Pontifícia Universidade Católica
do Rio de Janeiro

Cleber Oliveira Damasceno

Automated Synthesis of Optimal Decision Trees for Small Combinatorial Optimization Problems

Dissertação de Mestrado

Dissertation presented to the Programa de Pós-graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor : Prof. Thibaut Victor Gaston Vidal
Co-advisor: Prof. Eduardo Uchoa Barboza

Rio de Janeiro
July 2021

Pontifícia Universidade Católica
do Rio de Janeiro

Cleber Oliveira Damasceno

Automated Synthesis of Optimal Decision Trees for Small Combinatorial Optimization Problems

Dissertation presented to the Programa de Pós-graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee.

Prof. Thibaut Victor Gaston Vidal
Advisor
Departamento de Informática – PUC-Rio

Prof. Eduardo Uchoa Barboza
Co-advisor
Engenharia de Produção – UFF

Prof. Marcus Vinicius Soledade Poggi de Aragao
Departamento de Informática – PUC-Rio

Prof. Túlio Ângelo Machado Toffolo
Departamento de Computação – UFOP

Rio de Janeiro, July 28th, 2021

Cleber Oliveira Damasceno

Cleber Oliveira Damasceno holds a Bachelor Degree in Computer Science from the Pontifícia Universidade Católica de Minas Gerais, since 2018.

Bibliographic data

Damasceno, Cleber Oliveira

Automated Synthesis of Optimal Decision Trees for Small Combinatorial Optimization Problems / Cleber Oliveira Damasceno; advisor: Thibaut Victor Gaston Vidal; co-advisor: Eduardo Uchoa Barboza. – Rio de janeiro: PUC-Rio , Departamento de Informática, 2021.

v., 54 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. Otimização combinatória;. 3. Politopos;. 4. Diagramas de Voronoi;. 5. Busca por Vizinho mais Próximo;. 6. Modelo de Árvores de Decisão Lineares.. I. Vidal, Thibaut Victor Gaston. II. Barboza, Eduardo Uchoa. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

Acknowledgments

To my advisor Thibaut Vidal for the stimulus and partnership to carry out this work, and Eduardo Uchoa and Maximilian Schiffer for their ideas and constant support.

To my friends at TECGRAF and PUC-Rio, for all their help, education and support in these years.

To all my friends for always being there for me during all this time.

To CNPq, FAPERJ, and PUC-Rio, for the aids granted, without which this work could have been accomplished.

This study was financed in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) - Process 131104/2019-5.

This study was financed in part by the Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ) - Process E-26/200.289/2020.

This study was financed in part by the Coordenação de Aperfeiçoaento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Damasceno, Cleber Oliveira; Vidal, Thibaut Victor Gaston (Advisor); Barboza, Eduardo Uchoa (Co-Advisor). **Automated Synthesis of Optimal Decision Trees for Small Combinatorial Optimization Problems**. Rio de Janeiro, 2021. 54p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Classical complexity analysis for NP-hard problems is usually oriented to "worst-case" scenarios, considering only the asymptotic behavior. However, there are practical algorithms running in a reasonable time for many classic problems. Furthermore, there is evidence pointing towards polynomial algorithms in the linear decision tree model to solve these problems, although not explored much. In this work, we explore previous theoretical results. We show that the optimal solution for 0-1 combinatorial problems can be found by reducing these problems into a Nearest Neighbor Search over the set of corresponding *Voronoi vertices*. We use the hyperplanes delimiting these regions to systematically generate a decision tree that repeatedly splits the space until it can separate all solutions, guaranteeing an optimal answer. We run experiments to test the size limits for which we can build these trees for the cases of the 0-1 knapsack, weighted minimum cut, and symmetric traveling salesman. We manage to find the trees of these problems with sizes up to 10, 5, and 6, respectively. We also obtain the complete adjacency relations for the skeletons of the knapsack and traveling salesman polytopes up to size 10 and 7. Our approach consistently outperforms the enumeration method and the baseline methods for the weighted minimum cut and symmetric traveling salesman, providing optimal solutions within microseconds.

Keywords

Combinatorial Optimization; Polytopes; Voronoi Diagrams; Nearest Neighbor Search; Linear Decision Tree Model.

Resumo

Damasceno, Cleber Oliveira; Vidal, Thibaut Victor Gaston; Barboza, Eduardo Uchoa. **Síntese Automatizada de Árvores de Decisão Ótimas para Pequenos Problemas de Otimização Combinatória**. Rio de Janeiro, 2021. 54p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A análise de complexidade clássica para problemas NP-difíceis é geralmente orientada para cenários de "pior caso", considerando apenas o comportamento assintótico. No entanto, existem algoritmos práticos com execução em um tempo razoável para muitos problemas clássicos. Além disso, há evidências que apontam para algoritmos polinomiais no modelo de árvore de decisão linear para resolver esses problemas, embora não muito explorados. Neste trabalho, exploramos esses resultados teóricos anteriores. Mostramos que a solução ótima para problemas combinatórios 0-1 pode ser encontrada reduzindo esses problemas para uma Busca por Vizinho Mais Próximo sobre o conjunto de *vértices de Voronoi* correspondentes. Utilizamos os hiperplanos que delimitam essas regiões para gerar sistematicamente uma árvore de decisão que repetidamente divide o espaço até que possa separar todas as soluções, garantindo uma resposta ótima. Fazemos experimentos para testar os limites de tamanho para os quais podemos construir essas árvores para os casos do *0-1 knapsack*, *weighted minimum cut* e *symmetric traveling salesman*. Conseguimos encontrar as árvores desses problemas com tamanhos até 10, 5 e 6, respectivamente. Obtemos também as relações de adjacência completas para os esqueletos dos politopos do *knapsack* e do *traveling salesman* até os tamanhos 10 e 7. Nossa abordagem supera consistentemente o método de enumeração e os métodos *baseline* para o *weighted minimum cut* e *symmetric traveling salesman*, fornecendo soluções ótimas em microssegundos.

Palavras-chave

Otimização combinatória; Politopos; Diagramas de Voronoi; Busca por Vizinho mais Próximo; Modelo de Árvores de Decisão Lineares.

Table of contents

List of symbols

Sets

X – Set of points with 0-1 coordinates
$\phi(X)$ – Set of the transformed points of X with function $\phi(\mathbf{x}) = -2\mathbf{x} + \mathbf{1}$
$S(\sqrt{d})$ – Sphere centered at the origin with radius $\sqrt{d}$
$\mathcal{P}$ – Polytope associated to x, the convex hull of $\phi(X)$
h_{ij} – Bisector hyperplane of $\mathbf{p_i}$ and $\mathbf{p_j}$
V_i – The *Voronoi face* of $\mathbf{p_i}$
$\mathcal{H}$ – Set of all divider hyperplanes with $i < j$
$\mathcal{H}^+$ – Extended set of divider hyperplanes with $i \neq j$
χ – Set of delimiting hyperplanes, a subset of $\mathcal{H}^+$
χ_i – Set of hyperplanes delimiting V_i
χ_η – Set of hyperplanes delimiting the region in node η
$R(\chi)$ – Restricted region delimited by χ
N – Set of decision tree nodes
Δ_η – Set of divider hyperplanes of a node η
Σ_η – Set of solutions contained in the region of η
$\Gamma^L(\eta, h)$ – Set of solutions in η on the left of h
$\Gamma^R(\eta, h)$ – Set of solutions in η on the right of h

Other objects

$\mathcal{S}(\mathcal{P})$ – Skeleton graph of $\mathcal{P}$
$\overrightarrow{\mathbf{pq}}$ – Line segment connecting point $\mathbf{p}$ and $\mathbf{q}$
$\mathcal{V}$ – *Voronoi diagram* associated to $\mathcal{P}$
$\mathcal{D}$ – *Delaunay triangulation* associate to $\mathcal{P}$
η – Node in the decision tree
η_h^L – Node corresponding to the region on the left of h_η
η_h^R – Node corresponding to the region on the right of h_η
h_η – Best hyperplane to be used as comparison in node η

Functions

$f_{\mathbf{c}}(\mathbf{x})$ – Linear function defined as the inner product of $\mathbf{c}$ and $\mathbf{x}$
$h_{ij}(\mathbf{x})$ – Hyperplane function determining the position of a point
$\delta(\eta)$ – Minimal depth of tree that has η as root
$\delta(T)$ – Minimal depth of a tree T
$\hat{\mu}_{S_\eta}(x, y)$ – Heuristic function to evaluate split configurations

Values

m – Number of points in set X
d – Dimension of the point in X
r – Number of delimiting hyperplanes in a region, the size of χ
κ – Maximum number of hyperplanes to be considered in each node
s – Percentage of the hyperplanes to have the position calculated
H – Number of unique divider hyperplanes, the size of $\mathcal{H}$
D_η – Number of divider hyperplanes of a node η
S_η – Number of solutions contained in the region of η
δ_η – Number of comparisons to classify the solutions in η using h_η
d_η – Distance from the root

Relations

$h \mid \chi$ – h divides $R(\chi)$
$h \nmid \chi$ – h divides $R(\chi)$
$\mathbf{x} \prec h$ – $\mathbf{x}$ is on the left of hyperplane h
$\mathbf{x} \succ h$ – $\mathbf{x}$ is on the right of hyperplane h
$\chi \prec h$ – $R(\chi)$ is on the left of hyperplane h
$\chi \succ h$ – $R(\chi)$ is on the right of hyperplane h

But the wisdom of men is small, and the ways of nature are strange, and who shall put a bound to the things which may be found by those who seek for them?

Arthur Conan Doyle, *Lot No. 249.*

1
Introduction

Combinatorial Optimization (CO) problems play an important role in several areas such as operations research, artificial intelligence, and machine learning. Through the lenses of classical complexity analysis, these problems are studied considering their asymptotic behavior in "worst-case" scenarios. Since many of them are NP-hard, obtaining a solution for large instances of these problems is deemed impracticable, due to the time needed for the computation. However, for certain classes of problems, there has been progress on solution methods, making them practical for increasingly larger instances, useful in real-world applications. Moreover, in almost all real-world applications, the optimization goal is not to solve a single problem but to solve several instances of the same problem, with only slight data variations, depending on specific parameters.

Interestingly, even for some of these NP-hard problems, the existence of polynomial-time algorithms in the Random Access Machine (RAM) model (Kolinek, 1987; Meyer auf der Heide, 1984, 1988) shows that there are decision trees of polynomial depth to solve them. Still, these results have been mainly of theoretical interest and did not lead to readily applicable algorithms. In contrast, in this is work, we propose a framework to automatically synthesize such optimal decision trees, [1] i.e., in the Linear Decision Tree (LDT) model of computation (Aho and Hopcroft, 1974).

We push the boundaries of knowledge representation for small CO problems. Many previous studies (Boyd and Cunningham, 1991; Christof et al., 1991; Christof and Reinelt, 1996, 2001) aimed to determine complete descriptions (i.e., linear characterizations of the convex hull of the feasible solutions) of polytopes corresponding to combinatorial problems of small size. In a similar fashion, we pose the question of finding the most compact representation of a decision tree that returns the optimal solution for any input of a given size. As seen in this work, this task can be achieved for small-scale instances of some classical problems.

Our approach maps the set of solutions for a given problem into a *Voronoi diagram*. The solutions are represented as their corresponding *Voronoi regions*, and the problem is reduced to locating which region contains the query point.

[1] Note that when we refer to the optimally of these decision trees, we always mean that for any input, they always give the optimal solution. Optimally in the sense of minimizing the tree depth is not guaranteed.

First, we show that the optimal solution for any problem whose solution set is a subset of the 0-1 hypercube can be found by reducing it into a Nearest Neighbor Search (NNS) over the set of corresponding *Voronoi vertices*. Second, we use the adjacencies of the original problem's polytope to identify the hyperplanes that delimit the *Voronoi regions* and use them to generate a decision tree systematically. The tree uses the hyperplanes to repeatedly consider smaller regions at each level so that the regions in the leaves contain a single solution. We use three problems (0-1 knapsack, weighted minimum cut, and symmetric traveling salesman) to evaluate the method proposed and generate a set of instances used as a benchmark to compare the running time of our method with other known methods for these problems. In summary, we make the following contributions:

- We establish the equivalence between the class of 0-1 combinatorial optimization problems and the nearest neighbor search problem.
- We tackle the task of finding the adjacency relations for skeletons of the 0-1 knapsack and symmetric traveling salesman, describing them for sizes up to 10 and 7.
- We introduce a method of constructing decision trees for a given polytope. The generated tree can solve any instances for that polytope without further changes.
- We find decision trees for the three studied problems with sizes up to 10, 5, and 6, achieving a minimum depth for the sizes up to 4, 3, and 4, respectively. These trees consistently outperform the enumeration method for all problems, and the baseline methods for the weighted minimum cut and symmetric traveling salesman, providing optimal solutions within microseconds.

1.1 Related Work

To our knowledge, very few studies have focused on designing practical and fast algorithms to solve small instances of combinatorial optimization problems. Still, several tasks are closely related to this topic, and they will be discussed further in the next subsections.

The main works regarding this problem, although mainly theoretical, represent an important result in the field of computational complexity, asserting the existence of polynomial time complexity in the decision tree model of computation. On a more practical side, some works approach NP-complete problems of small sizes in hopes of either describing their structure or even

providing ways of solving them. Other methods use these NP-complete problems in subroutines with smaller sizes to obtain the solution for the original problem; there is interest in solving them as fast as possible in these cases. Finally, some studies approach the solutions of optimization problems with the lenses of machine learning, either applying methods to solve the problems or identifying strategies that will help solve them.

1.1.1 Decision Tree Model of Computation

Meyer auf der Heide (1984, 1988) has shown that some NP-complete problems are solvable in polynomial time in the LDT model of computation. Meyer auf der Heide (1984) and Kolinek (1987) use an approach based on identifying the location of a point relative to a set of hyperplanes to show that an algorithm in the decision tree model can solve some NP-complete problems in polynomial time (in the considered computation model). Meyer auf der Heide applies it to solve the n-dimensional knapsack problem, while Kolinek shows it for the cases of the traveling salesman problem, many other shortest path problems, and integer programming. Then, Meyer auf der Heide (1988) uses a construction process to convert programs in the RAM model to their equivalent in the LDT model to expand the number of NP-hard problems of fixed size that can be solved in polynomial time.

A polynomial-time complexity is possible because we do not consider the amount of time necessary to perform arithmetic operations and storage allocation when considering the decision tree model of computation (Aho and Hopcroft, 1974). We only account for the computational complexity associated with the comparisons in the tree nodes. The complexity, in this case, is the maximum number of comparisons that must be made to find a way from the root to one of the leaves (i.e., the tree depth). However, even when such trees have a polynomial depth, it is often the case that they have an exponential number of leaves (and total nodes). Therefore the space needed to allocate all instructions and the time to access the next operation can take an exponential time, as a function of the input size, on a standard computer.

Even though the memory size of modern computers has grown quite large, allowing the construction of such algorithms for several small instances, there have not been many practical advances in this area.

1.1.2 Tiny Problems

Without directly using the previous theoretical results, there are papers already considering problems with small sizes. Chikalov et al. (2013) study the problem of finding totally optimal decision trees, i.e., decision trees that are minimal in regards to space and time complexity, for small instances of boolean functions. In this case, the goal is to minimize three parameters: maximum depth, average depth, and the number of nodes. Note that the problem of finding an optimal decision tree is considered NP-complete (Hyafil and Rivest, 1976). Given the difficulty of the work, the authors managed to find good results for instances with less than six input variables. Still, they only proved total optimality regarding maximum depth and number of nodes for instances with up to four variables and total optimality regarding the three parameters for instances with up to three variables.

As small as it may seem, these results have a big impact when we consider the progressive study of these problems. An example that shows this slow progress is the task of finding "good" representations of the solution space for integer problems, i.e., finding all facet-defining inequalities of their convex hull. Studies on this have been pursued for decades, and each new complete representation (e.g., TSP with 8, 9, or 10 nodes) is seen as a breakthrough and methodological *tour de force*.

While the non-negative and sub tour facets can completely describe the TSP with up to 5 nodes, (Grötschel and Padberg, 1985), the instances with more nodes need more work. Boyd and Cunningham (1991) extended the descriptions for the cases with 6 and 7 nodes presenting two more classes of inequalities and proving that they completely describe the polytopes, but extrapolating their proof for larger sizes would be difficult with the present methods. Following closely, Christof et al. (1991) was able to describe the case with 8 nodes completely using computer code. After that, it took a few years, a lot more computational effort, and a new approach to characterize all facets of the TSP with 9 nodes (Christof and Reinelt, 1996). The last update we find from Christof and Reinelt (1996) presents a complete description for the case with 10 nodes. Decomposition and parallelization techniques were needed to achieve these results.

The TSP example shows us how difficult the process of expanding the boundaries of the knowledge of small CO problems can be. In this work, we focus on a slightly different task, that of producing a decision process, in the form of a decision tree that guarantees the solution's optimality and aims for minimum size.

1.1.3 Decomposition Algorithms

While solving these small problems as best as we can is an interesting topic by itself, there are situations in real-world applications that could greatly benefit from it. That is the case of several decomposition algorithms that have their efficiency connected to the speed to solve the small subproblems.

Many heuristics to solve the Vehicle Routing Problem (VRP) have been developed through the years, and some rely heavily on the quick solution of small CO problems. First, Taillard (1993) presents a heuristic for solving the VRP that decomposes the problem into several subproblems with smaller sizes. Each subproblem, modeled as a TSP, must be solved as quickly as possible to obtain an efficient algorithm. Then, Toffolo et al. (2019) introduces other heuristics, including one that presents the need to solve many instances of a variant of the TSP very quickly during the course of a local search. They even point to a situation where the repeated solution of the TSP seems promising, but the computational effort to do it is prohibitive. Finally, Arnold et al. (2021) introduces a local search strategy based on pattern mining that identifies useful high order moves. The subproblems used for completing solutions (after injecting a frequent pattern) are generally smaller (a few nodes) but must be solved quickly.

For the Minimum Spanning Tree (MST), Pettie and Ramachandran (2000) used optimal decision trees as a sub-component in their proof of the algorithm with best-possible complexity. They found that the smallest possible complexity for solving the problem is equal to the corresponding decision tree complexity.

Though the previous algorithms, especially the one for the MST, often involve fairly large subproblems, the need for fast methods can serve as motivation to improve the development of decision trees for larger problem sizes. The availability of such decision trees will represent a big breakthrough for these methods since the trees can take advantage of the problem's structure to speed up the computation times.

1.1.4 A Machine Learning Approach

Given the hard nature of several CO problems, many state-of-the-art methods for these problems rely on handcrafted heuristics for making some decisions that would be too expensive to compute otherwise. Machine Learning (ML) rises as a candidate to optimize the process in this scenario. Bengio et al. (2021) surveys recent attempts of applying ML to solve CO problems. They point that ML can be used in this context in two ways: (a) given the knowledge

of the optimization algorithm, we can replace some heavy computation steps by fast approximations; (b) without good knowledge, we use the learning process to explore the space of decisions and identify better policies. There are also two possibilities when we consider what we want to "learn". We can use generalization to either learn the solution itself or find a strategy to help solve the actual problem.

Bertsimas and Stellato (2021) use the strategies as the solutions themselves and aim to replace the process of solving the problem by the strategies. Like all classical ML methods, they use a set of possible solutions (restricted to distributions in certain ranges) for the training phase of their method. They use a probabilistic approach to identify the possible strategies and then build the corresponding decision tree to answer the problems with inputs in the chosen range.

Beunardeau et al. (2020) seeks to establish optimal strategies for identifying infected patients in a pool of tests. Unlike the classical ML methods, they do not use learning based on data/solution points; instead, the learning process is derived from the problem structure (they consider all possible solutions to determine the correspondence between input and answer). In this case, with no knowledge of the algorithm, the learning process is used to explore the space fully. The strategies represent the testing procedures that need to be performed to identify the infected cases optimally, and a meta procedure is used to identify all possible procedures and when to use each of them. They present results for the cases where there are 3 or 4 samples to be tested and show that for values greater than 4, the results could not be determined within a reasonable time.

These two types of machine learning (data-based and structure-based) represent an important distinction in achieving optimality. The classical data-based approach allows the exploration of much larger instances of problems. Still, it cannot offer any guarantee of optimality, especially when we consider the entire solution space instead of restricting it to a region like Bertsimas and Stellato (2021). On the other hand, the structure-based gives the possibility of optimally but with the trade-off of a much more limited range of instance sizes because we need to consider all possible solutions.

2
Preliminaries

Let $X \subset \{0,1\}^d$ be a set of m points, $S(\sqrt{d}) = \{\mathbf{x} \in \mathbb{R}^d \mid \|\mathbf{x}\| = \sqrt{d}\}$ the sphere centered at the origin with radius $\sqrt{d}$, and $f_{\mathbf{c}} : X \to \mathbb{R}$ be the linear function defined as $f_{\mathbf{c}}(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x}$ for $\mathbf{c} \in \mathbb{R}^d$.

Regarding the notation, to distinguish the elements of $\mathbb{R}$ from $\mathbb{R}^d$ we will always refer to the latter in bold font. Accordingly, while a and 1 are both real numbers, $\mathbf{a} = (a_1, a_2, \ldots, a_d)$ and $\mathbf{1} = (1, 1, \ldots, 1)$ are vectors in the dimension that makes sense.

The following proposition states a relation between linear transformations that will be useful in the next chapters.

Proposition 2.1 *Let $X \subset \mathbb{R}^d$ and $\phi(\mathbf{x}) = a\mathbf{x} + \mathbf{b}$ a fixed linear map. Then $\forall \mathbf{c} \in \mathbb{R}^d$ we have*

$$\mathbf{x}^* = \arg\min_{\mathbf{x} \in X} f_{\mathbf{c}}(\mathbf{x}) \Leftrightarrow \mathbf{x}^* = \arg\min_{\mathbf{x} \in X} f_{\mathbf{c}}(\phi(\mathbf{x})) \qquad \text{if } a > 0$$

$$\mathbf{x}^* = \arg\min_{\mathbf{x} \in X} f_{\mathbf{c}}(\mathbf{x}) \Leftrightarrow \mathbf{x}^* = \arg\max_{\mathbf{x} \in X} f_{\mathbf{c}}(\phi(\mathbf{x})) \qquad \text{if } a < 0$$

Remark 2.2 *Take $\phi(\mathbf{x}) = -2\mathbf{x} + \mathbf{1}$ and $-\phi(\mathbf{x}) = 2\mathbf{x} - \mathbf{1}$. Then, optimizing $f_{\mathbf{c}}$ over the set $X \subset \{0,1\}^d$ is the same as doing it over $-\phi(X)$, with the additional benefit that both $\phi(X)$ and $-\phi(X)$ are subsets of $S(\sqrt{d})$.*

2.1
Polytopes

Let $\mathcal{P} = \mathbf{Conv}(\phi(X))$ be the *polytope* associated to X, defined as the convex hull of $\phi(X)$. The *skeleton* of $\mathcal{P}$, denoted by $\mathcal{S}(\mathcal{P})$, is the undirected graph whose vertices are the points in $\mathcal{P}$ and the edges are the adjacent pairs of vertices[2]. We denote by $E(\mathcal{S})$ the set of edges in the *skeleton* and note that if $(\mathbf{p_i}, \mathbf{p_j}) \in E(\mathcal{S})$ then $(\mathbf{p_j}, \mathbf{p_i})$ is also in $E(\mathcal{S})$.

Two vertices $\mathbf{p_i}, \mathbf{p_j} \in \mathcal{P}$ are called *adjacent* if, and only if, they share a common edge (i.e., a 1-face of $\mathcal{P}$). The next lemma states a basic result regarding adjacency in polytopes. The first three equivalencies are presented by Gurgel and Wakabayashi (1997); here, we extend the third statement into a stronger version (the last statement).

[2]Unless specified, when we say a point in $\mathcal{P}$ we always mean a point in $\phi(X)$.

Lemma 2.3 (Adjacency on Polytopes) *Let $\mathcal{P} \subset \mathbb{R}^d$ be a polytope and let $\mathbf{p_i}$ and $\mathbf{p_j}$ be two distinct vertices of $\mathcal{P}$. Then the following are equivalent:*

(a) $\mathbf{p_i}$ *and* $\mathbf{p_j}$ *are adjacent*

(b) The point $\frac{1}{2}\mathbf{p_i} + \frac{1}{2}\mathbf{p_j}$ *can be represented uniquely as a convex combination of vertices of* $\mathcal{P}$

(c) Any point on $\overrightarrow{\mathbf{p_i}\mathbf{p_j}}$*, the line segment connecting* $\mathbf{p_i}$ *and* $\mathbf{p_j}$*, can be represented uniquely as a convex combination of vertices of* $\mathcal{P}$

(d) No point on $\overrightarrow{\mathbf{p_i}\mathbf{p_j}}$*, the line segment connecting* $\mathbf{p_i}$ *and* $\mathbf{p_j}$*, can be represented as a convex combination of vertices of* $\mathcal{P}$*, excluding* $\mathbf{p_i}$ *and* $\mathbf{p_j}$

Proof. We will prove that statements (c) and (d) are equivalent.

The direction (c) $\Rightarrow$ (d) is trivial. Since the unique representation of the line segment involves only the vertices $\mathbf{p_i}$ and $\mathbf{p_j}$, it follows that no representation containing the other vertices exclusively is possible.

We prove the direction (c) $\Leftarrow$ (d) using the contrapositive[3]. We state it as: "If there exists a point $\mathbf{p}$ on $\overrightarrow{\mathbf{p_i}\mathbf{p_j}}$ such that it has two distinct convex combinations of vertices of $\mathcal{P}$, then there is a point on $\overrightarrow{\mathbf{p_i}\mathbf{p_j}}$ that can be expressed as a combination of vertices of $\mathcal{P}$, disregarding $\mathbf{p_i}$ and $\mathbf{p_j}$". Furthermore, we will prove that these two points are the same.

Without loss of generality assume that $\mathbf{p} \in \overrightarrow{\mathbf{p_1}\mathbf{p_2}}$. First we will show that if $\mathbf{p}$ has two distinct convex combinations we can also write $\mathbf{p}$ in two other ways: (i) a convex combination using any vertex except $\mathbf{p_1}$; and (ii) a convex combination using any vertex except $\mathbf{p_2}$. Then, combining the two representations with the fact that $\mathbf{p} \in \overrightarrow{\mathbf{p_1}\mathbf{p_2}}$ we will derive a representation that contains neither $\mathbf{p_1}$ nor $\mathbf{p_2}$.

Let $\mathbf{p} \in \overrightarrow{\mathbf{p_1}\mathbf{p_2}}$ have two representations:

$$\mathbf{p} = \alpha_1\mathbf{p_1} + \alpha_2\mathbf{p_2} \qquad \alpha_1 + \alpha_2 = 1 \tag{2-1}$$

$$\mathbf{p} = \sum_{k=1}^{n} \beta_k\mathbf{p_k} \qquad \sum_{k=1}^{n} \beta_k = 1 \tag{2-2}$$

Note that we consider both $\alpha_1 \neq 0$ and $\alpha_2 \neq 0$, since otherwise $\mathbf{p}$ would be a vertex, which has a unique representation. Also $\alpha_1 \neq \beta_1$ and $\alpha_2 \neq \beta_2$ because no vertex can be expressed as a combination of any of the other vertices. Then, isolating $\mathbf{p_1}$ in (2-1) and replacing it into (2-2) gives us:

[3]The outline for this proof was given by M. Lavrov (personal communication, April 26, 2021).

$$\mathbf{p} = \frac{\alpha_1}{\alpha_1 - \beta_1}\left[\left(\beta_2 - \frac{\beta_1\alpha_2}{\alpha_1}\right)\mathbf{p_2} + \sum_{k=3}^{n}\beta_k\mathbf{p_k}\right] \tag{2-3}$$

The equation (2-3) is a linear combination of the vertices $\{\mathbf{p_2}, \mathbf{p_3}, \ldots, \mathbf{p_n}\}$. So we just need to show that the sum of the coefficients is equal to 1.

$$\begin{aligned}
\frac{\alpha_1}{\alpha_1 - \beta_1}\left[\left(\beta_2 - \frac{\beta_1\alpha_2}{\alpha_1}\right) + \sum_{k=3}^{n}\beta_k\right] &= \frac{\alpha_1}{\alpha_1 - \beta_1}\left[-\beta_1 - \frac{\beta_1\alpha_2}{\alpha_1} + \beta_1 + \beta_2 + \sum_{k=3}^{n}\beta_k\right] \\
&= \frac{\alpha_1}{\alpha_1 - \beta_1}\left[-\beta_1 - \frac{\beta_1\alpha_2}{\alpha_1} + \sum_{k=1}^{n}\beta_k\right] \\
&= \frac{\alpha_1}{\alpha_1 - \beta_1}\left[-\beta_1 - \frac{\beta_1\alpha_2}{\alpha_1} + 1\right] \\
&= \frac{\alpha_1}{\alpha_1 - \beta_1}\left[\frac{-\beta_1\alpha_1 - \beta_1\alpha_2 + \alpha_1}{\alpha_1}\right] \\
&= \frac{\alpha_1 - \beta_1(\alpha_1 + \alpha_2)}{\alpha_1 - \beta_1} \\
&= \frac{\alpha_1 - \beta_1}{\alpha_1 - \beta_1} \\
&= 1
\end{aligned}$$

Thus, we just showed that $\mathbf{p}$ has a representation as convex combination of the vertices $\{\mathbf{p_2}, \mathbf{p_3}, \ldots, \mathbf{p_n}\}$. Analogously, we can show that $\mathbf{p}$ also has a representation as convex combination of the vertices $\{\mathbf{p_1}, \mathbf{p_3}, \ldots, \mathbf{p_n}\}$. Now take:

$$\mathbf{p} = \beta_2\mathbf{p_2} + \sum_{k=3}^{n}\beta_k\mathbf{p_k} \qquad \beta_2 + \sum_{k=3}^{n}\beta_k = 1 \tag{2-4}$$

$$\mathbf{p} = \gamma_1\mathbf{p_1} + \sum_{k=3}^{n}\gamma_k\mathbf{p_k} \qquad \gamma_1 + \sum_{k=3}^{n}\gamma_k = 1 \tag{2-5}$$

Now, let $c_1 = \frac{\alpha_1}{\gamma_1}$ and $c_2 = \frac{\alpha_2}{\beta_2}$. Isolating $\mathbf{p_2}$ in (2-4) and $\mathbf{p_1}$ in (2-5) and replacing it into (2-2) gives us:

$$\mathbf{p} = \frac{1}{c_1 + c_2 - 1}\left[\sum_{k=3}^{n}(c_1\gamma_k + c_2\beta_k)\mathbf{p_k}\right] \tag{2-6}$$

Similarly, equation (2-6) is a linear combination of the vertices $\{\mathbf{p_3}, \mathbf{p_4}, \ldots, \mathbf{p_n}\}$ and we only need to show that the sum of coefficients is equals to 1.

$$\begin{aligned}
\frac{1}{c_1+c_2-1}\left[c_1\sum_{k=3}^{n}\gamma_k + c_2\sum_{k=3}^{n}\beta_k\right] &= \frac{1}{c_1+c_2-1}\left[c_1\left(-\gamma_1+\gamma_1+\sum_{k=3}^{n}\gamma_k\right) + c_2\left(-\beta_2+\beta_2+\sum_{k=3}^{n}\beta_k\right)\right] \\
&= \frac{1}{c_1+c_2-1}\left[c_1\left(-\gamma_1+1\right) + c_2\left(-\beta_2+1\right)\right] \\
&= \frac{c_1 - c_1\gamma_1 + c_2 - c_2\beta_2}{c_1+c_2-1} \\
&= \frac{c_1+c_2-\alpha_1-\alpha_2}{c_1+c_2-1} \\
&= \frac{c_1+c_2-1}{c_1+c_2-1} \\
&= 1
\end{aligned}$$

Therefore, $\mathbf{p}$ can be represented as convex combinations of the vertices $\{\mathbf{p_3}, \mathbf{p_4}, \ldots, \mathbf{p_n}\}$, which concludes the proof. ■

Note that, even with this definition of adjacency, the task of checking the adjacency of two vertices might still be complicated. There are polynomial-time algorithms to perform this task for some polytopes related to NP-hard problems. However, for other polytopes, such as the TSP, this task is an NP-complete problem (Papadimitriou, 1978). Therefore, we cannot expect to determine the skeletons for large polytopes so easily.

2.2 Voronoi Diagrams and Delaunay Triangulations

For an arbitrary finite set $Y \subset \mathbb{R}^d$ and any two points $\mathbf{p_i}, \mathbf{p_j} \in Y$, we define the *bisector* of $\mathbf{p_i}$ and $\mathbf{p_j}$ as

$$h_{ij} = \{\mathbf{x} \in \mathbb{R}^d \mid \|\mathbf{p_i} - \mathbf{x}\| = \|\mathbf{p_j} - \mathbf{x}\|\}$$

and the half-space containing $\mathbf{p_i}$ as

$$D_{ij} = \{\mathbf{x} \in \mathbb{R}^d \mid \|\mathbf{p_i} - \mathbf{x}\| < \|\mathbf{p_j} - \mathbf{x}\|\}$$

which is separated from the half-space D_{ji} containing $\mathbf{p_j}$ by h_{ij}.

Following Fortune (2017), we denote by

$$V_i = \bigcap_{\mathbf{p_j} \in Y, \mathbf{i} \neq \mathbf{j}} D_{ij}$$

the *Voronoi face* of $\mathbf{p_i}$ with respect to Y, the set of all points in $\mathbb{R}^d$ strictly closer to $\mathbf{p_i}$ than to any other point in Y. The boundary of a *Voronoi face* consists of *Voronoi edges* equidistant from two points and *Voronoi vertices*

equidistant from at least three points. Finally, the *Voronoi diagram* of Y, $\mathcal{V}(Y)$, is defined as the union of all *Voronoi faces*.

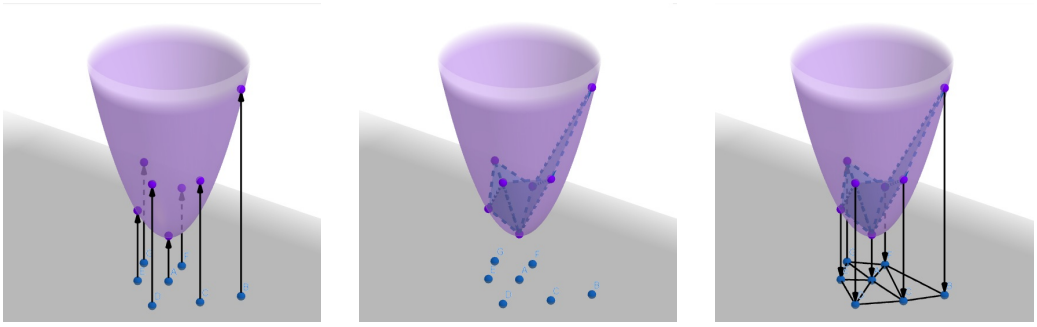
Let $B_i = \{\mathbf{p_j} \mid h_{ij}$ is the boundary of $V_i\}$. Then we can alternatively define the *Voronoi faces* as

$$V_i = \bigcap_{\mathbf{p_j} \in B_i} D_{ij} \tag{2-7}$$

which is a shorter representation, since $|B_i| < |Y|$.

The dual of a *Voronoi diagram*, the *Delaunay triangulation*, can be defined in terms of a convex hull in a higher dimension. Consider the *lifting map* $\lambda : \mathbb{R}^d \to \mathbb{R}^{d+1}$ defined by $\lambda(\mathbf{x}) = (x_1, x_2, \ldots, x_d, x_1^2 + x_2^2 + \cdots + x_d^2)$. Now take $\mathcal{Q} = \mathbf{Conv}(\lambda(Y))$, then the *Delaunay triangulation* of Y, $\mathcal{D}(Y)$, is exactly the orthogonal projection of $\mathcal{Q}$ into $\mathbb{R}^d$.

Figure 2.1 shows the effect of the *lifting map* in 2D. We can see that when the points are projected back into the original space, we add a *Delaunay edge* between the adjacent points on the convex hull.



(a) Lifting (b) Convex hull (c) Projection

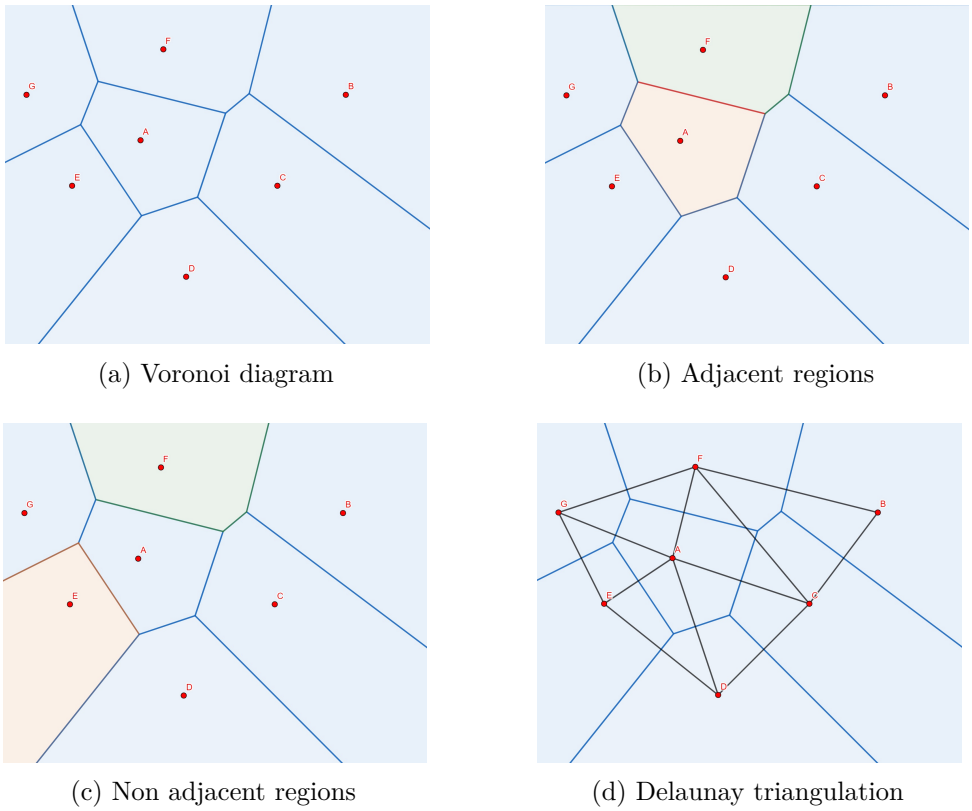
Figure 2.1: The *lifting map*: (a) the projection of Y into the unit paraboloid; (b) the computation of the lower convex hull of the projected points; (c) the projection back into the original space.

The *Delaunay edges*, which are the adjacent pairs of $\mathcal{Q}$, have a one-one correspondence with the *Voronoi edges*. So, we say that two regions V_i and V_j share a common *Voronoi edge* (called *adjacent*) if, and only if, there is a *Delaunay edge* connecting $\mathbf{p_i}$ with $\mathbf{p_j}$.

Figure 2.2 shows an example of the definitions for a better understanding. We can easily notice that adjacent regions, as A and F in 2.2b, have a common boundary (denoted by the red line) and have an edge between them in the Delaunay triangulation (2.2d). In contrast, nonadjacent regions (E and F in 2.2c) have no common boundaries and no edges connecting them.

Remark 2.4 *Considering* $Y = \phi(X)$ *we have* $Y \subset S(\sqrt{d})$ *and the set* h_{ij} *is also the hyperplane through the center of the segment* $\overrightarrow{\mathbf{p_i}\mathbf{p_j}}$ *with normal* $\vec{\mathbf{n}} = \mathbf{p_j} - \mathbf{p_j}$.

The segment $\overrightarrow{\mathbf{p_i p_j}}$ *is a chord of* $S(\sqrt{d})$ *so it follows that* h_{ij} *passes trough the origin, being alternatively defined as* $h_{ij} := \vec{\mathbf{n}} \cdot \mathbf{x} = 0$ *for* $\mathbf{x} \in \mathbb{R}^d$.



(a) Voronoi diagram

(b) Adjacent regions

(c) Non adjacent regions

(d) Delaunay triangulation

Figure 2.2: Example of a *Voronoi diagram*: (a) the original *Voronoi diagram*; (b) two adjacent regions are highlighted; in red, we can see their shared boundary; (c) two nonadjacent regions are highlighted; they do not share a boundary; (d) the corresponding *Delaunay triangulation*.

2.2.1 Voronoi Diagram of a Polytope

Let $\mathcal{V} = \mathcal{V}(\phi(X))$ be the unique *Voronoi diagram* associated to $\mathcal{P}$ with its correspondent *Delaunay triangulation* $\mathcal{D} = \mathcal{D}(\phi(X))$. The main result in this subsection regards the existing relation between $\mathcal{S}(\mathcal{P})$, the *skeleton* of $\mathcal{P}$, and $\mathcal{D}$. To achieve this, we need the following proposition presented by Ziegler (1995).

Proposition 2.5 (Affine Isomorphism) *Two polytopes* $\mathcal{P} \subset \mathbb{R}^d$ *and* $\mathcal{Q} \subset \mathbb{R}^e$ *are affinely isomorphic, denoted by* $\mathcal{P} \cong \mathcal{Q}$, *if there is an affine map* $f : \mathbb{R}^d \to \mathbb{R}^e$ *that is a bijection between the points of the two polytopes. (Note that such a map need not be injective or surjective on the spaces* $\mathbb{R}^d$ *and* $\mathbb{R}^e$*).*

Remark 2.6 (Affine Maps) *Let $Y \in \mathbb{R}^d$ and $Z \in \mathbb{R}^e$. We say that $f : Y \to Z$ is an affine map if there exists a* ***linear map*** $m_f : \mathbb{R}^d \to \mathbb{R}^e$ *such that $m_f(\mathbf{x} - \mathbf{y}) = f(\mathbf{x}) - f(\mathbf{y})$ for all $\mathbf{x}, \mathbf{y} \in Y$.*

Now we are ready to state the theorem that establishes the equivalence we want. Associated with Lemma 2.3 and Equation (2-7), this theorem provides a way to compute the *Voronoi faces* for a polytope, which will be explored in the next chapter.

Theorem 2.7 (Equivalence of Polytopes and Voronoi Diagrams) *Let X as defined previously, with its associated polytope $\mathcal{P}$. Then, $\mathcal{S}(\mathcal{P})$ is equivalent to $\mathcal{D}$, in the sense that $(\mathbf{p_i}, \mathbf{p_j})$ is an edge in $\mathcal{S}(\mathcal{P})$ if, and only if, $(\mathbf{p_i}, \mathbf{p_j})$ is a Delaunay edge.*

Proof. To prove this result we need to show that two points $\mathbf{p_i}, \mathbf{p_j} \in \mathcal{P}$ are adjacent in $\mathcal{P} \subset \mathbb{R}^d$ if, and only if, $\lambda(\mathbf{p_i})$ and $\lambda(\mathbf{p_j})$ are adjacent in $\mathcal{Q} \subset \mathbb{R}^{d+1}$. To achieve this we will prove a stronger result; we will show that $\mathcal{P} \cong \mathcal{Q}$, thus their structure (along with their adjacency relations) is equivalent.

Reminding that we are working with points in $\phi(X) \subset S(\sqrt{d})$, our candidate for the affine map is already given by the definition of $\mathcal{Q}$ as $\lambda(\mathbf{x}) = (x_1, x_2, \ldots, x_d, d)$. We have to prove that: (a) λ is an affine map and (b) it is a bijection between $\mathcal{P}$ and $\mathcal{Q}$. Since these conditions are quite clear from the definition of λ, we will not go into too much detail.
(a) Consider $m_\lambda(\mathbf{x}) = (x_1, x_2, \ldots, x_d, 0)$. Firstly, m_λ is a linear map:

$$\begin{aligned} m_\lambda(c\mathbf{x} + \mathbf{y}) &= (c \cdot x_1 + y_1, c \cdot x_2 + y_2, \ldots, c \cdot x_d + y_d, 0) \\ &= c \cdot (x_1, x_2, \ldots, x_d, 0) + (y_1, y_2, \ldots, y_d, 0) \\ &= c \cdot m_\lambda(\mathbf{x}) + m_\lambda(\mathbf{y}) \end{aligned}$$

We can also easily verify that

$$\begin{aligned} m_\lambda(\mathbf{x} - \mathbf{y}) &= (x_1 - y_1, x_2 - y_2, \ldots, x_d - y_d, 0) \\ &= (x_1 - y_1, x_2 - y_2, \ldots, x_d - y_d, d - d) \\ &= (x_1, x_2, \ldots, x_d, d) - (y_1, y_2, \ldots, y_d, d) \\ &= \lambda(\mathbf{x}) - \lambda(\mathbf{y}) \end{aligned}$$

Therefore, based on Remark 2.6, we have shown that λ is an affine map.

(b) The surjectivity of λ is trivial, because of the definition of $\mathcal{Q}$. To show injectivity take $\mathbf{x}, \mathbf{y} \in \mathcal{P}$ and assume that $\lambda(\mathbf{x}) = \lambda(\mathbf{y})$. Then,

$$\begin{aligned}\lambda(\mathbf{x}) = \lambda(\mathbf{y}) &\Rightarrow (x_1, x_2, \ldots, x_d, d) = (y_1, y_2, \ldots, y_d, d) \\ &\Rightarrow x_i = y_i, 1 \leq i \leq d \\ &\Rightarrow \mathbf{x} = \mathbf{y}\end{aligned}$$

From (a) and (b), we have shown the conditions established in Proposition 2.5. Therefore, the two polytopes are *affinely isomorphic*, and *skeleton* edges are equivalent to *Delaunay edges*. ■

2.3 Nearest Neighbor Search

Several combinatorial optimization problems can be thought of as search problems for the optimum of a linear functional over the vertices of a convex polytope. It happens that when the vertices are points of $\{0,1\}^d$ and our goal is minimizing the linear function, we can reduce our problem to a Nearest Neighbor Search. The Nearest Neighbor Search problem is defined as the task of finding the nearest point to a query point $\mathbf{c}$ among m points in $\mathbb{R}^d$, and it is closely related to *Voronoi diagrams*.

The next theorem will show the map we use to transform one problem into another, and it will prove the equivalence between them. Exploring this relation will give us a simple approach to search for the optimal vertex, as we will show in the next chapter.

Theorem 2.8 (Minimization and Nearest Neighbor Search) *Let X as defined previously and $\mathbf{c} \subset \mathbb{R}^d$. Then the following are equivalent:*

(a) $\mathbf{x}^* = \arg\min_{\mathbf{x} \in X} f_c(\mathbf{x})$

(b) $\phi(\mathbf{x}^*) = \arg\min_{\mathbf{x} \in \phi(X)} \|\mathbf{c} - \mathbf{x}\|^2$

Proof. For a fixed $\mathbf{c}$ and recalling that $\|\mathbf{x}\| = \sqrt{d}$ for all $\mathbf{x} \in \phi(X)$ we can apply Proposition 2.1:

$$\begin{aligned}\mathbf{x}^* = \arg\min_{\mathbf{x} \in X} f_{\mathbf{c}}(\mathbf{x}) &\Leftrightarrow \mathbf{x}^* = \arg\max_{\mathbf{x} \in X} f_{\mathbf{c}}(\phi(\mathbf{x})) \\ &\Leftrightarrow \mathbf{x}^* = \arg\min_{\mathbf{x} \in X} -f_{\mathbf{c}}(\phi(\mathbf{x})) \\ &\Leftrightarrow \phi(\mathbf{x}^*) = \arg\min_{\mathbf{x} \in \phi(X)} -f_{\mathbf{c}}(\mathbf{x}) \\ &\Leftrightarrow \phi(\mathbf{x}^*) = \arg\min_{\mathbf{x} \in \phi(X)} \|\mathbf{c} - \mathbf{x}\|^2\end{aligned}$$

■

Remark 2.9 *We can conclude from this equivalence that a point* $\mathbf{p_i} \in X$ *is the optimal solution for the minimization of* $f_{\mathbf{c}}(\mathbf{x})$ *if, and only if,* $\mathbf{c} \in V_i$*, where* V_i *is the associated Voronoi face with respect to* $\phi(X)$*.*

3
Construction

Given a set X, we compute $\mathcal{P}$ with its skeleton as described in Section 2.1. Then, for each pair of vertices, $(\mathbf{p_i}, \mathbf{p_j})$, we define the *bisector* h_{ij} as shown in Remark 2.4 with the associated function $h_{ij}(\mathbf{x}) = \vec{\mathbf{n}} \cdot \mathbf{x}$. Also, we define the sets of hyperplanes $\mathcal{H} = \{h_{ij} \mid i < j\}$ and $\mathcal{H}^+ = \{h_{ij} \mid i \neq j\}$, with $H = |\mathcal{H}|$.

For $\chi \subset \mathcal{H}^+$ with $|\chi| = r$, we call $R(\chi) = \{\mathbf{x} \mid h(\mathbf{x}) \geq 0 \quad \forall h \in \chi\}$ the r-restricted region of χ. Given a set χ and a hyperplane $h \in \mathcal{H}$, we say that h *divides* $R(\chi)$ and denote it by $h \mid \chi$ if there exits $\mathbf{a}, \mathbf{b} \in R(\chi)$ such that $h(\mathbf{a}) > 0$ and $h(\mathbf{b}) < 0$. Otherwise we say that h *does not* divide $R(\chi)$ and denote it by $h \nmid \chi$. Additionally, we say that a point $\mathbf{x}$ is *on the right* of h, notated by $\mathbf{x} \succ h$, if $h(\mathbf{x}) \geq 0$, while if $h(\mathbf{x}) < 0$ we say that $\mathbf{x}$ is *on the left* of h, notated by $\mathbf{x} \prec h$. Therefore, when $h \nmid \chi$ we can further classify their relation saying that $\chi \succ h$ ($\chi \prec h$) if for all $\mathbf{x} \in R(\chi)$ we have $\mathbf{x} \succ h$ ($\mathbf{x} \prec h$).

For each point $\mathbf{p_i} \in \mathcal{P}$ we define χ_i, the set of limiting hyperplanes, as

$$\chi_i = \{h_{ij} \mid (\mathbf{p_i}, \mathbf{p_j}) \in E(\mathcal{S})\}$$

and, consequently, we have $R(\chi_i) = \mathbf{cl}(V_i)$, where $\mathbf{cl}(\cdot)$ denotes the closure operator. Note that for a given edge $(\mathbf{p_i}, \mathbf{p_j})$ the hyperplane h_{ij} is in χ_i whereas the hyperplane h_{ji} is in χ_j. While $h_{ij} = h_{ji}$ as sets, when regarding their normal form they present opposite normal vectors and we say that $h_{ij} = -h_{ji}$.

Our goal is to define a decision tree that uses the hyperplanes in $\mathcal{H}$ as comparisons in each node. This tree must be constructed in a way that for any point $\mathbf{c} \in \mathbb{R}^d$ defining the function $f_{\mathbf{c}}$, the traversal of tree should provide the region V_i that contains $\mathbf{c}$.

We can easily construct such comparisons with a simple enumeration-like process, where in each node we consider the hyperplane h_{ij}, with $\mathbf{p_i}$ and $\mathbf{p_j}$ being possible solutions. In this way, at each step we decide whether the point $\mathbf{c}$ is closer to $\mathbf{p_i}$ or $\mathbf{p_j}$. Then the derived node on the right will have all previous solutions except $\mathbf{p_i}$, and the node on the left will have all previous solutions except $\mathbf{p_j}$. This process gives an initial upper bound of $O(n)$ hyperplane comparisons to solve the problem. To achieve the optimal solution with fewer comparisons, we construct a decision tree that will exploit the results of the previous comparisons to refine the search space.

3.1 The Decision Tree

A node η in the tree can be defined as the tuple $(\chi_\eta, h_\eta, \delta_\eta, \eta_h^L, \eta_h^R)$ with five elements: (a) the hyperplanes delimiting its region, with $R(\chi_\eta) \neq \varnothing$; (b) the best known hyperplane to be used as comparison; (c) the minimal known number of comparisons to classify the solutions using the chosen hyperplane; (d) the node corresponding to the region on the left of the hyperplane; and (e) the node corresponding to the region on the right of the hyperplane. Associated to each node η we also have the following properties:

- $\Delta_\eta = \{h \in \mathcal{H} \mid h \mid \chi_\eta\}$, the set of divider hyperplanes;
- $\Sigma_\eta = \{\mathbf{p_i} \in \mathcal{P} \mid R(\chi_\eta) \cap V_i \neq \varnothing\}$, the set of contained solutions;
- $D_\eta = |\Delta_\eta|$, the number of divider hyperplanes;
- $S_\eta = |\Sigma_\eta|$, the number of contained solutions;
- $d_\eta = |\chi_\eta|$, the distance from the root.

Initially, we can create a node with just the set of hyperplanes χ_η. In this case we write $\eta = (\chi_\eta)$ to say that $\eta = (\chi_\eta, h_\infty, \infty, \eta_\infty, \eta_\infty)$ where h_∞ and η_∞ are placeholders to indicate that those elements are not defined yet. When $\chi_\eta = \varnothing$ we say that $R(\chi_\eta) = \mathbb{R}^d$ and we denote the associated node by $\eta_0 = (\varnothing)$.

New nodes can be obtained from a given node η and a hyperplane h in two ways: (a) a right derivation, generating $\eta_h^R = (\chi_\eta \cup \{h\})$, the right node derived through h; and (b) a left derivation, generating $\eta_h^L = (\chi_\eta \cup \{-h\})$, the left node derived through h. When no solution is contained in the region derived from a node we also denote that by the special node η_∞.

Therefore we have three types of nodes: (a) unexplored nodes of the form $(\chi_\eta, h_\infty, \infty, \eta_\infty, \eta_\infty)$; (b) explored leaf nodes of the form $(\chi_\eta, h_\infty, 0, \eta_\infty, \eta_\infty)$; and (c) explored internal nodes $(\chi_\eta, h_\eta, \delta_\eta, \eta_h^L, \eta_h^R)$, with both η_h^L and η_h^R being explored nodes, and $\delta_\eta < \infty$.

We say that a set of nodes N defines a valid tree $T = T(N)$ if the following conditions are true:

- $\eta_0 \in N$;
- $\forall \eta \in N$, $\delta_\eta < \infty$;
- $\forall \eta \in N$, either $\eta_h^L \in N$ or $\eta_h^L = \eta_\infty$;
- $\forall \eta \in N$, either $\eta_h^R \in N$ or $\eta_h^R = \eta_\infty$;

Finally, for any given tree T we define its *depth* as $d(T) = \max_{\eta \in N} d_\eta$ where η is a node that be be reached from the root. Among all valid trees $\mathcal{T}$ our goal is two find the minimal tree T^* based on comparisons over the hyperplanes in $\mathcal{H}$ such that $T^* = \arg\min_{T \in \mathcal{T}} d(T)$.

3.2 The Algorithm

To construct the optimal decision tree we apply a Dynamic Programming approach, considering the function $\delta : N \to \mathbb{N}$, the minimal depth of a node η defined as:

$$\delta(\eta) = \begin{cases} 0 & \text{if } S_\eta = 1 \\ 1 & \text{if } S_\eta = 2 \\ \min_{h \in \Delta_\eta} \max\{\delta(\eta_h^L), \delta(\eta_h^R)\} + 1 & \text{otherwise} \end{cases} \tag{3-1}$$

Note that for any pair of points $\mathbf{p_i}, \mathbf{p_j} \in \Sigma_\eta$ with $i < j$, $h_{ij} \in \Delta_\eta$, which allows us to solve all cases, as in the worst case the enumeration method mentioned in the beginning of the chapter can be applied. Also, we know that $R(\chi_{\eta_h^R}) \subset R(\chi_\eta)$ and $R(\chi_{\eta_h^L}) \subset R(\chi_\eta)$ because we only consider the hyperplanes in Δ_η. This two conditions guarantees us that the algorithm will terminate, and we will find the optimal solution.

Using these functions, the basic process to compute the optimal tree is described in Algorithm 1. We consider the set of nodes N as a record in memory indexed by the corresponding regions. Associated with it we have the functions `MemorizeNode`(η) and `UpdateNode`(η). `MemorizeNode`(η) will update the values of $h_\eta, \delta_\eta, \eta_h^L, \eta_h^R$ for the region χ_η in N, or it will create a new record for the region χ_η if none is found. `UpdateNode`(η) will check whether there is a record for the given region in N and retrieve the information stored into the node η. Then we have $d(T^*) =$ `BuildTree`(η_0) and the tree can be obtained from N by starting at node η_0 and following their children.

3.3 Implementation

To implement the previous algorithm, we take advantage of previous values using additional structures that allow us to compute the sets and functions described in the earlier sections more efficiently. We also avoid keeping many copies of points and hyperplanes by storing them in a single place and keeping only the corresponding indexes.

We represent the regions as a list of the indexes of the hyperplanes that delimit it. Positive values indicate the half-space *on the right* of the hyperplane, whereas negative values indicate the half-space *on the left* of it. We also enforce an ordering of this list considering the absolute value of the indexes. Therefore, the list $[1, -3, 4, 5]$ is the unique representation for the region whose set of limiting hyperplanes is $\chi = \{h_1, -h_3, h_4, h_5\}$.

Algorithm 1: Calculate minimal tree

Input: A node η
Output: δ_η, the minimal depth of η

```
1  function BuildTree(η):
2      UpdateNode(η)
3      if δ_η < ∞ then
4          return δ_η
5      Compute Σ_η and Δ_η
6      if S_η = 1 then
7          δ_η ← 0
8          MemorizeNode(η)
9          return 0
10     else if S_η = 2 then
11         δ_{η^L_{h_ij}} ← 0
12         δ_{η^R_{h_ij}} ← 0
13         h_η ← h_ij
14         δ_η ← 1
15         MemorizeNode(η^L_{h_ij})
16         MemorizeNode(η^R_{h_ij})
17         MemorizeNode(η)
18         return 1
19     else
20         upperBound ← S_η − 1
21         lowerBound ← ceil(log(S_η))
22         foreach h ∈ Δ_η do
23             dLeft ← BuildTree(η^L_h)
24             if dLeft < δ_η then
25                 dRight ← BuildTree(η^R_h)
26                 if dRight < δ_η then
27                     h_η ← h
28                     δ_η ← max(dLeft, dRight) + 1
29                     upperBound ← δ_η
30             if upperBound = lowerBound then
31                 break
32         MemorizeNode(η)
33         return δ_η
```

With this representation we can store the set of nodes N as a hash table. We use the hash function BLAKE2 (Aumasson et al., 2013) to generate the hash digests of the lists (as strings). This hash function is faster than MD5, SHA-1, SHA-2, and SHA-3 and has been adopted by many projects due to its high speed, security, and simplicity. In our case, it will work as a quick way to consult the previously computed values.

The next subsections will go into more details of the implementation for the steps described before: (a) how we determine the adjacencies on a polytope; (b) how we decide whether a given hyperplane h *divides* the region $R(\chi)$; (c) how we keep track of Σ_η and Δ_η when deriving new nodes; and (d) how we sort Δ_η to improve the computations.

3.3.1 Checking adjacencies

Consider the set $\Lambda_{ij} = \{\lambda \in \mathbb{R}^m \mid \lambda \geq 0\}$ subjected to the additional following constraints:

$$\sum_{k \neq i,j} \lambda_k \mathbf{p_k} = \lambda_i \mathbf{p_i} + \lambda_j \mathbf{p_j} \qquad \mathbf{p_k} \in \mathcal{P} \tag{3-2}$$

$$\sum_{k \neq i,j} \lambda_k = 1 \qquad 1 \leq k \leq m \tag{3-3}$$

$$\lambda_i + \lambda_j = 1 \qquad 1 \leq i < j \leq m \tag{3-4}$$

Based on the last statement of Lemma 2.3 we know that $\mathbf{p_i}$ and $\mathbf{p_j}$ will be *adjacent* if, and only if, $\Lambda_{ij} = \varnothing$. Therefore, to determine the adjacencies on a polytope $\mathcal{P}$ we apply Algorithm 2, which uses a Linear Programming (LP) model, to obtain a list of all adjacent pairs.

Algorithm 2: Determine adjacencies

Input: Polytope $\mathcal{P}$
Output: Array containing the adjacent pairs

```
1 function DetermineAdjacencies(P):
2     adjacencies ← []
3     for i ← 0 to m do
4         for j ← i + 1 to m do
5             Create a LP with restrictions (3-2), (3-3), and (3-4)
6             if problem does not have a feasible solution then
7                 Insert (p_i, p_j) into adjacencies
8     return adjacencies
```

3.3.2 Determining the position of a region

As defined previously, to check whether $h \mid \chi$ we need to verify the existence of two distinct point $\mathbf{a}, \mathbf{b}$ in $R(\chi)$ located in opposite sides of h. In order words, these points must satisfy the following equations:

$$h_\chi(\mathbf{a}) \geq 0 \qquad \forall h_\chi \in \chi \tag{3-5}$$

$$h_\chi(\mathbf{b}) \geq 0 \qquad \forall h_\chi \in \chi \tag{3-6}$$

$$h(\mathbf{a}) \geq \epsilon \tag{3-7}$$

$$h(\mathbf{b}) \leq \epsilon \tag{3-8}$$

We accomplish this by dividing the intersection problem into two subproblems: `PositionR`$(h,\ \chi)$ and `PositionL`$(h,\ \chi)$. For each subproblem we create the corresponding LP model that we solve for feasibility. Algorithm 3 shows the two subproblems. Based on these subproblems we say that:

- $h \mid \chi$ if `PositionR`$(h,\ \chi)$ = True and `PositionL`$(h,\ \chi)$ = True
- $\chi \succ h$ if `PositionR`$(h,\ \chi)$ = True and `PositionL`$(h,\ \chi)$ = False
- $\chi \prec h$ if `PositionR`$(h,\ \chi)$ = False and `PositionL`$(h,\ \chi)$ = True

Algorithm 3: Determine positions

Input: Hyperplane h and region limiting hyperplanes χ
Output: Whether there is a point satisfying the constraints

```
1 function PositionR(h, χ):
2     Create a LP with restrictions (3-5) and (3-7)
3     return whether problem has a feasible solution
4 function PositionL(h, χ):
5     Create a LP with restrictions (3-6) and (3-8)
6     return whether problem has a feasible solution
```

Note that each linear programming model contains d variables (the space dimension) and $r+1$ equations, where $r \leq H$. We denote by $\ell(d, H)$ the number of operations to solve each subproblem, in the worst case.

3.3.3
Selecting contained solutions and divider hyperplanes

Given χ_η, the set of hyperplanes delimiting a node's region, recall that Σ_η is the set of solutions contained in $R(\chi_\eta)$, with $S_\eta = |\Sigma_\eta|$. Our goal is to compute the elements in this set in a way that we avoid unnecessary calculations. To achieve this, we show how to obtain the set of contained solutions for the derived nodes, $\Sigma_{\eta_h^L}$ and $\Sigma_{\eta_h^R}$, given Σ_η.

We want to determine the positions of the solutions in Σ_η relative to a hyperplane $h \in \Delta_\eta$, the set of dividing hyperplanes. Thus, for each solution $\mathbf{p_i} \in \Sigma_\eta$ we use the method described in Subsection 3.3.2 to say that:

- h *divides* V_i in $R(\chi_\eta)$, notated by $h \,|_{\chi_\eta}\, V_i$, if $h \mid (\chi_\eta \cup \chi_i)$
- V_i is *on the right* of h in $R(\chi_\eta)$, notated by $V_i \succ_{\chi_\eta} h$, if $(\chi_\eta \cup \chi_i) \succ h$
- V_i is *on the left* of h in $R(\chi_\eta)$, notated by $V_i \prec_{\chi_\eta} h$, if $(\chi_\eta \cup \chi_i) \prec h$

Now we define two functions to determine the set of solutions on each side of h in $R(\chi_\eta)$. $\Gamma^L : N \times \mathcal{H} \to 2^\Sigma$ gives the set of solutions of the left of h, while $\Gamma^R : N \times \mathcal{H} \to 2^\Sigma$ gives the set of solutions on the right. These functions are defined as:

$$\Gamma^L(\eta, h) = \{\mathbf{p_i} \in \Sigma_\eta \mid V_i \prec_{\chi_\eta} h\} \quad \text{and} \quad \Gamma^R(\eta, h) = \{\mathbf{p_i} \in \Sigma_\eta \mid V_i \succ_{\chi_\eta} h\}$$

Using these functions we keep track of the possible solutions in each node more easily. Instead of computing the set Σ_η for each node we start by defining $\Sigma_{\eta_0} \coloneqq \phi(X)$. Then each time we derive the nodes η_h^R and $d\eta_h^L$ from η we say that $\Sigma_{\eta_h^L} \coloneqq \Sigma_\eta \setminus \Gamma^R(\eta, h)$ and $\Sigma_{\eta_h^R} \coloneqq \Sigma_\eta \setminus \Gamma^L(\eta, h)$, with $S_{\eta_h^L} = |\Sigma_{\eta_h^L}|$ and $S_{\eta_h^R} = |\Sigma_{\eta_h^R}|$. Note that, if $h \,|_{\chi_\eta}\, V_i$ we have $\mathbf{p_i}$ as a possible solution on both nodes.

The computed values of the Γ functions can also determine the set of divider hyperplanes. If one side of a hyperplane h contains all the solutions and the other contains none, this hyperplane does not intersect the region defined by χ_η. So we alternatively say that $h \mid \chi_\eta$ if, and only if, both $S_{\eta_h^L}$ and $S_{\eta_h^R}$ are greater than zero. Start with $\Delta_{\eta_0} \coloneqq \mathcal{H}$. Then for each derivation we let $\Delta_{\eta_h} \coloneqq \{h' \in \Delta_\eta \mid h' \neq h \text{ and } h' \mid \chi_\eta\}$. Note that the set of diving hyperplanes is the same for both derived nodes.

To determine the sets of possible solutions relative to a single hyperplane, we need to solve S_η intersection problems instead of m if we considered the definition. To determine the divider hyperplanes, we must repeat this process for all hyperplanes in Δ_η. Therefore, in each node we execute this process with a complexity of $O(D_\eta \cdot S_\eta \cdot \ell(d, H))$ operations. Instead of computing Σ_η and

Δ_η each time, this approach allows us to exchange a fixed number of problems for a decreasing one since the sets Σ_η and $\Delta\eta$ are monotonically decreasing.

Now we improve the algorithm we have in two ways. First, we extend the representation of a node to include two more elements. A node η is now defined as the tuple $(\chi_\eta, h_\eta, \delta_\eta, \eta_h^L, \eta_h^R, \Sigma_\eta, \Delta_\eta)$. When created, the node $\eta = (\chi_\eta)$ represents the tuple $\eta = (\chi_\eta, h_\infty, \infty, \eta_\infty, \eta_\infty, \Sigma_\eta, \Delta_\eta)$, where the last two sets are obtained as described before. Second, we add another hash table indexed by the region χ_η, `P`, that will store the pair $(\Sigma_{\eta_h^L}, \Sigma_{\eta_h^R})$ for each hyperplane in Δ_η. We use a map that takes the indexes of the hyperplanes as keys and the previous pairs as values.

Algorithm 4 shows how the step described in line 5 of Algorithm 1 is implemented Also, when creating the nodes η_h^L and η_h^R, in lines 23 and 25, we set their solutions and hyperplanes using `P` as a reference. The solutions can be retrieved directly by accessing `P[`χ_η`][h]`. The set of divider hyperplanes is computed as the hyperplanes in Δ_η that have a nonempty list of left and right solutions in `P[`χ_η`]`.

Algorithm 4: Compute Σ and Δ for the children of η

Input: Node η

```
1  function ComputeSigmaDelta(η):
2      foreach h ∈ Δ_η do
3          if P[χ_η] does not contain key h then
4              Initialize leftSolutions and rightSolutions as an empty lists
5              foreach i ∈ Σ_η do
6                  if PositionR(h, χ_η ∪ χ_i) then
7                      Insert i into rightSolutions
8                  if PositionL(h, χ_η ∪ χ_i) then
9                      Insert i into leftSolutions
10             P[χ_η][h] ← (leftSolutions, rightSolutions)
```

3.3.4 Sorting the divider hyperplanes

Since we have to test all hyperplanes to find the best one, we propose a heuristic to select an order that should yield better results. This heuristic is based on the greedy intuition that the "best" hyperplane is the one that best separates the solutions, in the sense that the number of solutions on each side is as equal as possible while avoiding repeated solutions on both sides.

For a given node with region χ_η and a hyperplane h, we consider the values of S_η, $S_{\eta_h^L}$, and $S_{\eta_h^R}$ as defined in Subsection 3.3.3. Our goal is to define a

function $\hat{\mu}_{S_\eta} : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ such that $h^* = \arg\min_{h \in \Delta_\eta} \hat{\mu}_{S_\eta}(S_{\eta_h^L}, S_{\eta_h^R})$ is the "best" hyperplanes to separate the solutions in $R(\chi)$.

Given a pair of natural numbers (x, y), we call it a possible *split configuration* if $S_\eta \leq x + y \leq 2S_\eta$. We want $\hat{\mu}_{S_\eta}$ to represent a total ordering in the space of possible *split configurations*, where smaller values of $\hat{\mu}_{S_\eta}$ indicate better candidates. To achieve this we expect this function to have the the following properties:

(a) $\hat{\mu}_{S_\eta}(x, y) = \hat{\mu}_{S_\eta}(y, x)$

(b) $\hat{\mu}_{S_\eta}(x_1, y_1) = \hat{\mu}_{S_\eta}(x_2, y_2)$ if $(x_1, y_1) = (x_2, y_2)$ or $(x_1, y_1) = (y_2, x_2)$

(c) $\hat{\mu}_{S_\eta}(x_1, y) \leq \hat{\mu}_{S_\eta}(x_2, y)$ if $x_1 \leq x_2$

(d) $\hat{\mu}_{S_\eta}(x_1, y_1) \leq \hat{\mu}_{S_\eta}(x_2, y_2)$ if $x_1 + y_1 = x_2 + y_2$ and $x_2 \leq x_1 \leq y_1 \leq y_2$

Each of these properties indicates different characteristics we want: (a) indicates symmetry, because it should not matter the hyperplane side where the solutions lie, only their cardinality; (b) enforces a total ordering and guarantees that no additional tie-breaking criteria are required, since no two different configurations would have the same value of $\hat{\mu}_{S_\eta}$; (c) says that we should always prefer a configuration that has the least total number of solutions; and (d) known as the *Pigou-Dalton Transfer Principle* (Fleurbaey and Trannoy, 2003) stipulates that when two configurations have the same total number of solutions, the one with the smallest difference between the two sides should be preferred.

Let $z = \lceil \frac{S_\eta}{2} \rceil$, then we define a function $\hat{\mu}_{S_\eta}$ for $S_\eta \leq x + y \leq 2S_\eta$ as:

$$\hat{\mu}_{S_\eta}(x, y) = \begin{cases} (y - z + 1)^2 - (y - x) & \text{if } x \leq y \text{ and } s \text{ is even} \\ [(y - z + 1)^2 + (y - z + 1)] - (y - x) & \text{if } x \leq y \text{ and } s \text{ is odd} \\ \hat{\mu}_{S_\eta}(y, x) & \text{if } x > y \end{cases}$$

An example for $S_\eta = 8$ and $S_\eta = 7$ is shown in Table 3.1 and 3.2, respectively. Note that the values in the main diagonal (in blue) are defined by the first term in the equation, while the second term is responsible for the displacements of the previous values. We can verify in the table that the four properties hold for the function we defined:

(a) the last case guarantees the symmetry in the function definition;

(b) the table shows that the only repeated values occur for the pairs of symmetrical configurations;

(c) the values in all rows and columns are continuously increasing;

(d) the values in the diagonal defined by $x + y = k$ for some value of k decrease as it approaches the main diagonal.

	0	1	2	3	4	5	6	7	8
0	-	-	-	-	-	-	-	-	17
1	-	-	-	-	-	-	-	10	18
2	-	-	-	-	-	-	5	11	19
3	-	-	-	-	-	2	6	12	20
4	-	-	-	-	1	3	7	13	21
5	-	-	-	2	3	4	8	14	22
6	-	-	5	6	7	8	9	15	23
7	-	10	11	12	13	14	15	16	24
8	17	18	19	20	21	22	23	24	25

Table 3.1: Values of $\hat{\mu}_{S_\eta}$ for an even value of S_η

	0	1	2	3	4	5	6	7
0	-	-	-	-	-	-	-	13
1	-	-	-	-	-	-	7	14
2	-	-	-	-	-	3	8	15
3	-	-	-	-	1	4	9	16
4	-	-	-	1	2	5	10	17
5	-	-	3	4	5	6	11	18
6	-	7	8	9	10	11	12	19
7	13	14	15	16	17	18	19	20

Table 3.2: Values of $\hat{\mu}_{S_\eta}$ for an oddvalue of S_η

Before running Algorithm 1, we calculate all possible ordering tables of $\hat{\mu}_{S_\eta}$ and store than in a 3D array `L`, where `L[`S_η`]` references the values of $\hat{\mu}_{S_\eta}$ for S_η. Consequently, the value of $\hat{\mu}_{S_\eta}(x, y)$ is given by `L[`S_η`][x][y]`. When creating each node η, as described in the last paragraph of Subsection 3.3.3, we compute the value of $\hat{\mu}_\eta(h) = \hat{\mu}_{S_\eta}(S_{\eta_h^L}, S_{\eta_h^R})$ for each hyperplane in Δ_η, using the values stored in `P` as inputs.[4] Then we sort the set Δ_η in ascending order of the values $\hat{\mu}_\eta(h)$[5], so the first hyperplanes of Δ_η would always have the "best" results.

[4]If `P` does not have the values, we compute and store them as described in the last Subsection.

[5]Since the sets are implemented as lists, it makes sense to talk about their order.

3.4 Iterative Tree with Sampling

Constructing the optimal tree exploring all possible hyperplanes as described in Algorithm 1 can take much time, even with the structures we used. In this way, we need to evaluate a node thoroughly, find its minimal depth, and only then can we start taking advantage of the bounds we get. Besides that, we only get a valid tree at the end, which can take an impracticable amount of time.

To get around these issues, we propose a slightly different approach based on a relaxation of the function $\delta(\eta)$. We consider the function $\delta(\eta, \kappa)$ that is defined like in Equation 3-1 but instead of considering all hyperplanes in Δ_η it considers only the first κ hyperplanes in the ordered set. Note that a particular case for the iterative process happens when $\kappa = 1$. This case is equivalent to a greedy solution where at each node, we always choose h_η as the first element in Δ_η.

With this approach we can compute each node iteratively up to optimality, increasing the value of κ. At each iteration we obtain a valid tree T_κ such that $d(T_{\kappa_2}) \leq d(T_{\kappa_1})$ if $\kappa_2 > \kappa_1$. When dealing with the bounds, we can use the value of $d(T_{\kappa-1})$ as an upper bound for the minimum depth at iteration κ, so we do not have to explore nodes that are deeper than what we already have. However, we cannot update the lower bound anymore because our results are not optimal yet.

The iterative process allows us to obtain valid trees and limit the search space by using the upper bounds faster. Still, it takes much time because of the number of operations performed in the computation of contained solutions and divider hyperplanes ($D_\eta \cdot S_\eta \cdot \ell(d, H)$). A way to speed up the process is to avoid evaluating the positions of all D_η hyperplanes.

We introduce a strategy of randomly selecting a portion of the hyperplanes to evaluate at each iteration. This strategy is defined by the parameter s that represents the percentage of hyperplanes to be evaluated. Still, to guarantee that the number of solutions keeps decreasing at each iteration, we first favor the hyperplanes that split the pair of solutions in that node.

This strategy affects directly the behavior of Algorithm 4. Its updates version with all modifications is shown in Algorithm 5. A last extension is made to the node representation to include the number κ of hyperplanes computed. So our final representation for a node η is the tuple $\eta = (\chi_\eta, h_\infty, \infty, \eta_\infty, \eta_\infty, \Sigma_\eta, \Delta_\eta, \kappa_\eta)$, where κ_η is initialized with zero when the node is created.

Algorithm 5: Compute Σ and Δ for the children of η with sampling

Input: Node η and percentage of hyperplanes s

```
1  function ComputeSigmaDelta(η, s):
2      if P does not contain key χ_η then
3          Sort Δ_η to place the bisectors of the solutions in Σ_η first
4      evaluatedHyperplanes ← 0
5      numberHyperplanes ← min(1, ceil(s · S_η))
6      foreach h ∈ Δ_η do
7          if P[χ_η] does not contain key h then
8              Initialize leftSolutions and rightSolutions as an empty lists
9              foreach i ∈ Σ_η do
10                 if PositionR(h, χ_η ∪ χ_i) then
11                     Insert i into rightSolutions
12                 if PositionL(h, χ_η ∪ χ_i) then
13                     Insert i into leftSolutions
14             P[χ_η][h] ← (leftSolutions, rightSolutions)
15             Increment evaluatedHyperplanes
16         if evaluatedHyperplanes ≥ numberHyperplanes then
17             break
18     Sort the evaluated hyperplanes in Δ_η in ascending of order of μ̂_{S_η}
```

Algorithm 6 shows the final version of the algorithm to build trees iteratively and considering the sampling. For simplicity, in the base cases (lines 8 and 13), we just added the additional lines and omitted the others that are the same from Algorithm 1. The function `BuildTree`(η, κ, s) can be called iteratively starting with the value of $\kappa = 1$ up to $\kappa = H$. Note that the greedy case we mentioned before, with the addition of the sampling, is equivalent to calling `BuildTree`$(\eta, 1, 1.0)$.

Algorithm 6: Calculate minimal tree iteratively with sampling

Input: Node η, number of hyperplanes to consider κ, and percentage of hyperplanes s

Output: $\delta(\eta, \kappa)$, the depth of η considering κ hyperplanes

```
1  function BuildTree(η, κ, s):
2      UpdateNode(η)
3      if κ_η ≥ κ then
4          return δ_η
5      ComputeSigmaDelta(η, s)
6      if S_η = 1 then
7          κ_η ← H
8          ···
9      else if S_η = 2 then
10         κ_{η^L_{h_ij}} ← H
11         κ_{η^R_{h_ij}} ← H
12         κ_η ← H
13         ···
14     else
15         upperBound ← min(S_η − 1, δ_η)
16         lowerBound ← ceil(log(S_η))
17         foreach h_i ∈ Δ_η do
18             dLeft ← BuildTree(η^L_h, κ, s)
19             if dLeft < δ_η then
20                 dRight ← BuildTree(η^R_h, κ, s)
21                 if dRight < δ_η then
22                     h_η ← h
23                     δ_η ← max(dLeft, dRight) + 1
24                     upperBound ← δ_η
25             if upperBound = lowerBound or i ≥ κ then
26                 break
27         if upperBound = lowerBound then
28             κ_η ← H
29         else
30             κ_η ← κ
31         MemorizeNode(η)
32         return δ_η
```

4
Applications

The LDT is a good model of computation for dealing with many concrete problems in which the input data can be treated just as a vector in a d-dimensional Euclidean space, and all solutions of the problem correspond to a partitioning of the space into regions bounded by hyperplanes (Kolinek, 1987).

Among the problems with such geometrical character, we can find variations of the shortest paths (e.g., the travelling salesman), the knapsack, and integer linear programming, which are considered hard combinatorial problems, particularly when we allow the input entries to assume both positive and negative values.

4.1
Problems

We select three problems of different natures to study the behavior of our method. First, the 0-1 knapsack problem (in the case we consider) can be solved in polynomial time with a dynamic programming approach. Then, the weighted minimum cut problem has a known polynomial-time algorithm if the entries are positive. However, when we allow both positive and negative values, this problem becomes NP-complete. Lastly, the symmetric traveling salesman is NP-complete even with positive entries.

4.1.1
The 0-1 Knapsack Problem (KNP)

Given a set of n items numbered from 1 up to n, each with a weight w_i and a value v_i, the problem consists of choosing a collection of the items so that the total weight is less than or equal to a given limit W and the total value is as large as possible. We consider the particular case where each item has weight $w_i = i$ and the weight limit is $W = n$.

The space dimension d is the same as the number of items, and if we let $a(n)$ be the number of partitions of n into distinct parts[6], then the number of possible solutions[7] is

$$m = \sum_{1 \leq i \leq n} a(i).$$

[6] https://oeis.org/A000009
[7] https://oeis.org/A026906

This problem can be expressed in our model as: given an input $\mathbf{c} \in \mathbb{R}^d_+$ representing the values of each item and the set $X \subset \{0,1\}^d$ of possible solutions; our goal is to find $x^* \in X$ such that $f_{-\mathbf{c}}(x^*) = \min_{\mathbf{x}\in X} f_{-\mathbf{c}}(\mathbf{x})$. Each coordinate of x^* indicates whether the corresponding item is included in the optimal collection, and the total value is given by $f_{\mathbf{c}}(x^*)$.

4.1.2 The Weighted Minimum Cut Problem (CUT)

Given an undirected graph G with n nodes numbered from 1 up to n and positive weights on the edges, this problem consists in partitioning the nodes into two disjoint subsets such that the weights of the edges across the subsets are minimal. We consider the particular case where G is required to be connected.

In this case, the space dimension is the same as the number of node pairs $d = \binom{n}{2}$, and the number of possible solutions is half the number of subsets of the nodes excluding the empty set and the set itself, $m = 2^{n-1} - 1$.

In our model we say that, given an input $\mathbf{c} \in \mathbb{R}^d$ of the edge values of G where a value $c_i = 0$ indicates that the graph does not contain the edge i and the set of solutions $X \subset \{0,1\}^d$; our goal is to find $x^* \in X$ such that $f_{\mathbf{c}}(x^*) = \min_{\mathbf{x}\in X} f_{\mathbf{c}}(\mathbf{x})$. The nonzero coordinates of x^* indicates the edges to be removed to obtain the minimum cut, and $f_{\mathbf{c}}(x^*)$ is the total weight of that cut.

4.1.3 The Symmetric Travelling Salesman Problem (TSP)

Given a set of n cities numbered from 1 up to n and the distances between each pair of cities, the problem consists of finding the shortest possible route that visits each city exactly once and returns to the origin city.

In the symmetric case, the space dimension is also the same as the number of city pairs $d = \binom{n}{2}$, but the number of possible solutions is half the number of circular permutations, $m = \frac{(n-1)!}{2}$

Converting to our model: given an input $\mathbf{c} \in \mathbb{R}^d_+$ of the edge values of G and the set $X \subset \{0,1\}^d$ of possible solutions; our goal is to find $x^* \in X$ such that $f_{\mathbf{c}}(x^*) = \min_{\mathbf{x}\in X} f_{\mathbf{c}}(\mathbf{x})$. The nonzero coordinates of x^* indicate the edges in the optimal route, and $f_{\mathbf{c}}(x^*)$ is the total distance in that route.

4.2 Baseline Methods

To compare the performance of the proposed method, we selected a baseline method for each of the problems. Table 4.1 shows these methods

and their corresponding complexities in the worst case, replacing the general variables for their values in the cases we selected.

Problem	Method	Time Complexity
KNP	Dynamic programming	$O(n^2)$
CUT	Stoer–Wagner algorithm	$O(n^3 + n^2 \log n)$
TSP	Dynamic programming	$O(n^2 2^n)$

Table 4.1: Baseline methods and complexities

As previously mentioned, not all problems are NP-hard, so some of them present a polynomial-time complexity that we discuss now. The classical version of the 0-1 knapsack problem is NP-complete, and the dynamic programming approach to solve has a complexity of $O(nW)$. However, for the particular considered where $W = n$, it can be quickly solved in polynomial time in $O(n^2)$. The Stoer-Wagner algorithm (Stoer and Wagner, 1997) is a recursive algorithm to solve the weighted minimum cut problem in undirected weighted graphs. Its complexity is $O(V \cdot E + V^2 \log(V))$ with V being the number of nodes and E the number of edges. In our case, that complexity translates to $O(n^3 + n^2 \log n)$ since we consider graphs with about half of the edges, as we further detail in the experiments. Lastly, in the symmetric traveling salesman problem, the dynamic programming approach is also non-polynomial, and we maintain the usual time complexity of $O(n^2 2^n)$.

To use these methods, we implement both dynamic programming algorithms and use the Boost implementation of the Stoer-Wagner algorithm[8]. Note that while the Stoer-Wagner algorithm can only handle the cases with non-negative weights, the dynamic programming for the two problems mentioned can handle both positive and negative inputs.

[8] `https://www.boost.org/doc/libs/1_76_0/libs/graph/doc/stoer_wagner_min_cut.html`

5
Experiments and Results

In this chapter we explore the problems studied with the selected polytopes, detailing their characteristics and discussing the issues faced. We call a *problem* the tasks defined in Section 4.1 that can be executed for any case. A *polytope* is the particular case when we fix a value of n and an *instance* is the point whose values correspond to a particular objective function for a polytope. Thus, we say that KNP is the knapsack problem, KNP-n3 is the knapsack polytope for the case with 3 items, and $(0.5, 0.2, 0.4)$ is an instance for the KNP-n3 polytope.

The following sections will discuss: (a) the structure of the polytopes and the computation of their skeletons; (b) the construction process for the decision trees of each polytope; (c) the generation of the instances used in the benchmark sets; and (d) the performance evaluation of the method proposed.

5.1
Polytopes and Structure

When producing the decision tree of a polytope, the first step is to describe its skeleton completely, with all adjacent pairs. As mentioned before, the task of checking adjacency in some polytopes is an NP-complete problem (Papadimitriou, 1978). Therefore, we already face an issue regarding the computation of the skeletons. Due to their number of solutions, the skeletons for the KNP polytopes were straightforward to compute. For the CUT polytopes, there is a known result stating that every pair of vertices is adjacent (Barahona and Mahjoub, 1986). So for this polytope, no testing was required. However, the TSP polytopes represented quite an extensive task.

Table 5.1 presents the polytopes with basic attributes and additional information regarding their skeletons. In order we have: (a) the size n given by the problem; (b) the space dimension d where the points are inserted; (c) the number of possible solutions m; (d) the total number of unique hyperplanes H; (e) the number of unique hyperplanes corresponding to skeleton edges; (f) the number of edges in the skeleton graph; and (g) the average number of neighbors for each vertex.

From this data, we notice two interesting things. First, when we compare the number of skeleton hyperplanes with the number of edges (in the KNP and TSP polytopes), we note that their proportion seems to present a decreasing

Polytope	Size	Dimension	Solutions	Total Hyperplanes	Skeleton Hyperplanes	Edges	Average Neighbors
KNP-n3	3	3	4	6	6	6	3.00
KNP-n4	4	4	6	13	11	13	4.33
KNP-n5	5	5	9	27	20	29	6.44
KNP-n6	6	6	13	59	32	51	7.85
KNP-n7	7	7	18	110	53	91	10.11
KNP-n8	8	8	24	189	76	141	11.75
KNP-n9	9	9	32	327	123	233	14.56
KNP-n10	10	10	42	533	176	349	16.62
CUT-n3	3	3	3	3	3	3	2.00
CUT-n4	4	6	7	21	21	21	6.00
CUT-n5	5	10	15	105	105	105	14.00
CUT-n6	6	15	31	465	465	465	30.00
CUT-n7	7	21	63	1953	1953	1953	62.00
CUT-n8	8	28	127	8001	8001	8001	126.00
CUT-n9	9	36	255	32385	32385	32385	254.00
CUT-n10	10	45	511	130305	130305	130305	510.00
TSP-n4	4	6	3	3	3	3	2.00
TSP-n5	5	10	12	36	30	60	10.00
TSP-n6	6	15	60	1050	555	1230	41.00
TSP-n7	7	21	360	41835	9660	30240	168.00

Table 5.1: Complete description of polyoptes with skeleton attributes

tendency; this shows that (at least in these cases) the amount of symmetries increases along with its size. Second, we note very clearly that the number of skeleton hyperplanes gets smaller than the total number of hyperplanes. This is a good indicator that a different approach that considers only the skeleton hyperplanes might perform better when building the corresponding trees. Every limitation we impose on the number of hyperplanes is a good improvement since the main bottleneck of our method is determining the position of all hyperplanes relative to the region of interest.

5.2 Decision Tree Construction

In this step, we experiment with different ways of generating the decisions trees. Note that, while this part takes the most time, it has to be done just once. When we have a decision tree, we can use it to run all instances for a given polytope. To generate the trees we selected six algorithmic configurations with different values of the parameters κ and s defined on Section 3.4. Table 5.2 summarizes the parameters for each configuration. The first configuration corresponds to a greedy algorithm, as mentioned before. All other configurations take a value of $\kappa = H$ that allows the procure to iteratively keep testing more hyperplanes until it reaches the time limit.

Configuration	κ	s
Config #1	1	1.00
Config #2	H	0.10
Config #3	H	0.25
Config #4	H	0.50
Config #5	H	0.75
Config #6	H	1.00

Table 5.2: Algorithm parameters for each configuration

The code to build the trees was developed in Python 3.6 and can be accessed at `https://github.com/cleberoli/tiny-problems`. These experiments are run on a single thread of an Intel Gold 6148 Skylake 2.4 GHz processor with 40GB of RAM, running CentOs 7.8.2003. The limit established was 86400 seconds (one day), and the results obtained are summarized in Table 5.3. After completion, we tested the trees with the benchmark instances to ensure that they always give the optimal solution.

The table shows the information for each tree generated. Along with the number of solutions and hyperplanes (for comparison), we have for each configuration the depth d of the best tree obtained and the number of

			Config #1		Config #2		Config #3		Config #4		Config #5		Config #6	
Polytope	Solutions	Hyperplanes	d	κ	d	κ	d	κ	d	κ	d	κ	d	κ
KNP-n3	4	6	**3**	**1**	**3**	**1**	**3**	**1**	**3**	**1**	**3**	**1**	**3**	**1**
KNP-n4	6	13	4	1	**4**	**13**	**4**	**13**	**4**	**13**	**4**	**13**	**4**	**13**
KNP-n5	9	27	6	1	6	14	6	13	6	13	6	12	6	12
KNP-n6	13	59	7	1	8	6	8	5	7	5	7	4	7	4
KNP-n7	18	110	9	1	10	4	10	3	9	3	9	3	9	3
KNP-n8	24	189	10	1	12	2	12	2	10	2	10	2	10	2
KNP-n9	32	327	12	1	15	1	15	1	12	1	12	1	12	1
KNP-n10	42	533	15	1	18	1	18	1	15	1	15	1	15	1
CUT-n3	3	3	**2**	**1**	**2**	**1**	**2**	**1**	**2**	**1**	**2**	**1**	**2**	**1**
CUT-n4	7	21	6	1	6	15	6	15	6	15	6	15	6	15
CUT-n5	15	105	14	1	14	1	14	1	14	1	14	1	14	1
CUT-n6	31	465	-	-	-	-	-	-	-	-	-	-	-	-
TSP-n4	3	3	**2**	**1**	**2**	**1**	**2**	**1**	**2**	**1**	**2**	**1**	**2**	**1**
TSP-n5	12	36	8	1	8	7	8	7	8	7	8	7	8	7
TSP-n6	60	1050	-	-	22	1	-	-	-	-	-	-	-	-
TSP-n7	360	41835	-	-	-	-	-	-	-	-	-	-	-	-

Table 5.3: Decision tree construction results (bold values represent minimal trees)

hyperplanes κ tested in each node. We note that optimal trees, in the sense of minimizing the depth, were found only for one instance of the knapsack problem besides the trivial cases of each problem.

Analyzing these results, we note two expected behaviors: (a) the number of hyperplanes that could be tested during the time limit decreases when we either increase the problem size or the percentage of sampled hyperplanes; and (b) sampling fewer hyperplanes have a greater probability of not achieving the best depths. Another interesting result, seen in Config #4, Config #5, and Config #6, is that a value of $s \geq 0.50$, was sufficient to obtain the smallest depth found for the tested instances. Along with it, the greedy approach (Config #1) consistently achieves the best-known values. This behavior raises the question of whether the structure of these problems may offer support for obtaining optimal decision trees with a greedy algorithm. Another interesting fact is that when we compare the tree obtained with the greedy configuration with any other trees with the same depth, they stay the same, i.e., they do not change internal comparisons or arrangements.

Polytope	Minimum	Average	Maximum	Standard Deviation
KNP-n3	3	3.00	3	0.00
KNP-n4	4	4.00	4	0.00
KNP-n5	5	5.67	6	0.47
KNP-n6	6	6.67	7	0.47
KNP-n7	7	8.29	9	0.74
KNP-n8	8	9.47	10	0.75
KNP-n9	9	11.18	12	0.96
KNP-n10	10	12.78	15	1.29
CUT-n3	2	2.00	2	0.00
CUT-n4	6	6.00	6	0.00
CUT-n5	11	13.51	14	0.72
TSP-n4	2	2.00	2	0.00
TSP-n5	5	7.00	8	0.82
TSP-n6	10	16.54	22	1.92

Table 5.4: Statistics for the depth of the trees (through their structure)

Table 5.4 shows more information on the best trees (Config #1). We analyze some statistical metrics to understand better how is the internal structure of these trees. The overall behavior displayed is that small size polytopes have complete trees, but even when the size increases, they do not present many discrepancies among the depths of the leaves, as shown by the small standard deviations and gaps between the minimum and maximum depths. Exceptions start happening with larger sizes, and the biggest gap is found for the TSP with 6 nodes.

5.3 Generation of Benchmark Sets

To evaluate the performance of the methods, we generated benchmark sets for each polytope. Each benchmark set is made of randomly generated instances, amounting to 10000 points for each possible solution according to the descriptions in the Section 4.1.

Note that not all solutions can be attained if we consider the positivity constraints established in Section 4.1. For instance, in a knapsack problem with 3 items $(1, 0, 0)$ and $(1, 1, 0)$ are possible solutions. However, when the values are required to be positive, the first solution will never be chosen because the second will always have a larger total value. Therefore, we also generated a set of unrestricted instances for each polytope, generated without the positivity constraint, to cover all solutions.

Let $x \in U(I)$ denote that x is an independently and randomly generated value from the uniform distribution over the interval I. Now we describe in more detail the parameters for each problem:

- **Knapsack Problem:** an instance $\mathbf{c}$ in a KNP benchmark set is chosen such that $c_i \in U([0, 1])$ for all i and $\|\mathbf{c}\| = 1$, while a point $\mathbf{c}$ in the unrestricted benchmark set has $c_i \in U([-1, 1])$ for all i and $\|\mathbf{c}\| = 1$.
- **Cut Problem:** for these polytopes, we use the *Erdős–Rényi* model where $G(n, p)$ represents a graph G that has its n nodes randomly connected by edges with probability p. An instance $\mathbf{c}$ in a CUT benchmark is chosen such that $c_i \in U((0, 1])$ if i is an edge of a connected $G(n, 0.5)$ and $c_i = 0$ otherwise. In the unrestricted sets, $c_i \in U([-1, 0) \cup (0, 1])$ if i is an edge of the graph and $c_i = 0$ otherwise.
- **Travelling Salesman Problem:** an instance $\mathbf{c}$ in a TSP benchmark set is chosen such that $G = K_n$, the complete graph, with distances $c_i \in U([0, 1])$ for all i and $\|\mathbf{c}\| = 1$, with the additional constraint that for each triangle in G their corresponding distances respect the triangle inequality. For $\mathbf{c}$ in the unrestricted benchmark we just have $c_i \in U([-1, 1])$ for all i and $\|\mathbf{c}\| = 1$.

5.4 Performance Evaluation

To evaluate the performance of the resulting algorithms synthesized by our method, we implement their corresponding trees. We also implement the basic enumeration method and the baseline method as described in Section 4.2. The code for this evaluation was developed in C++ and can also be found in

the GitHub repository. These experiments were conducted on a computer with an Intel Core i7-8565U CPU @ 1.80GHz processor and 16.0 GB of memory, running Windows 10 Pro.

In this step, we compute the solutions for all instances in the benchmark sets using the three methods. To ensure accuracy, we run each benchmark several times so that each method takes an amount of time proportional to the number of solutions (in seconds).

Table 5.5 shows the average amount of time (in nanoseconds) to find the solution for a single benchmark instance. The shorter times are highlighted in bold font for easier comparison. We see that the algorithm synthesized with our method consistently outperforms the enumeration method, even for the CUT instances where the tree height is very close to the number of solutions. Our method is also better than the baselines for the CUT and TSP. The exception was the KNP problem which was not surprising since this dynamic programming method has a good complexity for the case. Still, we see that our method has a performance improvement with the increase of size, as shown by the baseline over tree ratio.

Polytope	Enumeration	Baseline	Tree	(Base/Tree) Ratio
KNP-n3	726	**129**	620	0.21
KNP-n4	939	**201**	724	0.28
KNP-n5	1295	**282**	800	0.35
KNP-n6	1786	**359**	887	0.40
KNP-n7	2658	**482**	976	0.49
KNP-n8	3535	**626**	1059	0.59
KNP-n9	4758	**767**	1148	0.67
KNP-n10	6316	**942**	1275	0.74
CUT-n3	196	24775	**158**	157.01
CUT-n4	679	34941	**319**	109.53
CUT-n5	2148	53700	**733**	73.30
TSP-n4	287	315	**209**	1.51
TSP-n5	1711	923	**465**	1.98
TSP-n6	11238	3107	**1006**	3.09

Table 5.5: Time comparison for different methods (in nanoseconds)

An unexpected behavior was the amount of time it took to solve the CUT with the baseline. A possible explanation for this event might have to do with the library used compared to the other baseline method that was implemented with simple structures. Alternatively, this could be an impact of the small sizes since the difference between the baseline and tree times gets smaller with the increase in size.

Finally, we ran the time comparisons with the unrestricted benchmark sets. Since not all baseline methods are equipped to deal with negative input values, we only compare the enumeration and tree methods. Note that the problems might not have a well-defined meaning when the inputs take negative values. However, in the perspective of our method, we only need the solutions to find the optimal one. We verify again that the trees always find optimal solutions, and they continue to outperform the enumeration algorithms, as seen in Table 5.6.

Polytope	Enumeration	Tree
KNP-n3	725	**606**
KNP-n4	1002	**720**
KNP-n5	1307	**810**
KNP-n6	1805	**1053**
KNP-n7	2497	**920**
KNP-n8	3499	**1141**
KNP-n9	4701	**1130**
KNP-n10	6628	**1309**
CUT-n3	204	**160**
CUT-n4	693	**328**
CUT-n5	2165	**786**
TSP-n4	296	**212**
TSP-n5	1692	**475**
TSP-n6	11152	**983**

Table 5.6: Time comparison with unrestricted benchmark (in nanoseconds)

6
Concluding Remarks

In this work, we explored the theoretical results that point to the existence of polynomial-time algorithms in the LDT model for NP-complete problems. We establish the equivalence between the class of 0-1 combinatorial optimization problems and the nearest neighbor search problem. Then we address the task of finding the adjacency relations for skeletons of the 0-1 knapsack and symmetric traveling salesman. In this process, we prove the equivalence of the *Delaunay triangulation* and *skeleton* graphs for the family of problems we study and use it to define the *Voronoi regions*.

We introduce a method that, given a polytope, constructs a decision tree algorithm that can solve any instances for that polytope without changes. With our experiments, we could find those trees for the three studied problems (KNP, CUT, and TSP) with sizes up to 10, 5, and 6, achieving a minimum depth for the sizes up to 4, 3, and 4, respectively. These trees consistently outperform the enumeration method for all problems, and the baseline methods for the weighted minimum cut and symmetric traveling salesman, providing optimal solutions within microseconds. Furthermore, our method presents the advantage of working with both positive and negative inputs without changing the time required to find the optimal solution.

The polytopes for which we managed to find the corresponding trees are considerably small and could not be practically used for problems like the ones described in Subsection 1.1.3. Still, we have other relevant results from the intermediate steps, like the complete definition of the skeleton of the TSP with up to 7 nodes. Again, this is a number race. We may be able to find it for polytopes with sizes 5 or 6. Still, each new size will pose challenges and require significant methodological refinements and a better understanding of the problems.

A clear path to continue this work is to refine our framework to process instances with larger sizes. One possible way of doing that would be to consider only the skeleton hyperplanes instead of all hyperplanes, like mentioned in Section 5.1. Another approach is investigating the possibility that a greedy algorithm could guarantee the minimal depth trees since they could achieve this in the experiments conducted. Also, a possible extension is to compare our method with other ML approaches (Kool et al., 2019; Wang et al., 2021) that can find solutions very quickly, even if they are not always optimal.

An alternative approach is to modify the trees to generate strategies to improve other algorithms instead of finding the optimal solutions like mentioned in Section 1.1.4. We could, for instance, use the hyperplanes down to a certain specified depth as strategies. These hyperplanes could be used by other solvers to limit the initial search space

Lastly, once we know that this class of CO problems can be thought of as an NNS problem and we have the hyperplanes that define those regions, we can tackle the decision version of these same problems a lot easier. Once we view the problem of verifying if a given solution is optimal as determining the location of a point relative to a set of hyperplanes, we can use works like the one from Meiser (1993). He presents a solution for the point location problem in arrangements of hyperplanes that has polynomial-time complexity.

7
References

A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1974. ISBN 0201000296. 1, 1.1.1

F. Arnold, Ítalo Santana, K. Sörensen, and T. Vidal. Pils: Exploring high-order neighborhoods by pattern mining and injection. *Pattern Recognition*, 116:107957, 2021. ISSN 0031-3203. doi: 10.1016/j.patcog.2021.107957. 1.1.3

J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. Blake2: Simpler, smaller, fast as md5. In M. Jacobson, M. Locasto, P. Mohassel, and R. Safavi-Naini, editors, *Applied Cryptography and Network Security*, pages 119–135. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-38980-1_8. 3.3

F. Barahona and A. R. Mahjoub. On the cut polytope. *Mathematical Programming*, 36(2):157–173, 1986. ISSN 1436-4646. doi: 10.1007/BF02592023. 5.1

Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021. ISSN 0377-2217. doi: 10.1016/j.ejor.2020.07.063. 1.1.4

D. Bertsimas and B. Stellato. The voice of optimization. *Machine Learning*, 110 (2):249–277, 2021. ISSN 1573-0565. doi: 10.1007/s10994-020-05893-5. 1.1.4

M. Beunardeau, Éric Brier, N. Cartier, A. Connolly, N. Courant, R. Géraud-Stewart, D. Naccache, and O. Yifrach-Stav. Optimal covid-19 pool testing with a priori information, 2020. 1.1.4

S. C. Boyd and W. H. Cunningham. Small travelling salesman polytopes. *Mathematics of Operations Research*, 16(2):259–271, 1991. ISSN 0364765X, 15265471. 1, 1.1.2

I. Chikalov, S. Hussain, and M. Moshkov. Totally optimal decision trees for monotone boolean functions with at most five variables. *Procedia Computer Science*, 22:359–365, 2013. ISSN 1877-0509. doi: 10.1016/j.procs.2013.09.113. 17th International Conference in Knowledge Based and Intelligent Information and Engineering Systems - KES2013. 1.1.2

T. Christof and G. Reinelt. Combinatorial optimization and small polytopes. *Top*, 4(1):1–53, 1996. ISSN 1863-8279. doi: 10.1007/BF02568602. 1, 1.1.2

T. Christof and G. Reinelt. Decomposition and parallelization techniques for enumerating the facets of combinatorial polytopes. *International Journal of Computational Geometry & Applications*, 11(4):423–437, 2001. doi: 10.1142/S0218195901000560. 1

T. Christof, M. Jünger, and G. Reinelt. A complete description of the traveling salesman polytope on 8 nodes. *Operations Research Letters*, 10(9):497–500, 1991. ISSN 0167-6377. doi: 10.1016/0167-6377(91)90067-Y. 1, 1.1.2

M. Fleurbaey and A. Trannoy. The impossibility of a paretian egalitarian. *Social Choice and Welfare*, 21(2):243–263, 2003. ISSN 1432-217X. doi: 10.1007/s00355-003-0258-2. 3.3.4

S. Fortune. Voronoi diagrams and delaunay triangulations. In F. Aurenhammer, R. Klein, and D.-T. Lee, editors, *Handbook of Discrete and Computational Geometry*, chapter 27, pages 702–721. CRC Press LLC, Boca Raton, FL, 2017. 2.2

M. Grötschel and M. W. Padberg. Polyhedral theory. In E. Lawler, J. Lenstra, A. R. Kan, and D. Shmoys, editors, *The traveling salesman problem*, chapter 8, pages 251–305. John Wiley & Sons, 1985. 1.1.2

M. Gurgel and Y. Wakabayashi. Adjacency of vertices of the complete pre-order polytope. *Discrete Mathematics*, 175:163–172, 1997. doi: 10.1016/S0012-365X(96)00143-4. 2.1

L. Hyafil and R. L. Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15–17, 1976. ISSN 0020-0190. doi: 10.1016/0020-0190(76)90095-8. 1.1.2

M. Kolinek. A polynomial-time linear decision tree for the traveling salesman problem and other np-complete problems. *Discrete & Computational Geometry*, 2(1):37–48, 1987. ISSN 1432-0444. doi: 10.1007/BF02187869. 1, 1.1.1, 4

W. Kool, H. van Hoof, and M. Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=ByxBFsRqYm`. 6

S. Meiser. Point location in arrangements of hyperplanes. *Information and Computation*, 106(2):286–303, 1993. ISSN 0890-5401. doi: 10.1006/inco.1993.1057. 6

F. Meyer auf der Heide. A polynomial linear search algorithm for the n-dimensional knapsack problem. *Journal of the ACM*, 31(3):668–676, 1984. ISSN 0004-5411. doi: 10.1145/828.322450. 1, 1.1.1

F. Meyer auf der Heide. Fast algorithms for n-dimensional restrictions of hard problems. *Journal of the ACM*, 35(3):740–747, 1988. ISSN 0004-5411. doi: 10.1145/44483.44490. 1, 1.1.1

C. H. Papadimitriou. The adjacency relation on the traveling salesman polytope is np-complete. *Mathematical Programming*, 14(1):312–324, 1978. ISSN 0025-5610. doi: 10.1007/BF01588973. 2.1, 5.1

S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. In U. Montanari, J. D. P. Rolim, and E. Welzl, editors, *Automata, Languages and Programming*, pages 49–60. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-45022-1. 1.1.3

M. Stoer and F. Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, July 1997. ISSN 0004-5411. doi: 10.1145/263867.263872. 4.2

E. Taillard. Parallel iterative search methods for vehicle routing problems. *Networks*, 23(8):661–673, 1993. doi: 10.1002/net.3230230804. 1.1.3

T. Toffolo, T. Vidal, and T. Wauters. Heuristics for vehicle routing problems: Sequence or set optimization? *Computers & Operations Research*, 105:118–131, 2019. ISSN 0305-0548. doi: 10.1016/j.cor.2018.12.023. 1.1.3

H. Wang, Z. Zong, T. Xia, S. Luo, M. Zheng, D. Jin, and Y. Li. Rewriting by generating: Learn heuristics for large-scale vehicle routing problems, 2021. URL `https://openreview.net/forum?id=xxWl2oEvP2h`. 6

G. M. Ziegler. *Lectures on Polytopes*. Springer-Verlag New York, USA, 1st edition, 1995. ISBN 9781461384311. doi: 10.1007/978-1-4613-8431-1. 2.2.1