Master's Degree in Informatics Engineering
Dissertation
Final Report

Algorithms for the Min-Max Regret Minimum Spanning Tree Problem

Noé Paulo Lopes Godinho
noe@student.dei.uc.pt

Advisor:
Luis Filipe C. Paquete

Date: July 3, 2017

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

UNIVERSITY OF COIMBRA

FACULTY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF INFORMATICS ENGINEERING

MASTER'S DEGREE IN INFORMATICS ENGINEERING

Algorithms for the Min-Max Regret Minimum Spanning Tree Problem

---

Author:
Noé Paulo Lopes Godinho

Advisor:
Luis Filipe C. Paquete

July 3, 2017

"An algorithm must be seen to be believed."

Donald Knuth, 1968

University of Coimbra

Abstract

Faculty of Science and Technology
Department of Informatics Engineering

Master's Degree in Informatics Engineering

Algorithms for the Min-Max Regret Minimum Spanning Tree Problem

by Noé Paulo Lopes Godinho

Uncertainty in optimization can be modelled with the concept of scenarios, where each scenario corresponds to possible values for each parameter of the problem. The Min-Max Regret criterion aims at obtaining a solution that minimizes the maximum deviation, over all possible scenarios, from the optimal value of each scenario. The study of this criterion is motivated by practical applications where an anticipation of the worst case is crucial. Well-known problems, such as the Shortest Path problem and the Minimum Spanning Tree become NP-hard with a Min-Max Regret criterion. Currently, there is a lack of knowledge on how to solve these problems in an efficient manner. This work consists in developing algorithms based on the Branch-and-Bound paradigm to solve the Minimum Spanning Tree problem under a Min-Max Regret criterion. An experimental analysis as well a comparison with a state-of-the-art pseudo-polynomial algorithm are also reported. The experimental results showed that this dissertation approach has better performance in most cases.

Keywords. min-max regret, branch-and-bound, minimum spanning tree, combinatorial optimization, graph algorithms, performance evaluation

Acknowledgements

It would not be possible to complete this dissertation without the help and support of a group of people who deserve to be in this acknowledgements list.

First, I would like to thank my advisor Luis Paquete who guided me through all this process, helped me and provided valuable suggestions to this dissertation conclusion.

Also, I thank the friends who helped and supported me through the most, if not all, of this dissertation: Carlos Diogo, Fernando Rocha, Filipe Sequeira, João Tiago Fernandes, João Silva, Paulo Pereira, Ernesto da Cruz, Luis Ventura, Ana Duarte, António Simões, João Campos and many others.

Finally, but not less important, I want to specially thank my mother who was there since the very beginning, never letting me give up of anything and providing all the support that only she could give.

Dedicated to my mother and all my friends who supported me

Contents

List of Figures

List of Tables

List of Algorithms

Acronymous

DTM Deterministic Turing Machine

NDTM Non Deterministic Turing Machine

MST Minimum Spanning Tree

MMR Min-Max Regret

B&B Branch-and-Bound

BeFS Best First Search

DeFS Depth First Search

Chapter 1

Introduction

Combinatorial optimization problems are widely found in several fields that include, but are not limited to, artificial intelligence, transportation systems, logistics and telecommunications [15, 27]. These problems consist of finding an optimal object from a finite set of objects. For most of them, it is not possible to perform an exhaustive search for all feasible solutions, since its execution time may be exponentially large [26]. For that reason, there has been a strong focus on the development of efficient algorithms to solve these problems.

Min-Max and Min-Max Regret (MMR) problems are widely used in decision theory and game theory [5, 29, 25]. These formulations model uncertainty on the objective function coefficients. The *regret* version of the problem is useful when the decision maker may feel regret if the wrong decision is made and it is taken into account when the problem is being solved. For example, if we make an investment, there is some expected profit. The profit obtained may be evaluated with some uncertainty due to various factors, such as, inflation, market evolution, etc. This uncertainty can take the form of scenarios, which may be *discrete* or *interval*. In the discrete scenario, the scenario is described explicitly by a vector of values in every scenario. In the interval scenario, each coefficient can take a number from a defined interval.

The Minimum Spanning Tree (MST) problem [14, 8, 22, 1] is a well-known problem of combinatorial optimization. Given a weighted undirected graph $G = (V, E)$, where V is the vertex set, E is the edge set and each edge has a positive weight $w(e)$, $e \in E$, the goal is to find a spanning tree with minimum total weight. A spanning tree is a graph without cycles, contains all the vertices in V and is connected. This formulation is used in several situations, such as network telecommunications. There exist two main algorithms to solve this problem: Kruskal algorithm and Prim algorithm [14]. The Min-Max Regret (MMR) Minimum Spanning Tree (MST) problem consists of finding an MST of a graph with several scenarios, i.e., more than one value per edge, minimizing the deviation from the optimal MST of each scenario. Given that the MMR formulation of the MST is NP-hard [3], those algorithms cannot be applied to solve it.

Branch-and-Bound (B&B) [10] is an algorithm paradigm that implicitly searches

for an optimum solution among all feasible solutions. This paradigm is used when a combinatorial optimization problem is known to be NP-hard. It is divided into two main parts: *branching* and *bounding*. The branching does an exhaustive enumeration of the sub-problems, while the bounding discards solutions to these sub-problems that do not provably lead to optimal solutions.

This dissertation proposes a solution method to solve the MMR formulation of the MST under the discrete scenario. In particular, this method consists of an implementation of an MST algorithm to obtain the optimal solution for each scenario, followed by a Branch-and-Bound (B&B) that uses two bound functions to obtain the best solution among all scenarios. The first bound function compares a partial solution to an upper bound that is the best solution found so far. If the value of the partial solution is greater than or equal to the upper bound, then the partial solution can be discarded since it can not lead to an optimal solution. Afterwards, the second bound function compares a lower bound of a partial solution to the upper bound. The lower bound is a point obtained by solving the MST for each scenario considering the set of vertices not considered in the partial solution. If the value of the partial solution plus the lower bound is greater than or equal to the upper bound, then the partial solution can be discarded. Moreover, an initial solution for the B&B is proposed that is obtained by calculating a weighted sum among scenarios. Finally, an experimental analysis is reported and the B&B framework is compared to an implementation of a generic pseudo-polynomial algorithm to solve MMR problems as described by Aissi et al. [4].

The remainder of this document is structured as follows: Chapter 2 describes the background of the main topics covered in this dissertation. Chapter 3 explains the main goal of this work and the algorithm proposed for this problem. Chapter 4 presents experimental analysis on a wide range of instances of this problem. Finally, Chapter 5 concludes this dissertation and discusses some future work.

Chapter 2

Background

This chapter describes the main topics covered in this work. It introduces a brief explanation of computational complexity, based on Garey and Johnson [26], Cormen et al. [14] and Sipsen [28] as well as the Minimum Spanning Tree (MST) problem and its most important algorithms, based on the descriptions given in Cormen et al. [14]. Afterwards, the Min-Max Regret (MMR) formulation is described, based on the work in Aissi et al. [5] for the MST problem. Finally, the Branch-and-Bound (B&B) approach for combinatorial optimization problems is introduced, based on the article of Clausen [10].

2.1 Computational Complexity

Computational complexity [26, 14, 28] focuses on classifying problems according to their difficulty of being solved. A problem is defined as a task to be solved, usually on a computer, by a series of steps, i.e., an algorithm. For convenience, this classification is only done in *decision problems*, i.e., problems that only have two solutions: *yes* or *no*. This facilitates the formal definition of the problems, since they are appropriate to be studied in a more precise manner in terms of theory of computation. Therefore, the counterpart of the problem is defined by a *language*, with a finite set of symbols. To correspond these languages to decision problems, and viceversa, an *encoding scheme* is defined. This encoding scheme defines the representation of the data into the language. To apply the representation to the decision problem, a *decoding* is applied. Therefore, the encoding defines the transformation of the data into the language and the decoding defines the transformation of the encoded data into the original data of the decision problem. An optimization problem may be easily turned into a decision problem by adding a constraint to the problem, making it possible to solve the question with a yes or no answer.

To formalize an algorithm, a *Deterministic Turing Machine (DTM)* model is used. A DTM is an abstract machine that is composed of a finite state control, a read-write head and a tape. This model manipulates symbols on the tape by using a set of rules. Despite being a simple model, it is capable of simulating any algorithm's logic [28].

There exist two important classes to classify the problems by time complexity: P and NP. Class P represents the set of problems for which there is a polynomial time DTM program that finds the yes/no answer to the problem p represented by a language L under an encoding scheme e. Class NP represents the set of problems for which a Non Deterministic Turing Machine (NDTM) program finds the yes/no answer to the problem p, in polynomial time. The NDTM implies some guess of the answer, since polynomial time solvability does not imply polynomial time verifiability. In a problem that belongs to the NP class, it is possible to check if a solution X belongs to the problem in polynomial time, but it is not possible to search all solutions, in polynomial time, to find the desired one.

The relation between P and NP is not yet known. If P $=$ NP, then there must be a deterministic algorithm that may solve any problem in polynomial time. Since, until now, no algorithm capable of solving any problem has been found, it is believed that P $\neq$ NP. Under this assumption, there are two important subsets of the NP class. First, there is the NP-complete class, which contains all the hardest problems that are in NP, i.e., a decision problem dp belongs to NP-complete only if dp belongs to NP and every problem in NP is reducible in polynomial time. Reducibility is the process of transforming one problem into another. Finally, there is the NP-hard class, which describes the hardest decision problems in NP, even the ones that may not be decidable, i.e., problems that are impossible to solve. It is conjectured that the NP-complete class is a subset of the NP-hard class and even problems that are not in NP belong to NP-hard.

2.2 Graph, Tree, Forest and Spanning Tree

In order to define an MST, is necessary to define a graph, a tree, a forest and a spanning tree. A graph G is an abstract mathematical structure composed of two finite sets, V and E, usually represented by a set of circles (V) and lines (E) as shown in Figure 2.1 (left) and denoted by $G = (V, E)$. V represents the set of vertices (or nodes), $|V|$ represents the number of vertices, E represents the set of edges and $|E|$ represents the number of edges in the graph. In a graph, each edge is a connection between two vertices.

There are several graph structures depending on the definition of the set E. If an edge consists of unordered pairs of vertices, then the graph is undirected, as shown in Figure 2.1 (left). If an edge consists of ordered pairs of vertices, then the graph (or digraph) is directed, as shown in Figure 2.2 (left). Finally, if an edge consists of repeated pairs of vertices, then the graph is a multigraph, as shown in Figure 2.2 (right). Also, it is possible to have weighted graphs. These graphs have weights associated to each edge, $w(e)$, $e \in E$, indicating a cost to transition from one vertex

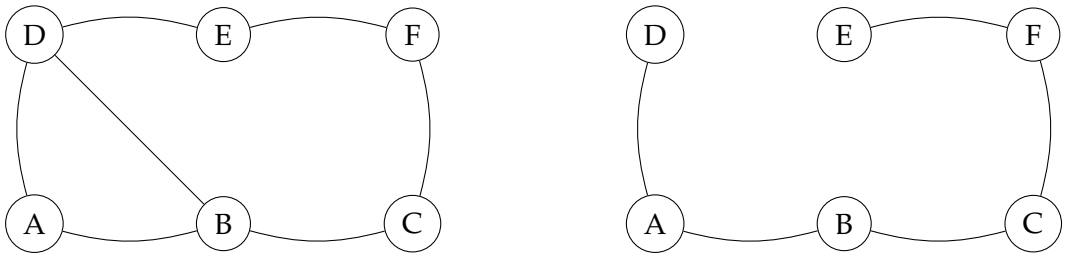


FIGURE 2.1: A graph with one cycle (left) and a spanning tree (right)

to another as shown in Figure 2.3. In this dissertation the focus is on undirected weighted graphs.

A graph can be sparse or dense depending on its number of edges [14]. A sparse graph has a low number of edges, i.e., $|E|$ is much lesser than $|V|^2$. A dense graph has a high number of edges, i.e., $|E|$ is close to $|V|^2$. An example of a dense graph is a complete graph [16], denoted by K_n, where n denotes the number of vertices $|V|$, every vertex is adjacent to every other vertex and its number of edges is $\frac{n(n-1)}{2}$.

A tree [16] is a connected graph without cycles. If it has $|V|$ vertices, then it is composed of $|V| - 1$ edges. In a connected graph there is a path between every pair of vertices. A cycle is a connection between two vertices where, at least, one has already been visited. A forest [16] is an undirected acyclic graph, where all of its components are trees.

Finally, a spanning tree is a tree that contains all the vertices in V. For example, Figure 2.1 shows a graph (left) and one of its spanning trees (right). Note that from this connected graph, it is possible to obtain several spanning trees. By Cayley's Formula [2], is possible to conclude that, on a complete graph, there are $|V|^{|V|-2}$ spanning trees.

There exist two methods to obtain a spanning tree from a graph: *building-up* and *cutting-down*. The building-up method starts with the original graph without edges and adds one edge at a time without creating cycles until a spanning tree is obtained. Using Figure 2.1 (right) as an example, the method chooses edges (A, D), (B, C), (E, F), (A, B) and (C, F). The cutting-down method starts with the original graph and removes edges keeping the connectedness of the resulting graph until a spanning tree is obtained. To obtain the spanning tree from the graph of Figure 2.1 (right), only edges (B, D) and (D, E) would need to be removed.

2.3 Minimum Spanning Tree

When the graph has weights associated to the edges, as seen Figure 2.3, one could be interested in finding an Minimum Spanning Tree (MST). The goal of an MST algorithm is to find the spanning tree of a graph whose total weight, that is, the sum

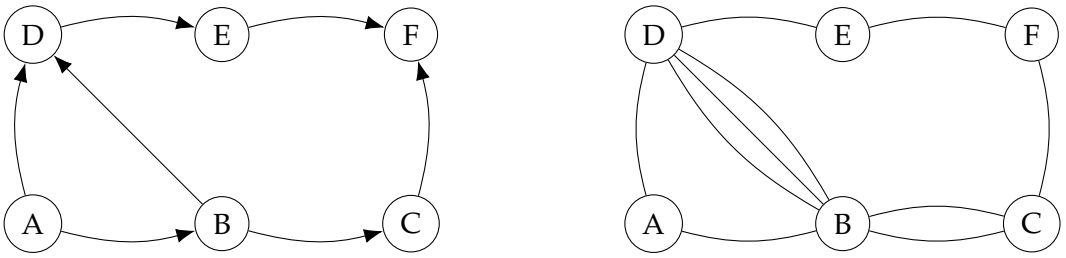


FIGURE 2.2: A digraph (left) and multigraph (right)

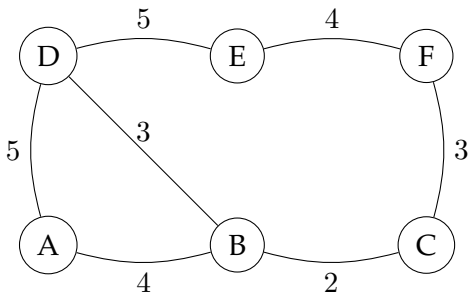


FIGURE 2.3: An undirected weighted graph

of the weights of the tree's edges, is as minimum as possible. This problem belongs to the P class since there are algorithms that solve it in polynomial amount of time. MST problems arise very often in network telecommunications, for instance, to minimize the amount of cable needed to connect a given number of routers. Supposing that the graph in Figure 2.3 is a network that connects several routers in a company, the routers can be connected as a spanning tree to avoid redundancy and as an MST to save on the length of cable.

There are two main algorithms to solve the MST problem in a connected undirected graph, Kruskal and Prim [14]. Although this document focuses on Prim's algorithm, an explanation of Kruskal's algorithm is also provided.

2.3.1 Kruskal Algorithm

Kruskal algorithm is a *greedy algorithm*. A greedy algorithm [14] is an algorithm that chooses the option that seems the best at each moment. Although in general, a greedy algorithm only finds approximations to the optimal solution, Kruskal algorithm ensures that the optimal solution is found. For efficiency reasons, it is assumed that the edges are sorted in a non-decreasing order. This algorithm is more efficient for sparse graphs [14].

To generate the MST, Kruskal algorithm goes through each edge $\{u, v\} \in G$ and checks if the vertices u and v are in different components of the forest. If they are, both components are joined into one. This ensures that the algorithm does not accept edges that cause cycles.

Algorithm 2.1: Kruskal Algorithm

```
Data: G(V, E)
Result: Set A with the MST

Function Kruskal(G)
    A ← ∅
    foreach vertex v ∈ G do
        MakeSet(v)
    G' ← Sort edges of G by non decreasing weight
    foreach edge {u, v} ∈ G' do
        if Find(u) ≠ Find(v) then
            A.append({u, v})
            Union(u, v)
    return A
```

In order to improve the efficiency of Kruskal algorithm, a disjoint-set data structure is used for maintaining the forest being created. This data structure uses a *Make-Set* operation, which creates $|V|$ subsets, each of which containing a distinct vertex in the forest, a *Find* operation that searches for the subset to which a specific element belongs to and a *Union* operation that joins the two subsets into a single one. The pseudo-code of Kruskal algorithm is described in Algorithm 2.1.

In the following, the several steps of Kruskal algorithm in the graph of Figure 2.3 are illustrated. First, the algorithm initializes the set with singletons, in this case, six sets with one different vertex in each singleton. Afterwards, the edges are sorted in a non-decreasing order of their weights. The order, in this example, is: $\{B, C\}$, $\{B, D\}$, $\{C, F\}$, $\{A, B\}$, $\{E, F\}$, $\{A, D\}$ and $\{D, E\}$. Finally, the algorithm follows the sorted list of edges and, for each vertex u and v, it compares if they belong to the same set using the *Find* operation. This ensures that the algorithm does not add a vertex to the set that causes a cycle. If they do not belong to the same set, the edge is added to the set containing the forest being created and the *Union* operation updates the *Union-Find* data structure by merging the two subsets into one, as shown in Figure 2.4. Its complexity is given by $\mathcal{O}(|E| \log |V|)$. The time complexity can be improved with the path compression technique. In this case, the time complexity becomes $\mathcal{O}(|V| \log |V|)$ [14].

2.3.2 Prim Algorithm

Prim algorithm is also a *greedy algorithm* that, at each iteration, adds an edge with minimum weight to the current tree such that no cycle is created, ensuring the correctness of the final tree. Also, it is similar to Dijkstra algorithm to find the shortest path in a graph. This algorithm is more efficient for dense graphs [14].

To generate the MST, Prim algorithm (see Algorithm 2.2) selects a random vertex u and builds up a tree until all vertices in G are chosen. This tree is generated by

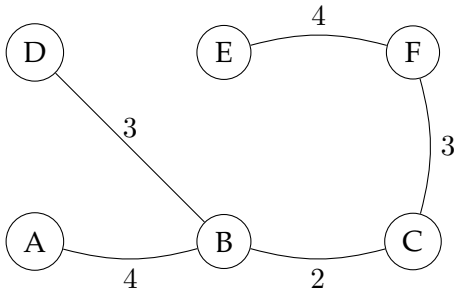


FIGURE 2.4: An MST from the graph in Figure 2.3

Algorithm 2.2: Prim Algorithm

Data: $G(V, E)$, root vertex r

Function `Prim(`G, r`)`
- **foreach** *vertex* $u \in G$ **do**
  - $u.key \leftarrow \infty$
  - $u.parent \leftarrow NULL$
- $r.key \leftarrow 0$
- $Q \leftarrow G$
- **while** $Q \neq \emptyset$ **do**
  - $u \leftarrow$ `ExtractMin` (Q)
  - **foreach** *edge* $\{u, v\} \in G$ **do**
    - **if** $v \in Q$ *and* `w(`u, v`)` $< v.key$ **then**
      - $v.parent \leftarrow u$
      - $v.key \leftarrow$ `w(`u, v`)`

choosing the lowest weighted edge that connects the vertex to the tree. To optimize the choice of the next edge, the algorithm implements a *min-priority queue*, a queue based on an attribute. This attribute is unique to each vertex and is the smallest weight of an edge incident to that vertex.

Figure 2.3 is used to show how this algorithm works. First, a random vertex is chosen, for example, A. Next, the edge with the lowest weight is chosen, $\{A, B\}$. Afterwards, $\{B, C\}$ is the edge with the lowest weight, so this will be the next edge added to the tree. Then, edges $\{B, D\}$, $\{C, F\}$ and $\{E, F\}$ are chosen in this order. The result, as in Kruskal algorithm, is an MST as shown in Figure 2.4. Its complexity, similarly to that of Kruskal algorithm, is $\mathcal{O}(|E| \log |V|)$, if a *binary heap* is used in the *min-priority queue*. A *binary heap* [14] is heap data structure that behaves as a binary tree. A heap is a tree that satisfies the following condition: a vertex u with a parent v has a value (*key*) ordered with respect to its parent's *key*. There are two types of heaps: *min-heap* and *max-heap*. The first minimizes the parent's *key*, while the former maximizes it. When a *fibonacci heap* is used, the complexity of Prim algorithm becomes $\mathcal{O}(|E| + |V| \log |V|)$ [14]. A *fibonacci heap* is a type of heap that has several operations with amortized constant time.

Note that different MSTs may be generated with these algorithms if there are edges with the same weight.

2.4 Min-Max Regret Criterion

In this dissertation, optimization problems under MMR criterion are considered. An optimization problem consists of a set of feasible solutions, an objective function and a set of constraints. A combinatorial optimization problem is an optimization problem with discrete variables and a finite set of feasible solutions [15]. The optimization problem with MMR criterion is a possible formulation when exists uncertainty on the objective function coefficients. An objective function f is a function to be minimized or maximized over the set X of feasible solutions. X is defined by a set of constraints of the problem. The uncertainty composes a scenario and the goal is to find a solution that minimizes the deviation between the value of the solution and the value of the optimal solution for each scenario. This way, the *regret* of a wrong decision is taken into account into the problem when the decision is being made.

This problem formulation differs from the usual *Min-Max* formulation whose goal is to obtain the solution at its best possible performance in the worst-case scenarios. The *Min-Max* formulation of a linear sum optimization problem is formulated as follows:

$$\min_{x \in X} \max_{s \in S} f(x, s) \tag{2.1}$$

where $f(x, s)$ is the value of a solution x under a scenario $s \in S$, S is the set of scenarios and X is the set of feasible solutions.

The i-th coefficient in a problem instance according to a scenario s is denoted by c_i^s. It is assumed that the value of a solution $x = \{x_1, \ldots, x_m\}$, where m denotes the number of coefficients, is calculated by $f(x, s) = \sum_{i=1}^{m} c_i^s x_i$. The values of this vector are defined by the types of scenarios considered, which will be described later in this section. The value of the optimal solution x_s^* in a scenario s is denoted as $f_s^* = f(x_s^*, s)$.

For a given solution $x \in X$ and scenario $s \in S$, the regret R is defined as $R(x, s) = f(x, s) - f_s^*$. Then, its $maximum$ regret $R_{max}(x)$ of a given solution x is defined as:

$$R_{\max}(x) = \max_{s \in S} R(x, s) \tag{2.2}$$

A possible way to state the MMR problem, as shown in [5] is as follows:

$$\min_{x \in X} R_{\max}(x) = \min_{x \in X} \max_{s \in S} \left(f(x, s) - f_s^* \right) \tag{2.3}$$

There are two natural ways of describing the set of all possible scenarios: *discrete scenario* and *interval scenario*. This document focuses on the discrete scenario case.

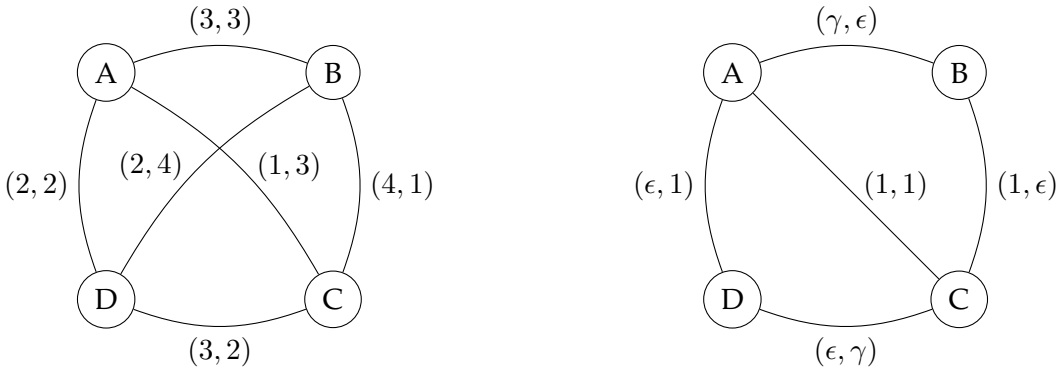


FIGURE 2.5: Two instances for Example 1 (left) and Example 2 (right)

2.4.1 Discrete Scenario

In the Min-Max Regret (MMR) Minimum Spanning Tree (MST) under the discrete scenario, c^s is a vector of size m, where each coefficient c_i^s corresponds to an edge on the graph in a scenario s. The coefficient is represented by $c^s = (c_1^s, \ldots, c_m^s)$, where $c_i^s \in \mathbb{N}, i = 1, \ldots, m$. This version assumes a finite set of scenarios $s \in S$. In this type of scenario, first, the optimal value f_s^* for each scenario $s \in S$ is obtained and, finally, an MST that solves Problem 2.3 is sought. In the following, the MST formulation with two examples, based on Figure 2.5 are illustrated.

Example 1 (left): in this example there is a graph with 4 vertices and 2 scenarios, $S = \{s_1, s_2\}$. For the first scenario s_1, the optimal value obtained by calculating the MST is $f_1^* = 5$, with edges $E_1 = \{\{A, C\}, \{A, D\}, \{B, D\}\}$. For the second scenario s_2, the optimal value is $f_2^* = 5$, with edges $E_2 = \{\{A, D\}, \{B, C\}, \{C, D\}\}$. The optimal MST for this MMR example is $MST = \{\{A, C\}, \{A, D\}, \{B, C\}\}$ with value $(7, 6)$ and $R_{max}(MST) = \max(7 - 5, 6 - 5) = 2$.

Example 2 (right): in this example there is a graph with 4 vertices and 2 scenarios, $S = \{s_1, s_2\}$. Assuming that $0 < \epsilon < 1$ and $\gamma > 1$ is a large integer value, for the first scenario s_1, the optimal value obtained by calculating the MST is $f_1^* = 1 + 2\epsilon$ with edges $E_1 = \{\{A, D\}, \{B, C\}, \{C, D\}\}$. For the second scenario s_2, the optimal value is $f_2^* = 1 + 2\epsilon$ with edges $E_2 = \{\{A, B\}, \{A, D\}, \{B, C\}\}$. The optimal MST for this MMR example is $MST = \{\{A, C\}, \{A, D\}, \{B, C\}\}$ with value $(2 + \epsilon, 2 + \epsilon)$ and $R_{max}(MST) = 1 - \epsilon$. This example shows that the optimal value of the MMRMST can be arbitrarily far from the optimal value for each scenario.

In the next Section, a general pseudo-polynomial algorithm to solve MMR discrete problems is described.

A General Pseudo-Polynomial Algorithm

Aissi et al [4] introduces a general pseudo-polynomial algorithm to solve MMR optimization problems. A pseudo-polynomial algorithm is an algorithm that is polynomial in its number of inputs but exponential in the length of the input. This length is the number of bits necessary to write the input. This algorithm solves a sequence of decision problems for every possible value of a modified objective function. This transformation consists in aggregating the several scenarios into one. The authors show that, once a solution is found for the related decision problem, it is also a solution for the original MMR problem. This decision problem needs to be solvable in polynomial or pseudo-polynomial time to ensure that the overall performance is pseudo-polynomial.

This algorithm (see Algorithm 2.3) starts by calculating the optimal value f_s^* for each scenario $s \in S$ and by generating an instance I'. Then, for an increasing order of all possible values $v = 0, \ldots, mM - \max_{s \in S} f_s^*$, where $M = \max_{i,s} c_i^s$, it decides if there exist a solution for instance I' with a specific value α_p for a p that describes a p-th scenario, $p = 1, \ldots, k$ and $k = |S|$, where $\alpha_p \leq f_p^* + v$ and there is a $q \leq k$, such that, $\alpha_q = f_q^* + v$. The authors show that these conditions can be transformed into a unique condition, as shown in the following equation:

$$\sum_{p=1}^{k} f(x,p)(mM+1)^{p-1} = \sum_{p=1}^{k} \alpha_p (mM+1)^{p-1} \tag{2.4}$$

Then, this problem can be solved by finding a solution with value $\sum_{p=1}^{k} \alpha_p (mM + 1)^{p-1}$ in an instance I'. This instance I' is obtained from the original instance problem by aggregating its coefficients among scenarios. This way, the coefficients in I' are obtained as follows:

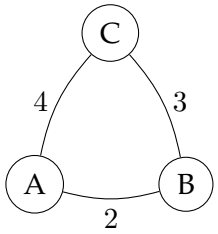
$$c_i' = \sum_{p=1}^{k} c_i^s (mM+1)^{p-1} \tag{2.5}$$

The `ValueFound` function in Algorithm 2.3 finds if the value obtained in each loop iteration given by Equation 2.4 corresponds to the value of a feasible solution. A way to ensure that this value is found, is to enumerate all feasible solutions in the modified instance I'. Since I' is an aggregation of all coefficients on the original instance, this problem has now only one scenario. Considering there is only an interest on the solution value, a possibility is to enumerate all possible values for all feasible spanning trees in I'. Barahona and Pulleyblank [6] show that it is possible to enumerate all possible values for all feasible spanning trees of a graph using a polynomial function. Let MX be a matrix $|V|$x$|V|$ constructed as shown in Equation 2.7. Let $D(MX)$ be the determinant of the matrix M obtained by removing the first

Algorithm 2.3: Pseudo-polynomial algorithm for the MMR problem [4]

Data: I', f_s^* for all $s \in S$
Result: Optimal solution v

Function `Pseudopolynomial` (I', f_s^*)
  $v \leftarrow 0$
  $test \leftarrow false$
  while $test \neq true$ **do**
    for $(\alpha_1, \ldots, \alpha_k) : \max\{\alpha_1 - f_1^*, \ldots, \alpha_k - f_k^*\} = v$ **do**
      $value \leftarrow \sum_{p=1}^{k} \alpha_p (mM+1)^{p-1}$
      if `ValueFound` $(I', value)$ **then**
        $test \leftarrow true$
    if $test \neq true$ **then**
      $v = v + 1$
  return v



	A	B	C
A	$x^2 + x^4$	$-x^2$	$-x^4$
B	$-x^2$	$x^2 + x^3$	$-x^3$
C	$-x^4$	$-x^3$	$x^3 + x^4$

FIGURE 2.6: Example graph (left) and its polynomial matrix MX (right)

row and column, corresponding to root vertex. The authors show that the equality in Equation 2.6 holds, where in this polynomial function, a_k corresponds to the number of spanning trees with value k.

$$D(M) = \sum a_k x^k \tag{2.6}$$

$$M_{ij}(x) = \begin{cases} -x^{w(j,i)} & \text{if } (j,i) \notin E,\ i \neq j \\ 0 & \text{if } (j,i) \notin E,\ i \neq j \end{cases}$$
$$M_{ii}(x) = \sum_{j \neq i} -M_{ij}(x) \tag{2.7}$$

In order to illustrate these results, an example is shown in Figure 2.6. In this example, a complete graph (left) with three vertices is represented. Matrix MX is shown in Figure 2.6 (right), using Equation 2.7. Then, the polynomial determinant, $D(MX) = ((x^2 \cdot x^3) \cdot (x^3 \cdot x^4)) - (x^3 \cdot x^3) = x^5 + x^6 + x^7$ is obtained. This solution indicates that there are three spanning trees in this graph with values 5, 6 and 7. These values correspond to the spanning trees $ST_1 = \{\{A, B\}, \{B, C\}\}$,

$ST_2 = \{\{A, B\}, \{A, C\}\}$ and $ST_3 = \{\{A, C\}, \{B, C\}\}$, respectively.

2.4.2 Interval Scenario

In the interval scenario case, each coefficient c_i has a value in a given interval $[\underline{c_i}, \bar{c_i}]$, where $0 \leq \underline{c_i} \leq \bar{c_i}$ and $i = 1, \ldots, n$. The set of scenarios S is obtained by the cartesian product of the intervals. Similarly to the discrete scenario case, the method to obtain the optimal solution is by calculating an optimal solution x_s^* for each scenario individually and then calculating $\min R_{\max}(x)$.

In the next section, several algorithms to solve MMR interval problems are presented.

Interval Algorithms

Makuchowski [21] introduces a perturbation algorithm for a MMRMST interval scenario formulation. A perturbation algorithm is a modification of an existing algorithm, usually a construction algorithm. This algorithm applies multiple random perturbations on the input data and then solves this problem with modified input data, using a base algorithm. Then, this solution is tested on the original input data. After performing n iterations, the algorithm returns the best solution found thus far. It should be noted that this algorithm is only able to return an approximation to the optimal solution.

Montemanni and Gambardella [23] describe an exact algorithm for the MMR shortest path formulation. In a shortest path problem [14] the goal is to finding the minimum weight path in a given weighted directed graph.

Kasperski and Zieliński [17] describe a polynomial time approximation algorithm for generic MMR interval problems. First, the algorithm calculates, for each coefficient, a fixed value with the midpoint of their interval values, $c_i^s = \frac{1}{2}(\underline{c_i} + \bar{c_i})$. Then, an algorithm to solve the problem is applied to only this new scenario s. The authors show that this algorithm has a performance ratio of at most 2, i.e., the obtained solution is, at most, 2 times worse than the optimal solution.

2.5 Branch-and-Bound

In B&B the goal is to search for the optimal solution, among an *interesting* set of candidate solutions. When all candidate solutions have been implicitly visited, the optimal solution is the best solution found so far. This search process can be seen as a decision tree. A decision tree is a method to display the search process of an algorithm by drawing a graph, as a tree. Each vertex of this tree corresponds to a state of the search. A subproblem is used to denote a new problem with additional constraints obtained from the original problem and it corresponds to a subspace of

the original space of solutions. There are several subproblems, where each one starts in a given vertex and contains all the subproblems that can be generated from the actual subproblem.

For the explanation, an optimization problem is assumed, with an objective function f, variables $\{x_1, \ldots, x_n\}$ and a set X of feasible solutions. Without loss of generality, minimization is assumed, as follows:

$$\min_{x \in X} f(x) \tag{2.8}$$

Usually, these problems have constraints. Let P denote the set of solutions that are obtained by relaxing some of these constraints, that is $X \subseteq P$. A bounding function $g(x)$, such that $g(x) \leq f(x)$, for all $x \in X$, is also defined.

There are critical subproblems that, when having a bounding function applied to them, the obtained value is strictly lesser than the optimal solution to the problem. At the end, they are used to prove optimality when the value has been discovered. There are four important steps in each iteration of the algorithm: *branching*, *bounding*, *selecting a candidate solution from the subset of available solutions* (also called the next subproblem) and *choosing a good initial solution*.

2.5.1 Branching

An important step on a B&B algorithm is branching. This consists of partitioning the search space based on the assignment of values to a certain variable. There are two types of branching: *dichotomic* and *polytomic*. When the search space is divided in two parts, the branching is dichotomic, otherwise it is polytomic. To ensure the convergence of a B&B algorithm it is necessary to ensure that the size of each subproblem is smaller than the original problem and the solutions are finite. This branching is also used in exhaustive enumeration algorithms.

2.5.2 Bounding

The bounding function is also an important component of a B&B algorithm because it makes it possible to discard (partial) solutions that will provably not lead to the optimum. The solutions are discarded based on a bounding function. A bounding function may be strong or weak, depending on the problem and the values obtained from it. There are three problems that are useful to discard solutions:

$$\min_{x \in X} g(x) \tag{2.9}$$

$$\min_{x \in P} f(x) \tag{2.10}$$

$$\min_{x \in P} g(x) \tag{2.11}$$

Problem 2.9 consists of finding the solution that minimizes the bounding function, i.e., the objective function is modified and ensures that $g(x) \leq f(x)$, for all, $x \in X$. Problem 2.10 consists in a relaxation, i.e., removing some constraints from the original problem or relaxing them. Finally, Problem 2.11 consists of combining the two strategies. This combination may seem weaker, but may be strong enough if P and g are chosen correctly. The following relation holds for the three problems:

$$\min_{x \in P} g(x) \leq \left\{ \begin{array}{l} \min_{x \in P} f(x) \\ \min_{x \in X} g(x) \end{array} \right\} \leq \min_{x \in X} f(x) \tag{2.12}$$

These three problems give lower bounds on the original problem and can be used to discard solutions during the search process if the optimal value is worse than the best solution value found so far.

2.5.3 Selection of a Candidate Solution

When selecting a candidate solution from the subset of available solutions there is a trade off between having a low number of explored solutions and not exceeding the memory of the computer. In order to handle this, two strategies have been proposed: *Best First Search (BeFS)* and *Depth First Search (DeFS)*. The BeFS strategy chooses the subproblem with the lowest bound. This strategy usually consumes more memory when the given problem has a large number of critical subproblems as they can not be discarded. Also, it needs to store the subproblems from each level. The alternative is to use a DeFS strategy, which requires much less memory since it only stores in memory the children of the actual solution in the space search. A problem occurs when the optimal solution is far from the actual solution, since it need to search for many other solutions until reaching a leaf. In order to avoid these problems, usually a combination of BeFS and DeFS is applied.

2.5.4 Good Initial Solution

A good initial solution allows the algorithm to discard more partial solutions. Since the best solution is set to infinity at the beginning, a lot more solutions are to be analyzed. The tighter the initial solution is, the more partial solutions that do not lead to the optimal solution are discarded. There are several strategies to obtain the initial solution, where the most common are using heuristics, such as Simulated Annealing, Genetic Algorithms or Tabu Search. This constrains the number of subproblems to be explored, specially on a DeFS strategy for selecting solutions. The arrangement of the previously defined steps depends on the problem or the strategy used to solve it. One strategy is to first calculate the bound of the selected solution and then branch on the solution if necessary, also called *lazy* strategy. If the *eager* strategy

is used instead, the branching is applied first, to subdivide the actual space of solutions into smaller subspaces. This way, at least two child solutions are constructed after being added constraints to the subproblem.

In the next chapter, a B&B framework implementation and several improvements are described.

Chapter 3

A Branch-and-Bound Framework

The purpose of this chapter is to describe a Branch-and-Bound (B&B) framework to solve the Min-Max Regret (MMR) Minimum Spanning Tree (MST) problem. First, it describes the implementation to obtain the optimal solution f_s^* for each scenario $s \in S$, where S denotes the set of scenarios, using Prim algorithm. Then, it describes the branching and selection of a candidate solution implementation to solve the MMRMST problem by enumerating all spanning trees in a graph, based on the algorithm described by Gabow and Myers [13] and two bound functions. Also, an implementation of an initial solution is described. Finally, a multithreading improvement is described.

3.1 Solving each Scenario

To obtain an optimal solution for each scenario $s \in S$ in the MMRMST problem, an MST algorithm needs to be implemented. As explained in Section 2.2, in this dissertation the focus is on undirected weighted complete graphs with positive weights, since they are the worst-case instances of undirected graphs. Therefore, the algorithm used is Prim algorithm (see Algorithm 2.2), since it performs better in dense graphs [14]. It is assumed that the optimal value for all scenarios are stored in an array called opt.

3.2 Branching and Selection of a Candidate Solution

In this section, the algorithm for generating all spanning trees in a graph is described. This algorithm is based on the approach of Gabow and Myers [13], since it does not generate duplicated spanning trees. It is the basis for the B&B that is proposed in this dissertation. The authors show that this algorithm has complexity $\mathcal{O}(|V|N)$, where N denotes the number of spanning trees in a given graph. As noted in Section 2.2, in a complete graph there are $N = |V|^{|V|-2}$ spanning trees, therefore its time complexity is exponential, $\mathcal{O}(|V|^{|V|-1})$, for complete graphs. It is important to note that this algorithm can be used in directed and undirected graphs.

This algorithm is called recursively and uses a method for detecting bridges using the Depth First Search (DeFS) principle. A bridge is an edge that when deleted increments the number of connected components in a graph, i.e., separates the graph into two or more subgraphs. It finds all spanning trees containing a given subtree in a starting vertex. It proceeds by choosing an edge e_i incident to a vertex visited and a non-visited. Then, all spanning trees containing that edge are found. Next, this edge is deleted, is chosen another edge and the same steps are done until a bridge is found. This bridge is found at the end of the series of steps by a bridge test and ensures that all spanning trees are found exactly once.

The pseudo-code of the approach to generate all spanning trees is given in Algorithm 3.1. The algorithm starts with a vertex r, which is marked as visited. During the construction, two lists, F and T, are implemented. These two lists act as a stack. Therefore, the selection of the candidate solution in each iteration uses a DeFS principle, as explained in Section 2.5.3. List F contains the candidate edges to the spanning tree being constructed and list T contains the actual spanning tree. In addition, there are two arrays (as global variables) used in this algorithm: an array $visited$ contains the vertices visited and the *opt* array is used to calculate the *maximum regret* of each partial solution. At the beginning, all edges $(r, u) \in G$ are added to list F. Afterwards, there exists a variable denoted by $best$, which contains the best solution found so far and is initialized to infinity.

The input of the algorithm is given by two parameters. First, it receives a value n that denotes the number of vertices visited at the moment. It is used to indicate when the actual spanning tree contains all the vertices. In its first call, is set to 1. Second, it receives an array val, which indicates the actual value for each scenario of the actual spanning tree T. It is used to calculate the *maximum regret* of the partial solution. In its first call, is set to 0 for all scenarios $s \in S$.

In each recursive function call, there is a verification to check if all vertices have been visited (line 2). When the algorithm starts, the number of vertices visited is one, therefore it jumps to the else condition (line 6). Then, a variable denoted by b is set to $false$. This variable is used in the while condition to check if the chosen edge is a bridge, to ensure that all spanning trees are found. Also, two local lists FF and $restore$ are set as empty. The first list is used to store all edges used and marked as visited in the graph, in order to restore them when the edge is a bridge. The second list is used to restore edges that are removed from the F list. When it enters the while condition, an edge e is popped from F (line 11). This edge has a vertex u already visited and a vertex v not visited. This vertex v is obtained by using function $Extract$ (line 12). Then, edge e is pushed into list T and v is marked as visited (lines 13-14).

Next, some operations on list F are performed. First, all edges $(v, w), w \notin T$, i.e.,

Algorithm 3.1: Branch Algorithm based on [13]

Data: $G(V, E)$, root vertex r
Result: Finds all spanning trees in G and returns the optimal solution for the MMRMST problem

```
1  Function Branch(n, val)
2      if n = |V| then
3          r_max ← R_max(val)
4          if best > r_max then
5              best ← r_max
6      else
7          b ← false
8          FF, restore ← ∅
9
10         while b = false do
11             e ← F.Pop()
12             v ← Extract (e)
13             T.Push(e)
14             visited[v] ← true
15             F.Push(v, w), w ∉ T
16             foreach edge (w, v) ∈ F do
17                 restore.Push(w, v)
18                 F.Remove(w, v)
19             foreach scenario s ∈ S do
20                 val[s] += ω(e, s)
21             Branch(n + 1, val)
22             foreach scenario s ∈ S do
23                 val[s] −= ω(e, s)
24             F.Remove(v, w), w ∉ T
25             foreach edge (w, v) ∈ restore do
26                 F.Push(w, v)
27                 restore.Remove(w, v)
28             visited[v] ← false
29             T.Pop (e)
30             G.Pop (e)
31             FF.Push(e)
32             b ← BridgeTest()
33
34         Restore each edge (u, v) from FF to F and G
```

all edges incident to v and incident to vertices that are not yet visited, are pushed into F (line 15). Now, all edges $(w, v) \in T$, i.e., all edges incident to v that were already visited, are removed from F and pushed into $restore$, together with their original position on list F (lines 16-18). This way, list F has only edges (u, w) such that $w \notin T$. Then, the val array is updated with the weight ω of edge e for each scenario $s \in S$ (lines 19-20).

Finally, the function is called recursively with updated val and n is incremented by 1 (line 21). After this recursive call is made and the algorithm backtracks, the

operations above are undone. Array val is update with the removal of the previous weight ω (lines 22-23). All edges $(v, w), w \notin T$ are removed and all edges previously removed and existing in $restore$ list are restored in their original positions (lines 25-27). Then, vertex v is marked as unvisited and edge e is removed from list T and removed from the graph and pushed into list FF (lines 28-31). This allows us to restore this edge, and all other edges used in the process, when a bridge is detected (line 34). Finally, the bridge test is performed in Function `BridgeTest` (line 32). This test is explained in the approach of Gabow and Myers [13], using the DeFS principle.

3.3 Bounding

In the following, two bounding functions are introduced. Let T be a partial solution, N_u denote the set of vertices for which a decision has not yet taken in the search process and UB be an upper bound on the optimal solution, for example, the best solution found during the search process. Finally, let $f(T) = R_{max}(T)$ (see Equation 2.2) and $f(UB) = R_{max}(UB)$.

Bound 1: If $f(T) \geq f(UB)$, T can be discarded as it will not lead to optimal solutions.

Let $s_1, s_2 \in S$ be two scenarios, $f^*_{s_1,s_2} = (\alpha, \beta)$ be the optimal solutions for each scenario s, $UB = (\gamma, \omega)$ and $T = (a, b)$. By Equation 2.2, $f(UB) = \max(\gamma - \alpha, \omega - \beta)$ and $f(T) = \max(a - \alpha, b - \beta)$. Without loss of generality, lets assume that $f(UB) = (\gamma - \alpha)$ and $f(T) = (b - \beta)$. Therefore, if $(b - \beta) \geq (\gamma - \alpha)$, this solution will not lead to an optimal solution, since the *maximum regret* of any extension of T is always greater.

Figure 3.1 illustrates an instance of the algorithm using *Bound 1*. The optimal solution value f^*_s is obtained by MST_1 and MST_2, therefore, $f^*_{s_1,s_2} = (1, 1)$. Also, the best solution found so far is $UB = (5, 6)$, $f(UB) = \max(5 - 1, 6 - 1) = 5$. Note that a partial solution with value T_1 can not be discarded, since it may lead to an optimal solution. This is easy to observe since $f(T_1) = \max(4 - 1, 4 - 1) = 3$ and $f(UB) = 5$, therefore $f(T_1) \ngeq f(UB)$. However, note that T_2 can be discarded, since $f(T_2) = \max(7 - 1, 8 - 1) = 7$, therefore $f(T_2) > f(UB)$. Finally, for similar reasons, T_3 can be discarded as well. This happens because its value is the same as $f(UB)$, $f(T_3) = \max(6 - 1, 5 - 1) = 5$, therefore $f(T_3) = f(UB)$.[1] Any solution that lies inside the red rectangle can not be discarded since the optimal solution value of the *maximum regret* value is lesser than the one obtained by UB.

[1] Note that only graphs with positive weights are considered.

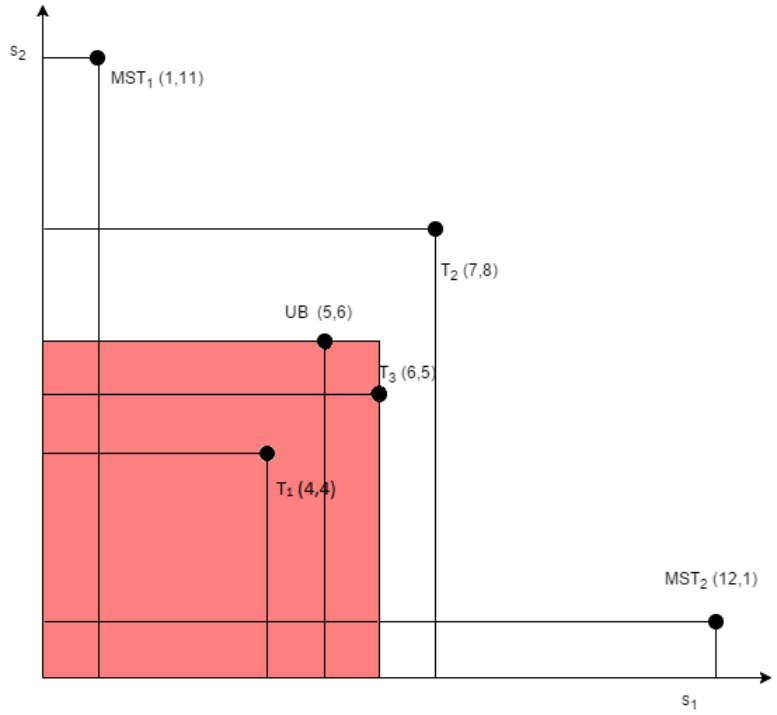


FIGURE 3.1: *Bound 1* example

Bound 2: Let LB be a lower bound. LB can be obtained by solving the MST for the subproblem defined by the vertices in N_u for each scenario $s \in S$. Then, if $f(T) + LB \geq f(UB)$, T can be discarded.

As shown for *Bound 1*, let $s_1, s_2 \in S$ be the scenarios, $f^*_{s_1,s_2} = (\alpha, \beta)$ be the optimal solutions for each scenario s, $UB = (\gamma, \omega)$ and $T = (a, b)$. Also, let $LB = (LB_a, LB_b)$. By Equation 2.2, $f(UB) = \max(\gamma - \alpha, \omega - \beta)$ and $f(T) + LB = \max(a + LB_a - \alpha, b + LB_b - \beta)$. Without loss of generality, lets assume that $f(UB) = (\gamma - \alpha)$ and $f(T) + LB = (b + LB_b - \beta)$. Therefore, if $(b + LB_b - \beta) \geq (\gamma - \alpha)$, this solution will not lead to an optimal solution, since the *maximum regret* of any extension of T is always greater. Also, each MST obtained is the best solution for each scenario s individually, therefore the actual weights are greater than or equal to the MST obtained.

Figure 3.2 illustrates an instance of the algorithm using *Bound 2*. The optimal solution f^*_s is obtained by MST_1 and MST_2, therefore, $f^*_{s_1,s_2} = (1, 1)$. Also, the best solution found so far is $UB = (5, 6)$, $f(UB) = \max(5 - 1, 6 - 1) = 5$. T_1 can not be discarded since it may lead to an optimal solution. This easy to observe since $f(T_1) + LB_1 = \max(3 + 2 - 1, 2 + 1 - 1) = 4$, therefore $f(T_1) + LB_1 \ngeq f(UB)$. T_2 can be discarded, since $f(T_2) + LB_2 = \max(4 + 3 - 1, 5 + 3 - 1) = 7$, therefore $f(T_2) + LB_2 > f(UB)$. The red dotted lines denote the solution obtained by LB with the vertices in N_u for scenarios s_1, s_2.

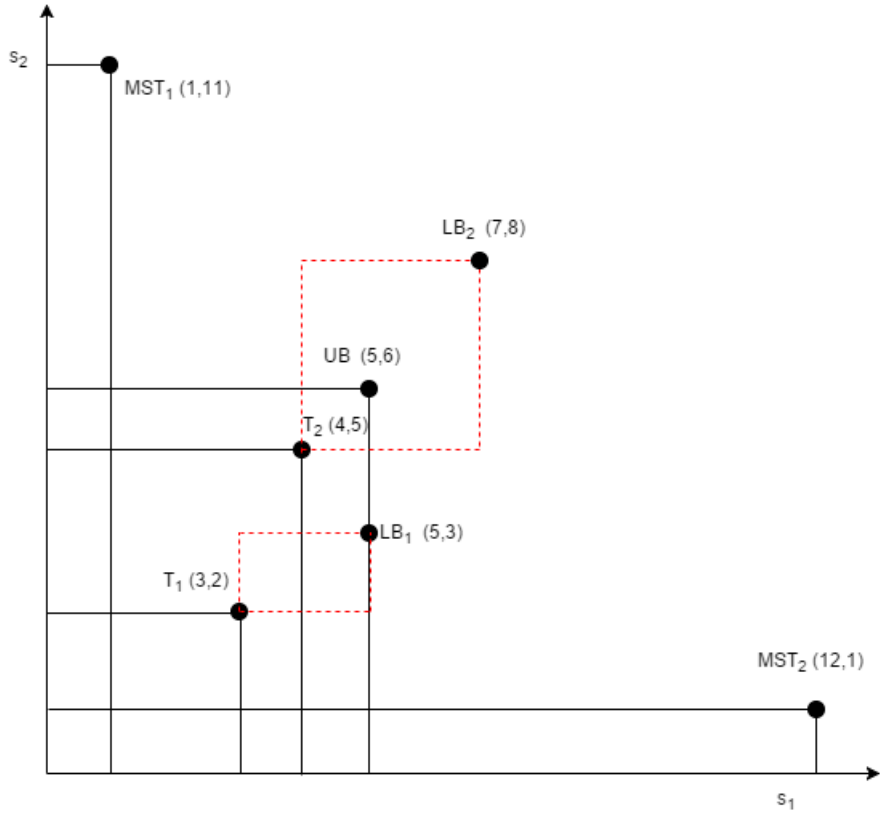


FIGURE 3.2: *Bound 2* example

These two bounds are implemented in the branching algorithm. For each function call, it verifies *Bound 1*, and if it fails, it verifies *Bound 2*. This may be implemented between lines 1 and 2 in Algorithm 3.1.

3.4 Initial Solution

To allow the algorithm to discard more partial solutions at the beginning, a procedure that computes an initial solution has been implemented. This procedure consists of finding a solution for a weighted sum problem with a single scenario where the coefficients are computated as follows:

$$c_i = \sum_{p=1}^{|S|} \omega_p c_i^p \tag{3.1}$$

where p indicates the p-th scenario, $p = 1, \ldots, |S|$, ω_p indicates a previously defined weight for each p-th scenario, such that $(\omega_1 + \ldots + \omega_{|S|}) = 1$. By solving the problem for different weights, different initial solutions can be provided. Prim algorithm can be used to solve the weighted sum problem, since it contains only one scenario and the initial solution is calculated using the *maximum regret* Equation 2.2. Note that this

problem corresponds to an MST problem and becomes the initial solution of $best$ in Algorithm 3.1.

3.5 Multithreading

A multithreading version of the B&B framework is implemented as well. The threads only share the optimal solution for each scenario f_s^*, the best solution found so far and the best initial solution found. Therefore, each thread has its own weight ω_p to obtain an initial solution independent from the other threads. This allows the algorithm to obtain a better initial solution, since the best from all threads is used as initial solution of $best$ in Algorithm 3.1. Also, each thread has a different vertex r.

Chapter 4

Experimental analysis

In this chapter an experimental analysis of the Branch-and-Bound (B&B) framework is described. As previously indicated, the graphs used in the experiments are *complete graphs*. To generate these random complete graphs, a *python* (version 3.5.2) script using *NumPy* [24] library was coded. Each graph instance is defined by: number of vertices, number of scenarios, minimum weight edge value, maximum weight edge value and graph number. The *minimum* and *maximum weight edge value* indicate the interval of the random values generated for each coefficient and scenario. The *graph number* is used to indicate the graph generated, since these graphs are generated only once and used for all the experiments.

The B&B framework is implemented in *C++11*, because of the need to use function *std::thread* to implement the multithreading, and compiled with *g++* version 5.4.0. It was tested in an Operating System *Ubuntu 16.04* with 4GB of memory RAM, a single core CPU with 2 virtual threads and a clock rate of 2GHz. For comparing the implementation of Prim algorithm, Boost version in C++ [11] is used. This library provides a generic implementation for several algorithms, including graph algorithms. It is implemented in C++98 in the same experimental setup as the B&B framework. For comparing the B&B approach with the pseudo-polynomial algorithm described in Section 2.4.1, the latter is implemented in C++98 and the determinant matrix calculation has been made with the support of GiNaC C++ [7] library. This algorithm has been tested in the same experimental setup. Finally, *R* (version 3.2.3) is used to analyze the obtained results. For the experimental analysis graphs with 4 to 15 vertices, two scenarios, and minimum and maximum weight value from 50 to 1000 were considered. Also, 100 graphs for each combination of parameters are generated. Note that the Time axis in all figures showing the average times are in *log scale*. For each run, the CPU time was collected.

A linear regression on the time to solve the instances with respect to the instance size is performed as well, which gives a further insight on how the algorithm scales. After constructing the model, is useful to know how well the equation fits the data and, therefore, there is an interest in knowing the *determination coefficient*, or R^2. This value is between 0 and 1, where 0 indicates that the data is not represented by the

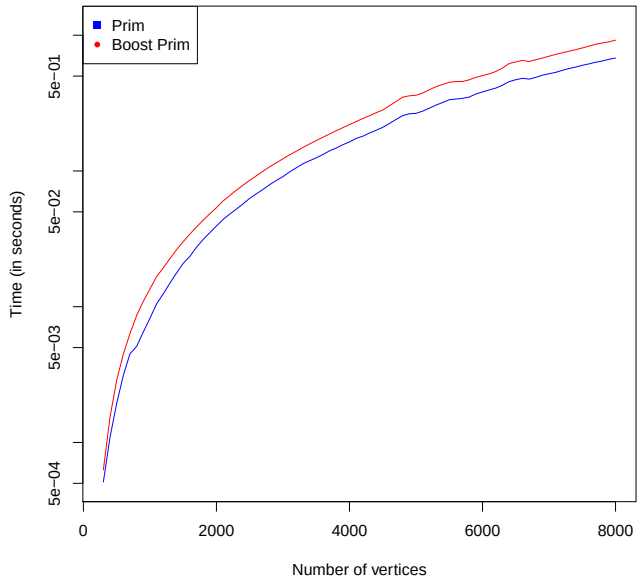


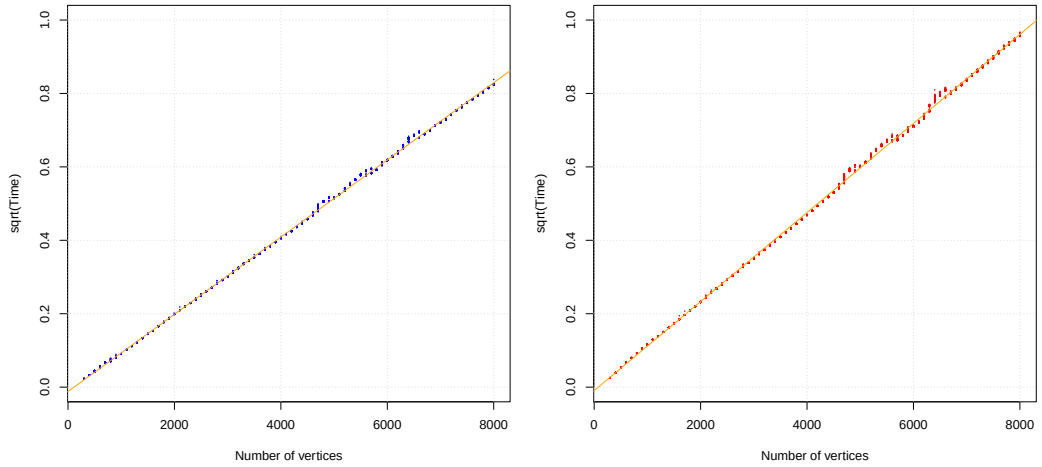
FIGURE 4.1: Prim and Boost Prim average time (in seconds) per number of vertices

regression model and 1 shows that the data is perfectly represented by the regression model. By using *boxcox* function in *R*, it is possible to derive a good transformation in the dependent variable, i.e., the CPU time. *lm* function in *R* is used to apply the linear regression model on the transformed data.

4.1 Prim Algorithm

Following the same order as the previous chapter, the performance of the implementation of Prim algorithm is analyzed. In Figure 4.1, the results of our implementation of Prim algorithm in the B&B framework and a Prim algorithm implemented in the Boost C++ [11] library are shown. In this experiment, graphs from 300 to 8000 vertices are considered. All the average times for each number of vertices and its standard deviation are in Tables A.4 and A.5. In this figure, is possible to observe a slightly better performance of the Prim implementation with increasing instance size.

Figure 4.2 shows the regression line according to a square root transformation of the dependent variable. This transformation suggests a quadratic behaviour, which makes sense since complete graphs are used. As is possible to observe, the regression line (in orange) is very close to the data collected for both algorithms, $R^2 = 0.9996$ for our Prim implementation and $R^2 = 0.9993$ for the Boost implementation.



(A) Prim algorithm (B) Boost Prim algorithm

FIGURE 4.2: Linear regression model from Prim algorithm, Figure 4.1

4.2 Bounding

In this section the results of the B&B implementation using only *Bound 1* (blue), using only *Bound 2* (red) and using both bounds (green) are analyzed. Note that it is not yet considered the effect of using an initial solution. For reference, the running time of B&B without using any bound (black) is also shown. In Figure 4.3, is possible to observe their behaviour. With few vertices, *Bound 1* performed better than *Bound 2*. However, as instance size grows, *Bound 2* presents better performance. When both bounds are used, the results are slightly better overall, since some cases are discarded first by *Bound 1* and, therefore, the use of *Bound 2* is avoided. Tables A.2 and A.3 show the average running time and standard deviation, respectively, of the bounds separately and together. When is compared the average time of only the B&B without and with bounds, is possible to observe a lot of improvement. For example, with 11 vertices, the approach without bounds takes an average of 2640 seconds, whereas for *Bound 1*, *Bound 2* and both bounds, the average is 4, 3 and 2 seconds, respectively.

Figure 4.4 shows the linear regression model obtained from the B&B. The best transformation for the dependent variable is logarithmic, which suggest an exponential behaviour as expected. Is possible to observe that the model still represents the data very well. For all cases, an R^2 larger than 0.96 was obtained.

4.3 Initial Solution

In the following, the effect of using an initial solution is considered, as explained in Section 3.4. $w_1 = w_2 = 0.5$ are considered. Both bounds are used since they gave

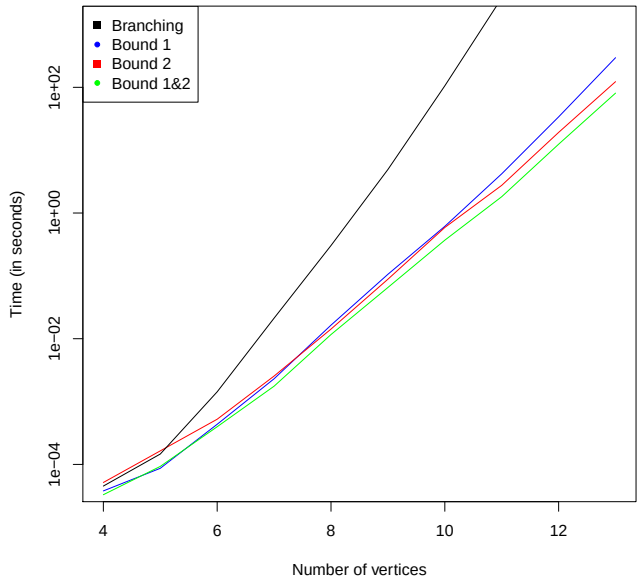


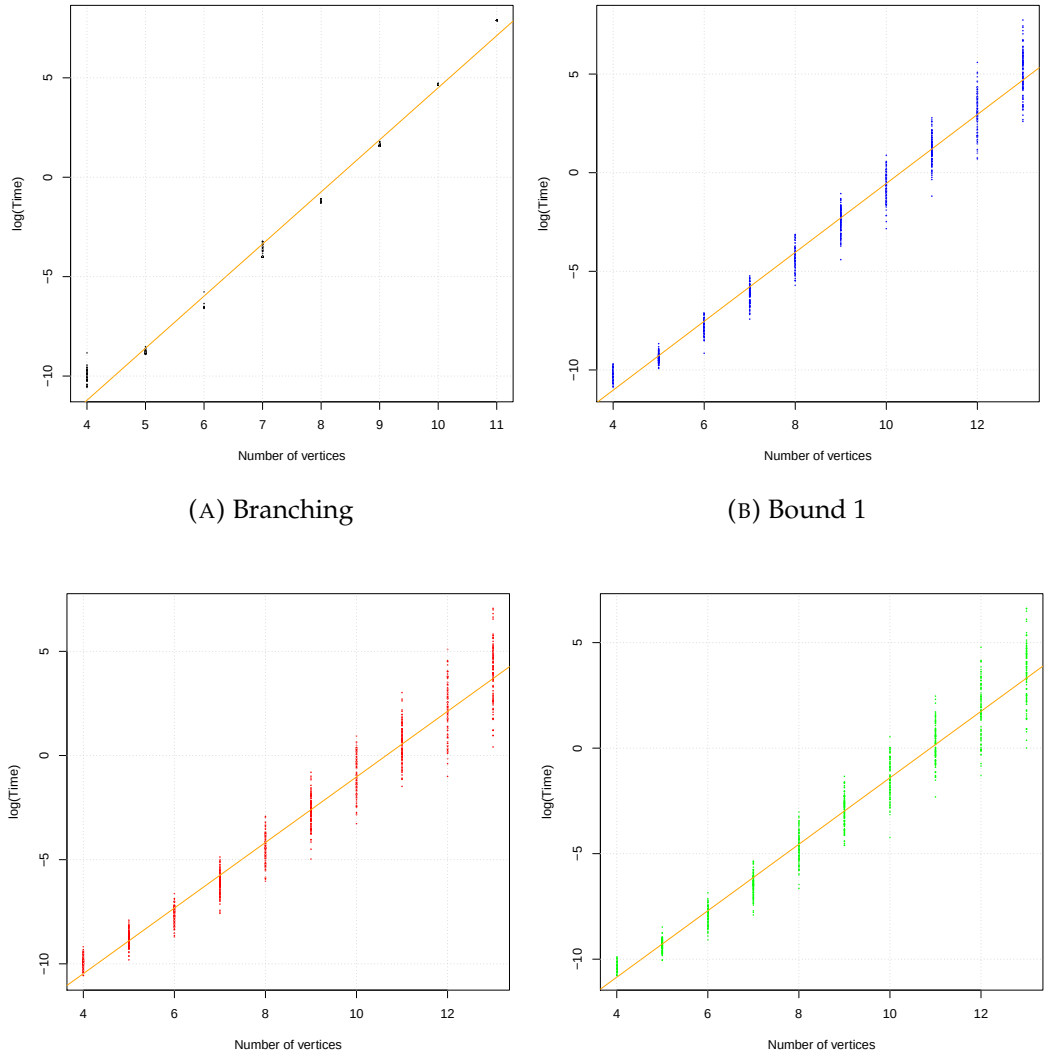
FIGURE 4.3: Branching only and bounds average time (in seconds) per number of vertices

the best performance, as shown in the previous section. Figure 4.5 (A) shows the execution times of the branching with both bounds, with and without the calculation of an initial solution. The number of vertices used are from 4 to 14. When an initial solution is used, the algorithm performs faster. Table A.2 shows the average running time of both approaches. Is possible to observe that using an initial solution brings an improvement of, approximately, 20%. For example, for 14 vertices, the running time is, in average, 419 seconds when an initial solution is used and 536 seconds when is not used. Table A.3 shows the standard deviation for both experiments. When the number of vertices increases, the standard deviation increases as well.

Figure 4.5 (B) shows the linear regression model obtained to this initial solution implementation. As before, the suggested dependent variable transformation is *logarithmic*. For this model, R^2 value is 0.9461.

4.4 Multithreading

As described in Section 3.5, a multithreading approach has been implemented with 4 threads. Each thread has his own w_1 and w_2 values for the calculation of the initial solution. For each thread, a different initial solution given by $w_1 = (0.8, 0.6, 0.4, 0.2)$ and $w_2 = (0.2, 0.4, 0.6, 0.8)$ is considered, as they allow the algorithm to start from distinct solutions. Figure 4.6 (A) shows the results of the experiments. As is possible to observe, there exist a performance increase on the multithreading approach as the number of vertices increase. Tables A.2 and A.3 show the average times and



(A) Branching

(B) Bound 1

(C) Bound 2

(D) Bound 1&2

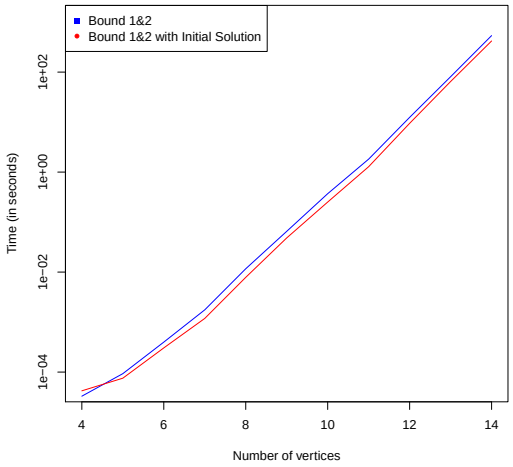
FIGURE 4.4: Linear regression model from B&B, Figure 4.3

standard deviation of the multithreading implementation, respectively. When multithreading is implemented, there is approximately 10% of performance increase on the B&B framework.

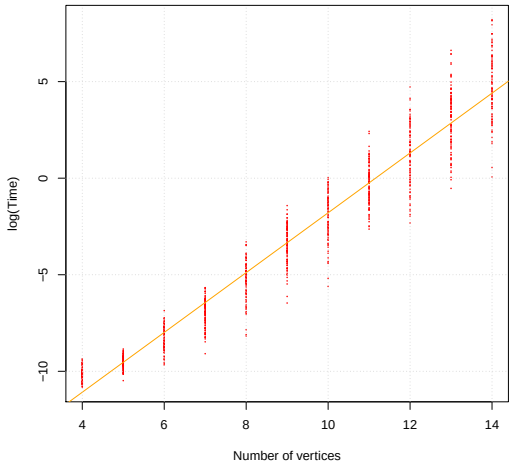
Figure 4.6 (B) shows the linear regression model obtained with logarithm transformation of the dependent variable with $R^2 = 0.8961$.

4.5 Comparison to Pseudo-Polynomial Algorithm

In the following, an experimental comparison of our best approach with the general pseudo-polynomial algorithm described in Section 2.4.1 is shown. As described in the previous section, the approach that shows the best results is the B&B implementation with both bounds, initial solution and multithreading. Figure 4.7 shows

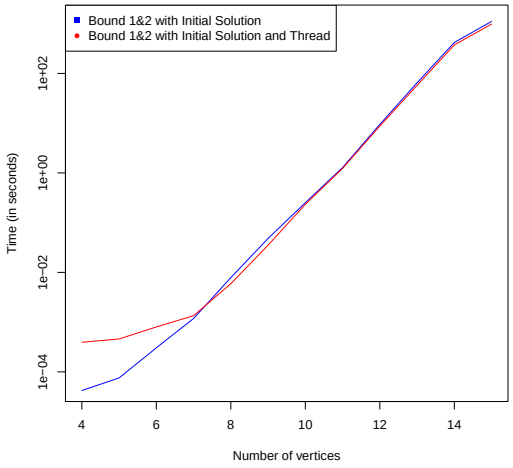


(A) Bounds without and with initial solution average time (in seconds) per number of vertices

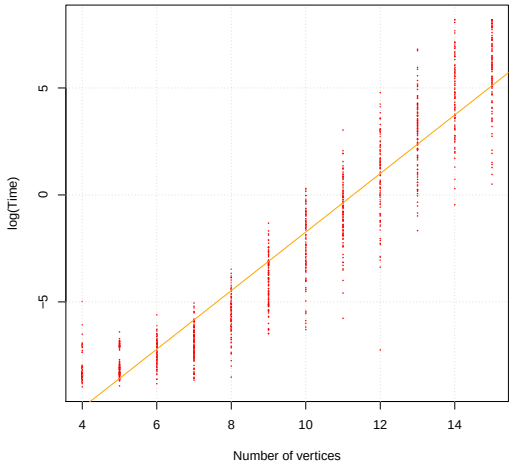


(B) Linear regression model from (A)

FIGURE 4.5: Bounds without and with initial solution (A) and its linear regression model (B)



(A) Initial solution and multithreading average time (in seconds) per number of vertices



(B) Linear regression model from (A)

FIGURE 4.6: Initial solution and multithreading (A) and its linear regression model (B)

the average time executions and the max value M, which denotes the *maximum edge weight* to be generated in the random graphs. In these experiment, values between 2 and 8 were used. This means that each edge has values between 1 and M. Also, 15 vertices were used for all of the generated graphs for this experiment. Is possible to observe that for some M values, the pseudo-polynomial algorithm performs slightly better than the B&B framework. This happens because it depends on the position

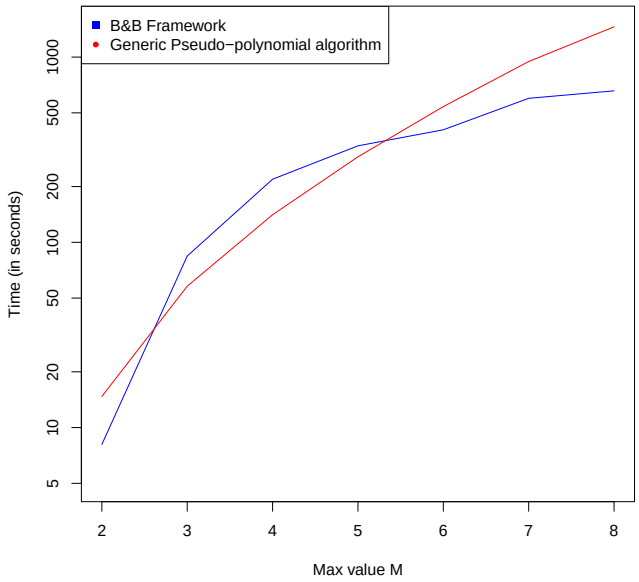


FIGURE 4.7: General pseudo-polynomial algorithm and B&B framework average time (in seconds) per max value M in a graph of 15 vertices

of the best solution in the search space, while the pseudo-polynomial algorithm depends on the value of the best solution itself, since it does an incremental search of the possible values. By increasing M, the running time of the pseudo-polynomial algorithm grows very fast, since it depends on the calculus of the determinant matrix. Table A.1 shows the average and standard deviation for both approaches. It is possible to observe a significant difference between both algorithms when M has values 7 and 8.

4.6 Discussion

In this chapter, the experimental results obtained by the B&B framework were discussed. Through the process, performance increases were verified in all experiments. The combination of the two bounds brings significantly better performance. In addition, starting from a good initial solution improved the performance even further, by approximately 20%. Finally, multithreading brought some improvement, although not as much as expected. This final approach was compared to the general pseudo-polynomial algorithm and presented significant better results as value M increases.

Chapter 5

Conclusion and Future Work

The goal of this dissertation is to implement a Branch-and-Bound (B&B) framework to solve the Min-Max Regret (MMR) Minimum Spanning Tree (MST) problem in a discrete scenario representation. For this, a B&B algorithm with two bounds, an initial solution and multithreading was implemented.

In the last chapter, an experimental analysis was shown. The experimental analysis indicated that our implementation of Prim algorithm is faster than the one available at the Boost library. Also, it is possible to observe a huge performance increase, when comparing the B&B with bounds and the B&B without bounds, i.e., only the branching. When an initial solution is used, an approximately 20% of performance is verified. Finally, when used multithreading with 4 threads, there is better average time execution on the same graphs tested, but this performance increase is close to 10%.

Finally, for the future work, there are some improvements that can be done in this algorithm to obtain better results. One possibility is the incremental calculation of *Bound 2*, for example, by using an hash table. This way it is not necessary to always run Prim algorithm to obtain the lower bound, since the value is already stored. Another possibility is an algorithm restart. When the algorithm is stuck during a lot of time without improving the best solution found, the algorithm may restart and begin in a new position in the search space.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1st edition, 1974.

[2] Martin Aigner and Günter M. Ziegler. *Proofs from the Book*. Springer, 4th edition, 2010.

[3] Hassene Aissi, Cristina Bazgan, and Daniel Vanderpooten. Approximation complexity of min-max (regret) version of shortest path, spanning tree and knapsack. In *Proceedings of the European Symposium on Algorithm*, pages 862–873. Springer, 2005.

[4] Hassene Aissi, Cristina Bazgan, and Daniel Vanderpooten. Pseudo-polynomial algorithms for min-max and min-max regret problems. *International Symposium on OR and Its Applications, Université Paris-Dauphine*, pages 171–178, 2005.

[5] Hassene Aissi, Cristina Bazgan, and Daniel Vanderpooten. Min-max and min-max regret versions of combinatorial optimization problems: a survey. *European Journal of Operational Research*, 197(2):427–438, 2009.

[6] J.F. Barahona and R. Pulleyblank. Exact arborescences, matching and cycles. *Discrete Applied Mathematics*, 16:91–99, 1987.

[7] Christian Bauer, Alexander Frink, Richard B. Kreckel, and et al. GiNaC C++ Libraries. `https://www.ginac.de/`.

[8] Cüneyt F. Bazlamaçci and Khalil S. Hindi. Minimum-weight spanning tree algorithms. a survey and empirical study. *Computers & Operations Research*, 28:767–785, 2001.

[9] Béla Bollobás. *Modern Graph Theory*. Springer, 1st edition, 1998.

[10] Jens Clausen. Branch and bound algorithms - principles and examples. *Department of Computer Science, University of Copenhagen, DK-2100 Copenhagen, Denmark*, 1999.

[11] Beman Dawes, David Abrahams, and et al. Boost C++ Libraries. `http://www.boost.org/`.

[12] J.R. Figueira, L. Paquete, M. Simões, and D. Vanderpooten. Algorithmic improvements on dynamic programming for the bi objective {0,1} knapsack problem. *Computational Optimization and Applications*, 56(1):97–111, 2013.

[13] Harold N. Gabow and Eugene W. Myers. Finding all spanning trees of directed and undirected graphs. *SIAM Journal on Computing*, 7(3):280–287, 1978.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[15] Christos H. Papadimitrou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, Inc., 1st edition, 1998.

[16] John M. Harris, Jeffry L. Hirst, and Michael J. Mossinghoff. *Combinatorics and Graph Theory*. Springer, 2nd edition, 2008.

[17] Adam Kasperski and PawełZieliński. An approximation algorithm for interval data minmax regret combinatorial optimization problems. *Information Processing Letters*, 97(5):177–180, 2006.

[18] P. Kouvelis and G. Yu. *Robust Discrete Optimization and its Applications*. Kluwer Academic Publishers, 1997.

[19] Daniel H. Larkin, Siddhartha Sen, and Robert E. Tarjan. A back-to-basics empirical study of priority queues. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 61–72. Society for Industrial and Applied Mathematics, 2014.

[20] David J. Lilja. *Measuring computer performance: A practitioner's guide*. Cambridge University Press, 1st edition, 2004.

[21] Mariusz Makuchowski. Perturbation algorithm for a minimax regret minimum spanning tree problem. *Operations Research & Decisions*, 24(1):37–49, 2014.

[22] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 1st edition, 2008.

[23] R. Montemanni and L.M. Gambardella. An exact algorithm for the robust shortest path problem with interval data. *Computers & Operations Research*, 31(10):1667 – 1680, 2004.

[24] Travis Oliphant. NumPy library,. `http://www.numpy.org`.

[25] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc, 1st edition, 1995.

[26] David S. Johnson and Michael R. Garey. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1st edition, 1979.

[27] Alexander Schrijver. *A Course in Combinatorial Optimization*. `http://homepages.cwi.nl/~lex/files/dict.pdf`, 2017.

[28] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2013.

[29] G. Yu. Min-max optimization of several classical discrete optimization problems. *Journal of Optimization Theory and Applications*, 98:221–242, 1998.

Appendix A

Tables

	Generic pseudo polynomial ($\bar{x}$)	Generic pseudo polynomial (σ)	B&B Framework ($\bar{x}$)	B&B Framework (σ)
1	0.098	0.026	0.001	0.001
2	14.723	0.573	8.108	40.563
3	57.965	3.098	84.326	198.771
4	140.915	11.375	219.141	518.135
5	289.467	26.243	331.819	722.151
6	540.375	51.108	405.109	696.086
7	946.274	100.143	599.055	947.901
8	1452.317	162.189	657.392	995.465

TABLE A.1: Generic pseudo-polynomial and B&B Framework algorithm executions. Rows indicate max value M.

	Branching	Bound 1	Bound 2	Bound 1 & 2	Bound 1 & 2 Initial Solution	Bound 1 & 2 Initial Solution Thread
4	4.523e-05	3.788e-05	5.158e-05	3.295e-05	4.219e-05	0.001
5	0.001	8.671e-05	0.001	9.298e-05	7.545e-05	0.001
6	0.001	0.001	0.001	0.001	0.001	0.001
7	0.021	0.002	0.002	0.002	0.001	0.001
8	0.303	0.016	0.014	0.012	0.008	0.006
9	4.923	0.106	0.088	0.065	0.048	0.035
10	105.697	0.614	0.588	0.370	0.252	0.234
11	2640.075	4.207	2.756	1.819	1.283	1.232
12	-	33.863	19.179	12.441	9.451	8.739
13	-	296.020	122.769	80.862	64.837	57.694
14	-	-	-	536.582	419.419	374.465
15	-	-	-	-	1111.931	982.766

TABLE A.2: Branch-and-Bound (B&B) execution with 2 scenarios (average). Rows indicate the number of the vertices in the graph.

	Branching	Bound 1	Bound 2	Bound 1 & 2	Bound 1 & 2 Initial Solution	Bound 1 & 2 Initial Solution Thread
4	1.55e-05	1.039e-05	1.564e-05	7.378e-06	1.481e-05	0.001
5	1.116e-05	2.075e-05	5.962e-05	2.568e-05	2.429e-05	0.001
6	0.001	0.001	0.001	0.001	0.001	0.001
7	0.006	0.001	0.001	0.001	0.001	0.001
8	0.026	0.010	0.010	0.009	0.007	0.006
9	0.230	0.060	0.076	0.049	0.043	0.043
10	1.851	0.440	0.510	0.316	0.237	0.292
11	23.870	3.131	3.109	1.938	1.707	2.454
12	-	38.519	25.387	17.288	15.324	16.645
13	-	369.464	204.151	134.958	125.171	135.472
14	-	-	-	780.867	738.971	680.036
15	-	-	-	-	1273.693	1191.521

TABLE A.3: Branch-and-Bound (B&B) execution with 2 scenarios (standard deviation). Rows indicate the number of the vertices in the graph.

	Prim ($\bar{x}$)	**Prim (σ)**	**Boost Prim ($\bar{x}$)**	**Boost Prim (σ)**
300	0.001	1.510e-05	0.001	1.473e-05
400	0.001	1.586e-05	0.002	1.695e-05
500	0.002	0.001	0.003	3.751e-05
600	0.003	0.001	0.004	0.001
700	0.005	4.949e-05	0.006	4.734e-05
800	0.005	0.001	0.009	8.168e-05
900	0.007	0.001	0.011	8.807e-05
1000	0.008	5.981e-05	0.014	6.957e-05
1100	0.010	5.154e-05	0.017	9.012e-05
1200	0.012	5.256e-05	0.019	9.201e-05
1300	0.015	9.235e-05	0.023	0.001
1400	0.018	8.085e-05	0.026	0.001
1500	0.021	0.001	0.030	0.001
1600	0.024	6.262e-05	0.034	0.001
1700	0.027	0.001	0.039	0.001
1800	0.031	0.001	0.043	0.001
1900	0.035	0.001	0.048	0.001
2000	0.039	0.001	0.054	0.001
2100	0.044	0.001	0.060	0.001
2200	0.048	0.001	0.066	0.001
2300	0.052	0.001	0.072	0.001
2400	0.057	0.001	0.079	0.001
2500	0.063	0.001	0.085	0.001
2600	0.068	0.001	0.092	0.001
2700	0.073	0.001	0.099	0.001
2800	0.079	0.001	0.107	0.001
2900	0.085	0.001	0.115	0.001
3000	0.090	0.001	0.122	0.001
3100	0.098	0.001	0.131	0.001
3200	0.105	0.001	0.139	0.001
3300	0.113	0.001	0.148	0.001
3400	0.118	0.001	0.158	0.001
3500	0.125	0.001	0.167	0.001
3600	0.132	0.001	0.177	0.001
3700	0.140	0.001	0.187	0.001
3800	0.147	0.001	0.197	0.001
3900	0.156	0.001	0.208	0.001
4000	0.163	0.001	0.219	0.001

TABLE A.4: Prim and Boost Prim algorithm executions. Rows indicate the number of vertices.

	Prim ($\bar{x}$)	**Prim (σ)**	**Boost Prim ($\bar{x}$)**	**Boost Prim (σ)**
4100	0.173	0.001	0.231	0.001
4200	0.180	0.001	0.243	0.001
4300	0.190	0.001	0.255	0.001
4400	0.199	0.001	0.268	0.001
4500	0.209	0.001	0.281	0.001
4600	0.223	0.002	0.302	0.004
4700	0.239	0.005	0.325	0.007
4800	0.255	0.001	0.349	0.002
4900	0.263	0.004	0.357	0.007
5000	0.265	0.001	0.361	0.001
5100	0.275	0.001	0.375	0.001
5200	0.290	0.003	0.396	0.006
5300	0.305	0.001	0.417	0.001
5400	0.319	0.001	0.433	0.002
5500	0.334	0.001	0.449	0.002
5600	0.338	0.006	0.456	0.008
5700	0.342	0.005	0.456	0.006
5800	0.350	0.001	0.469	0.001
5900	0.369	0.004	0.489	0.006
6000	0.382	0.001	0.504	0.001
6100	0.395	0.001	0.519	0.001
6200	0.408	0.002	0.540	0.006
6300	0.428	0.004	0.573	0.010
6400	0.455	0.006	0.615	0.010
6500	0.470	0.001	0.635	0.004
6600	0.481	0.005	0.653	0.008
6700	0.474	0.001	0.640	0.002
6800	0.489	0.001	0.659	0.002
6900	0.507	0.001	0.678	0.002
7000	0.520	0.001	0.700	0.002
7100	0.532	0.001	0.722	0.002
7200	0.551	0.001	0.741	0.003
7300	0.568	0.001	0.762	0.003
7400	0.582	0.001	0.782	0.003
7500	0.600	0.001	0.805	0.003
7600	0.614	0.001	0.831	0.003
7700	0.632	0.001	0.857	0.003
7800	0.646	0.001	0.876	0.003
7900	0.664	0.001	0.894	0.003
8000	0.678	0.002	0.919	0.004

TABLE A.5: Prim and Boost Prim algorithm executions (cont.). Rows indicate the number of vertices.