alpha \in (0, 1]$) non-trivial cycles of the set of non-trivial cycles of the solution are removed. Then, the solution is reconstructed in a similar way to the one performed in Algorithm 3.3. The main difference is that when the $ILP^{(2)}$ procedure is executed the first time, restrictions are added to avoid forming the removed cycles.

3.3 Experimental analysis

In this section, the results obtained by the proposed mathematical formulation and the matheuristic are presented. We analyze the same set of instances used in [47] for the

Algorithm 3.5: *Perturbation*

Input: A rainbow cycle cover R.
Output: A perturbed rainbow cycle cover R^*.

```
N ← non-trivial cycles in R
LC ← list formed by α|N| cycles in N
LR ← list of restrictions to avoid forming cycles in LC.
N ← N \ LC
T ← trivial cycles in R and the vertices in LC
R ← ∅
flag ← 0
while N ≠ ∅ do
    R ← R ∪ N
    H ← induced graph by T
    if flag = 0 then
        R' ← ILP^(2)(H, LR)
        flag ← 1
    else
        R' ← ILP^(2)(H)
    N ← non-trivial cycles in R'
    T ← trivial cycles in R'
R ← R ∪ T
return R
```

exact method. In addition, some of the instances used in [7] were analyzed (all results are available online [2]). These instances correspond to graphs with number of vertices ranging from 20 to 100, and densities varying between 0.1, 0.2 and 0.3. The number of colors is always smaller than the number of vertices and varies between 3 and 18. The tests were developed on an Intel (R) Core i5-4460S CPU @ 2.90GHz, with 6 Mb cache and 8 Gb of RAM using the operating system Fedora 22 and all methods were programmed in C ++ language using the gcc compiler. The IBM ILOG CPLEX 12.6 was used with a single thread of execution and 10800 seconds as time limit. All other CPLEX parameters were left to their default values.

3.3.1 Exact results

Our improved formulations (TC-RCC$^{IP}_{MOD}$ and RCC$^{IP}_{MOD}$) are the starting point for the development of our branch-and-cut algorithms. We chose to not apply constraints (3.9) to (3.11) and (3.15) for graphs with $|L| \leq 5$. Constraints (3.9) to (3.11) are basically used to reduce the symmetry of the problem and empirical results indicate a gain of

[2] `www.ic.uff.br/~yuri/files/RCC.zip`

performance by using (3.9) to (3.11) only for instances where $|L| \geq 6$. Besides that, note that constraints (3.15) are not really necessary for graphs with $|L| \leq 5$. This is because in order to have more than one cycle associated with the same variable γ_c, there should exist two different cycles with at least three edges, each one with different colors (as ruled by constraints (3.8)). So, if $|L| < 6$ then no variables γ_c may be associated with more than one cycle. On the other hand, if $|L| > 5$ then we include constraints (3.15) as lazy constraints, i.e., the separation problem is solved only when an integer solution is obtained.

In order to separate constraints (3.15), a depth first search algorithm is executed in each rainbow cycle associated with a non-zero variable γ_c with more than 5 vertices. If a cycle $H \subseteq E$ is found that contains all vertices associated with variable γ_c, then the constraint is not violated. Otherwise, an edge $f \notin H$ associated with γ_c (i.e. $x_f^c = 1$) is selected and a cut is generated, relating f, H and c as constraint (3.15). We only add one cut for each violated γ_c variable.

Computational results for the exact method are summarized in Tables 3.1 and 3.2. Each line presents average statistics over five instances. First, we analyze the impact of the reduction process described in the previous section over the set of instances. The results are shown in Table 3.1. The first four columns report the group identification (**ID**), number of vertices (**n**), number of edges (**m**) and number of colors (**l**) respectively. Columns 5, 6, 7 and 8 present, respectively, the average number of vertices (**n'**), edges (**m'**), co-classes used as constraints 3.16 ($\boldsymbol{C^C_{CUT}}$) and co-classes removed ($\boldsymbol{C^C_{DEL}}$), after the reduction process. Unfortunately, the method was not able to increase the number of connected components in the set of instances tested, but we could slightly reduce the number of vertices and edges in the graph, which is directly related to the number of variables and constraints. One can notice that the method is especially more effective in graphs with few edges and colors, with reductions of up to 45%, 22%, 13.5% and 11.2% (ID1, ID10, ID19 and ID28) in the number of vertices. Moreover, the number of deleted co-classes was very small and did not impact the numerical results significantly. On the other hand, CPLEX tends to solve the formulation faster with the inclusion of constraints (3.16).

Next, we conduct the experiment to determine the effectiveness of the modified formulations TC-RCC$^{IP}_{MOD}$ and RCC$^{IP}_{MOD}$. In order to be able to compare our results with [47], a value equal to $2\lfloor \frac{n}{3} \rfloor$ is given to the weight M in the TC-RCC$^{IP}_{MOD}$ formulation. In Table 3.2, columns 2 and 3 present, respectively, the average number of rainbow cycles

ID	n	m	l	n'	m'	C^C_{CUT}	C^C_{DEL}
1	20	39	3	11.0	23.8	2.2	1.6
2	20	39	6	14.6	31.4	4.0	0.8
3	20	39	11	16.4	35.0	4.8	0.0
4	20	58	3	16.4	50.0	0.8	1.0
5	20	58	6	18.4	55.6	1.4	0.0
6	20	58	12	18.6	56.2	1.4	0.0
7	20	77	4	19.8	76.8	0.4	0.0
8	20	77	7	19.6	76.4	0.2	0.2
9	20	77	13	19.8	76.8	0.4	0.0
10	30	74	4	23.4	63.2	3.4	2.0
11	30	74	7	27.0	70.6	5.8	0.2
12	30	74	13	27.2	70.8	5.8	0.2
13	30	117	4	27.6	112.0	1.8	0.4
14	30	117	7	29.2	115.8	2.6	0.2
15	30	117	14	29.2	115.8	2.6	0.4
16	30	161	4	29.4	160.2	0.8	0.0
17	30	161	8	29.4	160.2	0.8	0.2
18	30	161	15	29.6	160.6	1.0	0.0
19	40	118	4	34.6	110.6	3.4	1.2
20	40	118	7	35.0	111.4	3.4	0.8
21	40	118	14	36.2	114.2	4.2	0.0
22	40	196	4	38.4	193.2	1.0	0.2
23	40	196	8	38.6	194.2	0.6	0.0
24	40	196	16	39.2	195.2	1.0	0.0
25	40	274	5	39.2	273.0	0.0	0.2
26	40	274	9	39.4	273.4	0.2	0.0
27	40	274	17	39.4	273.4	0.2	0.0
28	50	173	4	44.4	164.6	4.6	0.6
29	50	173	8	45.8	168.2	5.2	0.4
30	50	173	15	46.2	169.0	5.6	0.2
31	50	295	5	48.2	291.8	1.2	0.4
32	50	295	9	48.4	292.6	1.0	0.8
33	50	295	17	49.2	294.2	1.8	0.0
34	50	418	5	49.8	417.8	0.8	0.0
35	50	418	9	49.6	417.4	0.6	0.2
36	50	418	18	49.8	417.8	0.8	0.0

Table 3.1: Reduction process for the RCC problem.

found (**TC-RCC**^{IP}) and average execution time in seconds obtained by the original formulation reported in [47]. The numbers in subscript indicate the number of instances not solved to optimality within 10800 seconds. Similarly, columns 4 and 5 present the same information for the modified formulation (**TC-RCC**$^{IP}_{MOD}$) strengthened by the preprocessing and cutting methods. We also provide information about the number of nodes examined in the search by the new approach (**nodes**), and, for each group of instances,

the best values (time and solution) for the TC-RCC problem are highlighted in boldface. Moreover, the last three columns in Table 3.2 display the same information as columns 4 to 6, but now regarding the RCC problem with the modified formulation RCC^{IP}_{MOD} (with $M = 1$). In our results, the time included the preprocessing time (always less than 1 seconds).

ID	**TC-RCC^{IP}**	$T(s)$	**TC-RCC$^{IP}_{MOD}$**	$T(s)$	***nodes***	**RCC$^{IP}_{MOD}$**	$T(s)$	***nodes***
1	18.00	0.01	18.00	**0.00**	0.0	18.00	0.00	0.0
2	13.60	0.14	13.60	**0.02**	3.2	13.60	0.01	0.0
3	10.60	0.18	10.60	**0.03**	0.0	10.60	0.03	0.0
4	14.80	0.13	14.80	**0.09**	61.2	14.80	0.13	41.0
5	9.80	1.76	9.80	**0.31**	171.2	9.80	0.22	123.8
6	6.80	1.71	6.80	**0.23**	61.0	6.80	0.23	43.0
7	10.00	2.41	10.00	**0.49**	449.0	10.00	0.40	387.8
8	6.80	5.86	6.80	**0.83**	667.0	6.60	0.52	365.8
9	4.80	5.71	4.80	**0.51**	147.0	4.60	0.35	117.8
10	21.80	2.59	21.80	**0.26**	201.4	21.80	0.31	180.0
11	19.40	15.16	19.40	**0.59**	162.0	19.40	0.45	80.0
12	15.40	9.33	15.40	**0.34**	15.6	15.40	0.32	9.6
13	18.20	21.38	18.20	**4.23**	1564.4	18.20	4.23	1212.6
14	13.00	56.52	13.00	**12.13**	4574.6	13.00	3.89	1163.6
15	9.60	54.07	9.60	**2.94**	850.6	9.60	2.42	718.2
16	15.20	180.5	15.20	**83.98**	7060.0	15.20	93.53	5922.4
17	8.00	210.39	8.00	**67.87**	23011.6	8.00	63.10	27000.6
18	5.40	55.79	5.40	**5.07**	1555.6	5.40	8.45	3486.2
19	30.60	10.67	30.60	**1.40**	719.2	30.60	1.30	768.6
20	24.80	129.83	24.80	**3.11**	1173.2	24.80	2.74	673.4
21	20.40	34.08	20.40	**1.36**	245.4	20.40	1.52	145.6
22	25.00	314.22	25.00	**83.01**	2456.4	25.00	82.55	3038.4
23	15.80	**361.94**	15.80$^{(1)}$	2424.49	407878.6	15.8	1613.69	332859.8
24	11.00	480.33	11.00	**169.70**	24912.4	11.00	23.53	4466.8
25	14.60$^{(2)}$	5136.55	14.60	**2881.00**	70865.8	14.60	1314.51	24876.6
26	8.80$^{(2)}$	6420.92	8.80$^{(1)}$	**4836.47**	307291.4	8.40	2358.98	158188.0
27	5.20$^{(1)}$	3185.14	5.20	**344.84**	21455.6	5.20	555.02	25825.2
28	35.80	526.1	35.80	**44.72**	1565.8	35.80	35.73	1505.0
29	30.40	1185.35	30.40	**107.87**	26816.0	30.20	59.81	15207.2
30	23.40	694.63	23.40	**98.95**	21368.4	23.00	45.53	13319.0
31	25.40$^{(1)}$	7354.71	25.40$^{(1)}$	**5529.30**	74803.6	25.20	1572.96	18156.4
32	18.40$^{(1)}$	**5558.08**	18.40$^{(3)}$	8099.75	533482.0	18.20$^{(1)}$	6548.15	515622.8
33	12.80	2695.05	12.80	**1635.61**	129143.8	12.40	736.82	79698.8
34	19.80$^{(5)}$	10800	**19.60**$^{(1)}$	**6165.31**	57837.8	19.60$^{(2)}$	7503.57	62218.4
35	12.80$^{(5)}$	10214.81	12.80$^{(4)}$	**9974.30**	691713.6	12.20$^{(3)}$	10306.44	449691.2
36	6.60$^{(3)}$	8954.48	**6.40**$^{(1)}$	**4664.18**	464425.6	6.40$^{(1)}$	2931.14	101895.4
	15.63$^{(20)}$	1796.68	**15.62**$^{(12)}$	**1312.37**		15.54$^{(7)}$	996.46	

Table 3.2: Results for the original and improved formulation for the TC-RCC and RCC problems.

The results show that the proposed method (using preprocessing and the modified mathematical formulation) spends less computational time to solve the 36 groups of instances (1312.37 sec) than the original formulation (1796.68 sec) with a slightly slower machine (in [47] the experiments were run using a Intel Xeon 3.07GHz with 96 GB of RAM

with IBM ILOG CPLEX 12.5). The exception is given by the groups of instances with ID equal to 23 and 32. Regarding group 23, only one instance ($Rand\_40\_196\_51\_8.rnd$) could not be solved into the time limit of 10800 seconds, which leads to a large displacement of the mean for the group. Disregarding this instance, the average resolution time for the remaining four graphs in this group was only 330.61 seconds. Besides that, we could prove optimality for 2 whole groups of instances (25 and 27), where 3 new optimal solutions were discovered. Furthermore, the upper bounds of groups 34 and 36 were improved by finding 6 new optimal solutions into these groups. In addition, 2 new optimal solutions were found in groups 26 and 35. On the other hand, the modified formulation failed to prove optimality in 3 already known optimal solutions reported in [47] (groups 23 and 32).

Regarding the results for the RCC problem, optimal solutions with fewer cycles were achieved in 4 groups of instances (8, 9, 29 and 30) when compared to the TC-RCC problem. Moreover, the mean time for RCC^{IP}_{MOD} was 24% faster than the time obtained by the TC-RCC^{IP}_{MOD} formulation. The reason for that lies in the fact that the big M factor in the TC-RCC formulation can lead to a main computational disadvantage. In the RCC, this weakness is avoided by setting $M = 1$ and eliminating this factor from the objective function.

3.3.2 Heuristics results

The computational results for the matheuristic are summarized in Table 3.3. As the graphs of these instances have small dimensions, we decided to use $\alpha = \frac{1}{3}$ in the perturbation procedure. The IBM ILOG CPLEX 12.6 was used to solve the model $ILP^{(q)}$with a single thread of execution and 30 seconds as time limit for graphs with 50 vertices or less and 60 seconds for graphs with more than 50 vertices.

In Table 3.3 each line presents average statistics over five instances. The first four columns report the group identification (**ID**), number of vertices (**n**), number of edges (**m**) and number of colors (**l**) respectively. Columns 5 and 6 present, respectively, the average number of rainbow cycles found and average execution time in seconds obtained by solving the problem using the mathematical formulation ILP presented in Section 3.1.

Columns 7 and 8 present the same information obtained using the matheuristic $ILS^{(ILP)}$. The last column shows the gap, calculated as:

$$gap = \frac{ILP - ILS^{(ILP)}}{ILP}$$

Table 3.3: Experiments results for $ILS^{(ILP)}$.

ID	n	m	l	ILP	ILP-time	$ILS^{(ILP)}$	$ILS^{(ILP)}$-time	gap
1	20	39	3	18.0	0.00	18.0	0.12	0.0
2			6	13.6	0.01	13.6	1.19	0.0
3			11	10.6	0.03	10.6	1.66	0.0
4	20	58	3	14.8	0.14	14.8	1.25	0.0
5			6	9.8	0.23	9.8	4.37	0.0
6			12	6.8	0.24	6.8	8.85	0.0
7	20	77	4	10.0	0.41	10.0	1.79	0.0
8			7	6.6	0.53	6.6	6.81	0.0
9			13	4.6	0.36	4.6	9.30	0.0
10	30	74	4	21.8	0.31	21.8	1.44	0.0
11			7	19.4	0.45	19.4	9.87	0.0
12			13	15.4	0.32	15.4	9.83	0.0
13	30	117	4	18.2	4.43	18.2	2.66	0.0
14			7	13.0	4.09	13.0	8.36	0.0
15			14	9.6	2.57	9.6	29.02	0.0
16	30	161	4	15.2	94.43	15.4	3.94	0.2
17			8	8.0	64.15	8.4	13.96	0.4
18			15	5.4	8.67	5.4	55.04	0.0
19	40	118	4	30.6	1.31	30.6	2.79	0.0
20			7	24.8	2.77	24.8	9.32	0.0
21			14	20.4	1.53	20.4	12.58	0.0
22	40	196	4	25.0	83.04	25.2	5.09	0.2
23			8	15.8	1717.42	15.8	15.41	0.0
24			16	11.0	23.78	11.0	41.63	0.0
25	40	274	5	14.6	1322.37	15.0	7.44	0.4
26			9	8.4	2388.52	9.4	22.59	1.0
27			17	5.2	561.52	5.6	55.63	0.4
28	50	173	4	35.8	35.80	35.8	3.15	0.0
29			8	30.2	60.29	30.2	22.59	0.0
30			15	23.0	45.75	23.0	33.53	0.0
31	50	295	5	25.2	825.95	26.0	8.72	0.8
32			9	18.2	505.61	19.0	24.32	0.8
33			17	12.4	938.91	12.6	50.65	0.2
34	50	418	5	19.6	7503.57	20.4	11.42	0.8
35			9	12.2	10306.44	13.4	25.96	1.2
36			18	6.4	2931.14	6.6	91.83	0.2
37	100	595	5	69.4	...	**69.0**	32.85	-0.4
38			10	56.4	...	**54.0**	151.77	-2.4
39			19	46.2	...	**46.0**	267.82	-0.2
40	100	1090	6	46.4	...	**43.8**	116.26	-2.6
41			11	36.6	...	**32.6**	335.23	-4.0
42			21	34.0	...	**23.4**	312.22	-10.6
43	100	1585	6	44.4	...	**33.6**	92.47	-10.8
44			11	33.6	...	**20.8**	445.28	-12.8
45			21	30.0	...	**10.8**	463.62	-19.2

When the matheuristic $ILS^{(ILP)}$ obtains a better result than the exact method ILP, values are highlighted in boldface.

For graphs with 20 and 30 vertices, the branch-and-cut used few seconds to obtain the exact solution. In these graphs, the matheuristic $ILS^{(ILP)}$ has spent more time and obtained the optimal values for all groups, except for the groups with id = 16 and id = 17. For graphs with 40 and 50 vertices, the matheuristic presents a very small gap related to the exact method and a maximum average time of 91.83 seconds (reached in the group with id = 36). The proposed method is quite effective in groups of graphs with 100 vertices. In these groups, the branch-and-cut fails to find the optimal values for all instances in a time limit of 3 hours (10800 seconds) of execution, and the proposed matheuristic got much better results in a maximum average time of 463.62 seconds.

3.3.3 Analyzing the impact of the strategies

To investigate how the strategies used affect the performance of the exact method, the result of the relaxation of the model in the root node was analyzed under several circumstances.

Table 3.4 shows the results of these experiments. The first column shows the identifier of the group of graphs. Column 2 shows the result of the best value obtained by our model (the same as column 7 in Table 3.2). Column 3 shows the result of the relaxation of the model described in Section 3.1 (RCC_B) without applying the restrictions associated to the co-class and without using the decomposition scheme (Algorithm 3.1). Column 4 shows the result of the relaxation of the RCC_B model without the decomposition scheme but adding the constraints relative to the co-classes that are detected in the graph. Column 5 shows the result of the relaxation of the RCC_B model by applying the decomposition scheme and without the restrictions associated with co-class. Finally, the last column shows the relaxation value of the proposed model using the decomposition algorithm and the co-class.

The results show that the use of the decomposition scheme and the use of constraints associated with the co-class impact the resolution of the model. In general, using only restrictions related to the co-class (R_2) turned out to be better than using only decomposition scheme (R_3) in 47.22% of the groups of instances. On the other hand, only applying the decomposition scheme turned out to be better than using only the restrictions related to the co-class in 25.00% of the groups of instances. When analyzing these results together with the results of Table 3.1, we can verify that groups of graphs with high number

of co-classes were benefited by the use of constraints (3.16), while graphs that have the largest number of co-classes removed were benefited by the decomposition scheme.

Table 3.4: Impact of the different strategies

ID	RCC	R_1	R_2	R_3	R_4
1	18.000	11.787	12.222	13.600	13.956
2	13.600	8.333	9.083	9.167	9.583
3	10.600	6.000	7.091	6.000	7.091
4	14.800	8.833	9.089	10.000	10.133
5	9.800	5.167	5.333	5.167	5.333
6	6.800	3.500	3.683	3.500	3.683
7	10.000	5.450	5.600	5.450	5.600
8	6.600	3.543	3.543	3.714	3.714
9	4.600	2.092	2.277	2.092	2.277
10	21.800	12.875	13.727	14.350	14.750
11	19.400	7.971	10.314	8.143	10.486
12	15.400	6.092	8.800	6.462	8.985
13	18.200	9.525	10.100	9.825	10.250
14	13.000	5.914	6.600	5.914	6.600
15	9.600	3.907	4.650	4.279	4.836
16	15.200	7.950	7.950	7.950	7.950
17	8.000	4.100	4.188	4.275	4.275
18	5.400	2.373	2.373	2.373	2.373
19	30.600	15.175	16.900	15.625	17.050
20	24.800	11.372	13.343	11.543	13.428
21	20.400	8.614	10.750	8.614	10.750
22	25.000	11.800	11.950	11.950	12.100
23	15.800	7.100	7.100	7.100	7.100
24	11.000	4.187	4.187	4.187	4.187
25	14.600	8.480	8.480	8.640	8.640
26	8.400	4.977	4.977	4.977	4.977
27	5.200	2.918	2.918	2.918	2.918
28	35.800	18.050	19.400	18.500	19.400
29	30.200	11.325	13.162	11.675	13.337
30	23.000	8.560	10.520	8.933	10.707
31	25.200	11.600	11.600	11.600	11.600
32	18.200	6.800	6.800	7.333	7.333
33	12.400	4.259	4.259	4.259	4.259
34	19.600	10.160	10.160	10.160	10.160
35	12.200	5.734	5.734	5.911	5.911
36	6.400	2.967	2.967	2.967	2.967

3.4 Conclusions

In this chapter we establish the difference between two interpretations of the Rainbow Cycle Cover (RCC) problem. With this objective, the Trivial Cycle RCC (TC-RCC) problem is defined as the problem of finding a rainbow cover for the graph with minimum number of trivial cycles (isolated vertices) and, as a secondary objective, to minimize the number of non-trivial cycles. We proposed a slight modification in the set of original constraints presented for the TC-RCC and RCC problems, together with a preprocessing and cutting method based on properties of edge-colored graphs. As a result of this preprocessing procedure and the inclusion of constraints (3.16), a more efficient method was developed and it was able to find 11 new optimal solutions (8 more than [47]) for instances that were unsolved for the TC-RCC.

Besides that, this methodology was used to study the RCC problem on the same set of instances. We pointed out that some of the optimal solutions found had fewer cycles than the optimal solution values obtained for the TC-RCC problem. Our goal here was to establish the difference between the two objective functions and give the reader a broader understanding of the problem domain.

Finally, a based ILS matheuristic was proposed for the RCC problem. According to the results, the matheuristic obtains very precise values for instances up to 50 vertices. Moreover, for graphs with 100 vertices, the performance of the matheuristic was superior to that of the branch-and-cut in terms of time and quality of the results

Chapter 4

The Rainbow Spanning Forest

A rainbow tree is a connected acyclic subgraph of a colored graph G with all its edges of different colors. Single vertices are considered as trivial rainbow trees. A rainbow spanning forest of G is defined as a disjoint collection of rainbow trees that covers all vertices of the graph, which means that each vertex can only belong to exactly one rainbow tree. Brualdi and Hollingsworth [6] studied the problem of finding rainbow spanning trees and forests in edge-colored complete bipartite graphs. A necessary and sufficient condition for the existence of a heterochromatic (rainbow) spanning tree in a graph was given by Suzuki [51]. Moreover, Carraher et al. [10] studied bounds for the number of edge-disjoint rainbow spanning trees. Formally, the Rainbow Spanning Forest problem is defined as:

Problem 3 (RSF problem) *Given an undirected colored graph G, the Rainbow Spanning Forest (RSF) problem consists of finding a rainbow spanning forest of G with the least number of rainbow trees.*

The RSF problem has been proved to be NP-hard in Li and Zhang [26] even for edge-colored graphs with two colors. Recently, the authors in [8] proved that this problem is NP-complete on trees and is solvable in polynomial time on paths, cycles and if the optimal solution value is equal to 1. Moreover, in Carrabs et al. [7], an integer mathematical formulation and a multi-start scheme based on a greedy algorithm were presented for this problem. They studied multicolored graphs with different densities and colors. Their formulation was able to solve problems with up to 50 vertices. In addition, the heuristic approach presented accurate results when compared to the exactly solution provided by their formulation.

In this section we present a modification of the mathematical formulation presented in [7], as well as a fast GRASP [41] metaheuristic for problem. The proposed modified

formulation was capable of solving 38 more instances than the original one. Furthermore, the GRASP method reduced the execution time in 87.0% while presenting better (in average) bounds for the tested set of instances.

The remainder of this chapter is organized as follows. Section 4.1 contains the proposed formulation while Section 4.2 presents the heuristic GRASP for RSF. In Section 4.3, experimental results obtained with the instances proposed in [7] are presented. Finally, in Section 4.4 the conclusions of this paper are discussed.

4.1 Mathematical formulation

Carrabs et al. [7] present a mathematical formulation based on the maximum number of connected components in a colored graph $G = (V, E, L)$. They concluded that if no upper bound is known, then the value $\overline{c} = n - 1$ can be used as an upper bound for the maximum number of connected components, where $n = |V|$. Since the structure of the graph is unknown, this value also represents an upper bound for the number of rainbow trees in a rainbow spanning forest. Their integer formulation uses the set of indices $I = \{0, \ldots, \overline{c} - 1\}$ that represents possible connected components (rainbow trees) in a solution. Similarly, the vertex set uses indices $V = \{0, ..., n - 1\}$, where the set of adjacent edges to the vertex $v \in V$ is denoted by $\delta(v)$, and the set of edges induced by a set $S \subseteq V$ is defined as $E(S) = \{\{u, v\} \in E \mid u, v \in S\}$. The binary variables used in their formulation are defined as follows:

y_v^c: 1, if and only if the vertex v belongs to the connected component c.
x_e^c: 1, if and only if the edge e belongs to the connected component c.
α_c: 1, if and only if c is a connected component in the solution.

And the following formulation:

$$\min \sum_{c \in I} \alpha_c \tag{4.1}$$

subject to:

$$\sum_{v \in V} y_v^c \leq (|L|+1)\alpha_c, \qquad c \in I \tag{4.2}$$

$$\sum_{v \in V} y_v^c \geq \alpha_c, \qquad c \in I \tag{4.3}$$

$$\sum_{c=0}^{\overline{c}-1} y_v^c = 1, \qquad v \in V \tag{4.4}$$

$$x_e^c \leq y_v^c, \qquad v \in V, e \in \delta(v), c \in I \tag{4.5}$$

$$\sum_{e \in E_l} x_e^c \leq \alpha_c, \qquad l \in L,\, c \in I \tag{4.6}$$

$$\sum_{c \in I} \sum_{e \in E(S)} x_e^c \leq |S| - 1, \qquad S \subset V,\, |S| \geq 2 \tag{4.7}$$

$$\sum_{e \in E} x_e^c = \sum_{v \in V} y_v^c - \alpha_c, \qquad c \in I \tag{4.8}$$

$$\alpha_{c+1} \leq \alpha_c, \qquad c \in \{0, \ldots, \overline{c}-2\} \tag{4.9}$$

$$y_1^1 = 1, \tag{4.10}$$

$$y_v^c \leq \sum_{w<v} y_w^{c-1}, \qquad v \in V \setminus \{0\},\, c \in \{2, \ldots, \overline{c}-1\} \tag{4.11}$$

$$y_v^c \in \{0,1\}, \qquad v \in V,\, c \in I \tag{4.12}$$

$$x_e^c \in \{0,1\}, \qquad e \in E,\, c \in I \tag{4.13}$$

$$\alpha_c \in \{0,1\}, \qquad c \in I \tag{4.14}$$

where $E_l = \{e \in E \mid f(e) = l\}$ defines the set of all edges associated with a color $l \in L$. As we mention in the previous chapter, $f : E \to L$ denotes a color function which assigns a color to each edge of E from the set of colors L.

The objective function (4.1) aims to minimize the number of rainbow trees in the forest. Constraints (4.2) express that the number of vertices belonging to a connected component, that is a tree, cannot be greater than the number of colors plus one, otherwise there will be repeated colours in the tree. Constraints (4.3) ensure that a connected component must have at least one vertex. Constraints (4.4) guarantee that a vertex will belong exactly to one tree in the forest. Constraints (4.5) impose that if a vertex does not belong to the connected component c, none of its adjacent edges can belong to that component.

Furthermore, constraints (4.6) impose that in each rainbow tree each $l \in L$ appears at most once. Constraints (4.7) are traditional subtour elimination inequalities while constraints (4.8) ensure a basic condition for a component to be a tree. Constraints (4.9)

to (4.11) are not necessary for the model, but their inclusion helps to eliminate symmetric solutions. Constraints (4.9) ensure that component c can be used only if component $c-1$ has been already assigned to a rainbow tree, while constraint (4.10) breaks symmetry by fixing a vertex in the first tree. Finally, constraints (4.11) impose that a vertex v can only belong to a connected component with index c if at least one vertex w, where $w < v$, belongs to the component with index $c-1$.

The authors in [7] also included the following constraints in order to strengthen the formulation:

$$y_v^c \leq \alpha_c, \qquad v \in V,\, c \in I \tag{4.15}$$

$$\sum_{e \in \delta_l(v)} x_e^c \leq y_v^c, \qquad v \in V,\, c \in I,\, l \in L \tag{4.16}$$

$$\sum_{c=0}^{\overline{c}-1} \left\{ x_e^c + \sum_{h \in \delta_l(u) \cup \delta_l(v)} x_h^c \right\} \leq 2, \qquad e = \{u, v\} \in E, l \in L \setminus \{f(e)\} \tag{4.17}$$

where $\delta_l(v) = \{e \in E_l \mid e = \{u, v\}\}$ defines the set of all edges of color l with one endpoint v.

Constraints (4.15) state that if a vertex belongs to a component c, then the variable representing that component must be used. Constraints (4.16) indicate that if a vertex belongs to a component, then at most one adjacent edge of each color will be used. Moreover constraints (4.17) ensure that an edge cannot have two adjacent edges with the same color.

4.1.1 A modified formulation for the RSF problem

The formulation presented by Carrabs et al. [7] is directly influenced by the maximum number of connected components $\overline{c}$. Actually, one can notice that the number of variables and constraints is proportional to this upper bound. Thus, in order to deal with this problem, we propose a slight modification in the previous formulation.

In our approach, we distinguish non-trivial trees, which have more than one vertex, from trivial trees that have only one vertex. In this scenario, $\overline{c}$ represents the maximum number of non-trivial trees in a solution. As a non-trivial tree must have at least two vertices, then the maximum number of non-trivial trees in a solution is set to $\overline{c} = \lfloor \frac{n}{2} \rfloor$. The following binary variables are defined:

y_v^c: 1, if and only if the vertex v belongs to the non-trivial tree c.
x_e^c: 1, if and only if the edge e belongs to the non-trivial tree c.
α_c: 1, if and only if c is a non-trivial tree in the solution.
ϕ_v: 1, if and only if the vertex v is isolated (trivial rainbow tree) in the solution.

and we propose the following reformulation:

$$\min \sum_{c \in I} \alpha_c + \sum_{v \in V} \phi_v \tag{4.18}$$

subject to:

$$(4.2), (4.6) - (4.9)$$

$$(4.11) - (4.14), (4.16)$$

$$\sum_{v \in V} y_v^c \geq 2\alpha_c, \qquad c \in I \tag{4.19}$$

$$\sum_{c=0}^{\overline{c}-1} y_v^c = 1 - \phi_v, \qquad v \in V \tag{4.20}$$

$$2x_e^c \leq y_u^c + y_v^c, \qquad e = \{u, v\} \in E, c \in I \tag{4.21}$$

$$\sum_{c=v+1}^{\overline{c}} y_v^c = 0, \qquad v \in V : v < \overline{c} \tag{4.22}$$

$$\alpha_0 = 1, \tag{4.23}$$

$$\phi_v \in \{0, 1\}, \qquad v \in V \tag{4.24}$$

Similar to the formulation presented by [36], our formulation analyses trivial and non-trivial components. The objective function aims to minimize the sum of the number of isolated vertices (trivial trees) and the number of non-trivial trees. Constraints (4.19) indicate that a non-trivial rainbow tree needs at least two vertices. Constraints (4.20) enforce that a vertex is an isolated vertex or belongs to exactly one non-trivial tree.

Constraints (4.21) replace constraints (4.5) to force that if an edge belongs to a connected component, then its extreme vertices also belong to the same component. Constraints (4.22) were proposed by [47] to solve the Rainbow Cycle Cover Problem and are used to reduce symmetry. These constraints impose that a vertex v, with index lower than $\overline{c}$, cannot belong to a non-trivial tree with index c, such that $c > v$. Finally, constraint (4.23) indicates that any edge-colored graph $G = (V, E, L)$, with $E \neq \emptyset$, has at least one non-trivial tree.

4.2 Heuristic approach

A greedy randomized adaptive search procedure (GRASP) [41] is a multi-start process, in which each GRASP iteration consists of two phases: a construction phase and a local search phase. In the construction phase, the method combines a greedy algorithm with a random component. Greedy algorithms are iterative procedures and, at each iteration, include in the solution the element that brings the best possible increment in its quality. However, this procedure does not always obtain an optimal solution. Therefore, a random component is included in the construction procedure to bring diversification into the solutions created, which can lead to good results. Once a solution is generated, a neighborhood is traversed in search of better solutions (local search phase). This process is repeated until it reaches the stop criterion. The best solution found among all iterations is returned as result.

We propose a GRASP heuristic aiming to provide quickly good quality solutions. Procedures for generating feasible solutions and local search are explained in the next sections.

4.2.1 Greedy randomized construction

Before proceeding with the method in detail, it will be convenient to introduce some additional notation. Let $G = (V, E, L)$ be an edge-colored graph. Given a forest $F = \{T_1, T_2, ..., T_k\}$, where each $T_i = (V_{T_i}, E_{T_i})$, $V_{T_i} \subseteq V$, $E_{T_i} \subseteq E$, is a non-trivial rainbow tree, we denote as $\overline{V} = \{v \in V \,|\, v \notin V_{T_1} \cup ...V_{T_k}\}$ the set of vertices that do not belong to any non-trivial tree. Also, for a vertex $v \in V$ and a set $S \subseteq V$ we define the set $\delta^S(v) = \{u \in S \,|\, \{u, v\} \in E\}$ as the set of adjacent vertices of v that belong to S and $c^S(v)$ as the number of different colors adjacent to v related to the neighbors in S.

Algorithm 4.1 shows the procedure used to generate a greedy randomized initial solution. In each step of the outer-loop (lines 5 to 19), the algorithm selects an isolated vertex from $\overline{V}$, and later tries to create a rainbow tree from this vertex (inner-loop between lines 12 to 17). In order to build this rainbow tree, we define a vertex weight function $w^S(v) = n \cdot c^S(v) + |\delta^S(v)|$ that establishes an order of candidate vertices $S \subseteq \overline{V}$ to be considered. Note that vertices with small weights have few neighbors in S and are connected by few different colors in this set. Thus, these vertices should be considered first to be in the rainbow tree. On the other hand, vertices with big weight values have a larger neighborhood in S, with many adjacent edges with different colors, and are more

Algorithm 4.1: GREEDY RANDOMIZED CONSTRUCTION

Input: The edge-colored graph $G = (V, E, L)$.
Output: A rainbow spanning forest F.

1 $it \leftarrow 1$
2 $F \leftarrow \emptyset$
3 $\overline{V} \leftarrow V$
4 **while** $\overline{V} \neq \emptyset$ **do**
5 $\quad w_{max}, w_{min} \leftarrow max_{v \in \overline{V}}\{w^{\overline{V}}(v)\},\ min_{v \in \overline{V}}\{w^{\overline{V}}(v)\}$
6 $\quad RCL^{\overline{V}} \leftarrow \{v \in \overline{V} \mid w^{\overline{V}}(v) \leq w_{min} + (w_{max} - w_{min}) \cdot \beta\}$
7 $\quad v \leftarrow$ random element from $RCL^{\overline{V}}$
8 $\quad V_{T_{it}} \leftarrow \{v\}$
9 $\quad E_{T_{it}} \leftarrow \emptyset$
10 $\quad \overline{V} \leftarrow \overline{V} \setminus \{v\}$
11 $\quad \overline{U} \leftarrow \delta^{\overline{V}}(v)$
12 $\quad$ **while** $\overline{U} \neq \emptyset$ **do**
13 $\quad\quad w_{max}, w_{min} \leftarrow max_{u \in \overline{U}}\{w^{\overline{V}}(u)\},\ min_{u \in \overline{U}}\{w^{\overline{V}}(u)\}$
14 $\quad\quad RLC^{\overline{U}} \leftarrow \{u \in \overline{U} \mid w^{\overline{V}}(u) \leq w_{min} + (w_{max} - w_{min}) \cdot \beta\}$
15 $\quad\quad u \leftarrow$ random element from $RLC^{\overline{U}}$
16 $\quad\quad V_{T_{it}} \leftarrow V_{T_{it}} \cup \{u\}$ and $E_{T_{it}} \leftarrow E_{T_{it}} \cup \{e_u\}$
17 $\quad\quad \overline{V} \leftarrow \overline{V} \setminus \{u\}$
18 $\quad\quad \overline{U} \leftarrow \{u \in \overline{V} \mid \{u\} \cup V_{T_{it}}$ and $\{e_u\} \cup E_{T_{it}}$ is a rainbow tree$\}$
19 $\quad F \leftarrow F \cup \{T_{it}\}$
20 $\quad it \leftarrow it + 1$
21 **return** F

easily handled.

The method starts by randomly selecting a initial vertex from a Restricted Candidate List $RCL^{\overline{V}}$ (line 7). We add to $RCL^{\overline{V}}$ those vertex whose weight $w^{\overline{V}}$ is lesser than or equal to a threshold value $w_{min} + (w_{max} - w_{min}) \cdot \beta$, where w_{min} and w_{max} are the minimum and maximum values of $w^{\overline{V}}$. The value of $\beta \in [0, 1]$ is used to control the size of the $RCL^{\overline{V}}$. When we choose small values of β the list $RCL^{\overline{V}}$ contains only vertices with the smallest weight values, i.e. vertices with few colors incident and low degree. Moreover, for larger values of β the cardinality of $RCL^{\overline{V}}$ will grow.

The initial vertex v is the starting point of a new rainbow tree T_{it}. At each iteration of the inner-loop (lines 12 to 17), the method tries to expand T_{it} by inserting a randomly selected vertex $u \in RCL^{\overline{U}}$. The Restricted Candidate List $RCL^{\overline{U}}$ is formed in the same fashion as $RCL^{\overline{V}}$, but now considers only vertices in the subset $\overline{U} \subseteq \overline{V}$ that can be used to expand T_{it} using vertex u and some edge e_u connecting u and the rainbow tree under construction, i.e., $T_{it} = (V_{T_{it}} \cup \{u\}, E_{T_{it}} \cup \{e_u\})$. It is important to note that the weight

is always calculated considering the vertices that do not belong to any tree already built ($\overline{V}$). The inner-loop is repeated until $\overline{U} = \emptyset$. At this point, the (maximal) rainbow tree T_{it} can not be enlarged and it is finally added to the forest F (line 18). The construction phase halts when there is no more isolated vertices left in $\overline{V}$.

As an example, consider the graph $G = (V, E, L)$ with $n = 5$ vertices in Figure 4.1. In this graph, the initial weights of the vertices appear in parentheses. The value of the evaluation function of vertex 2, for example, is $w^V(2) = 5 \cdot c^V(2) + \delta^V(2) = 5 \cdot 2 + 4 = 14$. Making good choices in each stage of the construction phase may determine the quality of the constructed solution. Figure 4.2 shows a forest with 3 trees as result of choosing the vertex with the highest value of the evaluation function in each step. The sequence of vertices for the first rainbow tree is $(2 \rightarrow 3 \rightarrow 1)$, while the second and third rainbow trees are composed by vertex 4 and 5, respectively.

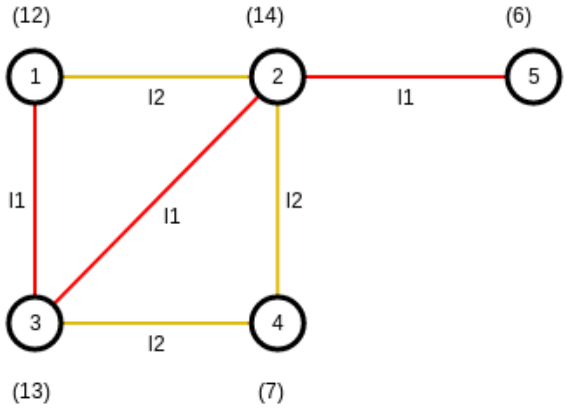


Figure 4.1: A graph with $n = 5$ vertices

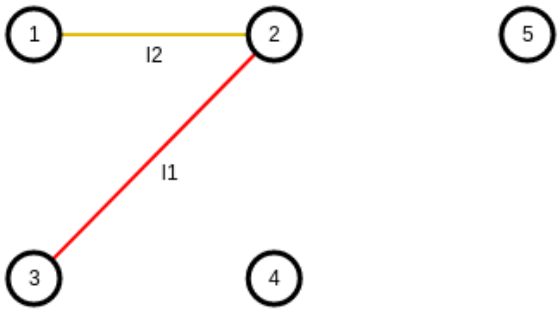


Figure 4.2: A rainbow forest with three rainbow trees

Figure 4.3 shows a forest with 2 trees, resulting of choosing the vertex with smaller value of the evaluation function in each step. In this case, the sequence of vertices for the first rainbow tree is $(5 \rightarrow 2 \rightarrow 4)$, and the second rainbow tree was composed using the sequence $(1 \rightarrow 3)$.

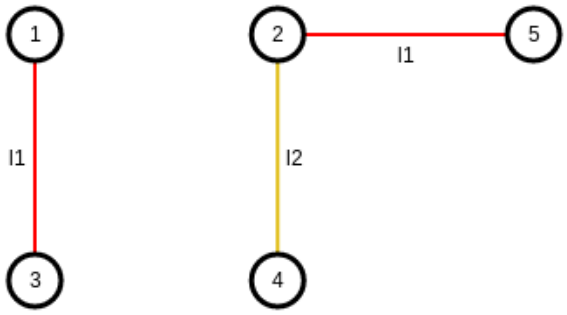


Figure 4.3: A rainbow forest with two rainbow trees

It is clear that depending on the value chosen for $\beta \in [0, 1]$, the impact on the diversity of the solutions will be different. If the value of β is close to 1, the algorithm would choose vertices from a larger list, which could deteriorate the quality of the constructed solutions and slow down the local search process. On the other hand, if a small value is chosen for β, then this may be too restrictive and decrease the diversity of the solutions, making the algorithm behave much like a greedy algorithm. Empirical tests were conducted on some of the instances with 30 and 40 vertices described in Section 4.3 to evaluate values of $\beta \in \{0.1, 0.2, ..., 0.8, 0.9\}$. The best quality results were obtained using $\beta = 0.2$ and $\beta = 0.3$ (slightly better with 0.3). We observe that any further increase beyond this value, the heuristic takes longer to converge to the same solution. For this reason, for the computational experiments, we used $\beta = 0.3$.

4.2.2 Local search

While at the expense of some computing time, the solution generated at the construction method can usually be improved by some greedy local search. Given a forest $F = \{T_1, T_2, ..., T_k\}$, three neighborhoods are defined by considering three movements. When two rainbow trees $T_i = (V_{T_i}, E_{T_i})$ and $T_j = (V_{T_j}, E_{T_j})$ do not have any color in common, we explore the Leaf_Merge and Edge_Merge movements. Otherwise, if T_i and T_j have exactly one common color, we only analyze the 2Edge_Merge movement. The

three neighborhoods are described bellow:

Leaf_Merge: This neighborhood considers all solutions that can be reached from merging trees T_i and T_j using a leaf vertex from a tree T_k, where $i \neq k \neq j$. More specifically, we look for a leaf vertex v of some tree $T_k = (V_{T_k}, E_{T_k})$, and vertices $u_i \in V_{T_i}$ and $u_j \in V_{T_j}$, where $\{v, u_i\}$ and $\{v, u_j\}$ are edges of the graph. Then, if the colors of $\{v, u_i\}$ and $\{v, u_j\}$ are not present in $E_{T_i} \cup E_{T_j}$, a new combined rainbow tree T_{ij} is created by removing vertex v from T_k and connecting trees T_i and T_j using the edges $\{v, u_i\}$ and $\{v, u_j\}$. Figure 4.4 shows a forest composed of three rainbow trees with vertices $V_{T_i} = \{6, 7\}$, $V_{T_j} = \{8, 9, 10\}$ and $V_{T_k} = \{1, 2, 3, 4, 5\}$. Dashed lines represent edges that do not belong to the forest but belong to the set of edges of the graph. Node 2 may be removed from T_k and the trees T_i and T_j may be combined using the edges $\{2, 7\}$ and $\{2, 10\}$. The combined tree is pictured in Figure 4.5.

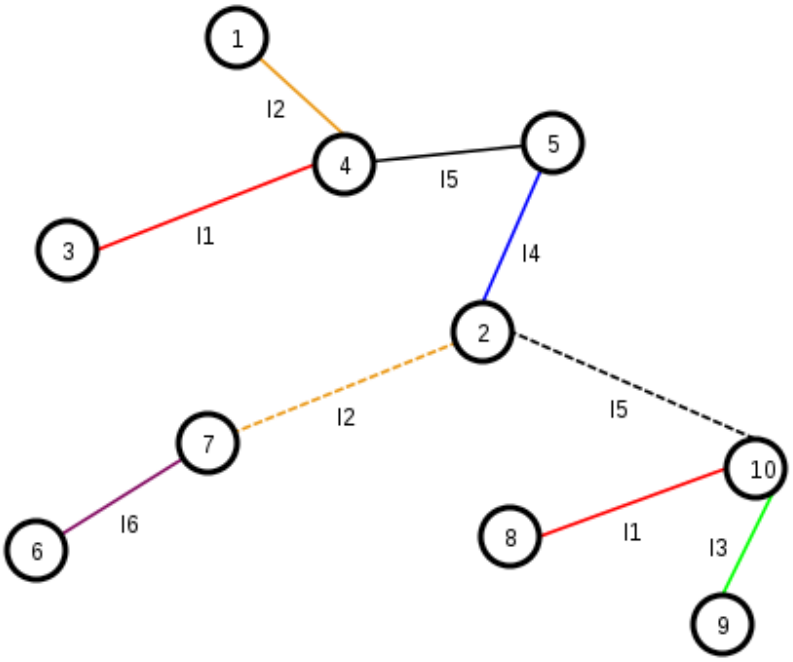


Figure 4.4: A rainbow forest with three rainbow trees

Edge_Merge: Similar to Leaf_Merge, this neighborhood considers all solutions that can be reached from merging trees T_i and T_j by directly connecting them. We first look for an edge $e_{ij} = \{u_i, u_j\}$ in G between the trees T_i and T_j, such that $u_i \in V_{T_i}$, $u_j \in V_{T_j}$, and create the combined tree T_{ij}. Note that the color of e_{ij} is already represented in $E_{T_i} \cup E_{T_j}$ by another edge, denoted $\overline{e}$ (or else T_i or T_j could be enlarged during the construction phase). Thus, in order to reach a feasible rainbow structure T_{ij}, we try to swap edge $\overline{e} \in E_{T_i} \cup E_{T_j}$ for another one $\overline{e'}$ with a color not present in the combined tree, such that, if $\overline{e'}$ is added then one of the trees (T_i or T_j) would have a cycle containing both $\overline{e}$ and $\overline{e'}$.

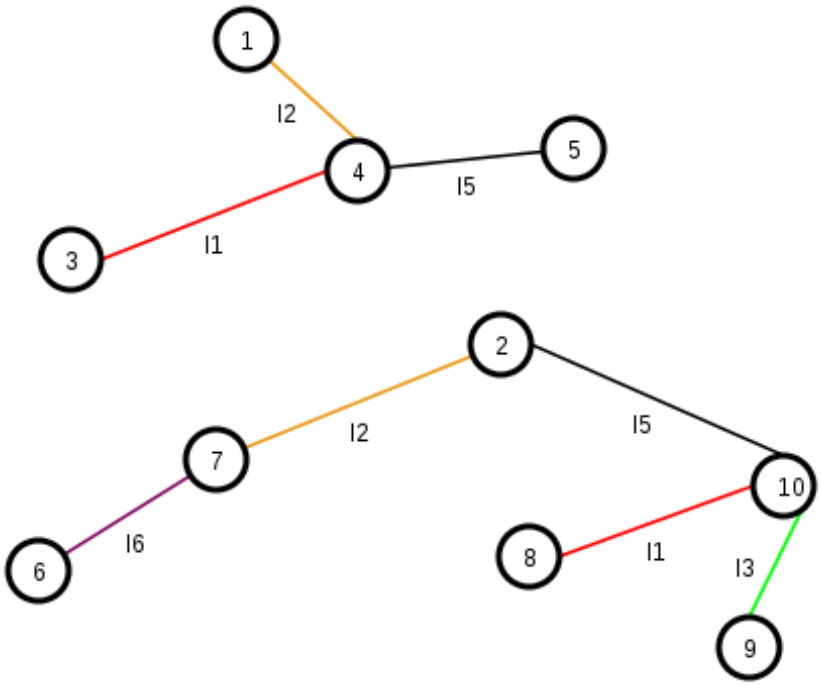


Figure 4.5: A rainbow forest with two rainbow trees after Leaf_Merge

Figure 4.6 shows a forest composed of two rainbow trees with vertices $V_{T_i} = \{1, 3, 4\}$ and $V_{T_j} = \{2, 5, 6, 7\}$. The edge $\{4, 7\}$ connects T_i and T_j. In order to combine the trees, edge $\{5, 7\}$, that has the same color of edge $\{4, 7\}$, is swapped by edge $\{2, 7\}$, which has a color not present in the solution and which, if added, creates a cycle in the tree T_j containing both $\{2, 7\}$ and $\{5, 7\}$. The combined tree is shown in Figure 4.7

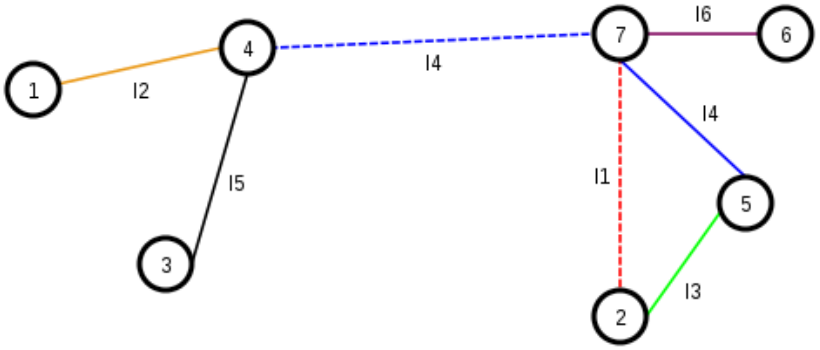


Figure 4.6: A rainbow forest with two rainbow trees

2Edge_Merge: Let $\overline{e}_i \in E_i$ and $\overline{e}_j \in E_j$ be the edges with the same color in T_i and T_j, respectively. The 2Edge_Merge neighborhood considers all solutions that can be reached from the current one by combining trees T_i and T_j using two different edges $e^1_{ij} = \{u^1_i, u^1_j\}$ and $e^2_{ij} = \{u^2_i, u^2_j\}$, such that $u^1_i, u^2_i \in V_{T_i}$ and $u^1_j, u^2_j \in V_{T_j}$. We specifically look for edges e^1_{ij} and e^2_{ij}, with colors not present in $E_{T_i} \cup E_{T_j}$, that will form a cycle with

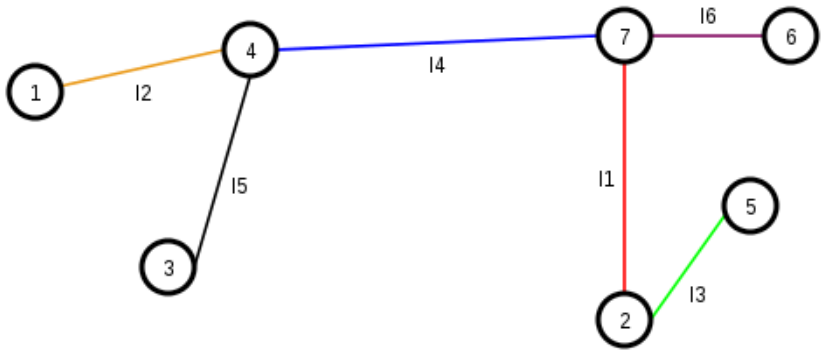


Figure 4.7: A rainbow forest with one rainbow tree after Edge_Merge

one of the edges with the same color ($\overline{e}_i$ or $\overline{e}_j$). In this way, the combined rainbow tree T_{ij} is created by removing the cycle edge $\overline{e}_i$ (or $\overline{e}_j$), and connecting T_i and T_j through edges e^1_{ij} and e^2_{ij}. Figure 4.8 shows a forest composed of two rainbow trees with vertices $V_{T_i} = \{3, 5, 7\}$, $V_{T_j} = \{1, 2, 4, 6\}$ and edges $\{5, 7\}$ and $\{2, 4\}$ with the same color. If we connect T_i and T_j through edges $\{1, 3\}$ and $\{2, 7\}$, a cycle is created which contains the edges $\{1, 3\}$, $\{3, 7\}$, $\{2, 7\}$, $\{2, 4\}$ and $\{1, 4\}$. Next, edge $\{2, 4\}$ is removed and the trees T_i and T_j are combined using the edges $\{1, 3\}$ and $\{2, 7\}$ as shown in Figure 4.9.

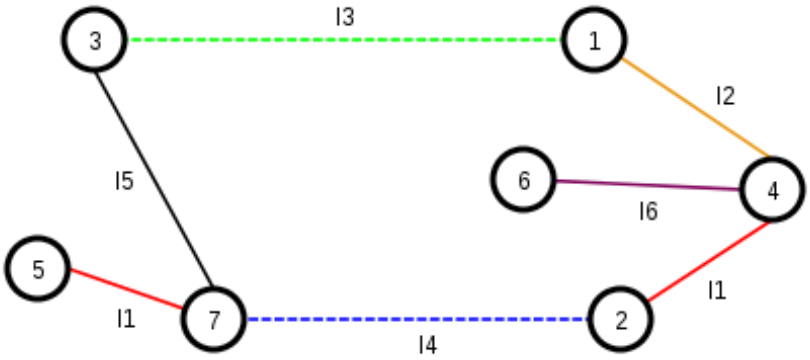


Figure 4.8: A rainbow forest with two rainbow trees

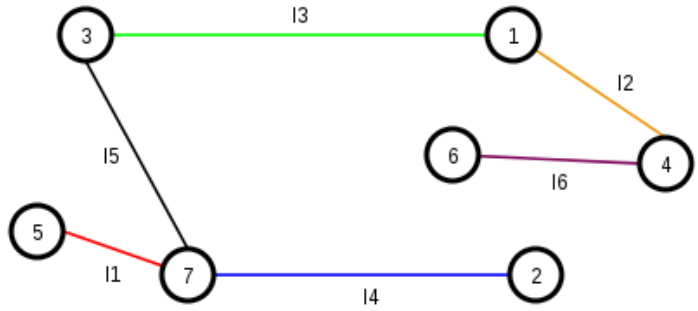


Figure 4.9: A rainbow forest with one rainbow tree after 2Edge_Merge

The local search phase of the proposed method is performed with a *Variable Neighbor-*

hood Descent (VND) heuristic [18]. The method sequentially explores the family of neighborhoods $\mathcal{N}$ in the following increasing-complexity order: Leaf_Merge ($\mathcal{N}_1$), Edge_Merge ($\mathcal{N}_2$) and 2Edge_Merge ($\mathcal{N}_3$). We denote as $N_k(F)$ the set of forests (solutions) that are neighbors of F on neighborhood N_k. Algorithm 4.2 illustrate the VND with a best improvement strategy in which the explored solution F' replaces F if $|F'| < |F|$ (line 4). In this case, the search is reinitialized from the first neighborhood ($\mathcal{N}_1$ =Leaf_Merge). On the other hand, if $|F'| = |F|$, then the neighborhood index is increased (line 8). The local search halts when all three neighborhoods have been explored without improving the current solution F.

Algorithm 4.2: VND

Input: Forest F and family of neighborhoods $\mathcal{N}_1$=Leaf_Merge, $\mathcal{N}_2$ =Edge_Merge and $\mathcal{N}_3$ =2Edge_Merge.
Output: Forest F after exploring the movements.

```
1 k ← 1
2 repeat
3 |   F' ← best neighbor of F in N_k(F)
4 |   if |F'| < |F| then
5 |   |   F ← F'
6 |   |   k ← 1
7 |   else
8 |   |   k ← k + 1
9 until k > 3
```

4.3 Computational experiments

In this section, the results obtained by the proposed modified mathematical formulation and the GRASP metaheuristic are presented (all results are available online [3]). We analyze the same set of instances used in [7]. These instances are divided into two groups: small scenarios and large scenarios. Small scenarios have instances that correspond to graphs with number of vertices ranging from 20 to 50, and densities varying between 0.1, 0.2 and 0.3. The number of colors is always smaller than the number of vertices and varies between 3 and 18. On the other hand, large scenarios have instances that correspond to graphs with number of vertices ranging from 100 to 400, and densities varying between 0.1, 0.2 and 0.3. The number of colors varies between 5 and 30.

The computational experiments in [7] were carried out on a Intel Xeon X5675 processor

[3] `www.ic.uff.br/~yuri/files/SFR.zip`

running at 3.07 GHz with 96 GB of RAM using IBM ILOG CPLEX 12.5. In contrast, our tests were executed on an Intel (R) Core i5-4460S CPU @ 2.90GHz, with 8 GB of RAM using a 64-bit Linux and all methods were programmed in C++ language. The IBM ILOG CPLEX 12.6 was used with a single thread of execution and 10800 seconds as time limit. All other CPLEX parameters were left to their default values. Using the data provided by the cross-platform processor benchmark Geekbench 3 [39], the performance ratio between their computer and ours is 1.033, so we consider them as similar machines.

4.3.1 Results of the heuristic

We compare in this section the performances of the GRASP method and the multi-start greedy heuristic presented in [7], denoted MS. Tables 4.1 and 4.2 present the results obtained for small and large scenarios respectively. The number of iterations of the GRASP method has been fixed at 100 for each instance and each one was executed 10 times. In these tables, each line presents average statistics over five instances. The first four columns report the group identification (**ID**), number of vertices (**n**), number of edges (**m**) and number of colors (**l**) respectively. Columns **MS**, **time** and **gap** present, for the multi-start heuristic proposed in [7], the average number of rainbow trees found, the average execution time in seconds and the gap between the results obtained by exact algorithms and the results obtained by the heuristic, respectively. The next three columns display the same information as the previous three columns, but for the proposed GRASP heuristic. Moreover, if a solution (or time) is strictly better than the previous bound (or time), it is represented in boldface.

In small scenarios (Table 4.1), the GRASP procedure was able to find solutions strictly better than those obtained by MS in 55.6% of the groups of instances, equal solutions for 33.3% of the groups and worse solutions in 11.1% of the groups of instances. Using a similar computing power (as it was shown at the beginning of this section), these results show that the GRASP heuristic was able to find good quality results faster than the MS heuristic (89.5% faster times).

A similar behavior was presented by the GRASP for large scenarios (Table 4.2). In this table we denote as "—" the gap for graphs with more than 100 vertices, because there are no results found by exact methods. In large scenarios, the GRASP procedure was able to find solutions strictly better than those obtained by MS in 83.3% of the groups of instances and worse solutions for 13.9%. This behavior is especially clear for the groups of instances with 300 and 400 vertices, which our method achieved better bounds in 17

ID	n	m	l	MS	time	gap	GRASP	time	gap
1	20	39	3	7.0	0.10	0.4	**6.6**	**0.01**	**0.0**
2			6	4.8	0.10	0.6	4.8	**0.01**	0.6
3			11	2.8	0.13	0.6	**2.6**	**0.01**	**0.4**
4	20	58	3	6.4	0.13	0.8	**5.8**	**0.01**	**0.2**
5			6	3.6	0.15	0.4	**3.4**	**0.01**	**0.2**
6			12	2.0	0.05	0.0	2.0	**0.01**	0.0
7	20	77	4	4.4	0.11	0.4	**4.2**	**0.00**	**0.2**
8			7	3.0	0.07	0.0	3.0	**0.01**	0.0
9			13	2.0	0.06	0.0	2.0	**0.01**	0.0
10	30	74	4	8.8	0.20	0.4	**8.6**	**0.02**	**0.2**
11			7	**5.4**	0.25	**0.4**	5.8	**0.02**	0.8
12			13	3.4	0.16	0.4	3.4	**0.03**	0.4
13	30	117	4	7.4	0.33	0.6	**7.0**	**0.02**	**0.0**
14			7	4.4	0.27	0.2	**4.2**	**0.02**	**0.0**
15			14	3.0	0.49	0.4	**2.8**	**0.02**	**0.2**
16	30	161	4	6.4	0.32	0.0	6.4	**0.01**	0.0
17			8	4.0	0.12	0.0	4.0	**0.02**	0.0
18			15	2.2	0.39	0.2	**2.0**	**0.03**	**0.0**
19	40	118	4	12.0	0.29	1.8	**11.0**	**0.04**	**0.8**
20			7	7.4	0.41	1.0	**7.2**	**0.04**	**0.8**
21			14	**4.0**	0.51	**0.8**	4.6	**0.05**	1.4
22	40	196	4	9.8	0.48	1.0	**9.0**	**0.03**	**0.2**
23			8	5.2	0.43	0.2	5.2	**0.04**	0.2
24			16	3.2	0.33	0.2	**3.0**	**0.06**	**0.0**
25	40	274	5	7.0	0.22	0.0	7.0	**0.03**	0.0
26			9	5.0	1.00	1.0	**4.2**	**0.01**	**0.2**
27			17	3.0	0.21	0.0	3.0	**0.06**	0.0
28	50	173	4	14.6	0.35	2.2	**13.6**	**0.07**	**1.2**
29			8	**9.2**	0.46	**1.2**	9.8	**0.08**	1.8
30			15	**5.4**	0.54	**1.0**	5.6	**0.10**	1.2
31	50	295	5	9.8	0.55	0.8	**9.4**	**0.06**	**0.4**
32			9	6.0	0.84	1.0	6.0	**0.06**	1.0
33			17	4.0	1.11	1.0	**3.0**	**0.09**	**0.0**
34	50	418	5	9.0	0.59	0.0	9.0	**0.05**	0.0
35			9	6.0	1.19	1.0	**5.4**	**0.04**	**0.4**
36			18	3.2	0.67	0.2	**3.0**	**0.11**	**0.0**
Avg.				5.7	0.38	0.6	**5.5**	**0.04**	**0.4**

Table 4.1: Heuristic results on small scenarios

ID	n	m	l	MS	time	gap	GRASP	time	gap
37	100	595	5	24.80	0.73	4.80	**24.60**	**0.62**	**4.60**
38			10	**11.80**	1.15	**1.80**	14.00	**0.58**	4.00
39			19	**7.20**	1.45	**1.80**	9.00	**0.94**	4.60
40	100	1090	6	15.60	1.52	0.60	**15.00**	**0.26**	**0.00**
41			11	10.20	2.00	1.20	**9.40**	**0.42**	**0.40**
42			21	6.20	2.64	1.20	**5.00**	**0.58**	**0.00**
43	100	1585	6	15.20	1.27	0.20	**15.00**	**0.30**	0.00
44			11	9.20	2.11	0.20	**9.00**	**0.37**	**0.00**
45			21	5.60	4.50	0.60	**5.00**	**0.65**	**0.00**
46	200	2190	6	**43.60**	**3.47**	—	44.00	8.49	—
47			12	**18.60**	**5.05**	—	21.80	7.24	—
48			23	13.20	**7.62**	—	13.20	7.63	—
49	200	4180	7	27.00	12.63	—	**25.20**	**0.86**	—
50			13	16.60	16.96	—	**15.00**	**2.17**	—
51			25	10.40	23.29	—	**8.00**	**3.43**	—
52	200	6170	7	26.20	25.26	—	**25.00**	**0.06**	—
53			13	15.20	25.50	—	**15.00**	**2.73**	—
54			26	9.00	45.37	—	**8.00**	**5.05**	—
55	300	4785	7	54.80	**12.32**	—	**51.00**	32.02	—
56			13	26.40	**19.26**	—	**25.20**	23.55	—
57			25	18.00	28.99	—	**15.20**	**26.98**	—
58	300	9270	7	41.80	50.44	—	**38.00**	**4.17**	—
59			14	22.00	82.13	—	**20.00**	**0.30**	—
60			27	14.20	96.14	—	**11.00**	**9.54**	—
61	300	13755	7	39.20	125.62	—	**38.00**	**6.93**	—
62			14	21.00	203.72	—	**20.00**	**0.33**	—
63			28	11.80	177.91	—	**11.00**	**17.86**	—
64	400	8380	7	79.20	**43.33**	—	**75.20**	156.11	—
65			14	31.80	69.59	—	**27.00**	**22.15**	—
66			27	22.80	97.05	—	**15.60**	**52.62**	—
67	400	16360	7	60.60	176.52	—	**50.00**	**0.23**	—
68			14	28.40	279.07	—	**24.40**	**17.60**	—
69			28	17.00	306.68	—	**14.00**	**24.26**	—
70	400	24340	8	45.20	247.19	—	**45.00**	**17.26**	—
71			15	26.00	857.59	—	**25.00**	**0.65**	—
72			30	**14.40**	718.85	—	19.40	**36.50**	—
Avg.				23.89	104.86	—	**22.39**	**13.65**	—

Table 4.2: Heuristic results on large scenarios

(out of 18) groups. Note that the MS heuristic presented a rapid increase in time as the size of the instances increases, while the computational time of the GRASP heuristic does not increase so fast as can be seen in Table 4.2. This leads to a GRASP method 87% faster than the MS heuristic to deal with large instances.

4.3.2 Results of the exact method

As discussed in [7], a good upper bound for the maximum number of connected components in a rainbow forest would help to decrease the number of variables and constraints used in the formulation. The results obtained by the heuristics could be used as an upper bound for the number of rainbow trees in a forest, and consequently, for the number of non-trivial rainbow trees.

Experiments were developed to compare the computational results obtained by our formulation (ILP^M) and our bounded formulation (denoted as ILP^B). This last formulation is based on ILP^M and uses, as an upper bound for the number of non-trivial rainbow trees in the forest, the minimum between the theoretical bound $\lfloor \frac{n}{2} \rfloor$ and the value obtained by the GRASP heuristic, which may be higher than the theoretical bound depending on the characteristics of the graph.

Table 4.3 shows the computational results of the proposed formulations ILP^M and ILP^B for small scenarios. The first four columns give the same information as described in Tables 4.1 and 4.2. Columns 5 and 6 present, respectively, the average number of rainbow trees found and average execution time in seconds obtained by the formulation reported in [7], denoted in this work as ILP. The numbers in subscript indicate the number of instances not solved to optimality within 10800 seconds. Similarly, columns 7 and 8, and columns 9 and 10 present the same information as columns 5 and 6, for the modified and bounded formulations, respectively. Again, if a solution (or time) is strictly better than the previous bound (or time), it is represented in boldface.

The results show that the formulation ILP^M was able to solve 8 more instances than the formulation presented in [7]. Moreover, ILP^M obtained faster average execution times than ILP (621.92 and 1332.29 seconds, respectively) for the small scenario instances. The best results were obtained by the ILP^B formulation which uses the results of the GRASP heuristic as bounds for the number of non-trivial trees. It was able to solve 11 more instances than ILP, and 3 more instances than ILP^M. Furthermore, the execution times to solve the instances were faster than the ones from formulations ILP^M, except for the groups of instances with ID 13, 28 and 36. In these groups, the formulation ILP^M had a

ID	n	m	l	ILP[7]	time	ILP^M	time	ILP^B	time
1	20	39	3	6.6	2.94	6.6	0.27	6.6	**0.26**
2			6	4.2	3.74	4.2	0.60	4.2	**0.33**
3			11	2.2	1.1	2.2	0.28	2.2	**0.10**
4	20	58	3	5.6	3.9	5.6	0.46	5.6	**0.22**
5			6	3.2	2.14	3.2	0.54	3.2	**0.07**
6			12	2.0	1.21	2.0	0.29	2.0	**0.05**
7	20	77	4	4.0	2.3	4.0	0.12	4.0	**0.08**
8			7	3.0	1.63	3.0	0.42	3.0	**0.09**
9			13	2.0	1.63	2.0	0.33	2.0	**0.05**
10	30	74	4	8.4	129.42	8.4	18.53	8.4	**17.60**
11			7	5.0	204.63	5.0	4.50	5.0	**2.93**
12			13	3.0	8.02	3.0	2.00	3.0	**0.26**
13	30	117	4	$6.8^{(1)}$	2199.73	6.8	**15.34**	6.8	31.73
14			7	4.2	16.72	4.2	1.92	4.2	**0.29**
15			14	2.6	51.51	2.6	2.06	2.6	**0.21**
16	30	161	4	6.4	36.88	6.4	1.15	6.4	**0.41**
17			8	4.0	17.27	4.0	3.66	4.0	**0.25**
18			15	2.0	18.84	2.0	2.91	2.0	**0.17**
19	40	118	4	10.2	2283.43	10.2	1172.35	10.2	**459.09**
20			7	$6.4^{(2)}$	4711.31	$6.4^{(1)}$	2195.53	6.4	**974.10**
21			14	$3.4^{(2)}$	4348.36	**3.2**	127.95	**3.2**	**79.00**
22	40	196	4	8.8	2336.74	8.8	56.88	8.8	**29.56**
23			8	5.0	121.37	5.0	12.19	5.0	**1.07**
24			16	3.0	82.75	3.0	7.91	3.0	**0.49**
25	40	274	5	7.0	89.99	7.0	8.25	7.0	**1.28**
26			9	4.0	86.37	4.0	10.65	4.0	**0.95**
27			17	3.0	139.92	3.0	13.70	3.0	**1.18**
28	50	173	4	$12.6^{(4)}$	8994.51	$\mathbf{12.4^{(3)}}$	**7458.11**	$\mathbf{12.4^{(3)}}$	7471.59
29			8	$9^{(4)}$	10652.42	$\mathbf{8^{(2)}}$	4503.49	$\mathbf{8^{(1)}}$	**3228.25**
30			15	$4.8^{(3)}$	6536.58	$\mathbf{4.4^{(2)}}$	4365.42	$\mathbf{4.4^{(2)}}$	**4326.28**
31	50	295	5	9.0	618.57	9.0	23.19	9.0	**7.88**
32			9	$5.2^{(1)}$	2318.95	$5.2^{(1)}$	2195.44	**5.0**	**8.73**
33			17	3.0	1009.18	3.0	51.76	3.0	**2.21**
34	50	418	5	9.0	294.84	9.0	20.60	9.0	**5.38**
35			9	5.0	295.96	5.0	33.08	5.0	**3.80**
36			18	3.0	337.71	3.0	**77.09**	3.0	326.61
Avg.				$5.2^{(17)}$	1332.29	$5.1^{(9)}$	621.92	$\mathbf{5.1^{(6)}}$	**471.74**

Table 4.3: Exact results on small scenarios

ID	n	m	l	ILP^M	time	ILP^B	time
37	100	595	5	$20.20^{(5)}$	10800.00	$\mathbf{20.00}^{(5)}$	10800.00
38			10	10.00	2559.58	10.00	**664.32**
39			19	5.40	2595.84	5.40	**822.59**
40	100	1090	6	$15.40^{(1)}$	4063.68	**15.00**	**151.28**
41			11	9.00	2926.83	9.00	**472.79**
42			21	$9.80^{(4)}$	9585.06	**5.00**	**1021.46**
43	100	1585	6	15.00	3870.17	15.00	**546.14**
44			11	$26.20^{(1)}$	6749.52	**9.00**	**2028.07**
45			21	$43.40^{(4)}$	9152.80	$\mathbf{5.00}^{(1)}$	**2500.75**
Avg.				$17.16^{(15)}$	5811.50	$\mathbf{10.24}^{(6)}$	**2498.14**

Table 4.4: Exacts results on large scenarios with 100 vertices

slightly better performance, using less time than formulation ILP^B (3.56 % faster).

Table 4.4 shows the results obtained by models ILP^M and ILP^B on large scenarios for graphs with 100 vertices. For these instances, the formulation proposed in [7] was reportedly not able to solve any instance. In this set of instances, ILP^M and ILP^B were able to solve 30 and 39 instances respectively. The exact approach quickly becomes not exploitable for instances of dimension greater than 200. Formulations ILP and ILP^M were not able to solve any of these instances, however the bounded formulation ILP^B reached the optimal solution in instance Rand_200_4180_79_7.rnd (optimal value 25 and time 7594.35 seconds).

4.4 Conclusions

In this chapter, methods were developed for obtaining exact and heuristic solutions for the Rainbow Spanning Forest problem. A GRASP heuristic was developed which has obtained better results both in terms of quality and computational time than the MS heuristic presented in [7]. A modified formulation was also proposed by considering only the non-trivial connected components in the forest. The computational results show the effectiveness of this modified formulation, capable of solving 38 more instances (8 in small scenarios and 30 in large scenarios) than the formulation previously presented in [7] within the time limit of 3 hours. Moreover, this modification enables to improve the bounds for unsolved instances and to reach shorter computational times. All these results were also presented by us in [35].

Chapter 5

Conclusions

In this thesis, exact and heuristic methods were developed to obtain solutions to hard problems in graphs. We elaborated different strategies to decompose and preprocess instances of the *Minimum d-Branch Vertices* problem and the *Rainbow Cycle Cover* problem. The obtained results show that preprocessing the instances brings great benefits to the exact resolution of these problems. Prefixing variables and reducing the size of the analyzed instances allowed to reduce the computational time used to solve them. Also valid inequalities and cuts were created that improved the computational time to find solutions to the developed models.

We developed mathematical models, considering the particular characteristics of each problem, that improved the results obtained so far for these problems. We defined the concept of co-classes of a graph and we used it to reduce the size of the instances and to create cuts and valid inequalities for the models. The obtained experimental results showed that using the concept of co-classes positively impacted the resolution effectiveness of the models.

Different types of integration between heuristic and exact methods were implemented. We used the results of heuristics to be upper bounds for the exact method in the *Minimum d-Branch Vertices* problem, and to reduce the number of variables and restrictions of the model of the *Rainbow Spanning Forest* problem. Experimental results showed that using heuristics to reduce the number of variables and/or inequalities of the model had a positive impact. We have also proposed a hybrid method to address the *Rainbow Cycle Cover* problem that turned out to be effective for larger instances.

In general, our heuristics obtained very good quality solutions and used less computational time than heuristics of previous works. Moreover, our exact methods found

solutions and new bounds for several unsolved instances, spending less computational time than previous methods.

Our future research will be oriented to the development of other strategies for the resolution of models, such as polyhedral analysis and column generation method. Also, we will study problems on rainbow paths, as well as the task of developing a fully heuristic method to solve the *Rainbow Cycle Cover* problem.

References

[1] Akgün, İ.; Tansel, B. Ç. Min-degree constrained minimum spanning tree problem: New formulation via Miller–Tucker–Zemlin constraints. *Computers & Operations Research 37*, 1 (2010), 72–82.

[2] Alexeev, B. On lengths of rainbow cycles. *Journal of Combinatorics 13*, 4 (2006), R105.

[3] Bastos, L.; Ochi, L. S.; Protti, F.; Subramanian, A.; Martins, I. C.; Pinheiro, R. G. S. Efficient algorithms for cluster editing. *Journal of Combinatorial Optimization 31*, 1 (2016), 347–371.

[4] Blum, C.; Puchinger, J.; Raidl, G. R.; Roli, A. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing 11*, 6 (2011), 4135–4151.

[5] Boschetti, M. A.; Maniezzo, V. Combining exact methods and heuristics. *Wiley Encyclopedia of Operations Research and Management Science* (2011).

[6] Brualdi, R. A.; Hollingsworth, S. Multicolored forests in complete bipartite graphs. *Discrete Mathematics 240*, 1-3 (2001), 239–245.

[7] Carrabs, F.; Cerrone, C.; Cerulli, R.; Silvestri, S. The rainbow spanning forest problem. *Soft Computing* (2017), 1–12.

[8] Carrabs, F.; Cerrone, C.; Cerulli, R.; Silvestri, S. On the complexity of rainbow spanning forest problem. *Optimization Letters* (2018), 1–12.

[9] Carrabs, F.; Cerulli, R.; Gaudioso, M.; Gentili, M. Lower and upper bounds for the spanning tree with minimum branch vertices. *Computational Optimization and Applications 56*, 2 (2013), 405–438.

[10] Carraher, J. M.; Hartke, S. G.; Horn, P. Edge-disjoint rainbow spanning trees in complete graphs. *European Journal of Combinatorics 57* (2016), 71–84.

[11] Cerulli, R.; Gentili, M.; Iossa, A. Bounded-degree spanning tree problems: models and new algorithms. *Computational Optimization and Applications 42*, 3 (2009), 353–370.

[12] Coelho, V.; Grasas, A.; Ramalhinho, H.; Coelho, I.; Souza, M.; Cruz, R. An ILS-based algorithm to solve a large-scale real heterogeneous fleet VRP with multi-trips and docking constraints. *European Journal of Operational Research 250*, 2 (2016), 367–376.

[13] Dantzig, G. B.; Ramser, J. H. The truck dispatching problem. *Management science 6*, 1 (1959), 80–91.

[14] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische mathematik 1*, 1 (1959), 269–271.

[15] Dupin, N.; Talbi, E.-G. Matheuristics for the discrete unit commitment problem with min-stop ramping constraints. In *Matheuristics 2016 - Proceedings of the Sixth International Workshop on Model-based Metaheuristics* (Brussels, Belgium, 2016), pp. 72–83.

[16] Garey, M. R.; Johnson, D. S. Computers and intractability. a guide to the theory of NP-completeness. a series of books in the Mathematical Sciences, 1979.

[17] Gargano, L.; Hell, P.; Stacho, L.; Vaccaro, U. Spanning trees with bounded number of branch vertices. In *International Colloquium on Automata, Languages, and Programming* (Berlin, Heidelberg, 2002), Springer, pp. 355–365.

[18] Hansen, P.; Mladenovic, N. Variable neighbourhood search: Principles and applications. *European Journal of Operational Research 120* (2001), 449–467.

[19] Hassin, R.; Monnot, J.; Segev, D. Approximation algorithms and hardness results for labeled connectivity problems. *Journal of Combinatorial Optimization 14*, 4 (2007), 437–453.

[20] Kano, M.; Li, X. Monochromatic and heterochromatic subgraphs in edge-colored graphs-a survey. *Graphs and Combinatorics 24*, 4 (2008), 237–263.

[21] Krumke, S. O.; Wirth, H.-C. On the minimum label spanning tree problem. *Information Processing Letters 66*, 2 (1998), 81–85.

[22] Lai, X.; Zhou, Y.; He, J.; Zhang, J. Performance analysis of evolutionary algorithms for the minimum label spanning tree problem. *IEEE Transactions on Evolutionary Computation 18*, 6 (2014), 860–872.

[23] Land, A. H.; Doig, A. G. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society* (1960), 497–520.

[24] Landete, M.; Marín, A.; Sainz-Pardo, J. L. Decomposition methods based on articulation vertices for degree-dependent spanning tree problems. *Computational Optimization and Applications 68*, 3 (2017), 749–773.

[25] Leggieri, V.; Haouari, M.; Triki, C. The steiner tree problem with delays: A compact formulation and reduction procedures. *Discrete Applied Mathematics 164* (2014), 178–190.

[26] Li, X.; Zhang, X. On the minimum monochromatic or multicolored subgraph partition problems. *Theoretical Computer Science 385*, 1-3 (2007), 1–10.

[27] Lourenço, H. R.; Martin, O. C.; Stützle, T. Iterated local search: Framework and applications. In *Handbook of Metaheuristics*. Springer, 2010, pp. 363–397.

[28] Maniezzo, V.; Stützle, T.; Voss, S. Matheuristics, volume 10 of annals of information systems, 2010.

[29] Marín, A. Exact and heuristic solutions for the minimum number of branch vertices spanning tree problem. *European Journal of Operational Research 245*, 3 (2015), 680–689.

[30] Martin, B.; Sánchez, Á.; Beltran-Royo, C.; Duarte, A. A matheuristic approach for solving the edge-disjoint paths problem. In *Matheuristics 2016 - Proceedings of the Sixth International Workshop on Model-based Metaheuristics* (Brussels, Belgium, 2016), p. 25.

[31] Martinez, L. C.; Da Cunha, A. S. The min-degree constrained minimum spanning tree problem: Formulations and branch-and-cut algorithm. *Discrete Applied Mathematics 164* (2014), 210–224.

[32] Melo, R. A.; Samer, P.; Urrutia, S. An effective decomposition approach and heuristics to generate spanning trees with a small number of branch vertices. *Computational Optimization and Applications* (2015), 1–24.

[33] Merabet, M.; Molnar, M. *Generalization of the Minimum Branch Vertices Spanning Tree Problem*. Tese de Doutorado, Nanyang Technological University, Singapore, 2016.

[34] Moreno, J.; Frota, Y.; Martins, S. An exact and heuristic approach for the d-minimum branch vertices problem. *Computational Optimization and Applications 71*, 3 (2018), 829–855.

[35] Moreno, J.; Martins, S.; Frota, Y. A new approach for the rainbow spanning forest problem. *Soft Computing* (2019), 1–10.

[36] Moreno, J.; Martins, S.; Frota, Y. A note on the rainbow cycle cover problem. *Networks 73*, 1 (2019), 38–47.

[37] Padberg, M.; Rinaldi, G. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review 33*, 1 (1991), 60–100.

[38] Pirkwieser, S.; Raidl, G. R. Multiple variable neighborhood search enriched with ilp techniques for the periodic vehicle routing problem with time windows. In *International Workshop on Hybrid Metaheuristics* (2009), Springer, pp. 45–59.

[39] Primate Labs. Geekbench 3. https://www.geekbench.com/geekbench3. [Online; accessed 16-May-2018].

[40] Puchinger, J.; Raidl, G. R. Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification. In *International Work-Conference on the Interplay Between Natural and Artificial Computation* (2005), Springer, pp. 41–53.

[41] Resende, M. G.; Ribeiro, C. C. *Optimization by GRASP: Greedy randomized adaptive search procedures*. Springer, 2016.

[42] Sastry, K.; Goldberg, D. E.; Kendall, G. Genetic algorithms. In *Search methodologies*. Springer, 2014, pp. 93–117.

[43] Schmidt, J. M. A simple test on 2-vertex-and 2-edge-connectivity. *Information Processing Letters 113*, 7 (2013), 241–244.

[44] Schmidt, J. M. A simple test on 2-vertex-and 2-edge-connectivity. *Information Processing Letters 113*, 7 (2013), 241–244.

[45] Silva, D. M.; Silva, R. M.; Mateus, G. R.; Gonçalves, J. F.; Resende, M. G.; Festa, P. An iterative refinement algorithm for the minimum branch vertices problem. In *International Symposium on Experimental Algorithms* (2011), Springer, pp. 421–433.

[46] Silva, R. M.; Silva, D. M.; Resende, M. G.; Mateus, G. R.; Gonçalves, J. F.; Festa, P. An edge-swap heuristic for generating spanning trees with minimum number of branch vertices. *Optimization Letters 8*, 4 (2014), 1225–1243.

[47] Silvestri, S.; Laporte, G.; Cerulli, R. The rainbow cycle cover problem. *Networks 68*, 4 (2016), 260–270.

[48] Silvestri, S.; Laporte, G.; Cerulli, R. A branch-and-cut algorithm for the minimum branch vertices spanning tree problem. *Computers & Operations Research 81* (2017), 322–332.

[49] Skiena, S. S. *The algorithm design manual: Text*, vol. 1. Springer Science & Business Media, 1998.

[50] Sundar, S.; Singh, A.; Rossi, A. New heuristics for two bounded-degree spanning tree problems. *Information Sciences 195* (2012), 226–240.

[51] Suzuki, K. A necessary and sufficient condition for the existence of a heterochromatic spanning tree in a graph. *Graphs and Combinatorics 22*, 2 (2006), 261–269.

[52] Taillard, E. Tabu search. In *Metaheuristics*. Springer, 2016, pp. 51–76.

[53] Toth, P.; Tramontani, A. An integer linear programming local search for capacitated vehicle routing problems. In *The vehicle routing problem: Latest advances and new challenges*. Springer, 2008, pp. 275–295.

[54] Vilar Jacob, V.; Arroyo, J. E. C. ILS Heuristics for the Single-Machine Scheduling Problem with Sequence-Dependent Family Setup Times to Minimize Total Tardiness. *Journal of Applied Mathematics 2016* (2016).

[55] Zhang, X.; Zhang, Z.-B.; Broersma, H.; Wen, X. On the complexity of edge-colored subgraph partitioning problems in network optimization. *Discrete Mathematics and Theoretical Computer Science 17*, 3 (2016), 227.