RECURSIVE OPTIMIZATION: EXACT AND EFFICIENT COMBINATORIAL OPTIMIZATION ALGORITHM DESIGN PRINCIPLES WITH APPLICATIONS TO MACHINE LEARNING

By

Xi He

A thesis submitted to the University of Birmingham for the degree of

DOCTOR OF PHILOSOPHY

School of Computer science

College of Engineering and Physical Sciences

University of Birmingham

April 24, 2025

Abstract

This thesis presents a generic algorithm design framework for solving combinatorial optimization problems, it subsumes the classical recursive optimization methods, such as *greedy*, *dynamic programming*, *divide-and-conquer*, and *branch-and-bound* (BnB) methods. In particular, this thesis focuses on solving combinatorial optimization problems in machine learning to demonstrate the effectiveness and practicality of this framework.

Our framework is grounded in Bird's theory of the algebra of programming, which is a relational formalism for deriving correct-by-construction algorithms from specifications. We introduce this theory through Haskell code, with a particular emphasis on its application to combinatorial optimization. Additionally, we reformulate the branch-and-bound method to integrate it formally into this framework, thereby establishing a unified approach for designing recursive combinatorial optimization algorithms.

More broadly, our theoretical foundation is an integration of *constructive algorithmics* (or *transformational programming*), *combinatorial generation*, and *combinatorial geometry*. These topics are integrated together to achieve our final goal—designing efficient combinatorial optimization algorithms, particularly for machine learning problems. They are interconnected in such a way that both geometric algorithms for addressing fundamental combinatorial geometry problems and efficient combinatorial generators can be structured or reformulated systematically using principles from constructive algorithmics. This approach facilitates the design of efficient (in terms of worse-case complexity and parallelizability) geometric algorithms and combinatorial generators that are sound and concise. Moreover, the geometric insights allow us to reveal the combinatorial properties of combinatorial problems, which allows us to significantly simplify the combinatorial complexity of the problem.

Our main contribution does not lie in advancing the three themes in the framework, although we have novel contributions to each of them. Instead, the primary contribution of this thesis lies in how to integrate these themes for solving combinatorial optimization problems in machine learning. To demonstrate the effectiveness of our framework, we address four fundamental problems in machine learning: *classification*, *clustering*, *decision tree*, and *empirical risk minimization for ReLU neural networks*. We provide a detailed analysis of the combinatorial properties of these problems, demonstrating that these problems can be solved in polynomial time. To the best of our knowledge, all algorithms we propose are the fastest known in terms of worst-case complexity, and their performance can be further improved through the acceleration techniques we introduce.

Finally, two example problems (the 0-1 loss linear classification problem and the K-medoids problem) are selected to show *end-to-end implementations* in Haskell. In our experiments, we compare our algorithm with state-of-the-art BnB algorithms. Our method not only provides provably exact solutions but also achieves significantly lower computational time. We demonstrate that the wall-clock time of our algorithm aligns with its worst-case time complexity analysis on synthetic datasets. In contrast, state-of-the-art BnB algorithms for these two problems exhibit exponential time complexity, even in cases where the brute-force algorithm has polynomial time complexity. Moreover, for the K-medoids problem, we demonstrate that a carefully designed brute-force algorithm, leveraging the generator proposed in this thesis, outperforms the state-of-the-art BnB algorithm across all datasets tested. Since a brute-force algorithm is guaranteed to yield the exact solution, our experiments revealed that the state-of-the-art BnB algorithm for the K-medoids problem often produces incorrect results. This highlights the importance of adopting a rigorous framework, like the one proposed here, when designing exact algorithms.

Moreover, our contribution to the study of interpretable machine learning is substantial. We believe that some of the algorithms proposed in this thesis will revolutionize the field of interpretable machine learning, as previous research has demonstrated the effectiveness of exact algorithms, even for simplified problems. The solution we propose is more generic (capable of solving a broader range of problems) and efficient (with polynomial-time complexity in the worst case, parallelizable, and amenable to further speed-up).

Acknowledgments

First and foremost, I extend my deepest gratitude to my parents, who have provided unwavering love and support throughout my PhD journey. Their steadfast belief in me has been a constant source of motivation, particularly during the most challenging moments, and I am forever grateful for their sacrifices and encouragement.

I am deeply grateful to my supervisor, Max, I could not have asked for a better supervisor. His dedication, insightful guidance, and exceptional support have surpassed all my expectations. When I first joined his research group, I was a naive student with no computer science background, filled with youthful confidence, yet eager to become an exceptional computer scientist. My first meeting with Max and his former students, Ugur Kayas and Yordan P. Raykov, left me overwhelmed, as I struggled to understand any of the terminology in their discussions. Thanks for the patients of Max, which let me becomes a immeasurably better researcher, presenter, writer that I could not imaging becomes when I first arrived. Beyond his role as a supervisor, Max has been a mentor who profoundly shaped both my academic and personal growth. He challenged me to think independently while offering guidance to navigate obstacles, fostering my development as an independent researcher. His openness to interdisciplinary approaches expanded my perspective, and his commitment to excellence, coupled with his willingness to go above and beyond, has profoundly influenced my academic journey.

I also express my sincere gratitude to my thesis examiners, Prof. Jeremy Gibbons and Dr. Shuo Wang, for their time, insightful questions, and valuable feedback, which significantly strengthened this work. I am particularly grateful to my external examiner, Prof. Jeremy Gibbons, whose expertise and engagement elevated the quality of this thesis. It was a true honor and fortune to have one of the greatest researcher in constructive algorithmics scrutinize my work. His insightful questions and thoughtful feedback not only refined my research but also reshaped my understanding of the field.

I am fortunate to have shared this journey with my colleagues, particularly Omer Eryilmaz, Feifei Zheng, Dhurim Cakiqi, Abdulrahman Aloyayri, Nawfal Zakar, Yusuf Bayka, whose friendship and support have been invaluable. I owe special thanks to Yi Miao for his generous assistance in helping me implement parallelized programs.

This thesis reflects the collective support of everyone mentioned and many others whose contributions, big and small, have made this achievement possible. Thank you all.

Contents

Part I

Background

The first chapter in this part begins with a brief overview of machine learning as motivation for studying exact algorithms. Understanding the motivation behind exact algorithms is crucial, as it provides the impetus for exploring more accurate and efficient solutions to complex learning problems. Current approaches for finding globally optimal exact solutions in machine learning rely solely on *branch-and-bound* (BnB) algorithms and *mixed-integer programming* (MIP) solvers. However, these *general-purpose algorithms* are far from perfect and usually have exponential complexity in the worst-case. The appreciation of the drawbacks of these general-purpose algorithms will highlight the importance of our research and point us in the right direction in order to solve these issues.

Following this, the chapter transitions into a thorough discussion of the prerequisite foundational knowledge essential for a deeper comprehension of the thesis.

The second chapter covers four key topics. In the first two sections, we discuss combinatorial optimization problems, addressing questions such as: How is a combinatorial optimization problem specified? What are the classical methods for solving these problems? How are these methods integrated into our framework, and what is the algorithm design process within this framework? The discussion here aims to provide an informal intuition rather than a rigorous exposition. The next two sections focus on *structured recursion*, *category theory*, and the programming language *Haskell*. As the title of this thesis suggests, the optimization methods discussed in this thesis are recursions, particularly structured recursions, which provide a guarantee of termination. Category theory and Haskell are the primary theoretical tool and programming language used in this thesis. Category theory is explored as a means of formalizing and abstracting algorithmic principles, while the *strongly-typed*, *side-effect-free* functional programming language Haskell enables us to produce rigorous and reliable code implementing those principles computationally.

A comprehensive overview of the thesis is provided in the third chapter, outlining the connections between the discussed topics and how they are combined to tackle difficult combinatorial problems. Together, the discussions in this part build a solid foundation for the subsequent exploration of the intriguing relationships between combinatorial optimization, algorithm design, combinatorial optimization and machine learning, in the remainder of the thesis.

Finally, the contributions of this thesis are summarized in chapter four.

I.1 Introduction

I.1.1 Machine learning

What is machine learning? Humans acquire knowledge by learning, and constantly learning lets us know how to think, understand, predict and make better decisions. The field of *artificial intelligence* (AI) tries to build intelligent machines which can think and act like humans. *Machine learning* (ML) is widely recognized as a subfield of AI, but today the terms machine learning and artificial intelligence are often used interchangeably. After many decades of steady progress, machine learning has now entered a phase of very rapid development. Applications of machine learning are becoming ubiquitous, it has had an impact on almost any field where computation plays a role.

Scientists often assume that stable underlying *mechanisms* exist in the world, organized to create natural observations [Pearl et al., 2016]. The main task of machine learning is to recover the underlying mechanism through an inductive learning process on a *finite* amount of observed data from the world. Problems in ML involving *labelled* datasets are called *supervised learning problems*. Typical examples of supervised learning problems include classification and regression. Algorithms for classification problems have output restricted to a finite set of values (usually a finite set of integers, called labels). Algorithms for regression problems have numerical, for instance real, values. On the other hand, problems involving unlabelled data are called *unsupervised learning problems*, the K-clustering problem and dimension reduction are examples.

The focus of machine learning research is to develop *accurate* models for prediction/prescription by designing *efficient* and *robust* algorithms. A common method to assess the quality of a model is to evaluate its *prediction accuracy* on *unseen* (*test*) *datasets*. Since the distribution of unseen data is typically unknown, we generally assume that it matches the distribution of the training data. Consequently, many robust algorithms aim to find a model with the lowest objective value within a given *hypothesis set* $\mathcal{H}$[1] on training data sets. This approach is known as *empirical risk minimization* (ERM). According to results in *statistical learning theory*, when the distribution of the training dataset is the same as the distribution of the test dataset, the ERM model not only performs best on the training data but also provides the *tightest upper bound* for the prediction error [Mohri et al., 2018].

Objectives of machine learning problems and associated optimization methods Due to the finite nature of training data, most machine learning tasks are defined to optimize a combinatorial objective function, which is determined by the combinatorial structure of the problem. For instance, in classification problems, the objective is to minimize the *number of misclassifications*. Similarly, in K-clustering problems, the objective is to find K centroids that minimize the *within-class distances* of data points assigned to each *centroid* or *cluster*. In optimization, solving a problem with a discrete objective is typically much more difficult than solving one with continuous-valued objective. Therefore, for most machine learning problems, finding the ERM solution with a combinatorial objective is computationally intractable [Little, 2019].

Alternatively, combinatorial objectives in ML are replaced with *convex surrogate* objective functions. Then, various continuous optimization methods, such as gradient descent (where the surrogate is differ-

[1] A set of functions mapping features to the set of predictions.

entiable), can be applied. On the other hand, solving ML problems using *combinatorial optimization methods* is particularly rare. This disparity arises from several factors:

1. **Problems are defined over continuous variables.** Although many ML problems possess a combinatorial nature, the independent variables are continuous [Bishop, 2006].

2. **Intractable combinatorics**. The combinatorics of many ML problems are exponentially large (or worse), and many of these problems have been proven to be NP-hard [Mohri et al., 2018, Little, 2019].

3. **Continuous optimization methods are well-studied**. Continuous optimization methods are well-established in ML research, and off-the-shelf convex optimization solvers are easy to use, often converging quickly with low polynomial-time complexity in the worst case [Boyd and Vandenberghe, 2004]. However, combinatorial optimization methods have not been systematically studied in ML research.

4. **The success of continuous optimization algorithms**. The most successful algorithms in ML research are all stochastic or continuous optimization algorithms [Hastie et al., 2009], such as *Markov chain Monte Carlo* (MCMC), the *expectation-maximization* (EM) algorithm, and gradient descent.

However, these continuous optimization algorithms provide no guarantee that the *exact* (*globally optimal*) solution can be obtained for difficult combinatorial optimization problems. In high-stakes or safety-critical applications, where errors are unacceptable or carry significant costs, we want the best possible solution given the specification. Only an exact algorithm can provide this guarantee. To address the lack of a universal framework for designing exact combinatorial optimization algorithms, the focus of this thesis is to develop a formalism that offers a generic solution for designing practical algorithms for solving combinatorial machine learning problems exactly.

Machine learning problems in this thesis The main topics of this thesis are combinatorial machine learning problems related to finite data points and *hyperplanes*. There are three reasons for focusing on these two objects. First, machine learning is inherently data-driven, and almost all problems in this field involve finite amounts of data. Second, the most successful models in machine learning, such as ReLU neural networks, *decision trees*, and *support vector machines*, are *linear* or *piecewise linear* (PWL) models. Therefore, studying the theory of hyperplanes will help us gain a better understanding of the generic combinatorial properties of these PWL models.

Finally, due to the geometric and topological simplicity of finite data points and hyperplanes, these subjects are well-studied in *combinatorial geometry*. Almost all problems involving finite sets of points or hyperplanes are combinatorial. However, strictly classifying a problem as either combinatorial or non-combinatorial is neither reasonable nor desirable [Edelsbrunner, 1987]. As we will discuss in Section I.2.1, many ML problems can be specified as *mixed continuous-discrete optimization problems* (MCDOP), where the objective functions can either be defined continuously or combinatorially. For instance, the well-known linear classification problem can be optimized either combinatorially or continuously. To find the optimal hyperplane in $\mathbb{R}^D$, using a continuous optimization method, we characterize it as a normal vector, a continuous parameter that can be optimized using general methods such as gradient descent.

Alternatively, although a hyperplane is a continuous geometric object, in classification problems, we are primarily concerned with the partition of data it induces. Since the dataset is always finite, we can characterize the hyperplane through these partitions. Given N data points, there are at most 2^N possible partitions, as a hyperplane can only divide the space into two connected regions. The definition of MCDOP is provided in equation (2).

I.1.2 Motivations

I.1.2.1 Why study exact machine learning algorithms for simple (interpretable) models?

The goal of learning

> *George Dantzig: The final test of any theory is its capacity to solve the problems which originated it.*

This quote [Dantzig, 2016] aptly states the importance of studying exact algorithms in machine learning, where the ultimate goal is to learn a model which is both accurate (*exact*) and easily understood (*interpretable*), so that we can be confident the predictions are *meaningful*. This is what exact algorithms can offer, a *provably exact algorithm* will always select the *best solution* in the hypothesis set. In other words, a *provably exact algorithm cannot be improved upon in terms of accuracy*. By contrast, while approximate algorithms offer advantages in terms of efficiency and computational scalability, the potential risks associated with inaccuracies and lack of reliability must be carefully considered.

In many practical engineering applications, it might not be important whether the globally optimal solution is obtained, but it always matters that a solution is as high-accuracy as is feasible. The choice between approximate and exact algorithms ultimately depends on the specific requirements of the application and the trade-offs between speed, accuracy, and reliability. Ideally, we aim to determine the necessary computational effort to achieve a specified level of accuracy. Given that our exact algorithms (introduced in Part II) have polynomial-time worst-case guarantee, this trade-off can be managed with a high degree of precision. By contrast, for most machine learning algorithms, this precise trade-off is generally not possible. For instance, deep learning models trained using *stochastic gradient descent* may require a few hours to achieve a desired level of accuracy, or several days for the same, yet no theory exists to predict this with certainty. It is far superior to have reliable information about the trade-off which comes from having a predictable, polynomial-time algorithm.

The myth of the accuracy versus interpretability trade-off Advancements in computer vision and natural language processing have led to a widespread belief that the most accurate models must be inherently *uninterpretable* and *overparameterized*. This has led to an unquestioned belief in the trade-off between accuracy and interpretability—namely, that an interpretable model cannot surpass an uninterpretable one in terms predictive accuracy. This misconception can be traced back to the early problems that machine learning research aimed to address. As Rudin and Radin [2019] note, "This belief stems from the historical use of machine learning in society: its modern techniques were born and bred for low-stakes decisions such as online advertising and web search where individual decisions do not deeply affect human lives."

A recent empirical finding is that models with highly complex hypothesis sets such as deep neural networks and random forests can be made to have zero or close to zero loss on the training data and yet still perform well out-of-sample. This effect seems particularly pronounced for high-dimensional data such as digital images in low-stakes decision problems (on tabular data, tree-based algorithms still outperform complex deep learning classifiers on various datasets, see Grinsztajn et al. 2022, Shwartz-Ziv and Armon 2022 for more details). This finding seems to contradict classical learning theory. An explanation for this phenomenon is that these apparently overparameterized models achieve excellent performance through *regularized interpolation* rather than through regularized statistical fitting of a decision boundary model [Belkin et al., 2018, 2019b,a]. For instance, AdaBoost and random forests which are maximally large (interpolating) decision trees achieve this same generalization behavior [Belkin et al., 2019a]. Such models are said to be operating in the *interpolation regime*.

This empirical observation has led to a somewhat blind belief that there exists a trade-off between accuracy and interpretability. However, this is not necessarily true for problems that have structured data[2] with meaningful features: there is often no significant difference in performance between more complex classifiers (deep neural networks, boosted decision trees, random forests) and much simpler classifiers (logistic regression, decision lists) after preprocessing [Rudin, 2019]. Therefore, a significant practical advantage for exact algorithms arises when we do not know the ground truth, but we would prefer a very simple model for the purposes of interpretability. In many high-stakes applications such as medical decision-making or criminal justice [Rudin and Radin, 2019, Holte, 1993, Rudin, 2019] it is important that the decisions made by the model are easily understood.

Moreover, in many scientific knowledge discovery domains, it is crucial to understand what has been "learned" from the data, rather than relying on a high-accuracy "black box" model whose predictions lack interpretability. For example, if an interpretable linear model makes it possible to identify a chemical reaction or systematically construct novel materials, this can be a useful contribution to scientific discovery in itself, possibly more useful than a complex nonlinear model which is hard to understand even if it has higher classification accuracy than a simple, interpretable linear model.

Do the exact solutions overfit the data? One of the main criticisms of exact algorithms is that finding the global optimal solution can lead to overfitting. This concern arises from the fact that, with small data sets, the generalization bounds in learning theory are not sufficiently tight, causing the accuracy of the resulting solution to be significantly affected by data quality and noise. Consequently, it seems reasonable to believe that exact algorithms may not be useful when dealing with small data sets.

We hold an opposing view: exact algorithms not only produce more accurate models relative to the ground truth but are also more robust to poor-quality data. For example, past research on the optimal classification tree problem has shown that when data sizes are small, approximate algorithms (such as the classification and regression tree (CART) algorithm) demonstrate poor approximation to the ground truth, whereas optimal decision tree algorithms perform significantly better [Bertsimas and Dunn, 2017]. However, as training data size increases, these approximate algorithms become as accurate out-of-sample as other methods [Bertsimas and Dunn, 2017].

[2]Structured data refers to data that is organized and designed in a specific way to be easily readable and understandable by both humans and machines.

In data-poor environments (i.e., data with high noise levels), approximate algorithms perform significantly worse than exact algorithms in terms of out-of-sample accuracy [Bertsimas and Dunn, 2017]. This provides strong evidence against the notion that optimal methods tend to overfit the training data in data-poor applications. Similar results have also been observed in research on the linear classification problem [He and Little, 2023b] and the K-medoids problem [He and Little, 2024], where the difference between exact and approximate algorithms is most significant with small data sizes and diminishes as data size increases.

I.1.2.2 Shortcomings of existing general-purpose exact algorithms

When obtaining exact solutions for problems with intractable combinatorics, *general-purpose algorithms*[3] such as branch-and-bound (BnB) algorithms and the off-the-shell mixed-integer programming (MIP) solvers (Groubi, GLPK, CPLEX for instance) are ubiquitous, but these algorithms normally possess exponential time and space complexity in the worst-case. The inclination to use general-purpose algorithms for obtaining exact solutions arises from the perceived intractable combinatorics of many ML problems, and most of these problems are classified as *NP-hard*[4], so that no known algorithm can solve all instances of the problem in polynomial time.

However, for many ML problems, the NP-hardness proofs have been constructed for problems which are not actually the same as the original definitions of the problems as used in practical ML applications. As a result, polynomial-time algorithms may exist for these problems, which means that they are not truly NP-hard but have been misclassified as "NP-hard problems." For instance, we have successfully developed two polynomial-time algorithms for solving the K-medoids problem [He and Little, 2024] and the 0-1 loss linear classification problem [He and Little, 2023b]. Similarly, polynomial time algorithms for solving the K-means problem have also been developed [Inaba et al., 1994, Tîrnăucă et al., 2018]. Therefore, if polynomial-time algorithms do exist for these seemingly intractable combinatorial problems, relying on general-purpose algorithms such as MIP solvers or BnB algorithms—both of which exhibit exponential complexity in the worst case and offer limited insight into the problem itself—can hinder our understanding of the fundamental principles involved, such as the combinatorial and the geometric properties of the underlying problem.

We have identified several limitations of these general-purpose algorithms which deserve closer examination:

1. **Lack of formal correctness proofs**. The correctness of these algorithms is often unclear or relies on tedious induction. Exact solutions require rigorous mathematical proof, yet many BnB studies rely on weak assertions or informal explanations that do not hold up under close scrutiny [Fokkinga, 1991]. A formal proof for exact combinatorial optimization algorithms should derive from an exhaustive search specification.

2. **Lack of systematic characterization**. Most existing general-purpose algorithms are designed in

[3] General-purpose algorithms are designed to solve a wide range of optimization problems, without being tailored to a specific domain. They are flexible and versatile, capable of handling various types of problem formulations and constraints.

[4] A problem is NP-hard if solving it efficiently would allow us to solve all problems in NP efficiently, but it is not necessarily in NP itself. In other words, NP-hard problems are at least as hard as the hardest problems in NP, but they may not have a solution that can be obtained in polynomial time.

an ad-hoc, intuitive manner. Algorithmic insights obtained from one particular problem are very hard to transfer to another.

3. **No worst-case guarantees**. Worst-case time and space complexity analysis is rarely reported in studies of BnB algorithms. Indeed, many of these general-purpose algorithms exhibit exponential (or even worse) worst-case time and space complexity, yet this critical aspect is often neither discussed nor analyzed rigorously. Consequently, existing studies on exact algorithms for many machine learning problems either omit time complexity analysis or overlook essential details. This omission undermines the reproducibility of findings and impedes the advancement of knowledge in this domain.

4. **No parallel implementations**. The success of many powerful algorithms can be attributed to their parallel implementations. *Embarrassingly parallel* programs—programs where processes require no communication or dependencies—are crucial for solving intractable combinatorial optimization problems, especially NP-hard problems, for which no polynomial-time algorithm is known. An embarrassingly parallel program might be the only feasible approach for solving large-scale problems. However, such powerful techniques are rarely discussed or used in the study of these general-purpose algorithms. Indeed, embarrassingly parallel implementation are very difficult to achieve for algorithms based on *generative recursions* (see Subsection I.2.3.2 for definition), such as *cutting-plane algorithms*.

5. **Informal acceleration techniques**. Although many acceleration techniques, such as upper bound-/lower bounds and dominance relations, are commonly used in BnB studies, they are rarely discussed formally. Detailed explanations of how to implement these techniques efficiently are often omitted.

6. **Lack of flexibility in problem definition**. General-purpose algorithms either struggle to incorporate various constraints, or the impact of these constraints on time and space complexity is often unclear.

To overcome the shortcomings of general-purpose algorithms, it is necessary to challenge the status quo and adopt new algorithm design methods. In this thesis, we take a fundamentally different approach by using a generic algorithm design formalism known as *transformational programming* or *constructive algorithmics*. By integrating concepts from *combinatorial geometry* and *combinatorial generation*, we introduce a novel framework for designing efficient and exact combinatorial optimization algorithms in a broader optimization context. We refer to this framework as the *recursive optimization framework* (ROF). We demonstrate that many seemingly intractable or even NP-hard problems can be solved exactly in polynomial time (with some parameters fixed), and by this aim to illuminate a path to designing reliable and tractable combinatorial optimization algorithms that are both logically sound and concise.

I.2 Foundations

In this section, we outline the foundational concepts frequently used throughout this thesis. The discussion here is intentionally informal rather than mathematically rigorous, focusing on providing an intuitive understanding of the essential principles. We will briefly explain key ideas such as *sequential decision processes*, *category theory*, *recursion*, and *combinatorial optimization*. More formal and detailed discussions of these concepts are provided in Part II.

We argue that it is preferable to derive algorithms from a correct specification, using simple, calculational steps, a process known as *program calculus* (or *transformational programming*). This requires treating programs as if they are mathematical functions, and it is for this reason we develop our algorithm in a *strongly-typed* and *side-effect free* functional programming language, specifically Haskell. A short introduction to Haskell is given in Subsection I.2.4.3.

I.2.1 Combinatorial optimization

I.2.1.1 Combinatorial optimization problem specification

In ML studies, the common task for ML algorithms is to learn an accurate model h in a *hypothesis set* $\mathcal{H}$, with respect to a data set $\mathcal{D}$ which consists of N *independent and identically distributed* (*i.i.d.*) data points (or data items) $\boldsymbol{x}_n$, $\forall n \in \{1, \ldots, N\} = \mathcal{N}$, where the data points $\boldsymbol{x}_n \in \mathbb{R}^D$ and D is the dimension of the *feature space*. For supervised learning problems, each data point is as sociated with a unique *true label* t_n. For regression tasks, t_n is a real value $\mathbb{R}$, and $t_n \in \{0, 1\}$ or $t_n \in \{0, 1, \ldots K\}$ for *binary* or *multiclass* classification tasks respectively. All true labels in this dataset $\mathcal{D}$ are represented by a vector $\boldsymbol{t} = (t_1, t_2, ..., t_N)^T$ or list $\boldsymbol{t} = [t_1, t_2, ..., t_N]$. The dataset with additional true labels vector is denoted by $\mathcal{D}_{\boldsymbol{t}}$.

The hypothesis h in $\mathcal{H}$ can either be defined *continuously* by a *continuous* parameter $\boldsymbol{\mu}$ in $\mathbb{R}^D$ or *combinatorially* by a *discrete* parameter (*combinatorial configuration*) s from a *combinatorial search space* $\mathcal{S}$. The task of a *learner* (algorithm) is to use dataset $\mathcal{D}$ to find a hypothesis $h_{\mathcal{D}} \in \mathcal{H}$ that has a small *generalization error*. However, the learner can measure the *empirical error* of a hypothesis on the dataset $\mathcal{D}_{\boldsymbol{t}}$ that minimizes the following *mixed continuous-discrete objective function*

$$E\left(s, \boldsymbol{\mu}, \theta\right) = \sum_{n \in \mathcal{N}} l\left(\boldsymbol{x}_n, t_n; \boldsymbol{\mu}, s, \theta\right), \tag{1}$$

where the loss function l has type $l : \mathbb{R}^D \times \mathbb{T} \times \mathbb{R}^D \times \mathcal{S} \times \Theta \to \mathbb{R}^+$, and $\theta : \Theta$ is the model *hyperparameter*. The objective function of this form is known as *linear objective function* [Björner and Ziegler, 1992]. For most ML problems, the hypothesis $h : \mathcal{H}$ is parameterized solely by a continuous parameter $\boldsymbol{\mu}$. Once the continuous parameter $\boldsymbol{\mu}$ is fixed, it is often that the discrete parameter s can be uniquely determined by $\boldsymbol{\mu}$. This correspondence is usually a *non-injective map*, meaning that many continuous parameters $\boldsymbol{\mu}$ can determine the same discrete parameter s. This is because the combinatorial search space $\mathcal{S}$ typically consists of a finite number of elements, whereas the continuous space $\mathbb{R}^D$ has *infinite* combinatorial complexity. This correspondence implies the existence of *equivalence relations* among different $\boldsymbol{\mu}$, defined in terms of their relationship to s.

The problems considered in this thesis can all be specified as the following *mixed continuous-discrete optimization problem* (MCDOP)

$$(\hat{s}, \hat{\boldsymbol{\mu}}) = \underset{s' \in p(\mathcal{S}), \boldsymbol{\mu}' \in \mathbb{R}^D}{\operatorname{argmin}} E(s', \boldsymbol{\mu}', \theta'), \tag{2}$$

where argmin returns one of the optimal solutions with respect to objective E, as optimization problems often have multiple optimal solutions. The *predicate function* $p : \mathcal{S} \to \mathbb{B}$ returns true if configuration $\mathcal{S}$ satisfies the constraints of the problem.

In this thesis, we consider these MCDOPs from a different perspective and start by specifying these problems in a different style. In the theory of *transformational programming* (*constructive algorithmics*) [Bird and De Moor, 1996, Jeuring, 1993], combinatorial optimization problems such as (2) are solved using the following *generate-evaluate-filter-select* (*exhaustive search*) paradigm

$$s^* = sel_E(filter_p(eval_E(gen(\mathcal{D})))), \tag{3}$$

where the generator function $gen : [\mathbb{R}^D] \to [\mathcal{S}]$ takes a list of data points $xs = [\mathbb{R}^D]$ (equivalent to the input dataset $\mathcal{D}$) and enumerates *all* possible *combinatorial configurations* s in search space $\mathcal{S}$ and stored them in a list. For most problems, *gen* is a recursive function so that the input of the generator can be replaced with the index set $\mathcal{N}$ and rewritten $gen(\mathcal{N})$. The *evaluator* $eval_E : [[\mathbb{R}^D]] \to [([\mathbb{R}^D], \mathbb{R})]$ computes the objective values $r = E(s)$ for *all* configurations[5] s generated by $gen(n)$ and returns a list of *tupled configurations* (s, r). The *filter* function $filter_p : [(\mathcal{S}, \mathbb{R})] \to [(\mathcal{S}, \mathbb{R})]$ filter out all infeasible configurations and retains only those which evaluate to true under the *predicate* $p : (\mathcal{S}, \mathbb{R}) \to Bool$. The predicate function receives a tuple (s, r) and returns true if configuration s satisfies the condition. Lastly, the *selector* $sel_E : [(\mathcal{S}, \mathbb{R})] \to (\mathcal{S}, \mathbb{R})$ select the best configuration s^* with respect to the objective value (the second term) returned by $eval_E : [[\mathbb{R}^D]] \to [([\mathbb{R}^D], \mathbb{R})]$.

In functional programming communities, (3) is often expressed in a more compact *point-free function composition style* as

$$mcdop = sel_E \cdot filter_p \cdot eval_E \cdot gen, \tag{4}$$

where $\cdot$ represents the *functional composition operator*, and this specification has type $mcdop : \mathcal{D} \to (\mathcal{S}, \mathbb{R})$. The astute reader may notice that the input $\mathcal{D}$ is left implicitly in (4), this is because, in a functional programming language, the type of the input can be inferred from the type declaration, as *mcdop* takes an input of type $\mathcal{D}$. We will see this style very often when we program in Haskell. Taking a different perspective, the specification *mcdop* can also be considered as a generic *program* for solving (2) exactly, it is known as a *brute-force algorithm*: by generating all possible configurations in the search space $\mathcal{S}$, evaluating the corresponding objective E for each, and selecting an optimal configuration, it is clear that it must solve the problem (2) exactly. However, program (4) is generally inefficient due to *combinatorial explosion*; the size of $gen(\mathcal{D})$ is often exponential (or worse) in the size of $\mathcal{D}$.

To make this exhaustive solution practical, the focus of this thesis is to develop a framework for designing correct-by-construction combinatorial optimization algorithms from specification (4), thus simultaneous correctness and efficiency is assured of the algorithm which computes the optimal solution.

[5] In some literature, $eval_E$ is applied to only one configuration; in such cases, the *map* function can be used to apply it to a list of configurations.

Furthermore, classical combinatorial optimization algorithms, such as the *greedy algorithm*, *dynamic programming* (DP), *divide-and-conquer* (D&C) and *branch-and-bound* (BnB) are unified within the same framework. In our framework, efficient recursive algorithms can hence be derived as long as the conditions for the corresponding theorems are satisfied. This approach allows us to rigorously construct efficient combinatorial optimization algorithms from provably correct specifications, avoiding tedious induction proofs or ambiguous, informal explanations.

I.2.1.2 Combinatorial generation and combinatorial optimization

Following the above discussion, (4) presents a universal approach for solving any combinatorial optimization problem, provided that the generator for the given problem is known. Indeed, one of the central topics of this thesis is to explain how to design an efficient combinatorial generator. This is because we have found that once we have *efficient brute-force algorithm*, an efficient combinatorial optimization algorithm for this problem follows almost immediately.

Some readers might find this claim surprising, given that brute-force algorithms are typically considered inefficient due to their exhaustive nature. Indeed, traditional brute-force methods are often grossly inefficient, especially for large-scale problems, as they fail to exploit any structure or heuristics to reduce the search space. The key insight is that both the efficient generator and the combinatorial optimization algorithm for a problem often require exploring the same principles but within different algebraic structures. We aim for this concern to be addressed through the evidence presented in this thesis.

One notable principle is *distributivity*, which, in its simplest form, states that $a \times b + a \times c = a \times (b + c)$. The left-hand side is equivalent to the right-hand side, but the left-hand side of this equation involves three arithmetic operations, whereas the right-hand side needs only two. In the context of optimization, the min and + operator also have distributivity, we have the property that $\min x + \min y = \min \{a + b \mid a \in x \wedge b \in y\}$. Assume x and y are length N lists, then the right side involves N^2 arithmetic operations whereas the left side involves only $2N+1$ computations. Distributivity also exists in the context of combinatorial generation, a detailed example will be provided in Subsection I.2.2.4.

Indeed, in the study of information theories [Aji and McEliece, 2000, Kschischang et al., 2001], it has long been recognized that a large family of fast algorithms, including *Viterbi' s algorithm* and the *fast Fourier transform* (FFT) can be derived from a *generalized distributivity law*. Similarly, distributivity plays an important role in designing efficient combinatorial generation algorithms. This similarity is not a coincidence. In Chapter II.2, we will discuss this correspondence more formally within our categorical framework. Our previous work on *polymorphic dynamic programming* also explains their correspondence by using a semiring algebraic framework [Little et al., 2024].

In this thesis, we will also explore how a generalization of distributivity—*monotonicity*—plays a crucial role in designing efficient exhaustive search algorithms, and consequently, efficient combinatorial optimization algorithms.

Furthermore, we will see that classical combinatorial optimization (CO) methods—such as greedy, DP, D&C, and BnB—are characterized by the structure of their corresponding combinatorial generators. One important class of generator is the *sequential decision process* (SDP), which involves subdividing the problem sequentially, leading to an iterative structure in the resulting algorithms. Greedy algorithms, BnB algorithms and some of the DP algorithms are all characterized by SDPs. By contrast, D&C methods

correspond to combinatorial generators with a different structure; they involve subdividing the original problem into subproblems in all possible ways (usually binary splits), rather than sequential decomposition as seen in SDP.

Characterizing different combinatorial optimization algorithms in terms of the structure of their corresponding generators offers several advantages:

1. **Focus on generator design**: It allows us to concentrate on designing an efficient combinatorial optimization generator, which is often simpler than directly designing an efficient combinatorial optimization algorithm.

2. **Reusability**: Since many combinatorial optimization problems share similar combinatorial structures, a generic combinatorial generator can be reused for various problems, whereas combinatorial optimization algorithms are typically problem-specific.

3. **Flexible design**: It is often the case that various generators exist for the same combinatorial structure, each with the same asymptotic complexity but different computational properties. This flexibility allows for the design of *tailored* combinatorial optimization algorithms that are more efficient for *specific tasks*.

I.2.1.3 What is an efficient combinatorial generator and where to find it?

Before discussing how to construct an efficient combinatorial generator, we must first answer the question of how to compare the efficiency of different generators. For most combinatorial structures, the size of the configuration space is usually polynomially or exponentially large. As a result, a generator with *optimal efficiency* would inherently exhibit polynomial or exponential time complexity, given the need to store such a large number of configurations in memory. Although each combinatorial structure can be generated in various ways by different generators, these generators typically share the *same* optimal asymptotic complexity when exhaustively generating all configurations. However, different generators are often designed for specific purposes rather than exhaustive generation alone. Consequently, they may have different constant factors, which are obscured by Big O notation, leading to differences in their practical performance. Therefore, relying solely on Big O notation is insufficient for a comprehensive analysis of the efficiency of these algorithms.

An ideal combinatorial generator should have *constant amortized time* (CAT), where the total computational effort scales linearly with the number of objects it produces. This means that, on average, generating each configuration requires a fixed amount of time, regardless of the set's size.

The key to constructing a CAT generator lies in identifying common substructures shared among configurations, which involves a *efficient factorization* of the combinatorics of the problem. This can be considered as a semantic interpretation of *Bellman's principle of optimality* [Bellman, 1954]. For instance, if we want to construct combinatorial structures $x = [a, b, c]$ and $y = [a, b, d]$, it is more efficient to first construct their shared part, $[a, b]$, and then construct x, y subsequently.

The majority of "optimally efficient" generators are closely related to a recursive program known as the *sequential decision process* (SDP). In Section II.1.2, we will first introduce a range of efficient SDP-based generators, including those for collections such as *permutations*, *subsets*, *K-permutations*, *K-combinations*

(K-subsets), *sequences*, and *partitions* (*segmentations*). These generators will serve as a comprehensive library that can be directly applied to combinatorial optimization tasks.

In addition to discussing existing combinatorial generators, in Chapter II.2 of Part II, we will explore the abstract forms of these generators and develop several generic principles for designing sophisticated combinatorial generators from basic ones. These basic SDP generators can serve as "atoms" used to create more complex "compounds" by applying various algebraic rules.

Given the critical importance of SDP in our work, we will first illustrate the basic concept to provide some intuition before the more detailed technical discussion in Section II.2.2 of Chapter II.2.

I.2.1.4 Sequential decision process

The sequential decision process is probably one of the most frequently used methods in combinatorial optimization and generation. It is also known as *sequential iteration* or *sequential recursion*. In the original paper of Bellman on dynamic programming, Bellman 1954 gives a somewhat vague characterization of the SDP as the general problem of sequential choice among several actions. De Moor [1995] gives a more precise definition of an SDP, the sequential nature is captured by expressing it as an instance of the operator `fold` from functional programming. For those not familiar with functional programming, in Haskell, `foldr` is defined as an

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e [] = e
foldr f e (a:xs) = a `f` (fold f e xs)
```

where `f :: a -> b -> a` is a binary operator with infix notation `` `f` ``[6], the function `foldr` traverses the *list* `(a:xs) = [a,a1,..,an]` (`:` is the operator that prepends an element to the list) recursively from right to left starting with the seed value `e`. Informally, `foldr f e [a,a1,..,an] = a `f` (a1 `f` ... (an `f` e))`. We will sometimes refer to the SDP defined solely by the `foldr` operator as the "*ordinary SDP*" to distinguish it from more sophisticated extensions of the SDP.

We have noted that greedy algorithms, DP, and the BnB method can all be characterized as SDPs. However, there are several notable properties that seem to be missing from this definition of SDP compared to existing DP and BnB algorithms.

First, many DP algorithms work with integer inputs, such as those used for calculating the *Fibonacci sequence* or *Catalan numbers*, which take a natural number $n \in \mathbb{N}$ as input. This can lead to issues, as *natural numbers* and *finite-length lists* are different data types, and many programs must be rewritten repeatedly for different data types. This occurs because many programming languages do not allow the programmer to abstract from the structure of the data that the program manipulates, and thus programs need to be rewritten for different datatypes.

Second, in the definition of `foldr`, each recursive step depends solely on the result of the previous step. Specifically, `foldr f e (a:xs)` depends solely on `a` and `foldr f e xs`, without considering substructures smaller than `xs`. By contrast, many DP problems involve *memoization* techniques that retain results from several previous steps. For instance, the Fibonacci sequence DP recursion $f(n) = f(n-1) + f(n-2)$, depends on the previous step $f(n-1)$ and previous step but one, $f(n-2)$.

[6] An infix notation for a binary function `f` apply its argument on the left and right side of the equation

Third, the current definition of SDP does not accommodate search strategies commonly used in the BnB method, such as *depth-first* or *best-first* strategies.

Nevertheless, these "missing parts" in the *ordinary SDP* defined by `foldr` operator can be incorporated by introducing the datatype-generic abstraction of SDP, known as a *catamorphism*, along with its extensions. Catamorphisms allow programmers to write statically-checkable generic type-independent programs that exploit the inherent structure of input data. In this generic recursive program, recursive datatypes are modeled by *polynomial functors*. This enables us to feed the program with arbitrary recursive datatypes defined by polynomial functors, and the structure of the recursion will be automatically determined by the structure of these datatypes. For instance, although the natural numbers $\mathbb{N}$ and lists are different datatypes, they share a similar structure: all natural numbers are successors of zero, and all lists can be constructed by inductively prepending new values to an empty list. In fact, both can be defined inductively through polynomial functors.

Furthermore, in Section II.2.8, we will show that different search strategies used in the BnB method can be derived from the specification of catamorphisms. This provides a formal proof for why different search strategies used in BnB algorithms are exhaustive, a fact that is usually demonstrated through weak assertions or informal explanations in the studies of BnB algorithms.

I.2.2 Combinatorial optimization algorithm design through a modern lens

In this section, we provide a high-level overview of combinatorial optimization algorithm design. The goal is to offer a brief explanation of how these methods are generally perceived within the community, while presenting a concise summary of key results from our framework before going to more rigorous details.

The discussion here is divided into four subsections. In the first Subsection, we present a detailed summary of how these combinatorial optimization methods are commonly understood. In the second Subsection, we summarize the key components of designing efficient combinatorial optimization algorithms, with efficiency being our sole concern. In the third Subsection, we summarize how these different combinatorial optimization methods are related to each other in terms of their *inclusion relations*. Finally, we examine the *rod-cutting problem*, a well-known problem solvable via DP methods. We demonstrate how the well-known DP algorithm for this problem can be derived from scratch, with the underlying design principles formally discussed in Chapter II.2 of Part II.

I.2.2.1 An overview of classical combinatorial optimization methods

Classical algorithm design textbooks typically present a "zoo" of algorithms. Algorithms with similar features, such as greedy or DP methods, are typically grouped into the same category. However, these categories are frequently defined ambiguously, leading to inconsistent classifications across the literature. Also, the correctness of these algorithms is often explained informally, making it challenging to understand how to design such algorithms from scratch and why they are correct.

Greedy methods Greedy method are widely used in many different combinatorial optimization problems. As the name "greedy" suggests, the *greedy strategy* always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally

optimal solution. However greedy methods do not always yield optimal solutions except for problems that satisfy the *greedy condition*, where greedy algorithms always yield exact solutions. It is known that the greedy condition can be characterized as a *matroid* (a collection of independent subsets that satisfies some axioms) [Schrijver et al., 2003] or greedoid [Björner and Ziegler, 1992]. A greedoid is a generalization of a matroid in the sense that it does not necessarily satisfy the *hereditary condition* (defined in II.2.6.1). Both greedoids and matroids are sufficient and necessary conditions for proving the correctness of greedy algorithms, albeit over different objective functions. Schrijver et al. [2003] prove that necessity and sufficiency of matroids hold for arbitrary non-negative objective functions, while Björner and Ziegler [1992] proved that the greedoid is a sufficient and necessary condition for a greedy algorithm over a compatible objective function and R-compatible linear functions (details of the definitions are out of the scope of this thesis, which can be found in Björner and Ziegler [1992]). Furthermore, a matroid is also the sufficient and necessary condition of greedy algorithm for all linear objective functions.

However, in practice, identifying whether a matroid exists in a problem is almost as challenging as determining whether the greedy condition is satisfied. In Section II.2.6, we will identify the greedy condition from a different perspective which is much easier to recognize in practice compared to finding a matroid. In short, a combinatorial optimization problem specified as (3) can be solved greedily if the selector sel_E can be fused into the generator *gen*.

Dynamic programming Dynamic programming (DP) is perhaps the most well-known problem-solving strategy for both *mathematical optimization* and *algorithm design*. It was first developed by Bellman [1954] in 1954. Interestingly, the name "dynamic programming" was chosen not for its descriptive accuracy but for political expediency. Bellman aimed to protect his work from the US Secretary of Defense Charles Wilson who was known for his hostility to mathematical research [Bellman, 1984].

Applications of DP have been found in numerous fields. Classical DP algorithms such as sequence alignment, the *Floyd–Warshall algorithm*, and the *matrix chain algorithm*, among others, have revolutionized many fields in scientific research, spanning from engineering to biology. The wide-ranging influence of DP across various fields has led to numerous misconceptions about the essential properties of DP. Definitions in the literature often differ significantly. This ambiguity originates from Bellman's initial description of DP, which was notably vague. The "archetype" dynamic programming approach described in Bellman [1954]'s original paper was proposed to find the optimal policy for a discrete decision process problem, and the main character for constructing DP is to identify *the principle of optimality*. Below is Bellman [1954]'s definition:

> **Principle of optimality**: *An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decisions.*

In this definition, terminology such as "policy", "decisions", and "initial states" are not rigorously defined. It was not until 1967 when Karp and Held [1967] presented the first rigorous definition of DP. In their discussion, DP was formally characterized as a sequential decision process combined with the principle of optimality. A *policy* is a sequence of decisions; Karp and Held [1967] demonstrated that the optimal policy can be found by identifying a *monotonicity* property, which provides a more rigorous characterization of Bellman's principle of optimality.

Today, it is widely accepted that an efficient recursive program incorporating the principle of optimality is insufficient to become a DP algorithm. The purported essential criteria for identifying a dynamic programming algorithm include ensuring both the principle of optimality and the use of *memoization techniques* in an efficient *recursive* program.

However, debates persist over whether memoization should be considered the defining characteristic of dynamic programming algorithms. In Karp and Held [1967]'s definition, the cost function in sequential decision process is characterized as a *step-by-step* recursive formula, thus it is a recursion that has access only to the previous step. In other words, the use of memoization is not applicable to this recursive definition. In 1995, De Moor [1995] provided a more elegant and concise definition of SDP as the fold operator, as defined above.

At the same time, there are some widely known DP algorithms that do not require memoization at all, for instance, the well-known *Viterbi decoding algorithm*, *Dijkstra's algorithm,* and *the Bellman-Ford algorithm*[7]. These algorithms are widely accepted as the DP algorithms, but there is no memoization involved.

Another contentious point that often sparks debate and confusion is the difference between DP algorithms and greedy algorithms. A notable example of this ongoing debate revolves around *Dijkstra's algorithm*, which a well-known method for solving the *shortest path problem*. For many years, researchers have disputed whether Dijkstra's algorithm should be classified as a greedy algorithm or a DP algorithm. The essential properties of DP were first rigorously outlined by Karp and Held [1967], and in their work, Dijkstra' s algorithm is considered the prototype DP algorithm. However, in one of the most popular textbooks on algorithm design by Cormen et al. [2022], which has more than **67000** citations, Dijkstra' s algorithm is categorized as a greedy algorithm! They justify this classification by noting that "Dijkstra' s algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S, we say that it uses a greedy strategy." Later, Huang [2008] asserted that Dijkstra' s algorithm is a DP algorithm again. The community has clearly not reached a consistent agreement on this matter, and the debate has persisted for decades. This intense dispute highlights the urgent need for a rigorous and formal definition to resolve the ambiguity—a goal we aim to achieve in this thesis.

It is important to recognize that monotonicity alone does not guarantee that the use of memoization will result in a valid dynamic programming algorithm. Even if subsolutions are repeated and the problem satisfies monotonicity, this does not guarantee the existence of a valid dynamic programming algorithm. A counterexample illustrating this issue will be presented in Section III.3.9. Although the problem satisfies monotonicity—meaning the optimal solutions to subproblems remain optimal after updating, which allows us to fuse the selector within the generator—we will show that demonstrate that employing the memoization technique is generally impractical for this problem.

Divide-and-conquer In the divide-and-conquer (D&C) method, the main problem is split into two equal-sized sub-problems, and then sub-problems are subdivided recursively until the sub-problems are small enough (base cases) that they are solved directly. Then the solutions to the sub-problems are

[7]For Bellman-Ford algorithm has a recursion $D_G^r = D_G^{r-1} \times D_G$. Some researchers might think that the distance matrix D_G between vertices is memoized and reused in later recursions. However, if we consider matrix multiplication as a fixed operation, then this recursion indeed depends solely on the results from the previous recursive step.

combined to produce the solution to the original problem. The D&C method is very powerful in practice. For many settings in which divide and conquer is applied, the natural brute-force algorithm may already be polynomial time, and the divide and conquer strategy serves to reduce the running time to a lower order polynomial.

In the later Subsection II.2.4.3, we will provide an alternative definition for the D&C method, which, instead of splitting the problem into two equal halves, considers all possible subdivisions of a problem (while ensuring that subproblems do not overlap). This approach allows us to relate to a very general abstraction of recursive programs—*hylomorphisms*. By doing so, we can clearly see how the classical D&C method (binary split) can be characterized as a special case. Indeed, almost all practical recursive algorithms can be characterized as special cases of hylomorphisms.

Branch-and-bound The Branch-and-Bound (BnB) method is one of the most widely used approaches for solving intractable combinatorial optimization problems. This method relies on the principle of decomposing a complex problem into smaller subproblems through branching rules, analogous to a decision processes in SDP. The BnB method systematically explores all possible solutions to identify the optimal one while avoiding exhaustive enumeration by *pruning* suboptimal solutions early. Suboptimal solutions are identified using bounding techniques, which involve estimating a *lower bound* and an *upper bound* for each subproblem. Assuming configurations with lower objective values are better, if the lower bound for a subproblem is worse than the current best solution, the corresponding solution can be discarded as suboptimal. Another essential aspect of the BnB method is the choice of search strategy. Common strategies include *breadth-first*, *depth-first*, and *best-first*, search. The efficiency of the algorithm can be significantly influenced by the choice of search strategy, affecting both the amount of time after which the optimal solution is found and the number of subproblems that need to be evaluated.

Given this definition, an astute reader might notice that the definition of the BnB method bears a resemblance to the definition of SDP. Indeed, the four main components of the BnB method—*branching rules*, *pruning*, *bounding techniques*, and *search strategies*—each have counterparts in SDP. While search strategies in SDP have not yet been discussed, we will address them in Section II.2.8, where we will explain how different search strategies can be derived from the original definition of SDP.

However, unlike SDP, which is characterized by a datatype-generic abstraction—the *catamorphism*, the BnB method lacks a similar generalization. This is primarily due to the fact that the BnB method has not been rigorously defined. The term "branch-and-bound" is often misapplied across various contexts. Researchers frequently label their algorithms as the BnB whenever they involve any form of branching rule. This leads to the misconception that the BnB method is exceedingly general. Moreover, some literature also classifies many DP algorithms as special cases of the BnB method [Ibaraki, 1977]. However, understanding the loosely defined principles given in BnB studies will not help us understand how to design DP algorithms.

Due to the high degree of similarity between the BnB method and SDP, we propose considering SDP as providing a rigorous definition for branch-and-bound (BnB) methods. This perspective allows us to abstract the BnB method within a formal setting and generalize its principles by studying the higher-level abstraction of SDP, i.e. the catamorphism. This approach is essential for the study of BnB algorithms, particularly since many existing studies on BnB algorithms rely on weak assertions or informal explanations

that do not withstand close scrutiny [Fokkinga, 1991]. Consequently, examining BnB algorithms within an elegant and rigorous algebraic framework will enable us to derive these algorithms in a provably correct and systematic manner. A detailed discussion of the BnB method will be presented in Section II.2.8.

I.2.2.2 Summary of key design components of efficient combinatorial algorithms

This section provides a summary of the principle factors of designing efficient and exact combinatorial algorithms. Here, we aim to provide a brief overview of the factor that affect program efficiency. These techniques and principles will be rigorously discussed in Part II.

1. **Identify combinatorics through geometry.** For most combinatorial optimization problems involving finite hyperplanes and data problems, it is possible to define the problem using combinatorial variables because the data is finite. However, the most straightforward combinatorial variables often involve a very large combinatorial search space $\mathcal{S}$. Identifying the intrinsic combinatorial structure of the problem through geometric insights can significantly reduce the problem's complexity. In Chapter II.3, we analyze the combinatorial structures of several commonly used machine learning models, such as *linear* (*hyperplane*) *models*, *polynomial hypersurface models*, and *Voronoi diagrams*. Additionally, we establish several theorems for enumerating these geometric objects with respect to a dataset $\mathcal{D}$. The results concerning hyperplanes and data should also be applicable to problems beyond machine learning.

2. **Efficient factorization.** After identifying the combinatorics of the problem, an efficient combinatorial generator can be determined if we can find an efficient factorization with respect to the problem's combinatorial structure. Often, the efficient generator for the underlying problem is already known; in such cases, we can use the existing generator or construct a more complex generator by combining basic ones.

3. **Fusion**. Once we have an efficient generator, an exhaustive search algorithm can be immediately constructed by applying (3). However, the computation involved can be greatly reduced by *reordering*. *Fusion* is a special case of computation reordering. In many applications, we can integrate the filtering or selection process within the generator. By doing this, we can achieve greater efficiency because most partial configurations can be eliminated without being fully generated. We will present various *fusion theorems* in Chapter II.2. Efficient combinatorial optimization algorithms can hence be derived by verifying the conditions of these theorems.

4. **Dominance relations.** The dominance relation, often encountered as various upper and lower bound techniques in BnB studies, is based on the concept that some partial configurations will never surpass others in terms of optimality. These suboptimal configurations can be discarded before being fully generated. Ingenious dominance relations are often highly specific to particular problems. An appropriate dominance relation for a problem can significantly impact program efficiency, and multiple dominance relations can be combined to form a more powerful one. We have identified two generic dominance relations, called *global upper bound* and *finite dominance relation*, which are ubiquitous in combinatorial machine learning problems. A more detailed discussion on dominance relations will be presented in Subsection II.2.6.3.

5. **Thinning**. This is a (relational or functional) program used to delete provably suboptimal configurations and is parameterized by dominance relations. It is roughly equivalent to what many authors refer to as a dominance relation. For many dominance relations, suboptimal configurations are identified by comparing them with other partial configurations. A naive implementation of thinning involves comparing every pair of configurations, resulting in quadratic time complexity. This task is non-trivial, as the number of configurations in each recursive generation step can be polynomial or exponential relative to the input size. An ideal thinning function should scan the current partial configurations in linear time. We implement four different thinning functions in Section II.2.6.

6. **Generation ordering**. There are two types of ordering related to generation. The first is *extending ordering*: this is also refered to as the *search strategy* in BnB studies. For a list of configurations, extending ordering determines the sequence in which configurations are updated. Well-known examples are depth-first search and breadth-first search.. The second is *arrangement ordering*: in SDPs, there are multiple decision functions applied to each configuration. The order in which these decision functions are applied determines the arrangement of configurations in the subsequent recursive steps. This is important because in real-world applications, some configurations may be more important than others, making it more profitable to test these configurations first. Additionally, a fixed arrangement ordering can help us save memory by storing the rank number of each configuration instead of the configuration itself. Detailed discussion will be presented in Section II.1.5 of Chapter II.1.

7. **Parallelism**. In solving intractable (NP-hard) combinatorial optimization problems, it is crucial to recognize the value of having a program that is embarrassingly parallelizable. By embarrassingly parallelizable, we mean that no communication is required between processors. This is important for two main reasons: First, the most successful algorithms used in practice are both embarrassingly parallelizable and have low-order polynomial worst-case complexity, such as backpropagation, fast Fourier transform, and sorting algorithms. Second, once a program is embarrassingly parallelizable, it almost guarantees a speed-up of P times when using P processors. As a result, for problems that are embarrassingly parallelizable, the focus shifts from how much "time" is required to solve the problem to how many "resources" we are willing to invest.

Our discussions here relate solely to improving time efficiency. However, in solving intractable combinatorial optimization problems, improving time efficiency often comes at the cost of sacrificing memory. Thus, we frequently encounter a need for a certain trade-off between time and space. A comprehensive discussion of this trade-off will be postponed to Chapter III.4.4 of Part III.

I.2.2.3 Relationships between different combinatorial optimization methods

Although algorithm design methods, such as greedy, DP, D&C and BnB, are often treated as distinct approaches, they are closely related to each other. In our framework, these methods are related according to their *level of abstraction*. This leads to an inclusion relationships as follows:

$$\text{SDP} \subseteq \text{Greedy algorithm} \subseteq \text{BnB} \subseteq \text{General SDP} \subseteq \text{DP}, \text{Classical D\&C} \subseteq \text{General D\&C}, \tag{5}$$

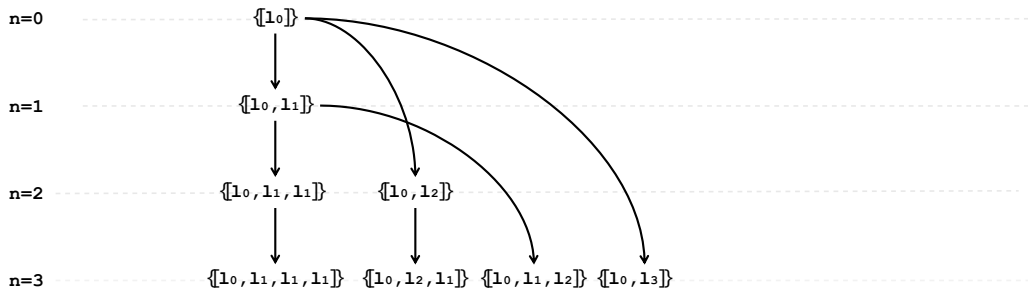


Figure 1: Rod-cutting problem generation tree. In this tree, the segments at each level depend on the segments from all preceding levels. For example, the length-three segments (configurations at level $n = 3$) are constructed using length-two, length-one, and empty segments. Configurations within the same level are grouped to form larger segments of equal length.

where "SDP" refers to the basic sequential decision process as defined in Subsection I.2.1.4. "*General SDP*" and "*General D&C*" refer to basic SDP and D&C augmented with various acceleration techniques, such as thinning and alternative search strategies. The comma between DP and Classical D&C means that they are part of the same hierarchy and are incomparable. In other words, both are encompassed by "General D&C," and both include General SDP.

This inclusion relation, outlined in (5) is derived from the inclusion relations of their abstractions. A more detailed exposition of this inclusion relation will be provided in Section II.2.9 of Chapter II.2.

I.2.2.4 Example: deriving an efficient dynamic programming algorithm from scratch

After introducing the key components of designing efficient algorithms, to provide some intuition about how to apply our algorithm design framework, we illustrate how to use our theory to construct a well-known dynamic programming algorithm for the rod-cutting problem from scratch. The aim of this example is intuition rather than formal reasoning, emphasizing how the overall algorithm design process should be conducted within our framework, i.e., demonstrating that an efficient algorithm can be derived from an initially inefficient exhaustive search specification.

Example 1. *Rod-cutting problem.* This is a classical combinatorial optimization problem that can be solved efficiently using DP. Assume we have a rod of length N that we want to cut into pieces. Pieces with different lengths are worth different amounts of money. A piece of length i is denoted as l_i, which is worth $w(i)$ dollars. The goal is to maximize the total amount of money obtained from cutting the rod.

The combinatorics of this problem are related to a basic combinatorial structure called *list partitioning* or *segmentation*, i.e., partitioning a list into disjoint *segments*. However, the number of possible non-empty segmentations for a list of size N is 2^{N-1}, hence the exhaustive search algorithm for the rod-cutting problem

is undoubtedly inefficient. Nevertheless, for the rod-cutting problem, we are only interested in the length of each segment. This fact allows for an efficient combinatorial factorization of the rod-cutting problem.

To construct an efficient combinatorial generator, we can analyze small cases first, and the general case can be derived inductively. Consider all possible rod pieces for a rod of length three. Define $\mathcal{S}_n$ to be all possible rod pieces of a length n rod. The all possible rod pieces for length three rod consist of

$$\mathcal{S}_3 = \{[l_0, l_1, l_1, l_1], [l_0, l_1, l_2], [l_0, l_2, l_1], [l_0, l_3]\}, \tag{6}$$

from which we establish the following equivalence relations

$$\begin{aligned}
\mathcal{S}_3 &= \{[l_0, l_1, l_1, l_1], [l_0, l_1, l_2], [l_0, l_2, l_1], [l_0, l_3]\} \\
&= \{[l_0, l_1, l_1, l_1]\} \cup \{[l_0, l_1, l_2]\} \cup \{[l_0, l_2, l_1]\} \cup \{[l_0, l_3]\} \\
&= \{\{[l_0, l_1, l_1]\} \circ \{[l_1]\} \cup \{[l_0, l_2]\} \circ \{[l_1]\} \cup \{[l_0, l_1]\} \circ \{[l_2]\} \cup \{[l_0]\} \circ \{[l_3]\}\} \\
&\Longrightarrow \circ \text{ distributes over } \cup \\
&= \{\{[l_0, l_1, l_1], [l_0, l_2]\} \circ \{[l_1]\} \cup \{[l_0, l_1]\} \circ \{[l_2]\} \cup \{[l_0]\} \circ \{[l_3]\}\} \\
&\Longrightarrow \text{definition of } S_n \\
&= \mathcal{S}_2 \circ \{[l_1]\} \cup \mathcal{S}_1 \circ \{[l_2]\} \cup \mathcal{S}_0 \circ \{[l_3]\},
\end{aligned} \tag{7}$$

where $\cup$ is the set join operator and $\circ$ is the Cartesian product of two sets, defined by concatenating each element in the first set with each element in another set, for instance, $\{[a], [b]\} \circ \{[c], [d]\} = \{[a, c], [a, d], [b, c], [b, d]\}$. The $\circ$ operator has a higher precedence than $\cup$. The generation tree for $\mathcal{S}_3$ is depicted in Fig. 1.

The derivation in (7) follows the following logic. The segments of size n can be constructed from all possible segments with sizes smaller than n. Segments at the same level in Fig. 1 are grouped together and joined with the same segments to construct larger segments in the next generation step. This grouping is possible because of the distributivity in semiring $(\{[\mathbb{X}]\}, \cup, \circ, \emptyset, \{[\,]\})$, known as *generator semiring* [Little et al., 2024], where symbol $\mathbb{X}$ represents a type variable. The segments at the same level share the same length, and these segments will still be the same after appending the same new segments. For instance, the segments of length two, $\{[l_0, l_1, l_1], [l_0, l_2]\}$ are grouped together by joining $\{[l_1]\}$ once, without need to join $\{[l_1]\}$ separately to $\{[l_0, l_1, l_1], [l_0, l_2]\}$, because all segments of size two can only join segments of size one to construct segments of size three.

Analogous to the above derivation, we can derive the following recursion inductively

$$\mathcal{S}_n = \bigcup_{1 \le i \le n} \mathcal{S}_{n-i} \circ \{[l_i]\}, \tag{8}$$

this factorization is a consequence of the principle of optimality, which exploits the distributivity between $\circ$ and $\cup$ operators.

If we replace semiring $(\{[\mathbb{X}]\}, \cup, \circ, \emptyset, \{[\,]\})$ with $(\mathbb{R}, \max, +, -\infty, 0)$ in recursion (7), we have following recursion

$$P_n = \max_{1 \le i \le n} P_{n-i} + w(l_i), \tag{9}$$

where $P_n = \max(W(\mathcal{S}_n))$ is the maximal profit over all possible rod pieces of length n, and $W(\mathcal{S}_n)$ evaluate the profit of each configuration in $\mathcal{S}_n$. The recursion (9) is the well-known DP algorithm for the

rod-cutting problem. To derive (9) we need to use the following distributivity

$$\max\{w(l_i) + w(s) \mid \forall s \in \mathcal{S}_{n-i}\} = w(l_i) + \max\{w(s) \mid \forall s \in \mathcal{S}_{n-i}\}. \tag{10}$$

This approach is also known as *semiring fusion* [Emoto et al., 2012]. Indeed, Little et al. [2024] have shown that we can freely swap any semirings if we have a recursion with type information similar to (8). This is a consequence of Wadler [1989]'s *free theorem*. Little et al. [2024]'s result is based on semiring distributivity; in Section II.2.5.5 in Chapter II.2, we will see how the distributivity can be generalized to *monotonicity* by using *relational algebra* [Bird and De Moor, 1996]. However, both monotonicity and distributivity are **sufficient conditions** for proving fusion, they are **not necessary** in certain special cases (see exercise 1.17 and exercise 7.6 and in [Bird and Gibbons, 2020]).

I.2.3 Structured recursion schemes

This thesis is focused on designing *recursive* optimization algorithms. *Structured recursion schemes* are well-studied in the functional programming community. Given the relevance of this field to our work, this section provides a brief introduction to the key concepts of structured recursion schemes and summarizes its developmental history. This will help readers gain a better understanding and appreciation of the abstract concepts introduced in Chapter II.2 of Part II.

I.2.3.1 What is recursion?

In the computer science community, the word "*recursion*" usually refers to as a function or a process that is defined in terms of itself. The Fibonacci sequence is a well-known recursion defined as

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2). \end{aligned} \tag{11}$$

Recursions are also used for definitions; many mathematical objects are formally defined by recursive rules. For example, the natural numbers are defined as follows: zero is defined as a singleton natural number, and the successor of any natural number is also a natural number. The singleton natural number zero serves as the "seed" for generating all other natural numbers.

Recursions can be classified in various ways. One common classification in computer science divides recursions into two categories—*structured recursion* and *generative recursion*—based on how the recursive procedure processes the data. Their definitions will be explained next.

I.2.3.2 Structured recursion and generative recursion

Structured recursion In *structured recursion*, the recursive call is made using a substructure of the input or the subsets of the input data. The Fibonacci sequence recursion (11) above is an example of a structured recursion. Structured recursions includes almost all recursion with tree structure, binary tree search and binary tree creation, are examples. Similarly, all algorithms based on SDP are structured recursion as well.

Structured recursions process progressively smaller portions of the input data until reaching the base cases, ensuring a *termination guarantee*. Consequently, structured recursions have been extensively studied in optimization problems, as guaranteeing termination is essential to prevent optimization programs from running indefinitely.

Generative recursion *Generative recursion*, in contrast, do not necessarily feed smaller inputs to their recursive calls. Instead, generative recursion creates an entirely new set of data from the given input. Consequently, proving the termination of generative recursions is often non-trivial. Examples of generative recursions include the *Newton-Raphson method* and the *Euclidean algorithm* (for calculating the greatest common divisor, GCD). For two integer a and b, and $a > b$, the GCD can be calculated using the following recursion:

$$\begin{aligned} GCD\,(b,0) &= b \\ GCD\,(a,b) &= GCD\,(b,r)\,, \end{aligned} \tag{12}$$

where $r = a \mod b$ is the remainder of a divide b. In this recursion, the remainder r is not the substructure of a or b, which is generated from them.

I.2.3.3 Developmental history of constructive algorithmics

In studies of recursive functions and recursive optimization algorithms, many results looked alike, but they could not be expressed as a single theorem. Over forty years ago, Hoare [1972] first observed that there are certain close analogies between the methods used for structuring data and the methods for structuring a program which processes that data.

The Bird-Meertens Formalism The pioneering work of Meertens [1986] and Bird [1987, 1989] created a new programming formalism for structuring data and the method for processing these data. This formalism is the calculus built around recursive/co-recursive datatypes and homomorphisms on those datatypes. These datatypes are typically various forms of trees or datatypes similar to trees (for example, natural number Nat, finite list $List$, binary trees $Btree$, etc.), and the homomorphisms on trees are often called *fold* operators. The advantage of deriving programs by using the fold operator is that we can use *initiality* to prove the equality of the program rather than tedious induction. It was later known as *constructive algorithmics*, *the Bird-Meertens formalism*, or *Squiggol formalism*, due to its use of "squiggly" symbols.

Initial algebras, recursive datatypes and catamorphisms Datatypes such as Nat, $List$, and $Btree$ look very similar, and their homomorphisms have the same recursive structures. It is reasonable to consider using abstract language to generalize all these datatypes. Later on, Goguen et al. [1975] demonstrated that abstract data types can be modeled using initial algebras, providing an enormous amount of abstract power to systematically construct data types.

This allows us to construct datatypes systematically. These datatypes modeled by initial algebras are called *recursive* (*inductive*) *datatypes*. Dually, the datatypes modeled by *terminal coalgebras* are called *co-recursive* (*coinductive*) *datatypes* (conatural numbers, colists, streams, etc.) [Fokkinga, 1992].

Homomorphisms between the *initial* **F**-algebra and **F**-algebras are called *catamorphisms*, as named by Meertens [1988]. Initial algebras are the abstraction of recursive datatypes, hence catamorphisms are the abstraction of homomorphisms of different datatypes, in other words, catamorphisms are an abstraction of the fold operator. Indeed, in the categorical concepts of **F**-algebras, **F**-algebra homomorphisms are generalizations of the corresponding concepts in universal algebra [Wechler, 2012]. These concepts will be explained in more detail in Section II.2.2.

The algebra of programming Subsequently, De Moor [1994], Bird and De Moor [1996] generalized the Bird-Meertens formalism to specifications viewed as input–output *relations* (non-deterministic functions) instead of *total functions*. This was later named the *algebra of programming*.

We are very familiar with the algebra of numbers, and we know how to manipulate numbers using algebraic rules. Similarly, the algebra of programs is similar to the algebra of numbers: we begin with the specification of a class of problems and apply certain algebraic rules (theorems) to reason about programs. The solution to the problem is obtained by verifying the conditions of these rules. This solution may take the form of a function, but more commonly, it is a relation characterized by a recursive program. A relational and recursive program can then be *refined* into a recursive program which is defined as a function.

Generalizing total functions to relations is an inevitable step in program derivation. De Moor [1994] provides two reasons for extending a total function framework to a relational one. First, pure functional programs are inadequate for optimization problems, as many such problems have *non-unique* solutions. Second, non-deterministic programs are beneficial in program derivation because **not** all functions have **inverses**, whereas **every** relation has a **converse**. Furthermore, the specifications of many problems can be more naturally expressed in terms of relations. We will provide a detailed discussion on the motivation for using a relational formalism instead of a functional one in Subsection II.2.5.1.

Structured recursive schemes - a zoo of recursive morphisms In the functional programming community, the relationship between data structure and program structure is often expressed through structured recursion schemes, which are widely used in functional languages such as Haskell. Structured recursions are often characterized by a variety of morphisms, which are programs operating over recursive (inductive) or co-recursive (co-inductive) *datatypes*. These morphisms combine to form structured recursion schemes. The simplest form of structured recursion, SDPs, is characterized by catamorphisms.

The catamorphism is the primitive recursive morphism that corresponds to the sequential decomposition of a problem. However, due to the limited expressiveness of the fold operator (catamorphism), many interesting and useful generalizations have been proposed, resulting in a diverse collection of recursive functions (morphisms) [Yang and Wu, 2022]. For instance, the *paramorphism* recursion models *primitive recursion* by allowing the recursive body to access not only the results of recursive calls but also the substructures on which these calls are made. *Zygomorphisms* are a variation of catamorphisms aided by an auxiliary function. *Histomorphisms* enable memoization techniques in recursive programs, making contextual information available to the body of the recursion.

Of particular note is the *hylomorphism*, a special type of morphism that encapsulates the essential properties of *divide-and-conquer* recursion. A more formal characterization of this approach will be pro-

vided after we introduce the notions of coalgebra and algebra in Subsection II.2.4.4. Hylomorphisms, as noted by Hu et al. [1996] are the foundation of almost all forms of recursion, including both structured and generative recursion. Nearly all practical recursive functions can be transformed into hylomorphisms.

Unifying structured recursive schemes The various generalizations of catamorphism can be perplexing to the uninitiated, as many of these morphisms appear quite similar. Numerous research effort has been invested to further generalize these recursive morphisms. The first attempt to unify them was the identification of recursion schemes from *comonads*, as proposed by Uustalu et al. [2001]. *Comonads* capture the general concept of "evaluation in context" [Milewski, 2018], allowing contextual information to be available for every recursive call. This pattern subsumes paramorphisms, zygomorphisms, and histomorphisms.

Hinze et al. [2013] proposed a further unification using *adjunctions*. Their approach stemmed from the observation that every adjunction induces a comonad, and every comonad can be factored into adjoint functors. This approach has proven to be well-founded, subsuming all morphisms in structured recursion schemes.

I.2.4 Category theory and Haskell

In this thesis, we use Haskell, a functional programming language, as the primary tool for illustration instead of employing mathematical-style functions. Despite its relative rarity in machine learning research compared to imperative languages such as C++, Python, or MATLAB, it offers several notable advantages. In particular, it is side effect-free, strongly-typed (which helps with code correctness), while its support for point-free functional composition and currying results in more concise and readable code [Bird and Gibbons, 2020]. Furthermore, functional programming languages are particularly suitable for implementing categorical concepts, providing a natural and convenient framework for expressing and exploring ideas from category theory. Additionally, Haskell's strengths in handling recursion, a central theme in our optimization strategies, make it a powerful and illustrative tool for our research.

I.2.4.1 Categories and functors

Of the branches of mathematics, category theory is one which perhaps fits the least comfortably in set-theoretic foundations [Univalent Foundations Program, 2013]. Category theory is an abstraction of composition and relations. This is particularly useful, as one way to view programs is as compositions of functions. Hence category theory is a powerful tool to model programming languages. Category theory abstracts structure and pattern as "categories" and studies the relation between different categories. A category is a collection of objects and morphisms between *objects*, and *morphisms* can be composed to create new morphisms. In other words, a category has the property that it is *closed* with respect to morphism composition. The formal definition of category is given as follows.

Definition 1. *Category.* A category $\mathcal{C}$ consists of

- A collection of objects usually denoted by uppercase letters X, Y, Z,
- A collection of morphisms usually denoted by lowercase letters f, g, h,

such that

- Each morphism has assigned two objects, called *source* and *target*, or *domain* and *codomain*. We denote the source and target of the morphism f by $s(f)$ and $t(f)$, respectively. If the morphism f has source X and target Y, we also write $f : X \to Y$.
- Each object X has a distinguished morphism $id_X : X \to X$, called the identity morphism.
- For each pair of morphisms f, g, such that $t(f) = s(g)$ there exists a specified morphism $g \circ f$, called the *composite* morphism, such that $s(g \circ f) = s(f)$ and $t(g \circ f) = t(f)$. Or, graphically,

$$f : X \to Y, g : Y \to Z \Longrightarrow g \circ f : X \to Z \tag{13}$$

Composition satisfies the following properties.

- *Unitality*: for every morphism $f : X \to Y$, the compositions $f \circ id_X$ and $id_Y \circ f$ are both equal to f.
- *Associativity*: for morphisms $f : X \to Y$, $g : Y \to Z$, $h : Z \to W$, the compositions $h \circ (g \circ f)$ and $(h \circ g) \circ f$, are equal.

Objects in category theory are abstract nebulous entities. All you can ever know about an object is how it relates to other objects—how it connects with them using arrows. If we want to single out a particular object in a category, we can only do this by describing its pattern of relationships with other objects (and itself). This is formalized through the *Yoneda Lemma*—an object in a category is no more and no less than its web of relationships with all other objects.

The essence of the category theory is composition. In programming, composability plays a crucial role; we compose pieces of code to create solutions to larger problems. Therefore, it is self-evident that programming is closely related to category theory. An example of a category is the *fictional category* **Hask** which considers the functional programming language Haskell itself as a category. Haskell has types, functions, identities, and compositions. In the category Hask, the objects correspond to types such as *Int*, *Double* or *String*, while the morphisms correspond to programs or functions. The input and output types of these functions represent the domain and codomain of the morphism. Two programs, f and g, are composable if and only if the output type of f matches the input type of g. A program that consumes an *Int* should not be able to accept a *String*.

However, Haskell does not strictly satisfy the definition of a category. In particular, the same function can have different implementations (code), which is not allowed in a true category. Nonetheless, while the Haskell language is not a perfect model for implementing categorical concepts, it is sufficient to illustrate the ideas presented in this thesis.

Categories can be considered as objects in larger categories, a 2-*category* which is *a category of categories*. The morphisms between categories are called *functors*.

Definition 2. *Functor.* A functor $\mathbf{F} : \mathcal{C} \to \mathcal{D}$ consists of two constituents:

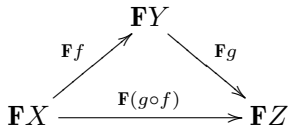
- For each object X of $\mathcal{C}$, an object $\mathbf{F}X$ of $\mathcal{D}$.
- For each morphism $f : X \to Y$ of $\mathcal{C}$, a morphism $\mathbf{F}f : \mathbf{F}X \to \mathbf{F}Y$ of $\mathcal{D}$.

and the following *functoriality axioms* hold:

- *Unitality*: for every object X of $\mathcal{C}$, $\mathbf{F}(id_X) = id_{\mathbf{F}X}$, where id_X is the identity morphism for object X in category $\mathcal{C}$ and $id_{\mathbf{F}X}$ is the identity morphism for object $\mathbf{F}X$ in category $\mathcal{D}$
- *Compositionality*: for every pair of composable morphisms f and g in $\mathcal{C}$, $g \circ f$ can be represented diagrammatically as

$$X \xrightarrow{f} Y \xrightarrow{g} Z$$

we have $\mathbf{F}(g \circ f) = \mathbf{F}g \cdot \mathbf{F}f$. That is, the following diagram commutes:



A functor between categories $\mathcal{C}$ and $\mathcal{D}$ must give an object of $\mathcal{D}$ for every object of $\mathcal{C}$. This leads us to consider how to construct new objects from existing ones. In the realm of programming, the mechanisms for how this is performed are "*type constructors*." A type constructor is a model for a *datatype* or *data structure*. Functors are therefore a special kind of type constructor that operates on both types and functions: functors map a type to a new type, and they also map a function defined on that type to a corresponding function that operates on the associated data structure.

We will explain how category theory assists in constructing *algebraic datatypes* by using *polynomial functors*. Further, relationships between functors are known as *natural transformations*. Natural transformations are highly useful for modeling *polymorphic* functions. Indeed, it has been proved that every natural transformation is a polymorphic function [Wadler, 1989].

I.2.4.2 Universal constructions

In programming, certain objects are made special or characterized by how they relate to others. Indeed, there is a common construction in category theory for defining objects in terms of its relationships (*universal property*), called the *universal construction*. A universal property is interpreted as the fact that objects are uniquely (up to isomorphism) specified by the way they interact with the rest of the category: for *all* objects A in $\mathcal{C}$, there is a *unique* morphism to it. For instance, in Haskell, the *unit type* `()` has the special property that for every other type `a` (in Haskell, we use lower-case letter `a` to represent types), there is a unique function to it, namely `\a -> ()`, an anonymous function that takes a variable `a` as input and returns an empty value. Moreover, up to isomorphism, the unit type is the *only* type with this property. We say that the unit type is defined by its *universal property*.

In this subsection, we will introduce four fundamental universal constructions that are particularly important in our exposition: *terminal objects*, *initial objects*, *products*, and *coproducts*. These universal constructions are well-modeled in Haskell, and we will explore their implementations when they are used in Chapter II.2.

Initial object and terminal object The simplest universal constructions are initial objects and terminal objects. The initial object is the object that has one and only one morphism going to any object

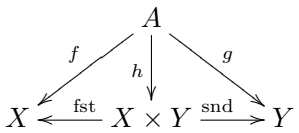
in the category. Dually, the terminal object is the object with one and only one morphism mapping to it from any object in the category. We normally use "0" to represent an initial object, and "1" to represent a terminal object.

The initial object is defined by its *mapping out* property. For instance, in a *partially ordered set* (*poset*) category, an initial object 0 is the smallest element, because there is only one morphism from $0 \to A$ for any A in the *poset* category. Another example is the category of sets and functions, the initial object is the empty set. The definition tells you that you can see this *shape* in *every* other object and in itself. This mapping out property will be very useful in our later discussions, we will model recursive datatypes in terms of initial objects.

Dually, the terminal object is defined by its *mapping in* property. For instance, in the poset category, a terminal object 1 is the greatest element, because there is only one morphism from $A \to 1$ in the poset category. In the category of sets, the terminal object is a singleton.

Product and coproduct Given two sets X and Y, we can always construct their Cartesian product $X \times Y$. This is the set whose elements are pairs (x, y), where $x \in X$ and $y \in Y$. Thinking categorically allows us to generalize the Cartesian product of sets, to do so, we need to think in terms of relationships. By analogy to the Cartesian product of sets, if we want to define a function $f : A \to X \times Y$, we can define a pair of functions $f_X : A \to X$ and $f_Y : A \to Y$, and then define $f(a) = (f_X(a), f_Y(a))$. In other words, a morphism $A \to X \times Y$ in a category of sets is the same as a pair of morphisms $A \to X$ and $A \to Y$, so a product in category theory is about a pair of relations (morphisms) rather than the elements. The product in a category $\mathcal{C}$ is defined as follows.

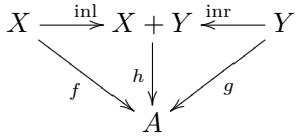
Definition 3. *Product.* Let X and X be objects in a category $\mathcal{C}$. A product of X and Y consists of three things: an object, denoted $X \times Y$ and two morphisms $\text{fst} : X \times Y \to X$ and $\text{snd} : X \times Y \to Y$, with the following universal property: For any other such three things, i.e. for any object A and morphisms $f : A \to X$ and $g : A \to Y$, there is a unique morphism $h : A \to X \times Y$ such that following diagram commutes



Categorically, we will frequently denote h by $h = \langle f, g \rangle$.

Dually, we can define the *coproduct* with the mapping-out property.

Definition 4. *Coproduct.* Let X and Y be objects in a category $\mathcal{C}$. A coproduct of x and y consists of three things: an object, denoted $X + Y$ and two morphisms $\text{inl} : X \to X + Y$ and $\text{inr} : Y \to X + Y$, with the following universal property: For any other such three things, i.e. for any object A and morphisms $f : X \to A$ and $g : Y \to A$, there is a unique morphism $h : X + Y \to A$ such that following diagram commutes



Categorically, we will frequently denote h by $h = [f, g]$.

I.2.4.3 Introduction to Haskell

Types and values A type is a kind of label that every expression has. It tells us in which class of things that expression fits. The expression `True` is a Boolean type `Bool`, `1` is an integer type `Int` or `Double`, etc. We will use only simple types, such as Booleans `Bool`, characters `Char`, strings `String`, numbers of various kinds (integers `Int`, double floating points `Double`), and lists `[a]`. Most of the functions we use can be found in Haskell' s standard *Prelude* (the Prelude library), or in the library `Data.List`. In Haskell, we use lower-case letters `a`, `b`, `c` to indicate type variables. A function $f : A \to B$ is the same as the function `f :: a -> b` in Haskell. Throughout the thesis, we will frequently employ this Haskell-style definition rather than the more familiar math style. However, it is worth noting that both *type variables* (variables used to define the function type)) and *value variables* (variables used to define the function body) in function definitions are represented with lower-case letters, which could potentially lead to confusion. For example, a function definition in Haskell might be represented as

```
f :: a -> b -> Bool
f a b = True
```

Although both the type variables and value variables of this function are denoted using lower-case letters, we can easily distinguish them: everything following `::` are types or type variables, while variables directly applied to a function after white space represent value variables.

List and tuple The type *list* is a *homogenous datatype*. It stores several elements of the *same* type. In Haskell, lists are denoted by square brackets, and the values in the lists are separated by commas. We can append an element at the beginning of a list using the `:` operator (also called the *cons* operator), or we can join two lists by using the `++` operator `[a,b] ++ [c,d] = [a,b,c,d]`. The length of a list is calculated by the function `length :: [a] -> Int` which counts the number of elements in a list.

Tuples are similar to lists in that they store multiple values in a single entity. However, tuples are used when you know exactly how many values you want to combine, and their type depends on both the number and types of the components. In other words, the tuple size is not changeable at run-time; it can therefore be statically type-checked in Haskell. Unlike lists, tuples do not require their components to be of the same type. In Haskell, tuples are denoted with parentheses, and their components are separated by commas. For instance, we can store two elements, `a :: Int` and `b :: Bool` with different types in a tuple `(a,b) :: (Int, Bool)`. By contrast, you cannot store elements with different types in a Haskell list.

Algebraic datatypes, type synonyms, and type inference Haskell's Prelude includes many built-in datatypes such as, `Bool`, `Int`, `Char`, `Maybe` etc. But how can you define your own datatypes? One way

to do this is by using the `data` keyword. For example, the built-in datatype `Bool` is defined as

```
data Bool = True | False
```

In this definition, the keyword `data` means that we are defining a new datatype. The part before the "`=`" specifies the type identifier, which in this case is `Bool`. The parts after the "`=`" are *value constructors*, which define the possible values that this type can take. The `|` is read as *or*. So we can read this as the `Bool` type can have a value of `True` or `False`. Both the type name and the value constructors have to be capital case.

Datatypes can be *parameterized*, for instance, `Maybe` is a *type constructor*, it is not a type. It receives a type as a parameter to construct a new type. It is defined as

```
data Maybe x = Nothing | Just x
```

where type `x` is the type variable, the *field* of the `Maybe` type constructor. Field, here refers to parameters. In this case, the variable of this datatype can be any type, for instance, the field `x` can be can be `Int`, or `String`, corresponding to type `Maybe Int`, or `Maybe String`, and we call the field with free variables the *free field* denoted by variable `x`. In programming, `Maybe x` means a value of type `x` within the context of possible failure attached. Another way to define datatype is to use `newtype` keyword and we will explain this in more detail when we use it later.

We can use the `data` keyword to construct more sophisticated datatypes that are parameterized by more than one type variables or *constant types*. For instance, we can define the datatype `Either` in Haskell as

```
data Either x = Left Int | Right x
```

where the first field of type constructor `Either` is fixed to `Int`, this field is then called a *constant field* or a variable of constant/fixed type. If this constant field is unknown, we denoted it as `a` to distinguish the free field `x`.

Type synonyms just give some types different names so that they make more sense to someone reading code and documentation. In Haskell Prelude, the type `String` is a synonym for a list of characters `[Char]`

```
type String = [Char]
```

Haskell has a *type inference* system, which means, for instance, that if we use a number, we do not have to tell Haskell it is a number. Haskell can infer missing type information where possible, so that, in most cases, we do not have to explicitly write out the types of functions or expressions. Therefore, if the codomain of a function `f` matches another function `g`'s domain, we can compose them without explicitly giving their types.

Haskell pattern matching, map and take functions Pattern matching in Haskell is a syntactic construct, it consists of specifying patterns to which some function should conform. When defining functions in Haskell, pattern matching allows defining separate function implementations for different patterns. This leads to concise, readable code. Here is a pattern-matching example in the definition of the function `map`:

```
map :: (a -> b) -> [a] -> [b]
```

```
map _ [] = []
map f (a:xs) = f a : map f xs
```

Here, the underscore `_` indicates that we do not care about the value of the first parameter of the `map` function. The `map` function receives a function of type `f :: a -> b` and a list of type `(a:xs) :: [a]`, then applies that function to every element in the list, producing a list of type `[b]`. When function `map` is called, the patterns will be checked from top to bottom, and when the input conforms to a given pattern, the corresponding function body will be used. So, the order in which you specify these patterns is important, it is always best to specify the most specific ones first and then the more general ones later.

Curried functions and infix section In Haskell, every function officially takes only one parameter. Consequently, it is not possible to define a function that takes multiple parameters commonly implemented in most programming languages, such as the form `f(x,y)`. Instead, all functions in Haskell that appear to take several parameters are actually *curried functions*.

To explain currying, consider the binary function `+ :: a -> a -> a`. This is equivalent to `+ :: a -> (a -> a)`, meaning `+` takes an input of type `a` and returns a *partially applied function* of type `a -> a`. This returned function then takes another input of type `a` and produces a result of type `a`. More specifically, the expression `(+) 1 2` first receives a value `1` and returns a function `(+ 1)`, the function `(+ 1)` receives another value `2`, and returns the summation of `1` and `2`.

Binary operators like $+$ and $*$ are typically written in *infix form* as $a+b$ and $a*b$ in most programming languages and in ordinary mathematics. However, in Haskell, binary functions and other functions are usually written in *prefix form*, as shown with the `(+)` function above. Haskell also allows binary functions to be written in infix form. For example, `a + b`, `a * b`, `a` ``max`` `b` express addition, multiplication and maximum in binary infix form. Any function in prefix form can also be partially applied by using a *section*. To section a function, it is surrounded with parentheses and supplied a parameter on one side. For instance, `(+ 1)` creates a function that takes one parameter and adds `1` to it. Therefore, `(+ 1) 2` is equivalent to `+ 1 2`.

Point-free style in Haskell In *point-wise* style, we describe a function by giving its application and arguments, for example:

```
f :: Double -> Double -> Integer
f x y = round (signum (dist x y))
```

This function calculates the distance between two data points, takes the sign of the result, and rounds it to the nearest integer.

On the other hand, in *point-free* style, a function is described solely in terms of function composition. It is common in functional programming languages to define functions as compositions of other functions, without explicitly mentioning their arguments. For example, we can define the function `f` above using point-free style as:

```
f :: Double -> Double -> Integer
f = ((round . signum) .) . dist
```

Function composition is indicated by the dot `.` operator in Haskell. This programming style reduces the number of parentheses needed, resulting in more readable code.

Fold functions In Subsection I.2.1.4, we defined the `foldr` function to illustrate SDP. This function, which is part of the Haskell Prelude, recursively folds a list from right to left. Similarly, there is a recursive function that folds a list from left to right, known as `foldl`. The `foldl` function (also called *left-fold*) can be defined recursively as follows:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f e [] = e
foldl f e (a:x) = foldl f (f e a) x
```

Similar to the definition of `foldr`, `f` is a binary function applied between the seed value `e` and the head of the list. This application produces a new accumulator value, and the binary function is then called with this new value and the next element in the list. The `foldl` function can be illustrated with the following, informal example

```
foldl (+) e [x1,x2,..,xN] == ((e + x1) + x2) + .. + xN)
```

Another useful function is `foldl1`, which applies only to non-empty lists

```
foldl1 f (a:x) = foldl f a x
```

List comprehension A set can be constructed from another set. This is known as *set comprehension* in set theory, for instance, $\mathcal{S} = \{2 \times x \mid x \in \mathbb{R},\ x \leq 10\}$. *List comprehension* is similar: it is a way to construct a list from existing lists. An example of list comprehension in Haskell is:

```
[f x | x <- xs, f <- fs]
```

which produces a list of values of the form `f x`, the operator `f` is taken one-by-one from a list of functions `fs` and the element `x` is taken one-by-one from a list `xs`. For instance, the expression `[f x | x <- [1,2,3], f <- [(+1),` produce the list `[2,3,4,3,4,5]`.

Polymorphic functions Some functions operate on elements of different types, regardless of the specific shape of the elements. These are known as *polymorphic functions* [Wadler, 1989]; "poly" means many or more than one. A good example to illustrate this is the `reverse` function

```
reverse :: forall a. [a] -> [a]
reverse l =  rev l []
  where
    rev []     a = a
    rev (x:xs) a = rev xs (x:a)
```

In this type declaration, `a` is a type variable that can represent any type. The `reverse` function takes a list and returns a new list with the elements in reverse order. For instance, `reverse [1,2,3] = [3,2,1]` and
`reverse "hello"= "olleh"`.

Functors in Haskell In Haskell, the standard functor class are *endofunctors* on the category **Hask**. Categorically, a functor maps between two categories if it maps every object in category $\mathcal{C}$ to another category $\mathcal{D}$. An endofunctor is a special type of functor where the source and target categories are the same, i.e., $\mathbf{F} : \mathcal{C} \to \mathcal{C}$. Thus a functor in Haskell is an endofunctor that maps objects in **Hask** to objects in **Hask**. Since objects in Haskell are types, a functor on objects corresponds to a polymorphic function on types—a function with type $\mathbf{F} : X \to \mathbf{F}X$. In Haskell, this is represented as `func :: x -> func x`, and one example of such a function is the `Maybe` type constructor.

Specifically, the functor on objects in Haskell corresponds to an algebraic datatype `func x`, which has exactly one free field `x` and is constructed using the `data` keyword. Functors with two free parameters are called bifunctors; however, the discussion of bifunctors is beyond the scope of this thesis.

Nevertheless, a functor is more than just a function on objects—it also involves the objects and the morphisms connecting them. A functor maps morphisms from the domain category to morphisms in the codomain category, preserving the structure of connections. Therefore, if a morphism $f : A \to B$ in the domain category connects object A and B, the functor $\mathbf{F}$ maps function f to a function $\mathbf{F}f : \mathbf{F}A \to \mathbf{F}B$ in the codomain category, which connects $\mathbf{F}A$ and $\mathbf{F}B$. In for the standard Haskell functor, we define the morphism part of the functor as

```
class Functor func where
    fmap :: (a -> b) -> func a -> func b
```

where the `class` keyword defines the *typeclass* in Haskell, which is not a class of types but rather a class of type constructors. For different datatypes, we need to define their `fmap` implementation separately . We can implement the functor instance for the `Maybe` datatype as follows

```
fmap _ Nothing = Nothing
fmap f (Just x) = Just (f x)
```

The definition of `fmap` for `Maybe` is very simple: if we map a value `Nothing`, then simply return `Nothing`. If the value is `Just x`, where `x` is some value, then `fmap` applies the function `f` to the contents of `Just x`. This is the same as the categorical notation of functor $\mathbf{F}(f) : (A \to B) \to (\mathbf{F}(A) \to \mathbf{F}(B))$.

Defining `fmap` for each new datatype can be tedious. Fortunately, Haskell has a built-in method for computing these automatically, using the `deriving` keyword. Because the standard functors in Haskell are polynomial functors, it is possible to automatically generate the functor behavior of datatypes within the functor class `class Functor`. This allows the definition of `fmap` for a particular datatype to be generated automatically. For example, the `fmap` for the `Maybe` datatype can be automatically derived by writing:

```
data Maybe x = Nothing | Just x deriving Functor
```

I.3 An overview of the thesis

This section outlines the structure of the research, the questions addressed, and the main results of the thesis. The research is organized into two main parts, Part II theories and Part III applications.

I.3.1 General theory -Principles for designing efficient combinatorial optimization algorithms

Part II explores three interrelated themes: combinatorial generation, constructive algorithmics, and geometry in machine learning.

Constructing efficient exhaustive search algorithms -Combinatorial generation

As discussed, designing an efficient combinatorial optimization algorithm hinges on developing effective brute-force algorithms, and efficient brute-force algorithms for most basic combinatorial structures are already known. In Chapter II.1, we present various efficient generators based on SDP. Additionally, abstract combinatorial generators in terms of different datatypes are illustrated in Section II.2.3, and these generators will be examined in a more structured and streamlined way in this section. Our goal is to provide a comprehensive library of generators for various combinatorial structures, which can be used directly when designing algorithms.

Algorithm design principles -Constructive algorithmics

This chapter introduces the core theory of the thesis. The terminology introduced in Section I.2.1 and Section I.2.2 will be explained more formally. There are several points of focus in this Chapter, the first part fixates on the *datatype-generic abstraction* of SDPs—*catamorphisms*, and includes a systematic study of how to construct catamorphism-based combinatorial generators. A significant portion of the remaining exposition in this chapter will introduce the *theory of the algebra of programming* [Bird and De Moor, 1996]. This theory, *a relational calculus formalism for program derivation*, will be explored with a particular focus on its application to the context of this thesis—combinatorial optimization. This formalism provides a systematic approach to deriving efficient programs from provably correct specifications, providing us with simple, cogent derivation steps. We will see in our applications that the algorithms designed using this formalism are *concise* and *readily comprehensible*.

Combinatorial geometry -Geometric algorithms and the essential combinatorial properties of ML models

In the final theme, *geometry*, we expound upon two key topics: how geometric insights can simplify the combinatorial complexity of apparently computationally intractable problems, and how the algorithmic design principles introduced in this thesis aid in solving fundamental problems in combinatorial geometry. We propose two novel *cell enumeration* algorithms, along with a reformulation of the well-known *reverse search algorithm* [Avis and Fukuda, 1996]. Our proposed algorithms and the reformulated reverse search algorithm are *optimally efficient* in terms

of worst-case complexity and *embarrassingly parallelizable*. By contrast, the parallelization method introduced by Avis and Fukuda [1996] requires a significant amount of communication between processors.

Furthermore, we explore the geometric and combinatorial properties of two important geometric objects: *Voronoi diagrams* and *hyperplanes*. Despite their simplicity, these objects are implicit to many machine learning models, as nearly all widely-used ML models can be represented as *piecewise linear functions*. Understanding these objects is crucial for grasping the combinatorial properties of many fundamental machine learning problems, which we will analyze in detail in the third part of this thesis.

I.3.2 Specialized theory -designing tractable algorithms for fundamental problems in machine learning

Part III analyzes the essential combinatorial properties of four critical problems in machine learning: the *classification* problem (with linear or polynomial hypersurface decision boundaries), *K-clustering* problems (including K-means and K-medoids), *empirical risk minimization* for *feedforward neural networks*, specifically those with rectified linear unit (ReLU) activation functions, and *decision tree* problems (with axis-parallel, hyperplane, and hypersurface decision boundaries). Our analysis reveals that all these problems have polynomial combinatorial complexity.

Consequently, polynomial-time combinatorial optimization algorithms for these problems can be derived directly by combining the basic combinatorial generators introduced in the general theory in this thesis. Furthermore, we also discuss specific acceleration techniques applicable to each problem. The use of these techniques enables the construction of algorithms that are provably faster than their worst-case complexity. In the best case, we can have nearly linear-time algorithms for some of these problems.

Classification problems

In Chapter III.1, we analyze the classical classification problem involving linear or polynomial hypersurface decision boundaries, with a particular focus on the 0-1 loss objective function, which aims to minimize the number of misclassified points. We will demonstrate that the combinatorial complexity of the linear classifier is $O\left(N^D\right)$, where D is the dimension of feature space. Two representations of the hyperplane will be examined, both of which show that an algorithm with $O\left(N^{D+1}\right)$ time complexity can be constructed to solve this problem. For polynomial hypersurface decision boundaries, the algorithm has a complexity of $O\left(N^{G+1}\right)$, where $G = \begin{pmatrix} D+W \\ D \end{pmatrix} - 1$ and W is the degree of the polynomial used to define the hypersurface. The term $\begin{pmatrix} N \\ K \end{pmatrix}$ denotes the ordinary binomial coefficient, defined as $\begin{pmatrix} N \\ K \end{pmatrix} = \frac{N!}{K! \times (N-K)!}$.

Empirical risk minimization for feedforward neural networks with ReLU activations

In Chapter III.2, we present the first algorithm for obtaining the empirical risk minimization (ERM) solution for 2-layer ReLU networks. The proposed algorithm has a worst-case com-

plexity of $O\left(N^{DK}\right)$, where K denotes the number of hidden nodes, and D is the dimension of feature space. Results from complexity theory indicate that this approach is optimal in terms of worst-case complexity. To extend the network with additional hidden layers, a greedy training approach can be employed, but the result will be non-optimal.

Decision tree problems

We propose **two** unified algorithms for solving decision tree problems that can handle *decision trees with binary feature data*, *axis-parallel splits*, *hyperplane splits*, and *polynomial hypersurface splits*. Assume there are K splitting rules in total and the dataset lies in $\mathbb{R}^D$ feature space, the worst-case complexity of the proposed algorithms is as follows: for decision trees with binary feature data, the complexity is $O\left(D^K\right)$; for axis-parallel decision trees, the complexity is $O\left((ND)^K\right)$, for hyperplane splits, it is $O\left(N^{KD}\right)$, and for polynomial hypersurface splits, it is $O\left(N^{KG}\right)$.

K-clustering problems

In Chapter III.4, we present two algorithms for solving the K-*means* and K-*medoids* problems separately. The algorithm for solving the K-means problem has a worst-case time complexity of $O\left(N^{K+(K-1)D}\right)$, where K is the number of clusters and D is the dimension of feature space. The K-medoids algorithm has a worst-case complexity of $O\left(N^{K+1}\right)$. Additionally, we propose a specialized algorithm for solving the 2-means problem, which achieves a reduced complexity of $O\left(N^{D+1}\right)$.

I.3.3 End-to-end implementation in Haskell

In the final section, we present the complete derivation of Haskell implementations from scratch, of two example problems: the 0-1 loss linear classification problem and the K-medoids problem. We provide comprehensive implementations in Haskell, along with detailed computational experiments. These examples use most of the techniques introduced in the general theory of this thesis.

I.4 Contributions

I.4.1 Publication and software

Publications

1. Xi He, Max A. Little. Provably optimal decision trees with arbitrary splitting rules in polynomial time. *ArXiv preprint, ArXiv:2503.01455*, 2025.

2. Xi He and Max A. Little. EKM: an exact, polynomial-time algorithm for the K-medoids problem. *ArXiv preprint ArXiv:2405.12237*, 2024.

3. Max A Little, Xi He, and Ugur Kayas. Polymorphic dynamic programming by algebraic shortcut fusion. *Formal Aspects of Computing*, May 2024.

4. Xi He, Waheed Ul Rahman, and Max A. Little. An efficient, provably exact algorithm for the 0-1 loss linear classification problem. *ArXiv preprint, ArXiv:2306.12344*, 2023.

Software

1. Xi He and Max A. Little. Exact 0-1 loss linear classification algorithms, April 2023.
URL: https://github.com/XiHegrt/E01Loss.

I.4.2 Contributions in detail

General theory

1. In Section II.1.4, we implement several *unproven* recursive definitions of ranking and unranking functions for various combinatorial Gray codes using Haskell. In contrast, the classical definitions were traditionally presented in the form of pseudo-code [Kreher and Stinson, 1999, Ruskey, 2003]. We included these generators for the sake of completeness in this thesis and verified their correctness through manual verification, despite their not being utilized elsewhere in this work. The reason for their inclusion is their importance to the study of combinatorial generation; it would be a shame to omit discussion of them. A constructive algorithmic proof of their correctness would be an intriguing topic to explore in future research.

2. In Section II.1.5, we combine the use of a combinatorial Gray codes and sequential decision process generators, which can generate configurations in Gray code ordering *without* explicitly running the unranking function. This generator proves to be very useful in applications involving nested generators, as it enables us to store configurations using their integer representations, thereby saving an enormous amount of memory. To the best of our knowledge, this type of generator has not been previously reported in the literature.

3. In Subsection II.2.3.3, we present two generic join-list generators for generating K-combinations and K-permutations based on arrays, making them well-suited for modern hardware optimized for array-based operations. Operations on these contiguous memory structures reduce the probability of cache misses, thereby enhancing the efficiency of our program. Alternative methods are either based

on lists [Bird, 1987, Kreher and Stinson, 1999, Jeuring, 1993] or employ a one-by-one enumeration approach [Ruskey, 2003].

4. In Subsection II.2.3.4, we propose *cross-product fusion principles* for combining simple catamorphism generators into complex ones. We believe the potential implications of this generator are far more significant than the examples discussed in this thesis. Introducing the cross-product fusion is a key step in completing the design of a combinatorial generator, which we believe has the potential to revolutionize the study of combinatorial generation and has huge influence on the fundamental study of combinatorics and combinatorial optimization. For more details, we will discuss the potential implications of these principles in Conclusion V.

5. In Subsection II.2.6.3, we introduce two novel and generic dominance relations prevalent in machine learning. The first, termed the *global* upper *bound*, has been informally employed in studies of branch-and-bound (BnB) algorithms, such as those by He and Little [2023b,a], Nguyen and Sanner [2013], Lin et al. [2020], Hu et al. [2019], Zhang et al. [2023], Demirović et al. [2022], Aglin et al. [2021, 2020], Demirović and Stuckey [2021], Nijssen and Fromont [2010]. However, a rigorous formalization has not been previously reported. The second, called the *finite dominance relation*, was applied by us in a linear classification problem He and Little [2023a]. The version presented in Subsection II.2.6.3 is far more generic; a special case of this dominance relation appears in Zhang et al. [2023].

6. In Section II.2.8, we reformulate the branch-and-bound method using catamorphisms, where different search strategies and bounding techniques can be specified by defining two relations as inputs to the program. In contrast, previous discussions in functional programming community either focus on deriving BnB algorithm for a particular problem [Bird and Hughes, 1987] or address only depth-first search without including any discussion of bounding techniques [Fokkinga, 1991] .

7. In Section II.3.2, we present a detailed analysis of the relations (including counting and incidence relations) concerning dichotomies and cells of the dual arrangement in *inhomogeneous coordinates*, whereas the classical understanding in combinatorial geometry and oriented matroid theory is based on *homogeneous coordinates* (projective space) [Edelsbrunner, 1987, Fukuda, 2016].

8. We provide a novel linear classification theorem 12 for solving the linear classification problem with an arbitrary objective by enumerating all cells (totaling $2\sum_{d=0}^{D}\binom{N-1}{d}$ for N data points in $\mathbb{R}^D$) in the dual space, which is a direct consequence of our analysis of the incidence relations between dichotomies and cells of an arrangement. Furthermore, in Theorem 15, we show that the linear classification problem with a 0-1 loss objective can be solved exactly by enumerating $\binom{N}{D}$ combinations, which is significantly fewer than the number of cells in the dual space.

9. The geometric analysis given by Gerstner and Holtz [2006] for enumerating cells of an arrangement *perfectly matches* the K-combination generator (`kcombs`) described in Subsection II.2.3.3. The way it organizes combinations—grouping combinations of the same size into a single list—perfectly aligns with the need to enumerate cells of an arrangement. Although we have not yet conducted experiments to test the performance of cell enumeration algorithms based on `kcombs`, it is difficult to imagine

whether any new approach in the future could outperform this one in the task of cell enumeration. Our assumptions regarding the superior efficiency stem from the perspective that this geometric representation of hyperplanes is more efficient than the classical one given by Avis and Fukuda [1996], which requires running a linear program to determine the hyperplane. In contrast, our representation relies only on matrix inversion, and the combination generator `kcombs` we propose is the most efficient one available on modern hardware.

10. In Section II.3.3, we also present another cell enumeration algorithm based on linear programming (referred to as the LP-based method), but we later discovered that Rada and Cerny [2018] presented a *dual version* of our algorithm five years earlier. Another distinction is that Rada and Cerny [2018] implement their algorithm using depth-first search, but we employ a breadth-first approach. Since our algorithm is derived from the geometry of point configurations rather than hyperplane arrangements and can be implemented using the 0-1 loss assignment generator over join-list catamorphism in Subsection II.2.3.3, we still consider it a novel contribution. Moreover, applying this cell enumeration algorithm to solve the linear classification problem [He and Little, 2023a], is new.

11. We also reformulated the well-known reverse search algorithm for generating cells of an arrangement in a breadth-first way, which is typically expressed in a depth-first manner in the literature Avis and Fukuda [1996]. The breadth-first version is easier to parallelize compared to the depth-first version, as the depth-first approach involves backtracking, which inevitably introduces communication overhead between processors.

12. In Chapter III.1, we present the first rigorously proven algorithm for solving the 0-1 loss linear classification problem with polynomial-time complexity, whereas the state-of-the-art ad-hoc BnB algorithm proposed by Nguyen and Sanner [2013] exhibits exponential complexity in the worst case (empirical analysis can be found in Chapter IV.1) and is not proven correct yet. In Subsection III.1.4.3, we also discuss how to use a similar algorithmic process for solving the problem with a margin loss objective; the algorithm for solving this problem is also new.

13. In Chapter III.2, we present the *first* algorithm for solving the ERM problem for a 2-layer ReLU neural network based on a *nested-combination* (combinations of combinations) generator. To the best of our knowledge, the discussion on the *recursive nested-combination* is novel, and no such generator has been presented in the literature before, except for the trivial generator using ranking and unranking functions.

14. In Chapter III.3, we provide the first axiomatic definition of *proper decision trees*. These axioms establish a firm mathematical foundation for studying decision tree problems. We prove that only proper decision trees can be uniquely characterized as K-permutations. As a result of this achievement, we develop the first provably correct polynomial-time algorithm for solving the optimal decision tree problem with arbitrary splitting rules that adhere to the axioms and any objective functions that can be specified in a given form. Our results contradict several unproven claims in the literature, such as the incorrect characterization of decision trees that fail to satisfy the axioms as K-permutations [Hu et al., 2019], and the use of Catalan number-style recursion in solving decision tree problems [Demirović et al., 2022, Hu et al., 2019, Lin et al., 2020, Zhang et al., 2023, Aglin et al., 2021, 2020,

Nijssen and Fromont, 2010]. More importantly, we demonstrate that, while a dynamic programming recursion exists for this problem, the use of memoization is generally impractical in terms of space complexity. This finding questions the application of memoization techniques commonly used in the study of decision tree problems with binary feature data [Demirović et al., 2022, Hu et al., 2019, Lin et al., 2020, Zhang et al., 2023, Aglin et al., 2021, 2020, Nijssen and Fromont, 2010].

15. In Chapter III.4, we improved Tîrnăucă et al. [2018]'s result in solving K-means problem from a worst-case complexity of $O\left(N^{K+(K-1)D+1}\right)$ to $O\left(N^{K+(K-1)D-1}\right)$. Although the exponent term does not differ much, our approach has a much smaller constant term, because Tîrnăucă et al. [2018]'s algorithm explicitly enumerates $K+(K-1)\,D+1$ equations out of $NK\,(K-1)\,/2$ equations, whereas our approach enumerates $K+(K-1)\,D-1$ out of $(K-1)\,N$ possible equations.

16. Furthermore, our discussion demonstrating that solving the 2-means problem is equivalent to the classification problem is novel, and the resulting algorithm for solving the 2-means problem is the fastest available. Additionally, in Chapter IV.2, we demonstrate that the naive brute-force algorithm for solving the K-medoids problem based on the `kcombs` generator is more efficient than the state-of-the-art BnB algorithm [Ren et al., 2022] in all tested experiments. Moreover, we found that Ren et al. [2022]'s algorithm frequently produces erroneous solutions on datasets that were not included in their experiments. Furthermore, Ren et al. [2022]'s algorithm exhibits exponential complexity in the worst case, whereas the worst-case complexity of the brute-force algorithm is polynomial.

17. In the final Part IV, we present the Haskell implementations for the 0-1 loss linear classification and the K-medoids problem.

Part II

General theory: Principles for designing efficient combinatorial optimization algorithms

This part explains the core theoretical framework of this thesis. The discussion focus on three interrelated themes: *combinatorial generation*, *constructive algorithmics*, and *geometry in machine learning*. These themes form the foundation for the algorithm design principles discussed throughout the thesis.

The first theme, **combinatorial generation**, focuses on the development of efficient, exhaustive search algorithms. This section presents a range of advanced combinatorial generators based on *sequential decision processes* (SDPs). These SDP-based generators will be further refined and presented in a more structured and abstract form after we introduce the datatype-generic abstraction of SDPs in the following chapter. In addition to introducing these SDP-based generators, this theme explores the most common classes of combinatorial generation techniques in the study of *combinatorial generation* [Kreher and Stinson, 1999, Ruskey, 2003]. Each class of generator serves a distinct purpose and is tailored to specific optimization scenarios, making them uniquely suited to particular contexts and not easily replaceable by others. Furthermore, we introduce a new class of generators called *integer sequential decision process generators* (I-SDPs), which integrate the strengths of existing generators. These new generators offer a better balance in the time-space trade-off, enhancing the design of exact algorithms. The aim of this theme is, therefore, to provide a comprehensive library of exhaustive search algorithms that can be directly employed in the design of efficient combinatorial optimization algorithms.

The second theme, **constructive algorithmics**, introduces the central theory of this thesis. Here, the SDP introduced earlier is formalized as a datatype-generic program known as a *catamorphism*. We therefore reformulate the generators defined over SDPs in a more structured manner using catamorphisms over *cons-list* and *join-list* datatypes. At the same time, we provide a set of rules for constructing complex catamorphism generators from basic ones, enabling the systematic design of more sophisticated generators for more complex combinatorial structures. Furthermore, we also explore Bird and De Moor [1996]'s *theory of the algebra of programming*, with particular a focus on combinatorial optimization. Bird and De Moor [1996]'s theory is a relational calculus formalism that enables the derivation of efficient programs from provably correct, but initially inefficient, specifications. This formalism not only streamlines the design of algorithms but also ensures that they are both concise and comprehensible, illustrating the conceptual and practical value of systematic algorithmic derivation.

Finally, the third theme, **combinatorial geometry**, explores the essential combinatorial properties of many machine learning models by identifying equivalence classes through geometry. While efficient combinatorial generation is critical, the inherent geometry of problems often introduces equivalence relations that can significantly reduce the complexity of the search space. This chapter emphasizes the importance of understanding these geometric relationships, particularly through the study of *Voronoi diagrams* and *hyperplanes*. These geometric objects, despite their apparent simplicity, form the fundamental

building blocks of many successful machine learning models, which are often represented as piecewise linear functions.

Together, these themes provide a comprehensive framework for understanding and advancing the design of algorithms at the disciplinary intersection of combinatorial optimization and machine learning, offering both conceptual advances and new, practical tools for tackling complex problems in these fields.

II.1 Combinatorial generation

Developing a systematic approach to constructing efficient factorizations for various combinatorial structures is a non-trivial task. Fortunately, efficient factorizations for many basic combinatorial structures, such as *permutations*, *sublists*, *list partitions*, and *multiclass assignments*, already exist. This chapter focuses on presenting a comprehensive discussion of three distinct classes of combinatorial generators: sequential decision processes, *lexicographical generation*, and *combinatorial Gray codes* (CGC). Each class of generator has unique advantages that make it suitable for specific contexts. The aim of this chapter is to provide a library of generators that can be easily applied to solve problems involving different combinatorics.

The SDP generators will be discussed in a more abstract manner in Section II.2.3. Moreover, we will see how these basic SDP generators can serve as the "atoms" for constructing more complex "compound" generators in Subsection II.2.3.4.

II.1.1 Datatypes (containers)

Combinatorial structures consist of a collection of elements. Containers that store these combinatorial structures are called *datatypes*. Readers with a computer science background may be very familiar with *data structures*. Datatypes are *abstractions* of data structures, each datatype can implemented using different kinds of data structures. For instance, the datatype *list* can be implemented by an array, a single-linked list, and so forth [Cormen et al., 2022, Kleinberg and Tardos, 2006].

Among the numerous datatypes, we are particularly interested in a class of datatypes known as the *Boom-hierarchy* [Bunkenburg, 1994]. Datatypes within this family exhibit favorable *algebraic* properties. Notably, algebraic theories based on *sets*, *bags*, and *lists* have been extensively developed in previous literature [Bird, 1989]. These three datatypes are particularly significant for combinatorial generation.

In this section, we briefly introduce several datatypes that feature prominently in this thesis and explore their algebraic properties. For a more formal and detailed discussion on algebraic datatypes, refer to Section II.2.2, where each datatype is modeled categorically through *polynomial functors*.

Each datatype in the Boom-hierarchy is the free algebra of its binary operator $\cup$, called *"join"*, and unit $\emptyset$ [8], called "*empty*." Different datatypes are distinguished by the algebraic laws satisfied by their join operator $\cup$. The four laws considered in the Boom-hierarchy are

$$\begin{aligned} \text{Unit:}\quad & \emptyset \cup a = a = a \cup \emptyset \\ \text{Associativity:}\quad & a \cup (b \cup c) = (a \cup b) \cup c \\ \text{Commutativity:}\quad & a \cup b = b \cup a \\ \text{Idempotence:}\quad & a \cup a = a. \end{aligned} \tag{14}$$

We can classify different datatypes by identifying the laws its binary operator $\cup$ satisfies, there are four laws in total, hence there are 2^4 number of datatypes in the Boom hierarchy. Adding a law to an algebra can be thought of as partitioning the *carrier* of the algebra into equivalence classes induced by that law

[8] We overload the notation $\cup$ and $\emptyset$ to represent the concatenation operator and unit for all datatypes in the Boom hierarchy family not just set.

and regarding each equivalence class as one element [Bunkenburg, 1994]. For instance, set concatenation has commutativity, hence $\{1, 2\}$ and $\{2, 1\}$ belong to the same equivalence classes.

Sequences (lists) A (finite) list or sequence is an ordered collection of values of the same type which are called the *elements* or *items* of the list. We shall use letters a, b, c, ..., at the beginning of the alphabet to denote elements of lists, and letters x, y, z at the end of the alphabet to denote the lists themselves. On some occasions, we want to describe a list of lists, which are denoted by compound symbols xs, ys, and zs.

The join operator $\cup$ for list is associative and has a unit but is *not* commutative, so we have $x \cup (y \cup z) = (x \cup y) \cup z$ and $x \cup \emptyset = \emptyset \cup x = x$, but $x \cup y \neq y \cup x$ does not hold in general.

In many imperative programming languages, such as Python or C++, lists can store values of different types. In our research here, we do not allow a list to store values with different types. What this means is that we can have lists of numbers, lists of characters, and even lists of functions, but we shall never mix two distinct types of values in the same list. This can simplify the type information of a list. A list of integers will be considered as $[Int]$, in Haskell, it is `[Int]`. We use the symbol $\emptyset$ or $[\,]$ to denote the empty list, in Haskell, an empty list is rendered as `[]`.

Sets and bags By definition, a finite set is a collection of elements in which the order of the values is ignored and there are no duplicate values in the collection. Hence *all laws* in (14) are satisfied by the join operator of the finite set datatype. We write the elements of a set in brace brackets and symbol $\emptyset$ or $\{\,\}$ to denote the empty set. For instance, set $S = \{1, 2\}$ means set S contains values 1 and 2.

Similarly, *bags* are like sets without the idempotent property, or lists without ordering, and bags are sometimes called *multisets*. Hence bags satisfy associativity, and commutativity and have a unit. We use $\wr 1, 1, 2, 3 \int$ to represent a bag with two 1, one 2 and one 3, and symbol $\emptyset$ or $\wr\, \int$ to denote the empty set. So we have $x \cup (y \cup z) = (x \cup y) \cup z$, $x \cup \emptyset = \emptyset \cup x = x$ and $x \cup y = y \cup x$, for all bags x, y.

Nested containers Containers can be nested, i.e., contained inside each other. For instance, $xs = \{[4, 2], [\,], [3, 6]\}$ is a set of lists, whereas, $ys = [\{4, 2\}, \{\,\}, \{3, 6\}]$ is a list of sets. In xs, while the ordering in which the lists appear in the set does not matter, there can be no duplicate lists, so $\{[4, 2], [\,], [3, 6]\} = \{[\,], [4, 2], [3, 6]\}$. Whereas, in ys, the ordering in which the sets appear in the list matters and there can be duplicate sets, but the ordering of the elements in each of the lists does not matter. As examples, $[\{2, 4\}, \{\}, \{3, 6\}]$ is the same as ys but $[\{\,\}, \{2, 4\}, \{3, 6\}]$ is not.

II.1.2 Sequential decision processes for basic combinatorial structures

Generators based on the sequential decision process are perhaps the most common type of combinatorial generator. It is valued for its simplicity, generality, efficiency, and ease of abstraction. The simplicity arises from its sequential nature, and the SDP-based generators for most of the basic combinatorial structures are *optimally efficient* in terms of asymptotic complexity. Furthermore, SDPs can be given a datatype-generic abstraction, which will be introduced in Chapter II.2. Due to these unique advantages and their significance in combinatorial optimization, SDP generators will be our primary focus in the discussion on combinatorial generation.

II.1.2.1 Sequential decision process combinatorial generator in Haskell

We have defined the SDP in Subsection I.2.1.4. When applying SDP to combinatorial generation, it is often the case that the combinatorial objects must satisfy certain constraints. Therefore, it is necessary to incorporate a *filtering process* to exclude infeasible configurations. Thus, the SDP generator for combinatorial generation can be defined as follows [De Moor, 1995]

```
sdp_gen :: ([a] -> Bool) -> [[a] -> a -> [a]] -> [[a]] -> [a] -> [[a]]
sdp_gen p fs e = filter p . foldl (choice fs) e
 where choice fs xs a = [f x a | x <- xs, f <- fs]
```

where the seed value `e` is the starting point of the recursion, and the `choice` function reflects the "decision" process in SDP. In each recursive step of `foldl` function, we have the choice of applying any of the operators from a list of decision functions `fs`[9] to all partial configurations so far produced in `xs`. The `filter` function takes a predicate function `p :: a -> Bool` (predicate function receives an argument and returns a Boolean value) and filters out all configuration that do not satisfy the predicate.

When a filtering process is required, direct use of `sdp_gen` program is inefficient because the number of configurations returned by `foldl` is usually exponential large or worse. In some applications, we can *fuse* the filtering process inside the `foldl` function, the fused SDP generator can be defined as

```
sdp_filtgen p fs e = foldl (choicefilt fs) e
 where choicefilt fs xs a = filter p [f x a | x <- xs, f <- fs]
```

Indeed, this filter fusion is possible if the update function `f` reflects `p`. Readers who are familiar with the theory of lists may recognize that when `f` is the "`snoc`" operator (which will be defined in Section II.2.2), `p` is called prefix-closed if `p (snoc x a) = q (snoc x a) && p x` [Bird, 1987, De Moor, 1995], where the original predicate `p` with respect to updated configuration `f x a` is true if and only if the auxiliary predicate `q` with respect to `f x a` is true and `p` with respect to prefix `x` is also true. The prefix-closed condition is sometimes expressed as `p (f x a)` $\Longrightarrow$ `p x` ($\Longrightarrow$ denotes logical implication) in the literature [Bird, 1987]. Note, if $A \implies B$ then also $\neg B \implies \neg A$. In practice, calculating `q (f x a)` is often more efficient than directly calculating `p (f x a)`. For instance, the well-known *eight queens problem* has predicate `p` which checks that no queen is attacked by any other queens, whereas the auxiliary predicate `q` checks whether the newly added queen does not attack the others.

II.1.2.2 Sublists, sequences and K-sublists

Sublists A list is a sequence of elements and a list with some missing elements is called a sublist—in other words, a sublist is a portion of a larger list potentially with gaps. We denote $\mathcal{S}_{\text{subs}}$ as the set of all sublists of a given list. To construct a sublist, each item has two possibilities, it can either remain unchanged or be deleted. Therefore, for a list with N elements, there are 2^N choices. Thus, the number of all possible sublists or subsets for a list of length N is 2^N. From this observation, we can define two choice functions

```
append x a =  x ++ [a]
ignore x a = x
```

[9] The decision function list can be either fixed or parameterized.

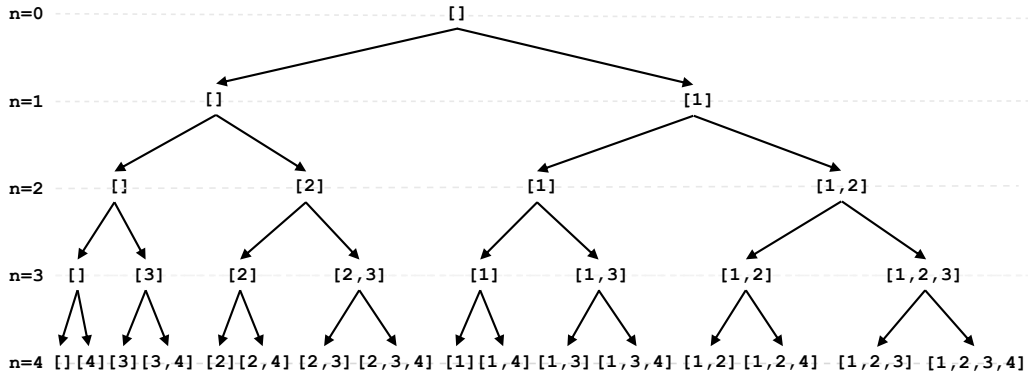


Figure 2: The generation tree for a sequential decision process (SDP) sublist generator with the input $[1, 2, 3, 4]$. At each recursive step, we either append a new element to the end of each partial configuration (sublist) or leave the configuration unchanged.

which append a new element `a` to a partial configuration `x` or ignore this element, respectively. These two functions serve as the decision functions in the recursive process. Thus, the list of decision functions for the sublist generator is given by

```
subs_fs :: [[a] -> a -> [a]]
subs_fs = [ignore, append]
```

The SDP generator for sublists is then rendered as

```
sdp_subs :: [a] -> [[a]]
sdp_subs = sdp_gen subs_p subs_fs subs_e
 where
  subs_p = const True
  subs_e = [[]]
```

where `subs_p` and `subs_e` are the predicate and seed value in sublists SDP generator.

Given a list `[1,2,3] :: [Int]`, `sdp_subs [1,2,3]` evaluates to `[[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]] :: [[Int]]`. Fig. 2 draws the *generation process tree* for `sdp_subs [1,2,3]`.

Sequences Generating a sequence recursively is similar to sublist generation. However, in this case, the `ignore` function is not needed, as the goal is to recover the complete input sequence. Consequently, the generator for sequences is straightforward, involving only a single decision. The SDP generator simply reconstructs the input data, so it can be defined as follows

```
sdp_seq :: [a] -> [[a]]
```

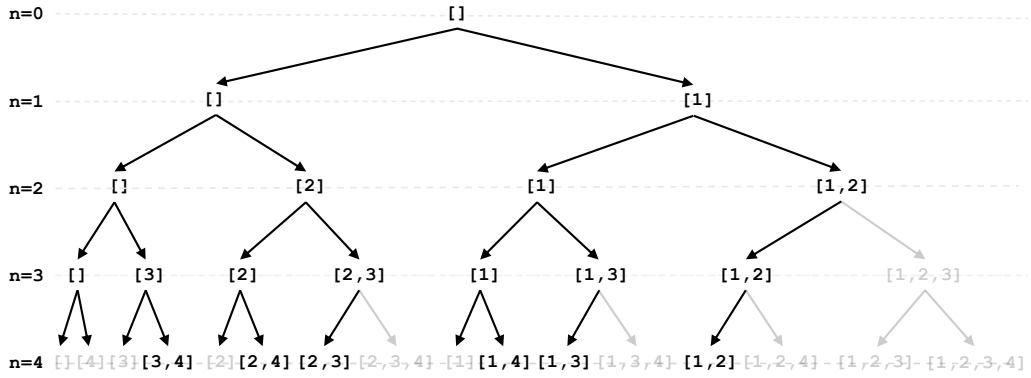


Figure 3: The sequential decision process generation tree for the 2-combination of list $[1, 2, 3, 4]$. The shaded sublists are filtered sublists, which violate the predicate `ksub_q = (k >=) . length`.

```
sdp_seq = sdp_gen seq_p seq_fs seq_e
 where
  seq_fs = [append]
  seq_p = const True
  seq_e = [[]]
```

Invoking `sdp_seq [1,2,3]` will return `[1,2,3]`. Although the sequence generator seems very trivial, in optimization problems, it is often necessary to evaluate the objective value of a partial configuration. To make our program more efficient, we need a method to update this objective value recursively. However `sdp_seq` has quadratic time complexity because the use of `++` is linear time in Haskell. To achieve a linear-time implementation, one can define `sdp_gen` based on the `foldr` operator and define the `append` function based on `:`.

K-sublists The terms "K-sublists" and "K-combinations" are synonymous, both representing sublists of size K. Below are two novel generators of K-sublists/combinations. To differentiate between them, we refer to the generator developed in this Chapter as the K-sublist generator `ksubs`, while another generator described in Section II.2.3 is termed the K-combination generator `kcombs`.

The K-combinations of a length N list is a sublist of it with fixed length K, such that $K \leq N$. It is well-known that the number of all possible K-combinations for a length N list is the same as the *binomial coefficient*, denoted as C_K^N or $\begin{pmatrix} N \\ K \end{pmatrix}$,

$$C_K^N = \begin{pmatrix} N \\ K \end{pmatrix} = \frac{N!}{K!\,(N-K)!} = \frac{N\,(N-1)\ldots(N-K+1)}{K\,(K-1)\ldots 1}. \tag{15}$$

We denote $\mathcal{S}_{\text{ksubs}}$ as the set of all K-sublists of a given list. The K-combinations generator can be

easily obtained from the sublists generator by filtering out all configurations that have a length not equal to K, thus we can construct the K-sublists generator as

```
ksubs_p k = (k ==) . length
sdp_ksubs k = sdp_gen (ksubs_p k) subs_fs subs_e
```

However, this program is not efficient, because we need to apply a predicate to every sublist generated by the sublist generator, and there are $2^N$of them. Moreover, the predicate `ksubs_p` is **not** prefix-closed, because a length K sublist `f a x` does not imply that sublist `x` has length K. Fortunately, the predicate `ksubs_p = (== k) . length` can be relaxed to `ksubs_q = (k >=) . length`, and predicate `ksubs_q` is prefixed-closed. Then a more efficient combination generator can be constructed by using the fused SDP generator `sdp_filtgen`

```
ksubs_q k = (k >=) . length
sdp_ksubs' k = filter (ksubs_p k) . (sdp_filtgen (ksubs_q k) subs_fs
   subs_e)
```

To illustrate, `sdp_ksubs' 2 [1,2,3,4]` evaluates to `[[3,4],[2,4],[2,3],[1,4],[1,3],[1,2]]`, the generation tree is depicted in Fig. 3.

II.1.2.3 Assignments

Binary assignments There exists a one-to-one correspondence between the sublists of a list $[a_1, a_2, \ldots, a_N]$ and the *0-1 assignment lists* $[x_1, x_2, \ldots, x_N] \in \{0, 1\}^N$, whenever the element a_n exist in the sublists, x_n equals 1 and 0 otherwise. The binary assignments are also called the *characteristic vector* of a sublist [Kreher and Stinson, 1999]. For instance, the sublist $[1, 2]$ of list $[1, 2, 3]$ has characteristic vector $[1, 1, 0]$. Therefore the number of binary assignments is the same as the number of sublists. We denote the set of all possible binary assignments of a length N list as $\mathcal{S}_{\text{basgns}}$. Therefore, the binary assignment generator is nearly identical to the sublist generator, with the decision functions modified to

```
asgn1 x a =  x ++ [1]
asgn0 x a = x ++ [0]
```

The binary assignment generator can hence be defined as

```
sdp_basgns :: Num a => [a] -> [[a]]
sdp_basgns = sdp_gen basgns_p basgns_fs basgns_e
 where
  basgns_fs = [asgn0, asgn1]
  basgns_p = const True
  basgns_e = [[]]
```

Evaluating `sdp_basgns [1,2,3]` gives us
`[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]]`.

Multiary assignments This is a direct generalization of binary assignments. Instead of binary labels 0 and 1, multiary assignments use a set of M labels, $\mathcal{M} = \{1, 2 \ldots M\}$. Similarly, for each element in the

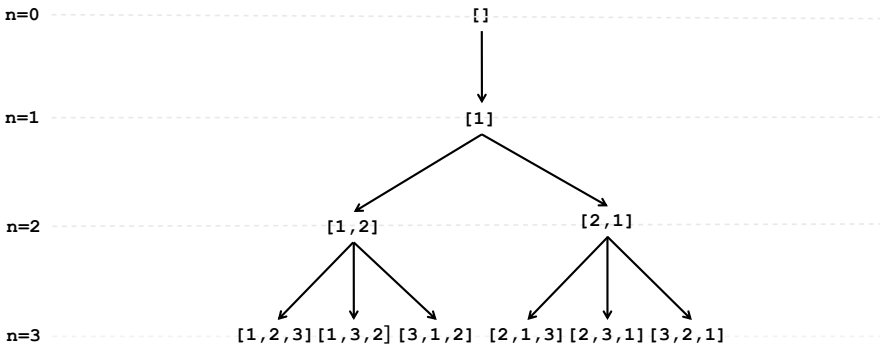


Figure 4: The sequential decision process generation tree for the permutations of list $[1, 2, 3]$. In each recursive step, a new element is inserted into every possible position between elements. For a list of length n list, there are $n + 1$possible choice of insertion.

list, there are M possible choices, resulting in a total of M^N possible M-ary assignments. We denote the set of all multiary assignments as $\mathcal{S}_{\text{masgns}}$.

The decision functions for the multiary assignments SDP generator consist of M decisions

```
asgnm i x a = x ++ [i]
masgns_fs m = [asgnm i | i <- [0..(m-1)]]
```

hence the SDP generator for multiary assignments is rendered as

```
sdp_masgns m = sdp_gen masgns_p (masgns_fs m) masgns_e
 where
  masgns_p = const True
  masgns_e = [[]]
```

Evaluating `sdp_masgns 2 [1,2,3]` gives us
`[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]]`.

II.1.2.4 Permutations

Permutations are probably the most widely encountered combinatorial structure. Perhaps more algorithms have been developed for generating permutations than any other kind of combinatorial structure [Sedgewick, 1977]. A permutation for a given list is a rearrangement of its elements. Given a list $[1, 2, \ldots, N]$, we denote a permutation π of list $[1, 2, \ldots, N]$ with $[\pi(1), \pi(2), \ldots, \pi(N)]$, so $\pi(i)$ changes the position for element i. This is often called *one-line notation*. For instance, a permutation $[3, 2, 1]$ for list $[1, 2, 3]$, has $\pi(1) = 3$, $\pi(2) = 2$, and $\pi(3) = 1$. We denote the set of all possible permutations as $\mathcal{S}_{\text{perms}}$ of a given list/set.

There are two classical approaches for generating permutations, based on *selecting* and *inserting*. In this subsection, we focus on insertion. The selection approach, which can also be used for constructing K-permutations algorithm, will be discussed it in the next Subsection.

Partial permutations are generated this way by inserting a new element into an existing partial permutation. For instance, inserting the new element 3 into the partial permutations $[1, 2]$ and $[2, 1]$,

obtains all possible permutations $[[3,1,2],[1,3,2],[1,2,3],[3,2,1],[2,3,1],[2,1,3]]$ for a list $[1,2,3]$. We can define the insertion function in Haskell as

```
insertKth :: Int -> a -> [a] -> [a]
insertKth k a x = (take k x) ++ [a] ++ (drop k x)
```

the `insertKth` function insert a new element `a` to the k_{th} position of list `x`. The decision functions for the permutations generator differ from those of the previous generators. In all the SDP generators discussed so far, the number of decisions for all configurations at each step is fixed. However, when inserting an element into a list of length n, there are $n+1$ possible positions for insertion. Thus, the number of decisions in the permutations SDP generator varies according to the stage of the recursion. We can define the decision functions for the permutations generator as being parameterized by an integer `k`

```
perms_fs :: [a] -> [a -> [a] -> [a]]
perms_fs x = [insertKth i | i <- [0..length x]]
```

The `choice` function defined above can only accommodate fixed-sized decision choice lists, `fs`. To accommodate varying-sized decision function lists, we need to modify the SDP generator. This leads us to the following new SDP generator

```
sdp_gen2 p fs e = foldl (choice2 fs) e
 where choice2 fs xs a = filter p [f a x | x <- xs, f <- fs x]
```

Then we can define the permutation SDP generator as

```
sdp_perms :: [a] -> [[a]]
sdp_perms = sdp_gen2 perms_p perms_fs perms_e
 where
  perms_p = const True
  perms_e = [[]]
```

Evaluating `sdp_perms [1,2,3]` give us `[[3,2,1],[2,3,1],[2,1,3],[3,1,2],[1,3,2],[1,2,3]]`. Fig. 4 depicts the generation tree for `sdp_perms [1,2,3]`.

Alternatively, rather than have a choice of one-shot extension functions, Bird and Gibbons [2020] provide another definition for the SDP generator

```
sdp_gen' p f e = foldl (step f) e
  where step f xs a = filter p (concatMap (f a) xs)
```

which uses a single function to produce multiple extensions. Thus, we can redefine the permutation generator as

```
sdp_perms' :: [a] -> [[a]]
sdp_perms' = sdp_gen' perms_p perms_f perms_e
 where
  perms_f a x = [insertKth i a x | i <- [0 .. length x]]
  perms_p = const True
  perms_e = [[]]
```

which provides a more general and robust definition.

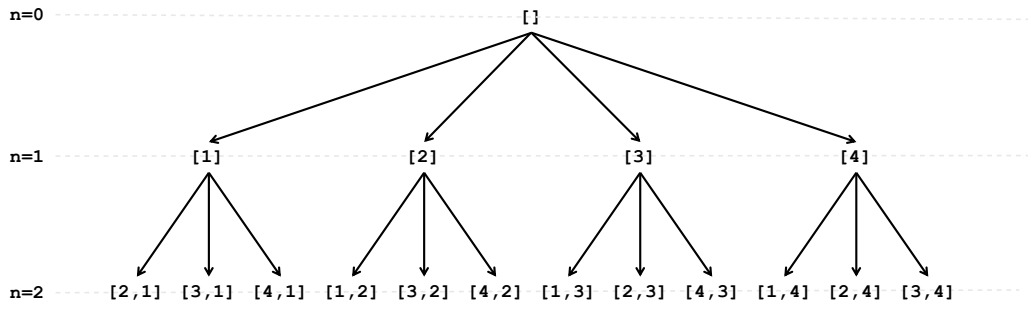


Figure 5: The sequential decision process generation tree for the 2-permutations of list $[1, 2, 3, 4]$. This process recursively selects new elements from the input list and appends them to the end of the current configuration. If the current configuration has n elements, there are $N - n$ possible choice of selection.

II.1.2.5 K-permutations

K-permutations are essentially permutations of$K$-combinations. In other words, we have `kperms k = concatMap . kcombs k`. This combinatorial structure is frequently used in many ML problems, such as the decision tree problem [Hu et al., 2019] and rule list [Angelino et al., 2018]. The most widely encountered approach for generating K-permutations can be understood as a recursive *selection* process, the first element select from the given list $[1, 2, \ldots, N]$ have N choice. In the next step, we select another element from the input except for the one we chose in the first step. If we only repeat K step of this process, then we have a K-permutations generator, the number of the K-permutations is $N \times (N-1) \times \ldots \times (N-K) = \frac{N!}{(N-K)!}$. When $K = N$, we have $N \times (N-1) \times \ldots \times 1 = N!$ all possible permutations. We denote the set of all possible K-permutations of a length N list as $\mathcal{S}_{\text{kperms}}$.

Under this description, the update functions are straightforward prepend operations, which select an element from the input and prepend it to the existing configurations. However, the recursive selection process ensures that only elements not yet selected are permitted. Consequently, we can define a predicate

```
kperms_p x = not (elem (head x) (tail x))
```

which checks whether the newly selected element `head x` already exists in the current configuration `tail x`. Instead of defining the SDP using the fold operator, we explicitly define it with a for loop to align with our description of the "selection process", which can be defined in Haskell as

```
for :: Int -> (a->a) -> a -> a
for 0 f x       = x
for (n+1) f x   = for n f (f x)
```

Then the new SDP generator `sdp_gen3` is defined as

```
sdp_gen3 k p e x = for k (choice3 x) e
  where choice3 x ys = filter p [ a:y | a<-x, y<-ys ]
```

the operation `a:y` can be understood as the decision function `f`.

Then, the K-permutations SDP generator can be defined as follows

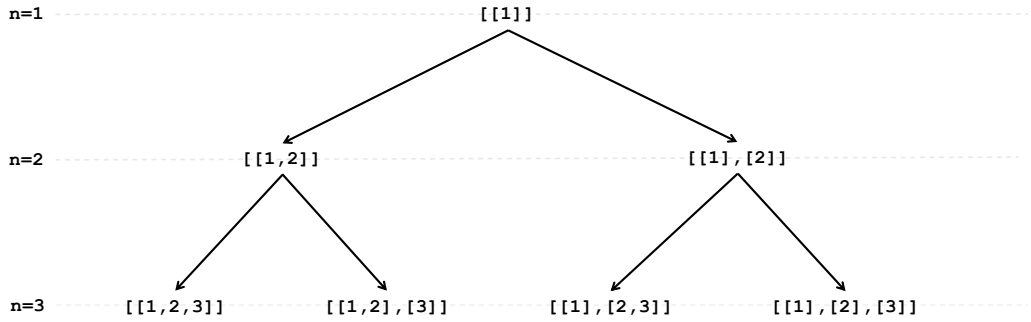


Figure 6: The sequential decision process generation tree for the list partitions of list $[1, 2, 3]$. In each recursive step, a new element is either appended to the last element of a configuration or used to create a singleton list containing this new element.

```
sdp_kperms k = sdp_gen3 k kperms_p kperms_e
 where
  kperms_p = const True
  kperms_e = [[]]
```

Evaluating `sdp_kperms 2 [1,2,3]` returns `[[2,1],[3,1],[1,2],[3,2],[1,3],[2,3]]`. Fig. 5 draws the generation tree for `sdp_kperms 2 [1,2,3]`.

II.1.2.6 List partitions

A *partition* of a list divides it into non-empty contiguous *segments*. For instance, the set of all possible list partitions of $[1, 2, 3]$ is

$$\{[[1, 2, 3]], [[1, 2], [3]], [[1], [2, 3]], [[1], [2], [3]]\}. \tag{16}$$

The number of partitions for a given list is equal to the number of ways to create contiguous, non-empty segments. This can be understood as inserting partitions into the spaces between adjacent elements in the list. For instance, the partitions for the list $[1, 2, 3]$ can be described as follows:

$$\begin{aligned} 123 \\ 12 \mid 3 \\ 1 \mid 23 \\ 1 \mid 2 \mid 3 \end{aligned}$$

This approach provides a recursive algorithm similar to the sublist SDP generator. For each space between two adjacent elements, we can either insert a separator $\mid$ or ignore it. This is equivalent to saying that we either append a new element to the last segment or create a new segment. This gives us two decision functions, which can be defined in Haskell as follows

```
appdlast :: [[a]] -> a -> [[a]]
appdlast x a =  init x ++ [(last x) ++ [a]]
```

```
extend :: [[a]] -> a -> [[a]]
extend x a = x ++ [[a]]
```

where `x` is a non-empty list.

Thus, we can define the SDP generator for list partitions as follows

```
sdp_parts :: [a] -> [[[a]]]
sdp_parts x = sdp_filtgen parts_p parts_fs (parts_e x) (drop 1 x)
 where
  parts_fs = [appdlast, extend]
  parts_p = const True
  parts_e x = [[take 1 x]]
```

The SDP generator for list partitions is very similar to the sublist generator, except list partitions start from a non-empty seed value, `parts_e x = [[take 1 x]]`, which is the first element of input list `x`, since segments cannot be empty. Evaluating `sdp_parts [1,2,3]` gives `[[[1,2,3]],[[1,2],[3]],[[1],[2,3]],[[1],[2],[3`

Fig. 6 depicts the generation tree for `sdp_parts [1,2,3]`.

II.1.3 Lexicographic generation

Generating combinatorial structures based on SDPs is relatively rare in the study of combinatorial generation. A significant proportion of research in the field focuses on generating combinatorial configurations by assuming an intrinsic *ranking* or *ordering* among all configurations. The most common ranking involves sorting the configurations based on some order relation. The most common one is *lexicographic ordering*, which is based on the familiar concept of the ordering of words in dictionaries. For example word "Apple" the predecessor of "Application" in a dictionary. The symbol $\prec_l$ is used to represent lexicographic ordering between values, and $<_l$ denotes lexicographic ordering between configurations.

Definition 5. *Lexicographic ordering.* The list $[a_1, a_2, \ldots, a_N] <_l [b_1, b_2, \ldots, b_M]$ if either

1. There exists an $n \in \mathcal{N}$, such that $a_n \prec_l b_n$ and $a_i = b_i, \forall i = 1, 2, \ldots, n-1$, or,
2. The size $N < M$ and $a_n = b_n, \forall n \in \mathcal{N}$.

We use distinct symbols to differentiate between the ordering of individual values and the ordering of configurations (i.e., lists of values) for clarity. In Haskell, both the ordering of values and the ordering of lists of values belong to the `ord` typeclass for comparison. Thus, we can compare them using `<`.

Using ordering among configurations, it is possible to define a *ranking function* and its inverse, an *unranking function*. These two functions define a bijection between a configuration and its intrinsic rank. In other words, the ranking and unranking functions have type

$$\begin{aligned} rank &: \mathcal{S} \to \mathcal{R} \\ unrank &: \mathcal{R} \to \mathcal{S}, \end{aligned} \tag{17}$$

where $\mathcal{S}$ is a set of all configurations in the same combinatorial structure of a given length, and $\mathcal{R} = \{1, 2, \ldots, |\mathcal{S}|\}$. For instance, when $\mathcal{S}$ is the set of all sublists for a length N list, then $\mathcal{R} = \{1, 2, \ldots, 2^N\}$.

In the literature, the primary use of a lexicographical generator is to generate random configurations with equal probability based on their rank. Thus, lexicographical generators are often designed for random generation rather than for exhaustive generation of all possible configurations. Although the asymptotic complexity of this generator is the same as others, it is rarely used for generating all possible configurations due to a large constant factor hidden in the Big O notation, which makes it less practical for exhaustive generation tasks.

Despite the inefficiency of the lexicographical generator in exhaustive generation, it has the following two potential uses in combinatorial optimization:

1. **Random generation for producing upper bounds**: We can randomly select integers from set $\mathcal{R}$, and then use the unranking function to obtain the corresponding configurations. The objective values of these configurations can serve as the upper bound.

2. **Storing configurations by their integer representation:** In combinatorial generation, the size of configuration space $\mathcal{S}$ is typically polynomial or even exponential in the data size N, with each configuration requiring $O(N)$ space to store. Storing all these configurations can be memory-intensive. Instead of saving the configurations themselves, which demand substantial memory, it is often more efficient to store their integer representations using the rank function. This approach reduces memory usage by saving only the integer representation of a configuration rather than the entire configuration.

3. **Quick indexing**: In many combinatorial optimization tasks, it is often necessary to associate additional information with each configuration. This information is typically pre-stored. By organizing the information to align with the intrinsic ranking of the combinatorial objects, it becomes possible to access this information in constant time.

The second use can be combined with the SDP generator. During each recursive generation step, numerous candidate configurations arise, and storing these configurations in their integer representation (integer format) can significantly reduce memory usage. We will refer to the SDP generator that replaces combinatorial configurations with their integer representations during the recursive generation steps as the *integer SDP generator*. The third application will turns out to be extremely useful in applications involving nested combinatorial objects, such as the ReLU network problem III.2 and the decision tree problem III.3 .

However, lexicographic ordering is not the most ideal for constructing such an SDP generator. Embedding lexicographic ordering directly into SDP generation is inefficient because it requires frequent transformations between the configuration and its integer representation.

Instead, configurations generated by SDP are naturally associated with an ordering known as *minimal change ordering*. Combinatorial generation with minimal change ordering makes use of so-called *combinatorial Gray codes* (CGCs). Indeed, minimal change ordering generation agrees with the principle of optimality but within the context of combinatorial generation, we will see examples shortly. Thus combinatorial Gray code generation can be considered a special case of SDP generation, and this will be explained in the next section.

II.1.4 Combinatorial Gray codes

Lexicographic generation is inefficient for generating all possible configurations because consecutive configurations in lexicographic order can be combinatorially very different. For instance, the sublists of list $[1,2,3]$ in lexicographic order are $[\,]$, $[1]$, $[1,2]$, $[1,2,3]$, $[1,3]$, $[2]$, $[2,3]$ and $[3]$. Here, configurations $[1,3]$, $[2]$ are adjacent but combinatorially distinct. Therefore, lexicographic generation is not well-suited for exhaustive generation.

To address this issue, research in combinatorial generation has explored the *minimal change ordering*. When generating all 2^N sublists sequentially, it is often desirable to do so in a manner where any two consecutive configurations have the *smallest* possible distance [10] between them.

Minimal change ordering organizes configurations so that, at each recursive step, only one new element is incorporated into each configuration. This approach employs a similar principle to that used in SDP generation, with the key difference being the ordering of configurations. In basic SDP generation, the ordering of configurations is not a concern, whereas, in combinatorial Gray codes (CGCs), the ordering is critical.

There is a vast amount of research on CGC generators, and there is not the space to survey all of it in this thesis. Instead, a few simple and classical examples are discussed below. More importantly, the underlying principles of these generators can be further explored within the novel framework of this thesis. To streamline the discussion, it will focus solely on the *recursion*, *ranking*, and *unranking algorithms* for each Gray code algorithm. Note that, all recursion, ranking, and unranking functions mentioned in this section are standard practice in the classical combinatorial generation book. However, our implementation is different from the pseudo-code presented in Kreher and Stinson [1999], we construct these functions from the description given by Kreher and Stinson [1999]. While our resulting implementation is more succinct and has undergone manual verification to ensure correctness, a constructive proof is still required. Given that this thesis does not employ CGC generators, ranking, or unranking functions elsewhere, we leave this proof as an interesting topic for future research.

II.1.4.1 Sublists

Definition 6. *Sublist distance.* Given a list of sublists $\mathcal{S}_{\text{subs}}$, and two sublists $s_1, s_2 \in \mathcal{S}_{\text{subs}}$, the *symmetric difference* between two sublists s_1 and s_2 is given by

$$s_1 \triangle s_2 = (s_1 - s_2) \cup (s_2 - s_1)\,, \tag{18}$$

where $s_1 - s_2$ is the list difference defined as $[x \mid x \in s_1 \wedge x \notin s_2]$, for instance $[1,2,3] - [2] = [1,3]$. Using this, the distance between two sublists is the size of their symmetric difference

$$d_{\text{sub}}(s_1, s_2) = |s_1 \triangle s_2|\,. \tag{19}$$

Alternatively, if we represent the sublists $\mathcal{S}_{\text{subs}}$ as the binary assignments $\mathcal{S}_{\text{basgns}}$ that described in Subsection II.1.2.3, $d_{\text{sub}}(s_1, s_2)$ equals the number of entries $s_1', s_2' \in \mathcal{S}_{\text{basgns}}$ that are different. This is also called the *edit distance* $d_{\text{edit}}(s_1', s_2')$. In other words, $d_{\text{sub}}(s_1, s_2) = d_{\text{edit}}(s_1', s_2')$, which represents the number of elements that need to be added to and/or deleted from one sublist in order to obtain the other.

[10] The definition of "distance" varies depending on the combinatorial structure.

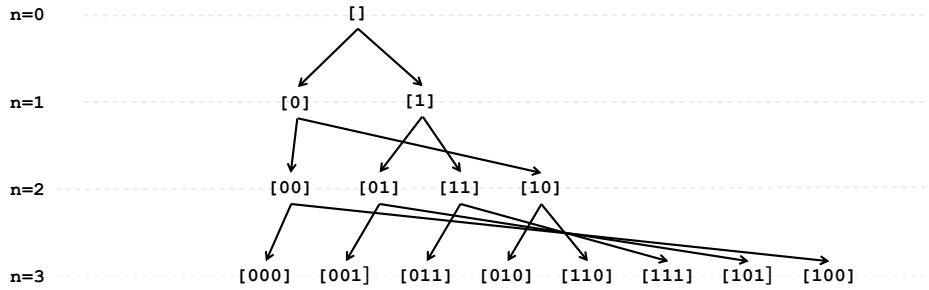


Figure 7: The combinatorial Gray code generation tree for the binary assignment of a length 3 list.

The minimal change ordering for two sublists $s_1, s_2 \in \mathcal{S}_{\text{subs}}$ has precisely the characteristic that $d_{\text{sub}}(s_1, s_2) = 1$ between two configurations s_1 and s_2 occurring sequentially in the ordering. An example of minimal change ordering for sublists of list $[1, 2, 3]$ is

$$[\,], [3], [2, 3], [2], [1, 2], [1, 2, 3], [1, 3], [1]. \tag{20}$$

Similarly, the binary assignments in minimal change ordering have Gray code

$$[0, 0, 0], [0, 0, 1]\,[0, 1, 1]\,[0, 1, 0]\,[1, 1, 0]\,[1, 1, 1], [1, 0, 1], [1, 0, 0]. \tag{21}$$

Geometrically, binary assignments can be visualized as the vertices of an N-dimensional unit cube. Each *Hamiltonian path* through this N-dimensional unit cube represents a minimal change ordering, where the edges connecting adjacent vertices have an edit distance of one, and each vertex is visited exactly once. Due to the diversity of Hamiltonian paths in sublist generation problems, considerable research has been devoted to various constructions for Gray codes. For sublist generation, we will examine a particularly simple class of Gray codes next.

The binary reflected Gray code The binary reflected Gray code for all possible 0-1 binary assignments for a given length N list is denoted as G^N, and is defined as

$$G^N = \left[G_0^N, G_1^N, G_2^N, \ldots, G_{2^N-1}^N, \right], \tag{22}$$

where $G_r^N \in \mathcal{S}_{\text{basgns}}$ is the characteristic vector of a sublists in $\mathcal{S}_{\text{subs}}$, and $0 \leq r \leq 2^N - 1$. The Gray code G^Ncan be obtained by the recursive function

$$\begin{aligned} G^0 &= [[\,]] \\ G^n &= \left[[0] \cup G_0^{n-1}, \ldots, [0] \cup G_{2^{n-1}-1}^{n-1}, [1] \cup G_{2^{n-1}-1}^{n-1}, \ldots, [1] \cup G_0^{n-1}\right], \end{aligned} \tag{23}$$

or equivalently

$$\begin{aligned} G^0 &= [[\,]] \\ G^n &= [[0]] \circ G^{n-1} \cup [[1]] \circ rev\left(G^{n-1}\right), \end{aligned} \tag{24}$$

where $\circ$ is the Cartesian product of two lists of lists, for instance, $[[a], [b]] \circ [[c], [d]] = [[a, c], [a, d], [b, c], [b, d]]$ and the function $rev : [a] \to [a]$ that reverses a list. The appearance of the reverse function here is the reason why it is called a *reflected* Gray code. The Gray code G^n is constructed from G^{n-1} by two steps:

first append a new element [0] to every configuration in G^{n-1} and then append a new element [1] to every element of $rev\left(G^{n-1}\right)$. The recursion (24) is a CGC that can be proved through induction [Kreher and Stinson, 1999].

The next three Gray codes generated by recursion (24) are

$$\begin{aligned} G^1 &= [[0],[1]] \\ G^2 &= [[0,0],[0,1],[1,1],[1,0]] \\ G^3 &= [[0,0,0],[0,0,1],[0,1,1],[0,1,0],[1,1,0],[1,1,1],[1,0,1],[1,0,0]]. \end{aligned} \tag{25}$$

One way to understand the binary reflected Gray code for sublists generation is that CGC is a special SDP with different generative ordering, the generation tree for the binary reflected Gray code G^3 is drawn in Fig. 7.

We can implement the binary reflected Gray code generator in Haskell as

```
brgc :: Int -> [[Int]]
brgc 0 = [[]]
brgc n = ([[0]] `crj` cnfgs) ++ ([[1]] `crj` (reverse cnfgs))
 where cnfgs = brgc (n-1)
```

where `crj x y = [a ++ b | a <- x, b <- y]` is the Cartesian product operator.

Next, the ranking and unranking functions with respect to the ordering used in CGC are examined. These two functions are crucial for constructing efficient integer SDP generators, and their role will be elaborated upon in the subsequent sections.

Ranking and unranking functions Constructing ranking and unranking functions for the binary reflected Gray code is associated to the following lemma.

Lemma 1. Suppose that $N \geq 1$ is an integer, $0 \leq r \leq 2^N - 1$, and suppose that $b_N, b_{N-1} \ldots b_0$ is the binary representation of integer r, such that $\left(r = \sum_{i=0}^{N} b_i 2^i\right) \wedge (b_N = 0)$, and $G_r^N = [a_{N-1}, \ldots, a_0] \in G^N$ is a 0-1 binary assignment for all possible assignments generated by G^N. Then,

$$a_j = (b_j + b_{j+1}) \mod 2, \tag{26}$$

and

$$b_j = \sum_{i=j}^{N-1} a_i \mod 2, \tag{27}$$

for $j = 0, 1, \ldots, N-1$.

The consequence of above lemma gives rise to the definition of unranking and ranking functions. The definition is given here briefly, more detailed explanations can be found in Kreher and Stinson [1999].

Definition 7. *Unranking function.* Assume a rank $r \in \mathcal{R}$. From Lemma 1, the unranking function $unrank : \mathcal{R} \times \mathcal{N} \to \mathcal{S}_{\text{subs}}$ for the binary reflected Gray code can be defined as

$$unrank\,(r, N) = G_r^N = [a_{N-1}, \ldots, a_0], \tag{28}$$

where each value a_n is defined as

$$a_n = \begin{cases} 1 & b_n \neq b_{n+1} \\ 0 & \text{otherwise} \end{cases}, \forall n \in \{0, \ldots, N-1\}. \tag{29}$$

We can implement the unranking function in Haskell as

```
unrank_brgc :: Int -> Int -> [Int]
unrank_brgc n r = [asgn i| i <-[0..n-1]]
 where
  asgn i = if bits!!i /= bits!!(i+1) then 1 else 0
  bits = padWithZeros (n+1) (binary r)
```

where `binary` returns the binary representation of integer `r`, defined as

```
binary n = if n == 0 then [] else binary q ++ [r]
 where (q,r) = n `divMod` 2
```

However, `binary` function will only return the binary representation. What we want is a length `n+1` list (as $b_N, b_{N-1} \ldots b_0$ consists of $N+1$ elements), we thus pad `binary r` with zeros by using

```
padWithZeros n xs = replicate (n - length xs) 0 ++ xs
```

Running `unrank_brgc 3 5` gives us `[1,1,1]`.

Definition 8. *Ranking function.* Given a binary assignment $G_r^N = [a_{N-1}, \ldots, a_0] \in G^N$. From Lemma 1, the unranking function $rank : \mathcal{S}_{\text{subs}} \times \mathcal{N} \to \mathcal{R}$ for the binary reflected Gray code is given by,

$$rank\left(G_r^N, N\right) = r, \tag{30}$$

where $\left(r = \sum_{i=0}^{N} b_i 2^i\right) \wedge (b_N = 0)$, and $b_j = \sum_{i=j}^{N-1} a_i \mod 2$.

Similarly, the ranking function can be implemented as following

```
rank_brgc :: Int -> [Int] -> Int
rank_brgc n x = snd $ foldl' update (0, 0) (reverse [0..(n-1)])
  where
    t = [i | (i, flag) <- zip [1..] x, flag == 1]
    update (b, r) i =
        let b' = if (n - i) `elem` t then 1 - b else b
            r' = if b' == 1 then r + (2 ^ i) else r
        in (b', r')
```

Running `rank_brgc 3 [1,1,1]` gives us the ranking for binary assignment `[1,1,1]`, which is `5`.

II.1.4.2 *K*-sublists

The symmetric difference for any two configurations in $\mathcal{S}_{\text{ksubs}}$ is greater than 2, i.e., $d_{\text{subs}}(s_1, s_2) \geq 2, \forall s_1, s_2 \in \mathcal{S}_{\text{ksubs}}$. Hence, the minimal change ordering on $\mathcal{S}_{\text{ksubs}}$ must be redefined to ensure that any two consecutive combinations have a distance of one. This special ordering is known as the *revolving door ordering.*

The revolving door algorithm Generating K-sublists using this minimal change ordering is closely related to *Pascal's formula* for binomial coefficients,

$$C_K^N = C_{K-1}^{N-1} + C_K^{N-1}, \tag{31}$$

this identity can be proved by observing that the C_K^N K-sublists/K-subsets can be partitioned into two disjoint sub-collections, the C_{K-1}^{N-1} $(K-1)$-sublists contains the element N, and the C_K^{N-1} K-sublists that do not contain the element N.

Therefore, a recursive program for generating the K-sublists of the list $[1, 2, \ldots, N]$ in revolving door ordering follows a pattern similar to Pascal's formula (31). Define G_k^n as the list of K-sublists in revolving door ordering. Note that C_K^N denotes the collection of K-sublists, irrespective of the ordering. Given G_{k-1}^{n-1}, and G_k^{n-1}, the list G_k^n, $0 \leq k \leq N$ is defined as

$$\begin{aligned} G_0^n &= [[\,]] \\ G_k^n &= G_k^{n-1} \cup rev\left(G_{k-1}^{n-1}\right) \circ [[n]] \\ G_n^n &= [[1, 2, \ldots, N]]\,. \end{aligned} \tag{32}$$

In Haskell, we can implement the revolving door algorithm as

```
revdoor :: Int -> Int -> [[Int]]
revdoor _ 0 = [[]]
revdoor n k
 | n == k = [[1..n]]
 | otherwise = (revdoor (n-1) k) ++ (map (++[n]) $ reverse (revdoor (n-1)
    (k-1)))
```

Executing `revdoor 3 2` yields
`[[1,2,3],[1,3,4],[2,3,4],[1,2,4],[1,4,5],[2,4,5],[3,4,5],[1,3,5],[2,3,5],[1,2,5]]`.

Ranking and unranking functions

Definition 9. *Ranking function.* Given a sublist $[a_1, a_2, \ldots, a_K]$ of list $[1, 2, \ldots, N]$, and $a_1 < a_2, \ldots, < a_K$. We can define the ranking function $rank : \mathcal{S}_{\text{ksubs}} \times \mathcal{N} \to \mathcal{R}$ as

$$rank\left([a_1, a_2, \ldots, a_K], K\right) = \begin{cases} \sum_{i=1}^{K} (-1)^{K-i} \begin{pmatrix} a_i \\ i \end{pmatrix} & \text{if } K \text{ is even} \\ \sum_{i=1}^{K} (-1)^{K-i} \begin{pmatrix} a_i \\ i \end{pmatrix} - 1 & \text{if } K \text{ is odd.} \end{cases} \tag{33}$$

Similarly, we can define unranking function as

```
rank_revol :: [Int] -> Int
rank_revol x
 | even k = sum [(-1)^(k-i)*(x!!(i-1) `choose` i) |i<-[1..k]]
 | odd k = sum [(-1)^(k-i)*(x!!(i-1) `choose` i) |i<-[1..k]] - 1
  where k = length x
```

where the `choose` operator represents the ordinary binomial coefficient, defined as

```
factorial n = product [1..n]

choose :: Int -> Int -> Int
n `choose` k
    | k < 0     = 0
    | k > n     = 0
    | otherwise = factorial n `div` (factorial k * factorial (n-k))
```

Running `rank_revol [1,4,5]` gives us 4.

Definition 10. *Unranking function.* Given a rank $r \in \mathcal{R}$, the length of the sublists $K \in \mathcal{N}$ and the length of the original list $N \in \mathcal{N}$. We can define the unranking function $unrank : \mathcal{R} \times \mathcal{N} \times \mathcal{N} \to \mathcal{S}_{\text{ksubs}}$ as modified

$$unrank\,(K, N, r) = \begin{cases} \emptyset & \text{if } K = 0 \\ unrank\left(K-1, N, \begin{pmatrix} x+1 \\ K \end{pmatrix} - r - 1\right) \cup [x+1] & \text{if } K \geq 1, \end{cases} \tag{34}$$

where x is the smallest integer such that $\begin{pmatrix} x \\ K \end{pmatrix} \leq r$.

```
unrank_revol :: Int -> Int -> Int -> [Int]
unrank_revol n k r
 | k <= 0 = []
 |otherwise = (unrank_revol n (k-1) ((x `choose` k)-r-1)) ++ [x]
 where
  x = (findUntil n k r) + 1
```

where `findUntil` returns the smallest integer `n` such that `n `choose` k <= r` , defined as

```
findUntil n k r
  | n `choose` k <= r = n
  | otherwise         = findUntil (n - 1) k r
```

Running `unrank_revol 5 3 4` gives `[1,4,5]`.

II.1.4.3 Permutations

The Trotter-Johnson algorithm Any two permutations must differ in at least two positions. The minimal change ordering defined on permutations is when one configuration can be obtained by an *adjacent transposition* (by exchanging two adjacent elements). The *Trotter-Johnson algorithm* is a Gray code for permutation generation. Let G^N denote the list of all permutations of $[1, \ldots N]$ arranged in Gray code order. G^N is constructed recursively from G^{N-1}. For instance, for $G^2 = [[1,2],[2,1]]$, G^3 is constructed

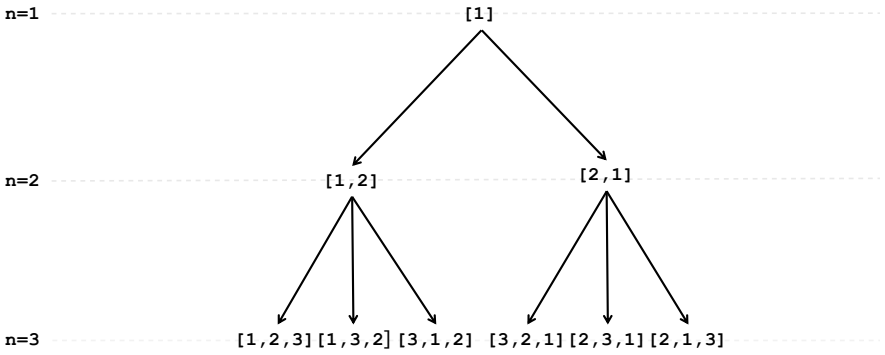


Figure 8: The combinatorial Gray code generation tree for the permutations of list $[1, 2, 3]$.

from G^2, as follows

$$\begin{array}{cccc} & 1 & & 2 & 3 \\ & 1 & 3 & 2 & \\ 3 & 1 & & 2 & \\ 3 & 2 & & 1 & \\ & 2 & 3 & 1 & \\ & 2 & & 1 & 3 \end{array}$$

Thus $G^3 = [[1,2,3],[1,3,2],[3,1,2],[3,2,1],[2,3,1],[2,1,3]]$. The generation tree of G^3 is illustrated in Fig. 8.

Based on the description provided by Kreher and Stinson [1999] and the recursive generation pattern of configurations, we present a succinct yet *unproven* recursive implementation in Haskell

```
tja :: Int -> [[Int]]
tja 1 = [[1]]
tja n = concat $ evenApply reverse $ map (interleave n) tja_last
 where tja_last = (tja (n-1))
```

where `evenApply f` receives a list and apply function `f` to the elements with even indexes

```
evenApply f [] = []
evenApply f [a] = [f a]
evenApply f (x:y:xs) = f x : y : evenApply f xs
```

The function `intlv a x` returns a list of all possible ways of inserting the element `a` into the list, which can be defined in Haskell as

```
interleave :: a -> [a] -> [[a]]
interleave a [] = [[a]]
interleave a (b:x) = [a:b:x] ++ map (b:) (interleave a x)
```

For instance, running `interleave 4 [1,2,3]` gives us `[[4,1,2,3],[1,4,2,3],[1,2,4,3],[1,2,3,4]]`.

Indeed, the `tja` definition we present here differs from both the definition provided in the combina-

torial generation textbook [Kreher and Stinson, 1999] and the loopless[11] functional definition proposed by Bird [2010], which is based on recursively swaps adjacent elements. We construct this definition by observing how the configurations should be updated. One could employ a similar induction approach, such as that outlined by Kreher and Stinson [1999] to prove it. However, a constructive algorithmic proof for this generator would be an intriguing topic to explore in future research.

Ranking and unranking functions Both ranking and unranking functions for the Trotter-Johnson ordering can be calculated recursively, this will help in constructing an integer SDP generator. Observe that descendants for a configuration G_r^{N-1} will have rank which lies between $n \times r$ and $nr + n - 1$, because all configurations $G_{r'}^{N-1}$ with rank $r' < r$ will have descendants that precedes the decedents of G_r^{N-1}, and all configurations $G_{r'}^{N-1}$ with rank $r' > r$ will have descendants which come after the descendants of G_r^{N-1}. Now the ranking and unranking functions can be defined recursively.

Definition 11. *Ranking function.* Given a permutation $[\pi(1), \pi(2), \ldots, \pi(N)] \in \mathcal{S}_{\text{perms}}$, the ranking function $rank : \mathcal{N} \times \mathcal{S}_{\text{perms}} \to \mathcal{R}$ for the Trotter-Johnson algorithm can be defined recursively as

$$rank([\pi(1), \pi(2), \ldots, \pi(N)], N) = N \times rank([\pi(1), \ldots, \pi(k-1), \pi(k+1), \ldots, \pi(N)], N-1) + c, \tag{35}$$

where $\pi(k) = N$ and

$$c = \begin{cases} n - k & \text{if } rank([\pi(1), \ldots, \pi(k-1), \pi(k+1), \ldots, \pi(N)], N-1) \text{ is even} \\ k - 1 & \text{if } rank([\pi(1), \ldots, \pi(k-1), \pi(k+1), \ldots, \pi(N)], N-1) \text{ is odd.} \end{cases} \tag{36}$$

Again, we can implement this ranking function in Haskell as

```
rank_tja :: [Int] -> Int
rank_tja [] = 0
rank_tja x = n * rank_tj l + c
  where
    (l, k) = del n x
    c = if even (rank_tj l) then n - k  else k -1
    n = length x

del :: Int -> [Int] -> ([Int],Int)
del n x = (take (k-1) x ++ drop k x,k)
 where k = fst $ head $ filter ((== n) . snd) (zip [1..] x)
```

Definition 12. *Unranking function.* Given a rank $r \in \mathcal{R}$, the unranking function $unrank : \mathcal{N} \times \mathcal{R} \to \mathcal{S}_{\text{perms}}$ for the Trotter-Johnson algorithm can be defined recursively as

$$\begin{aligned} unrank(1, r) &= [1] \\ unrank(N, r) &= insertKth(k, n, unrank(N-1, r')), \end{aligned} \tag{37}$$

[11] Suppose that each pattern is obtained from its predecessor by a single transition. An algorithm for generating all patterns is called loopless if the first transition is produced in linear time and each subsequent transition in constant time.

where $r' = \lfloor r/N \rfloor$ and $insert\,(k, n, x)$ is equivalent to the `insertKth` function defined earlier, which inserts the element n into the kth position of the list x. The value of k is determined by:

$$k = \begin{cases} N - r - N \times r' - 1 & \text{if } r' \text{ is even} \\ r - N \times r' & \text{if } r' \text{ is odd.} \end{cases} \tag{38}$$

Similarly, the Haskell implementation of this function is rendered as

```
unrank_tja :: Int -> Int -> [Int]
unrank_tja 1 _ = [1]
unrank_tja n r = insertKth k n (unrank_tj (n-1) r')
 where
  r' = r `div` n
  k = if odd r' then r-n*r' else n-r-n*r'-1
```

II.1.5 Integer sequential decision process combinatorial generator

Embedding an ordering into configurations allows us to replace configurations with their integer representations during SDP generation. To demonstrate the potential memory savings of this approach, consider the example of permutation generation. A permutation is a list of length N given by $\left[x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(N)}\right]$, where the subscripts $(\pi\,(1), \pi\,(2), \ldots, \pi\,(N))$ represent a permutation of $(1, 2, \ldots, N)$. Each data item is typically a double-precision floating-point number (or, if not, likely a 64-bit pointer), which consumes 64 bits of memory. Consequently, storing each permutation requires $64 \times N$ bits. However, the integer representation of a configuration requires *at most* $\log\left(|\mathcal{S}_{\text{perms}}|\right)$ bits.

The SDP combinatorial generation process, which includes an embedded ordering (such as lexicographic or any other type of ordering) for each configuration, will be called the *ordered SDP combinatorial generator*. Replacing each configuration with the integer representing its rank obtains the *integer SDP combinatorial generator*.

This section, provides a step-by-step derivation to construct two ordered SDP combinatorial generators: one for the binary reflected Gray code ordering and another for the revolving door ordering. Following this, their corresponding integer SDP generators can be easily derived.

II.1.5.1 The binary reflected Gray code SDP generator

The binary reflected Gray code generator is given above in mathematical form (23). In this Subsection, the binary reflected Gray code is further investigated by implementing the recursion (23) as an ordered SDP generator, and its integer SDP combinatorial generator can subsequently be implemented.

Ordered SDP combinatorial generator According to the binary reflected Gray code recursion (23), the corresponding two SDP update functions are given by,

```
left_upd :: [[Int]] -> Int -> [[Int]]
left_upd xs a = map (0:) xs
```

```
right_upd :: [[Int]] -> Int -> [[Int]]
right_upd xs a = reverse $ map (1:) xs
```

where the `left_upd` function receives an input configuration list `xs` and keeps it unchanged. The `right_upd` function appends the new element `a` to the front of every configuration in `xs`, then reverses the ordering of this list to maintain the binary reflected Gray code ordering.

Therefore, the SDP generator for generating sublists with binary reflected Gray code ordering is rendered as

```
sdp_gen4 p fs e = foldl (choice fs) e
 where choice fs xs a = filter p $ concatMap (\f -> f xs a) fs

sdp_brgc :: [Int] -> [[Int]]
sdp_brgc = sdp_gen4 brgc_p brgc_fs brgc_e
 where
  brgc_fs = [left_upd,right_upd]
  brgc_p = const True
  brgc_e = [[]]
```

Evaluating `sdp_brgc = [1,2,3]` generates
`[[0,0,0],[0,0,1],[0,1,1],[0,1,0],[1,1,0],[1,1,1],[1,0,1],[1,0,0]]`.

Integer SDP combinatorial generator Following the construction of the ordered SDP generator, a corresponding integer SDP generator can be developed by adapting the configuration update function to utilize the integer update pattern outlined in 30. Based on this modification, the updated functions can be defined as follows

```
left_upd_int :: [Int] -> Int -> [Int]
left_upd_int x n = x

right_upd_int :: [Int] -> Int -> [Int]
right_upd_int x n = reverse $ map (l-) x
 where l = 2^n - 1
```

where `n` denotes the recursion stage. For instance, evaluating `right_upd [0,1,2,3] 3` returns `[4,5,6,7]`. While this approach successfully generates the integer SDP from the ordered SDP based on observed outcomes, a direct analytical derivation of this relationship has not yet been established. A formal deduction using equational reasoning remains to be developed, and the current construction relies on empirical validation through specific instances, with a complete proof proposed as a future research direction.

The corresponding integer SDP generator is thus defined as

```
sdp_brgc_int :: [Int] -> [Int]
sdp_brgc_int = sdp_gen4 brgc_p_int brgc_fs_int brgc_e_int
 where
  brgc_fs_int = [left_upd_int,right_upd_int]
```

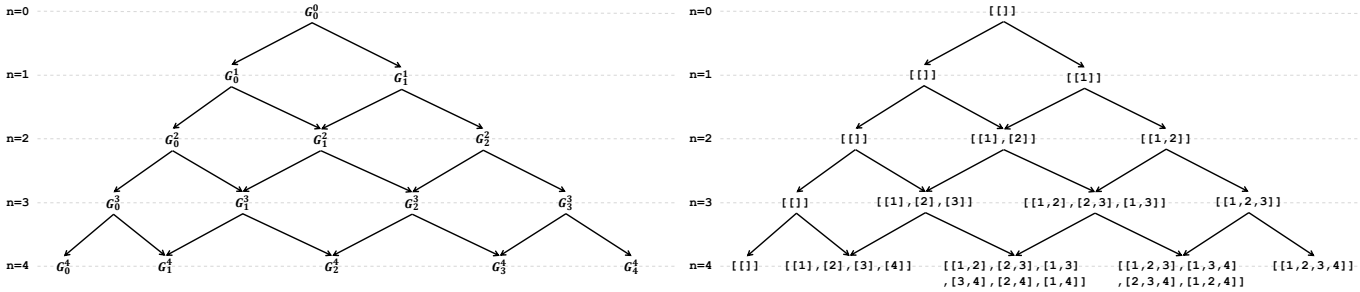


Figure 9: The sequential decision process generation tree for all k-combinations, $0 \leq k \leq N$, of the input list $[1, 2, 3, 4]$ in revolving door ordering. The left panel depicts the generation tree representing the combinations of the same size in a configuration G_k^n. The combinations contained in G_k^n are illustrated in the right panel of the figure.

```
brgc_p_int = const True
brgc_e_int = [0]
```

Evaluating `sdp_brgc_int [1,2,3]` will simply return the rank list `[0,1,2,3,4,5,6,7]`; running the unranking function on these integers would recover the original configurations.

II.1.5.2 K-combination SDP generator with revolving door ordering

A K-sublist generator `ksubs` was introduced above. Given a list `x :: [a]`, `ksubs` will return a list of all possible K-sublists. In other words, the K-sublist generator has a type `ksubs x :: [[a]]`. In this section, a new K-sublist generator is introduced, which will generate all possible sublists, but the sublists of the same length, i.e., combinations, will be grouped in the same list, and these lists are elements of the outer lists. Therefore, this generator will have a output type `[[[a]]]` instead of `[[a]]`. To discriminate `ksubs` and `subs`, we name this generator as `kcombs`, which connotes "K-combinations generator."

Ordered SDP combinatorial generator As discussed in Subsection II.1.4.2, the Gray code for generating K-combinations is closely related to Pascal's formula (31). It is not obvious how to construct an SDP generator based on this formula because a collection of k-combinations C_k^n depends on two collections of $(k-1)$-combinations C_{k-1}^{n-1}, C_k^{n-1} , instead of a single sublist which only depends on a single sublist in the previous step. One way to resolve this issue is to consider a collection of all size k-combinations C_k^n as a single configuration, then one configuration in this level is determined by *two* configurations in the last level. This is different from all SDP generators in Section II.1.2, where each configuration in one level is determined by exactly *one* configuration in the last level.

Addressing the above issue requires generalizing the ordinary SDP generators defined in Section II.1.2. It is still possible to consider that each configuration in the current level is updated to exactly *one* configuration in the next level by *one* decision function, but the updated configurations that are *adjacent* to each other should be "merged" together to become a single configuration.

Now, assume each configuration is a *list* of k-combinations in revolving door ordering, represented by

G_k^n. Each configuration is associated with two update functions `upd_left` and `upd_right` that generalize the `ignore` and `append` functions used in `subs` generator. They are defined as

```
upd_left :: [[a]] -> a -> [[a]]
upd_left xs a = xs

upd_right :: [[a]] -> a -> [[a]]
upd_right xs a = map (++ [a]) (reverse xs)
```

where the `upd_left` function applies `ignore` function to every k-combination in `xs`. Each element in `xs` represents a list of k-combinations G_k^n. Since the `ignore` function essentially does nothing, the configuration `xs` remains unchanged, so `upd_left [[1],[2]] 3 = [[1],[2]]`. Similarly, the `upd_right` applies the `append` function to every k-combination in configuration `(reverse xs)`, the use of `reverse` here maintains the revolving door ordering, to mimic the update rule of recursion (32). To illustrate, `upd_left [[1],[2]] 3 = [[2,3], [1,3]]`.

Next, some configurations generated by applying `upd_left` and `upd_right` functions must be merged. For instance, applying `upd_left` and `upd_right` to a list of configurations $[G_0^1, G_0^1] = [[[0]], [[1]]]$ gives $[[[0]], [[2]], [[1]], [[1, 2]]]$. Clearly, $[[2]]$, $[[1]]$ should belong to a single configuration in G_1^2 because both of them are *size-one* combinations. Thus, it can be inferred that the right update of G_{k-1}^{n-1} and the left update of G_k^{n-1} should belong to the same configuration G_k^n according to Pascal's formula (31). More generally, the left panel of Fig. 9 depicts the generation tree of the K-sublist SDP generator with revolving door ordering.

On merging the right update of G_{k-1}^{n-1} and the left update of G_k^{n-1}, their ordering must be exchanged to maintain the revolving door order. This is achieved using a helper function, `revjoin`, which joins two lists in reverse order

```
revjoin :: [[a]] -> [[a]] -> [[a]]
revjoin x y = y ++ x
```

for instance, `revjoin [[2]] [[1]] = [[1],[2]]`.

Observed that two special configurations G_0^n and G_n^n are only related to G_0^{n-1} and G_{n-1}^{n-1}, hence the left update of the first configuration and the right update of the last configuration do not need to be merged with any other updates. Following this observation leads to a function

```
sort_revol :: [[[a]]] -> [[[a]]]
sort_revol xs = case xs of
  [[]]     -> [[]]
  (x:ys)   -> [x] ++ mappair revjoin (init ys) ++ [last ys]
```

which sorts and merges the adjacent configurations by `revjoin` and obtains a new list of configurations G_k^n, $0 \leq k \leq K$, such that the k-combinations in G_k^n satisfy the revolving door ordering. The `mappair` function is defined as

```
mappair :: (a -> a -> a) -> [a] -> [a]
mappair _ [] = []
mappair f (x:y:rest) = f x y : mappair f rest
```

Bringing the above together leads to the following generator for all k-combinations, $0 \leq k \leq N$, in revolving door ordering

```
sdp_gen5 p fs e = foldl (choice5 fs) e
 where choice5 fs xs a = filter p $ sort_revol [f x a | x <- xs, f <- fs]

sdp_combs_revol :: [a] -> [[[a]]]
sdp_combs_revol = sdp_gen5 combs_revol_p combs_revol_fs combs_revol_e
 where
  combs_revol_fs = [upd_left, upd_right]
  combs_revol_p = const True
  combs_revol_e = [[[]]]
```

Evaluating `sdp_combs_revol [1,2,3,4]` gives all possible k-combinations, for all $0 \leq k \leq K$ in revolving door ordering $[G_0^4, G_1^4, G_2^4, G_3^4, G_4^4]$. The right panel of Fig. 9 shows the generation tree for `sdp_combs_revol [1,2,3,4]`.

The K-combination generator can thus be defined as

```
sdp_kcombs_revol k = (!!k) . sdp_combs_revol
```

For instance, evaluating `sdp_kcombs_revol [1,2,3,4]` returns `[[1,2],[2,3],[1,3],[3,4],[2,4],[1,4]]`.

Integer SDP combinatorial generator For all k-combinations, $0 \leq k \leq N$, generation, the integer SDP generator will generate a list of rank lists that start at rank zero. To distinguish the rank list for different k, tuple each rank list with an integer k. Then, modify the `upd_left` and `upd_right` functions above as

```
upd_left_int :: ([Int],Int) -> Int -> ([Int],Int)
upd_left_int (x,k) n = (x,k)

upd_right_int :: ([Int],Int) -> Int -> ([Int],Int)
upd_right_int (x,k) n = (reverse $ map (l-) x, k+1)
 where l = (n `choose` (k+1)) -1
```

where `n` is the index for the recursive stage which represents the `n`-th level in the generation tree (see Fig. 9) and `bc` is the *binomial coefficient* for `n `choose` (k+1)`. The `choose` function is defined as

```
factorial :: Int -> Int
factorial n = product [1..n]

choose :: Int -> Int -> Int
n `choose` k
    | k < 0     = 0
    | k > n     = 0
    | otherwise = factorial n `div` (factorial k * factorial (n-k))
```

Next, modify the `revjoin` and `sort_revol_int` accordingly:

```
revjoin_int :: ([Int],Int) -> ([Int],Int) -> ([Int],Int)
revjoin_int (x,k1) (y,k2) = (y ++ x, k1)

sort_revol_int :: [([Int],Int)] -> [([Int],Int)]
sort_revol_int xs = case xs of
  [([],0)]     -> [([],0)]
  (x:ys)       -> [x] ++ mappair revjoin_int (init ys) ++ [last ys]
```

Finally, the integer SDP combinatorial generator with revolving door ordering is given by

```
sdp_gen5_int p fs e = foldl (choice5 fs) e
 where choice5 fs xs a = filter p $ sort_revol_int [f x a | x <- xs, f <-
    fs]

sdp_combs_revol_int :: [Int] -> [([Int],Int)]
sdp_combs_revol_int = sdp_gen5_int combs_revol_int_p combs_revol_int_fs
   combs_revol_int_e
 where
  combs_revol_int_fs = [upd_left_int, upd_right_int]
  combs_revol_int_p = const True
  combs_revol_int_e = [([0],0)]
```

Evaluating `sdp_combs_revol_int [1,2,3,4]` gives
`[([0],0),([0,1,2,3],1),([0,1,2,3,4,5],2),([0,1,2,3],3),([0],4)]`, where the second element in each tuple represents the size of the sublists.

Similarly, the integer K-combination generator `sdp_kcombs_revol_int` is defined as

```
sdp_kcombs_revol_int k = (!!k) . sdp_combs_revol_int
```

II.1.6 Chapter discussion

This chapter introduced a variety of efficient combinatorial generators based on SDPs as well as several CGC generators. Some of the SDP generators discussed here have been previously explored in the literature, either within the scope of combinatorial generation studies [Kreher and Stinson, 1999, Ruskey, 2003] or in the context of constructive algorithmics [Jeuring, 1993, Bird and De Moor, 1996]. The novel contribution of this chapter lies in the integration of SDP generation with ranking functions, enabling the design of integer SDP generators, which generate configurations with CGC ordering by using an SDP generator *without* explicitly executing the ranking function.

Nevertheless, several limitations remain regarding SDP generators. First, although the SDP generators introduced in this chapter are optimally efficient in terms of worst-case complexity, it will still be difficult to solve large-scale problems using them. In machine learning, distributed algorithms that require minimal or no communication between processes are often necessary to handle large-scale problems effectively, as witnessed in gradient descent algorithms used for training deep neural networks. Unfortunately, none of the generators introduced in this chapter are immediately *embarrassingly parallelizable*.

Additionally, SDP generators encounter significant limitations when applied to problems involving complex combinatorial structures. The main limitation of these generators is their requirement that the input sequence be provided before executing the generator. While this may not initially seem disadvantageous, as input data is typically known for most problems, it poses challenges for many complex combinatorial optimization problems that require constructing combinatorial structures within other combinatorial structures—essentially, *nested combinatorial generators*. Such nested generators frequently arise in machine learning, where problems often involve nested combinatorics, such as those involving piecewise linear functions. To execute a nested generator using SDP, it becomes necessary to store all possible configurations that are output by the **first** SDP generator before they can be used by the **second** SDP generator. However, this approach is impractical for most problems involving nested combinatorics, as the configuration size of a single combinatorial structure can grow exponentially, polynomially or worse. Storing all these configurations is both inefficient and memory-intensive.

Fortunately, both issues can be effectively addressed through a datatype-generic generalization of the SDP—the catamorphism. This generalization allows us to define combinatorial generators in a more succinct and conceptually principled manner. Moreover, the generality offered by catamorphisms enables the design of recursive generators with desirable algebraic properties, as determined by the datatype that defines the catamorphism's recursive structure. For instance, catamorphisms defined over the *join-list* datatype, with its inherent *associativity*, allow us to decompose problems in arbitrary ways, unlike the sequential decomposition inherent in SDPs. This flexibility facilitates, for instance, the construction of embarrassingly parallelizable recursive generators.

II.2 Constructive algorithmics

II.2.1 What is constructive algorithmics and why we need to care about it?

Don Knuth: Premature optimization being the root of all evil in programming [Knuth, 1974]

Before exploring the details of constructive algorithmics, it is necessary to first discuss some motivations for studying it. In programming, a recursive function $f : A \to B$ is a function that calls itself within its own definition. The most common definition for recursion can take the abstract form

$$f = \varphi\,(f) \tag{39}$$

where function φ is arbitrary function with type $\varphi : A \to B \to (A \to B)$.

The recursion function abstraction (39) has three immediate problems:

1. **No unique solution guarantee.** Although function φ can arbitrary as long as it has the required type, (39) is not guaranteed to be a meaningful definition for all possible φ, it may have a unique solution up to isomorphism or we can prove there is *no canonical choice.*.

2. **Lack of structure**. Because φ can have arbitrary form, it is impossible to know the structure of the recursion from the type information of φ only. Most of the time, the elaborate definitions for φ are very messy, it is very difficult to understand or gain any insights from the definition of one particular recursive algorithm. Therefore, it is unlikely that the design process for a particular recursion can be reused to guide the design of recursions for different problems.

3. **Lack of formal verification**. This problem is an inherent consequence of the second problem above. Due to the difficulty in observing the structure of recursion, the study of BnB algorithms rarely adopt a systematic and formal approach to program correctness verification. As a result, proofs of program correctness often rely on ad-hoc induction, making them difficult to conduct and prone to errors.

Most BnB algorithms are recursive functions and are typically presented in the form (39). This is because the design of BnB algorithms often relies on intuitive insights rather than systematic principles to support their derivation. Consequently, proofs for BnB algorithms frequently depend on weak assertions or informal explanations that do not hold up under close scrutiny. Formal proofs for these algorithms usually require induction and tend to be excessively lengthy, making them challenging to understand.

Instead of abstracting the recursion in the form of (39), this thesis employs a more structured abstraction for recursion, *hylomorphisms*. The recursion specified by a hylomorphism takes the form

$$f = \phi \cdot \mathbf{F} f \cdot \psi \tag{40}$$

where $\mathbf{F} : \mathcal{C} \to \mathcal{C}$ is called the *base functor* which determines the structure of the recursion, and the function $\psi : A \to \mathbf{F}A$ and $\phi : \mathbf{F}B \to B$ are called $\mathbf{F}$-*coalgebra* and $\mathbf{F}$-*algebra* respectively., these two names originate from the study of *universal algebra*. The recursion (40) is a more structured instance of (39), in other words, most practical recursive function defined by specifying an operator φ can be equivalently defined by specifying an appropriate $\mathbf{F}$, ψ and ϕ.

The focus of this chapter is the study of the theory of *structured recursion* in the form of (40). There are several benefits to examining recursions of this form: first, the sophisticated definition provided by (40) offers deeper insights into the structure of the recursion; it ensures that the recursion will terminate; and it allows the program to be reasoned about using its *calculational* properties.

The study of these structured recursions is part of *constructive algorithmics* or *transformational programming*, originally explored by Gerhart [1975], Burstall and Darlington [1977], Meertens [1986], Bird [1987], Balzer [1985], Bauer et al. [1985]. The theory of constructive algorithmics involves deriving programs from specifications. These specifications detail the clearest and most understandable program for the problem possible, disregarding efficiency concerns. Subsequently, an efficient program is derived without altering the results or meaning of the original specification. In other words, constructive algorithmics serves as a *calculus for programs*.

The value of this approach is in its **separation** of the concerns of *correctness* and of *efficiency* and *implementability*. It provides a flexible framework for formally verifying the correctness of algorithms, ensuring that they meet specified requirements and constraints. This is crucial for *safety-critical systems* or *high-stakes problems*, where incorrect behavior can have serious adverse effects. Constructive algorithmics was specifically developed to design reliable programs for these tasks. It provides a foundation for formal methods in algorithm design, and encourages a deeper understanding of algorithms by demanding rigorous proofs of correctness. This approach can yield insights into algorithmic properties and behaviors that might otherwise remain obscure.

II.2.2 Algebraic datatypes and catamorphisms

Datatypes are abstractions of data structures. For instance, the finite list is a datatype, while the single-linked list, the double-linked list etc. serve as concrete implementations of the finite list. Many data types are defined inductively, such as finite lists, binary trees, and natural numbers. It is reasonable to ask if we can find an conceptually principled language to unify these datatypes, using which it will be possible to construct similar recursive datatypes for free.

In this section, datatypes are abstracted categorically using $\mathbf{F}$-*algebras*. An $\mathbf{F}$-algebra is an arrow with type $alg : \mathbf{F}A \to A$; the object A is called the *carrier* of the initial algebra, and functor $\mathbf{F}$ is referred to as the *base functor*, which is defined in terms of *polynomial functors*. In Haskell, the algebra is rendered as `alg :: func a -> a`, with the syntax constraining type declarations to lowercase letters.

Given a polynomial functor $\mathbf{F}$, the $\mathbf{F}$-algebras, along with homomorphisms between different $\mathbf{F}$-algebras, form a category $\mathbf{Alg}(\mathbf{F})$. This category contains an *initial algebra*, which serves as the initial object in the category of $\mathbf{F}$-algebras, with their carriers representing the *least fixed point* for the given base functor $\mathbf{F}$. Some researchers will directly call the initial $\mathbf{F}$-algebras as the models for recursive datatypes, rather than the carriers of the initial $\mathbf{F}$-algebras, and we adopt the same terminology in our discussion as well.

Recursive datatypes are formally defined through the principle of least fixed points. In the category $\mathbf{Alg}(\mathbf{F})$, the homomorphisms between different datatypes ($\mathbf{F}$-algebras) are called $\mathbf{F}$-homomorphisms. These $\mathbf{F}$-homomorphisms are *structure-preserving maps* that generalize *homomorphisms* in universal algebra. Furthermore, when the domain of the $\mathbf{F}$-homomorphisms is an initial algebra in the category of $\mathbf{F}$-algebras, these $\mathbf{F}$-homomorphisms are called *catamorphisms*, which can be implemented as a recursive program and

the recursive structures of catamorphisms depend upon the structure of the input datatypes. In other words, the catamorphism is a datatype-generic recursive program. Additionally, these constructions can be dualized, enabling the construction of co-recursive or co-inductive datatypes, such as *infinite lists* (*streams*).

Before we provide the formal definitions for the terminology introduced above, we will first explain the main concepts of this chapter by constructing a well-known example in computer science: the *snoc-list*, which is just the list constructed, one element at a time, from left to right. Through this example, we aim to illustrate the following concepts:

- Datatypes can be constructed algebraically by polynomial functors.
- A recursive datatype can be modeled by the least fixed point of a base functor $\mathbf{F}$.
- The $\mathbf{F}$-homomorphism from the initial algebra to other algebras in this category can be implemented as recursive programs (catamorphisms).

II.2.2.1 An illustrative example: snoc-list

The snoc-list is the list built from right to left, for instance, a list $[1, 2, 3]$ of natural numbers can be constructed from the empty seed $[\,]$ and then appending each element from left to right, i.e $[1, 2, 3] = [\,] : 1 : 2 : 3$. In Haskell, we can define a snoc-list by the following datatype

```
data ListFl x a = Nil | Snoc x a
```

recall the previous Subsection I.2.4.3, the keyword `data` means we are defining a new datatype. The term on the left-hand side of the definition is referred to as a *type constructor* because it is used to construct a new datatype, with two parameters `a` and `x`. On the right-hand side are the *value constructors* which are used to produce the "*terms*" or "*data*" for this datatype.

Categorically, the type constructor `ListFl a` can be characterized as a polynomial functor $\mathbf{F_A} = \mathbf{1} + \mathbf{id} \times \mathbf{A}$, we will explain in more detail what this functor represents in the next section. The word "*polynomial*" refers to how the construction of the polynomial functor mimics the construction of ordinary polynomials, consisting of "*plus*" and "*times*" operations with respect to "constant terms" and "identities." The finite list is indeed a built-in datatype in Haskell, namely `[a]`. We define it explicitly to illustrate the concept of the recursive datatype, so `Snoc (Snoc (Snoc Nil 1) 2) 3` is the same as `[] : 1 : 2 : 3` in Haskell (the list constructor operator (or simply "cons" operator) `:` can only be applied from right to left, we apply it from left to right here as an analogy).

Similarly, we can also define a list by constructing it from the right to left or we can take the empty list as the seed and then join it with elements from both the right and left. These lists are the so-called *cons-list* and *join-list*.

Can the type variable `x` in datatype `ListFl` be an arbitrary type? The answer is affirmative: we can even substitute the type constructor itself as the argument for the value constructor. Then the datatype is parameterized by only one parameter `a`, which is rendered as

```
data Listl a  = Nil | Snoc (Listl a) a
```

and this definition exemplifies a *recursive datatype*. `Listl a` is called the *least fixed point*[12] of the functor `ListFl a`. This substitution can be made more generally for any polynomial functor, because we can prove that `Listl a` is isomorphic to `ListFl (Listl a) a`, that is `Listl a` ≅ `ListFl (Listl a) a`. This is a consequence of *Lambek's lemma* [Lambek, 1968]. The systematic derivation of a recursive datatype from a non-recursive definition can be accomplished more broadly by introducing the theory of **F**-algebras.

Assuming we want to map from the datatype `Listl a` to `Int`. For instance, the `length` function, which calculates the length of the list (from left to right), can be implemented as follows

```
length xs = go xs 0
  where go [] acc = acc
        go (_:xs) acc = go xs (acc + 1)
```

The `length` function serves as a homomorphism from snoc-list to integers, it recursively calculates the length of a list from the left to right. Equivalently, we can define the length function more compactly by using the `foldl` operator,

```
length :: [a] -> Int
length x = foldl (\acc a -> acc + 1) 0 x
```

where the function `\acc a -> acc +1` is called a *lambda expression*. Lambda expressions are just anonymous functions that are used because we need some functions only once, and the `foldl` operator is defined as

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f e [] = e
foldl f e (a : x) = foldl f (f e a) x
```

Indeed, `foldl` is precisely the catamorphism for the category of `ListFl a`-algebras. We can see the `foldl` operator is a program defined by two patterns, the first pattern is the empty case, and the second pattern is the non-empty case. This precisely corresponds to the definition of `ListFl a` datatype, which is defined by two terms, `Nil` and `Snoc (Listl a) a`. As we mentioned, catamorphisms are the **F**-homomorphisms with initial algebras as the domain algebra, in this case, the initial algebra is the `ListFl a` datatype.

The advantage of implementing recursive functions as a special case of a *datatype-generic recursive program* is two-fold:

- First, other than using `ListFl a` as the input, we can feed the catamorphism with more complex recursive datatypes as the input, these datatypes are constructed by using different polynomial functors. The recursive structure of catamorphism is automatically determined by the structure of the input datatype.
- Second, a datatype-generic recursive program—such as a catamorphism—allows us to produce modular programs, in the sense that we do not need to write a new program when we encounter a new

[12] It is a fixed point, meaning that $X \cong \mathbf{F}(X)$, for some functor **F**. The term "least" connotes that it possesses a unique algebraic mapping to any other fixed point of the functor. In Haskell, `data` keyword define the least fixed points, ensuring that recursion always terminates and all values are finite. In set theory, the least fixed $\mu\mathbf{F}$ is the smallest solution closed under the functor **F**, usually constructed as an initial algebra. Dually, corecursion corresponds to greatest fixed points ν, which model infinite structures (such as streams) and are constructed as final coalgebras.

problem, we just need to design a new algebra. For example, in the second definition of `length`, we only need to define the recursive update function `f = \acc a -> acc +1` and the seed value `e = 0`. Functions constructed using the generic program `foldl` are simple and compact.

II.2.2.2 Polynomial functors

Recall that, a functor is called *endofunctor* if it has a type $\mathbf{F} : \mathcal{C} \to \mathcal{C}$ which is a functor from a category to the category itself. Polynomial functors are endofunctors constructed through four fundamental algebraic rules and basic functors in an inductive manner, discussed next. These functors are crucial for defining initial algebras, which are used to model recursive datatypes. Throughout this paper, our focus will be exclusively on polynomial functors.

Identity and constant functor The identity functor **id**, maps every object and morphism to itself, that is, $\mathbf{id}X = X$, and $\mathbf{id}f = f$. Similarly, the constant functor $\mathbf{A}$ maps every object in this category to the same object $\mathbf{A}$, and morphisms to the identity function with respect to object A, i.e., $\mathbf{A}X = A$ and $\mathbf{A}f = id_A$.

Product functor Given two arrows $f : C \to A$ and $g : C \to B$, we use the symbol $\langle\ \rangle$ to denote the categorical *pairing arrow* $\langle f, g\rangle : C \to A \times B$ that applies two arrows f, g to the same object, where $A \times B$ called the *product* of two objects A and B. Some texts will also use $f \vartriangle g$ to denote $\langle f, g\rangle$. There are two *projection* arrows $fst : A \times B \to C$, $snd : A \times B \to C$ associated to pairing arrow $\langle f, g\rangle$. In Haskell, we can define them as

```
pair :: (c -> a) -> (c -> b) -> c -> (a,b)
pair f g = \a -> (f a, g a)

fst :: (a,b) -> a
fst (a,b) = a

snd :: (a,b) -> b
snd (a,b) = b
```

Note that this is not a fully defined product, as we have yet to verify the universal property discussed in I.2.4.2. However, it is safe to use in the context of this thesis.

If all objects in a category $\mathcal{C}$ have products, then we say the category $\mathcal{C}$ *has products*. Therefore the bifunctor $\times : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ can be used to define a *product category*, this bifunctor maps two objects A and B to the Cartesian product of A and B. For example, if object A, B are lists, in Haskell, the product functor on objects is rendered as

```
cp :: [a] -> [a] -> [(a,a)]
cp xs ys = [(x, y) | x <- xs, y <- ys]
```

The product functor on morphisms called the *cross operator*[13], it has type $cross\,(f, g) : A \times B \to C \times D$

[13]The pairing arrow $\langle f, g\rangle : C \to A \times B$ can not be the product functor on morphisms, because the input type does not match.

for $f : A \to C$ and $g : B \to D$. Categorically, we write the cross operator as $f \times g$. We can combine the projection operator and the pairing arrow to define the morphisms on product category

```
cross :: (a -> c) -> (b -> d) -> (a,b) -> (c,d)
cross f g = pair (f . fst) (g . snd)
```

which is equivalent to

```
cross f g = \(x, y) -> (f x, g y)
```

Coproduct functor A coproduct is a construction that generalizes the notion of a disjoint union of sets. In Haskell, the coproduct functor on objects is modeled by

```
data  Coproduct a b = Left a | Right b
```

Dually, given two arrows $f : A \to C$ and $g : A \to C$, we can define the dual of the paring operator as *copair* $(f, g) : A + B \to C$. Categorically, we use the symbol $[\,]$ to denote the coparing arrow as $[f, g] : A + B \to C$, it pronounced "case f or g", so it is also called *case arrow*. Some literature will also use $f \triangledown g$ to denote$[f, g]$. Similarly, there also exists two arrows `inl` $:: A \to A + B$ and `inr` $:: B \to A + B$, which maps an object to a coproduct. In Haskell, these functions are defined as

```
inl :: a -> Coproduct a b
inl a = Left a

inr :: b -> Coproduct a b
inr b = Right b

copair :: (a -> c) -> (b -> c) -> Coproduct a b -> c
copair f g (Left a) = f a
copair f g (Right b) = g b
```

The case arrow $[f, g] : A + B \to C$ is implemented using pattern matching in Haskell.
Similarly, the morphisms between coproduct objects is rendered as

```
cocross :: (a -> c) -> (b -> d) -> Coproduct a b -> Coproduct c d
cocross = copair (inl . f) (inr . g)
```

In the category of finite sets and total functions, denote as **Set**, if A is an object of size m and B is an object of size n, then $A + B$ has size $m + n$ while $A \times B$ has size $m \times n$.

Example 2. The snoc-list functor `ListFl a` is defined as $\mathbf{F_A} = \mathbf{1} + \mathbf{id} \times \mathbf{A}$ categorically, the mapping on object X is defined as $\mathbf{F_A}(X) = 1 + X \times A$ and mapping on morphism f is defined as $\mathbf{F_A}(f) = id_1 + f \times id_A$ where $\mathbf{1}$ is the constant functor for the terminal object 1. In the category **Set**, the terminal object can be understood as a singleton set, say $1 = \{c\}$, functor $\mathbf{1}$ maps every object to a unique element (for instance, the element `Nil` in the above definition) and every function to the identity function id_1 over the object 1. Similarly, the functor $\mathbf{A}$ maps every other object in the category to the same object A, and morphisms to the identity function id_A over the object A. **id** is the identity functor, which maps an object or a morphism to itself.

II.2.2.3 F-algebras and universal algebra

As mentioned, the theory of **F**-algebras is a generalization of classical universal algebra theory. Connections and differences between them are contained in this discussion, so as to provide better understanding of the theory of **F**-algebras in later sections.

Universal algebra describes specific algebraic structures, such as *groups*, *monoids* or *rings*. A monoid *homomorphism* $h : \mathbb{M}_1 \to \mathbb{M}_2$ between two monoids $\mathbb{M}_1 = (\mathbb{R}, \times, 1)$, $\mathbb{M}_2 = (\mathbb{R}, +, 0)$ is a *structure map* that satisfies $h(a \times b) = h(a) + h(b)$, for $\forall a, b \in \mathbb{R}$ and $h(1) = 0$. In the case of ordinary $+$, $\times$ operation on the set of real numbers, $h = \log$. Generalizing the idea of monoids and monoid homomorphisms can be achieved using the theory of **F**-algebras.

Assume $\mathcal{C}$ be a category and **F** an endofunctor on $\mathcal{C}$. An **F**-algebra is an arrow `alg` : $\mathbf{F}A \to A$ with *carrier* A. In Haskell, we can implement functor **F** as `func` which is an instance of typeclass `Functor`, in other words, `func` is a datatype that can derive a functor instance. Hence, the `func`-algebras (represent **F**-algebras mathematically) are defined as `alg :: Functor func =>func a -> a`.

A morphism between algebras `alg1 :: func a -> a` and `alg2 :: func b -> b` is an **F**-homomorphism, denoted as `h`, defined by the mapping between their carriers `h :: a -> b` such that `h . alg1 = alg2 . (fmap h)`, recall that `fmap` is the morphism part of the functor.

In the context of **F**-algebra theory, the squared functor $\mathbf{F} = \mathbf{id} \times \mathbf{id}$ is an appropriate model for monoid operations, its object part and morphisms part are defined as $\mathbf{F}X = (X)^2 = X \times X$ and $\mathbf{F}\mathtt{h} = (\mathtt{h})^2 = \mathtt{h} \times \mathtt{h}$. In Haskell, we can implemented the squared functor $\mathbf{F} = \mathbf{id} \times \mathbf{id}$ using type synonyms of Haskell built-in tuples

```
type Sqr x = (x,x)
```

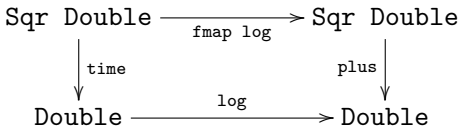
Then the monoid operation $\times : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$ can hence be defined as an `Sqr`-algebra `alg :: Sqr Double -> Double`, which can be implemented in Haskell as follows

```
times :: Sqr Double -> Double
times (a, b) = a * b
```

Similarly, the monoid operation $+$ is defined as

```
plus :: Sqr Double -> Double
plus (a, b) = a + b
```

The monoid homomorphism log is precisely the **F**-homomorphism from `time` to `plus` such that `log . time = plus . (fmap log)`, which makes the following diagram commute

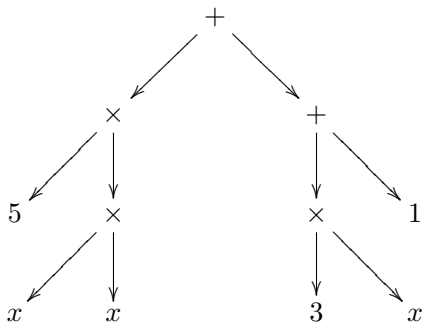


Additionally, the homomorphism preserves identity `log 1 = 0`.

The properties of **F**-algebra which generalizes universal algebra, are two-fold: Firstly, while all operators (except the identity and the negation) in universal algebra theory are binary, operators with arbitrary arity can be constructed in the theory of **F**-algebras. Secondly, the theory of **F**-algebras permits different

types for the arguments of the **F**-algebras. For instance, defining a binary operation $x \oplus y$ is a monoid operation defined as $\mathbb{M} = (\mathbb{X}, \oplus, id_{\oplus})$, x, y are elements in a same set $\mathbb{X}$, i.e., they have the same type. On the contrary, in **F**-algebras, the arguments of an algebra are allowed to have different types. For instance, the initial algebra on snoc-list has type `Snoc :: (Listl a) -> a -> (Listl a)` in Haskell, which receives an argument of type `ListF a` and then an argument `a`.

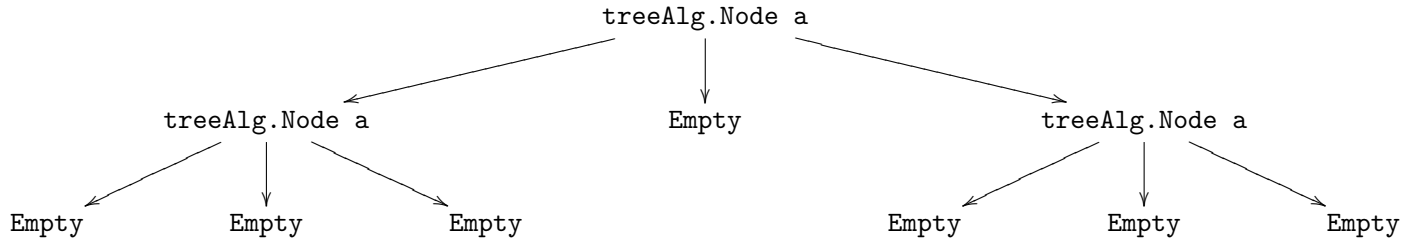
In computer science, arithmetic expressions like $5x^2 + 3x + 1$ can be represented as the following expression tree



and this tree is evaluated from the bottom up to reconstruct the expression. This tree is always a binary tree, as all non-leaf nodes represent binary operations, while the leaves are constrained to constants and variables. In the theory of **F**-algebras, it is easy to define a *ternary* tree using the polynomial functor $\mathbf{F_A} = \mathbf{1} + \mathbf{A} \times \mathbf{id} \times \mathbf{id} \times \mathbf{id}$. In Haskell, the object part of this functor is defined by

```
data TtreeF a x = Empty | Node a x x x
```

A `Ttree`-algebra `treeAlg`, the catamorphisms will recursively evaluate the tree by applying the algebra from the bottom to up. The recursive computation of a catamorphism can be drawn as follows:



In this diagram, each node represents a computation step performed by the algebra `alg`. The evaluation proceeds from the leaves (bottom) of the tree towards the root (top). This tree diagram demonstrates the generalizability of **F**-algebras, the `Ttree`-algebra `treeAlg` receives three elements, and the initial `Ttree`-algebra `Node` contains elements of two different types.

II.2.2.4 Catamorphism characterization theorem

Fixed point operator The initial **F**-algebra $in : \mathbf{F}\mu\mathbf{F} \to \mu\mathbf{F}$ is the *initial object* in the category $\mathbf{Alg}(\mathbf{F})$[Malcolm, 1990, Bird and De Moor, 1996], where type $\mu\mathbf{F}$ is the carrier. The carrier of the initial **F**-algebra is the least fixed point of functor **F**. The carrier $\mu\mathbf{F}$ of the initial **F**-algebras $in : \mathbf{F}\mu\mathbf{F} \to \mu\mathbf{F}$ models the recursive datatype. In the literature, initial **F**-algebras are commonly explained as models for

recursive datatypes, and we adopt this convention. According to Lambek's lemma [Lambek, 1968], the initial property induces an isomorphism $\mu\mathbf{F} \cong \mathbf{F}\mu\mathbf{F}$. Therefore, we can apply functor $\mathbf{F}$ an infinite number of times to the datatype $\mu\mathbf{F}$, and it will still be isomorphic to itself. Hence $\mu\mathbf{F}$ is a recursive datatype.

The fixed point operator $\mu : \mathbf{F} \to \mu\mathbf{F}$ can be considered as a function which receives a functor and returns the fixed point of this functor, we call $\mu\mathbf{F}$ the recursive datatype defined by functor $\mathbf{F}$ or the fixed point of functor $\mathbf{F}$. We can implement the fixed point operator μ in Haskell as

```
newtype Mu func = In {out :: func (Mu func)}
```

where keyword `newtype` is similar to `data` type constructor, but `newtype` can receive only *one* argument. The braces on the right-hand side are called *record syntax*. In particular, `In` is called a *record*, and `out` is called the *field* of this record. By using record syntax, we do not need to define the accessor for the component separately. Here, `In` and `out` can be understood as two arrows with the types `In :: func (Mu func) -> (Mu func)` and its reversed arrow `out :: Mu func -> func (Mu func)`, these two arrows form the initial/terminal object of the category $Alg(\mathbf{F})$.

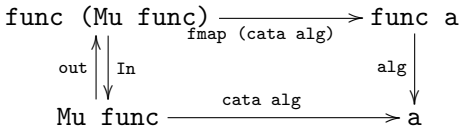
As an example, in the above section, we have explicitly defined the recursive definition for the snoc-list functor `ListFl a` as `Listl a`, now we can equivalently define `Listl a` by just calling `Mu (ListFl a)`, the least fixed point of functor `ListFl a` will automatically be generated by `Mu (ListFl a)`.

Initiality and catamorphism As discussed in Subsection I.2.4.2, an initial object in category $\mathcal{C}$ has a unique morphism to other objects in this category. Hence there is a *unique* $\mathbf{F}$-homomorphisms (unique up to isomorphism) from $\mu\mathbf{F} \to A$ which maps an initial algebra $\texttt{in} : \mathbf{F}\mu\mathbf{F} \to \mu\mathbf{F}$ to any algebra $\texttt{alg} : \mathbf{F}A \to A$ in the category $Alg(\mathbf{F})$. This $\mathbf{F}$-homomorphism is called the *catamorphism*.

In Haskell, the catamorphism is an arrow from `Mu func -> a`, which maps the initial algebra `In :: func (Mu func) -> (Mu func)` to an algebra `alg :: func a -> a`, such that `(cata alg) . In = alg . fmap (cata alg)`. We can apply the terminal algebra `out` on both sides of the equation. Then the structure condition for the `F`-homomorphisms `cata alg` can be rendered as

$$\texttt{cata alg} = \texttt{alg . fmap (cata alg) . out}, \tag{41}$$

this is called *catamorphism characterization theorem*. In other words, the following diagram commutes



The definition of catamorphism immediately implies the *reflection law* `cata In = id`. This holds because a catamorphism always maps an initial algebra `In` to another algebra `alg`, so that with `alg = In`, the catamorphism maps the initial algebra `In` to itself, thus the catamorphism becomes the identity function.

Given an $\mathbf{F}$-algebra `alg :: Functor func =>func a -> a`, where base functor `func` is constrained to Haskell's *functor class* `Functor`, the catamorphism characterization theorem (41) can be defined as

```
cata ::Functor func => (func a -> a) -> Mu func -> a
cata alg = alg . fmap (cata alg) . out
```

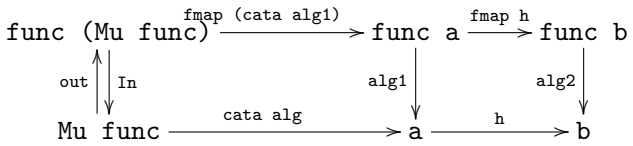
where the catamorphism `cata` takes an algebra `alg :: Functor func =>func a -> a` and a fixed point type `Mu func` as input. Although the fixed point input is not explicitly given in the definition of the function, it is reflected in the type declaration. This is again because of the use of *currying*. In the case here, the catamorphism `cata` receives an algebra `alg :: func a -> a` and then returns a partial function of type `Mu func -> a`.

One of the most important corollaries of the catamorphism characterization theorem is the *catamorphism fusion law*. Fusion gives the condition that has to be satisfied in order to "fuse" the composition of a function with a catamorphism into a new catamorphism.

Corollary 1. *Catamorphism Fusion Theorem.* Given a *function* `h :: a -> b`, an algebra `alg1 :: func a -> a` and another algebra `alg2 :: func b -> b`, the fusion law states the following implication

$$\texttt{h . (cata alg1)} = \texttt{cata alg2} \Longleftarrow \texttt{h . alg1 = alg2 . (fmap h)}, \tag{42}$$

and the following diagram commutes



The proof of the fusion law beyond the scope of this thesis, interested readers can refer to [Bird and De Moor, 1996, Jeuring, 1993]. The fusion law (42) is also sometimes referred to as the *promotion law*. *Distributivity* and *associativity* in universal algebra can be seen as special cases of this law. For example, the distributivity of `h` over `alg` can be reformulated as `h . alg = alg . (fmap h)` over squared functor. This is another demonstration for why the theory of **F**-algebras generalizes universal algebra.

Fusion is a special case of *reordering of computation*. A function `h` satisfying the right-hand side equality can replace `alg1` with a new algebra `alg2`. The program `cata alg2` can be more efficient than `h . (cata alg1)`. In the context of combinatorial optimization problems, `cata alg1` represents a combinatorial generator `gen`, and the function `h` performs some computations on the configurations generated by `gen`. In ML applications, computations such as filtering or evaluation on combinatorial configurations, might be fusable with the generator `gen`. Indeed, both the evaluator and the filter can be fused under certain conditions, as discussed in detail later.

Catamorphisms are the most fundamental type of datatype-generic program, the functor is the *syntax* of the program, which determines the structure of the recursion. Various useful datatypes will be discussed in the next section, and how these datatypes determine the structure of the recursion will be elaborated upon.

F-algebras represent the *semantics* of programs, which determine the content of the recursion. In the context of this thesis, the content includes recursive optimization algorithms and recursive combinatorial generators. Various useful algebras for combinatorial generation will be presented in Section II.2.3.

II.2.2.5 Various useful recursive datatypes

To highlight the effectiveness of polynomial functors, several datatypes that are frequently used in ML research are introduced below, which also demonstrates both their utility and limitations.

Cons-list In snoc-list we construct a list from the left to right. The cons-list is essentially a dual definition of the snoc-list. Categorically, it is defined by functor $\mathbf{F_A} = \mathbf{1} + \mathbf{A} \times \mathbf{id}$. The cons-list functor on-objects part is implemented as

```
data ListFr a x = Nil | Cons a x
```

The recursive definition for cons-list can be derived by applying the fixed point operator to the cons-list functor definition above, `Mu (ListFr a)`. Equivalently, we can define the recursive definition of cons-list explicitly as

```
data Listr a = Nil | Cons a (Listr a)
```

Similarly, the catamorphism implied by this datatype has the same functionality as the `foldr` operator, which is defined as

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e [] = []
foldr f e a:x = f a (foldr f e x)
```

The functor on morphism part for cons-list functor is $\mathbf{F_A} f = id_1 + id_A \times f$. As discussed in Subsection I.2.4.3, one way to derive the functor `fmap` for functor `ListFr a` is by using the `deriving` keyword after the datatype declaration. We can also define the `fmap` for datatype `ListFr a` explicitly as

```
fmap :: (y -> x) -> ListFr a y -> ListFr b x
fmap _ Nil = Nil
fmap f (Cons a x) = Cons a (f x)
```

Join-list The join-list datatype is similar to a binary tree, but the binary tree datatype does *not* have *associativity*. Rather than constructing a list from right to left or left to right, we can assume there exists a seed element, and then the list is constructed from both sides. The polynomial functor for defining the join-list datatype can be expressed as $\mathbf{F_A} = \mathbf{1} + \mathbf{A} + \mathbf{id} \times \mathbf{id}$.

In Haskell, the join-list functor and its recursive definition are rendered as

```
data ListFj a x = Nil | Single a | Join x x
data Listj a = Nil | Single a | Join (Listj a) (Listj a)
```

similar to the cons/snoc-list case, the recursive definition can be equivalently defined by `Mu (ListFj a)`.

Categorically, the morphisms part of the mapping is defined as $\mathbf{F_A}(f) = id_1 + id_A + f \times f$. The explicit Haskell implementation of `fmap` based on `ListFj a` functor is given as

```
fmap :: (Functor func) => (a  -> b)  -> func a -> func b
fmap f Nil = Nil
fmap f (Single a) = Single a
fmap f (Join x y) = Join (f x) (f y)
```

Using the join-list datatype, lists can be freely joined from both sides. This gives freedom to split the list in an arbitrary way and then join them together using the `Join` operator. Therefore, the catamorphism with a join-list as input follows a recursive pattern: it allows the problem to be split into arbitrary

contiguous segments, solves each sub-problem independently, and then the solutions are "joined" together using a `ListFj`-algebra. In particular, if this algebra is associative, this allows the construction of an *embarrassingly parallelizable recursive program*. This fact will be investigated further and used to construct a few embarrassingly parallelizable catamorphism generators based on the join-list datatype.

As previously discussed, Haskell provides a built-in list datatype `[a]`. If we want to work with lists, it would be much more convenient to use the built-in list `[a]` directly with the `cata alg` function. However, `cata alg` has a type `Mu f -> a`, it will be more convenient if `cata alg` has type `[a] -> a`, as it would be cumbersome to write an input like `Join (Single 1) (Join (Single 2) ((Single 3)))` instead of `[1,2,3]`. To achieve this, we need a helper function that transforms the built-in list datatype `[a]` into the recursive list datatype `Mu f`. For example, the transformation from `[a]` to `Mu (ListFj a)` can be achieved through the following recursive function

```
conv :: [a] -> Mu (ListFj a)
conv []  = In (Nil)
conv [a] = In (Single a)
conv x   = In (Join (conv (take m x)) (conv (drop m x)))
 where m = (length x) `div` 2
```

where the `conv` function recursively splits a list into two equal halves. However, this definition requires three traversals of the list. As our primary focus here is to illustrate these concepts. A more efficient implementation that traverses the list only once is presented in Section 5.2 of Bird and Gibbons [2020].

Then, the join-list catamorphism with built-in list `[a]` as input can hence be defined as

```
cata' alg = cata alg . conv
```

Note that a new `conv` must be defined for each new functor.

Rather than define a conversion function `conv`, it is much more convenient to define a suitable terminal algebra `out`. This ad-hoc `out` function on the join-list can be defined as

```
out :: [a] -> ListFj a [a]
out []    = Nil
out [a]   = Single a
out x = Join (take m x) (drop m x)
 where m = (length x) `div` 2
```

In the following discussion, we will implicitly assume that we have defined the ad-hoc `out` function when we are working on (Haskell) lists.

Similarly, for the cons-list, a suitable terminal algebra is

```
out :: [a] -> ListFr a [a]
out [] = Nil
out (a:x) = Cons a x
```

Note that the `out` function defined here is specific to the join-list functor `ListFj a` and the cons-list functor `ListFr a`. By contrast, the `out` field in the definition of the record `Mu f` is polymorphic

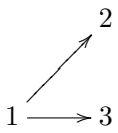
with respect to any polynomial functor and can be viewed as an imaginary function with the type `out :: Mu f -> f(Mu f)`.

Algebraic directed graphs A *directed graph* is a ubiquitous datatype in ML research. For instance, the directed graph is widely used in modeling *probabilistic* or *causal relations*. The most common definition for the graph is to define a pair $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. However, this definition suffers from the problem that it is easy to define *malformed graphs*, i.e., an edge refers to a non-existent vertex. For instance, given the graph defined as $G = (\{1\}, \{(1, 2)\})$, the vertex set $\{1\}$ contains only one vertex, but the edge $(1, 2)$ refers to the non-exist vertex 2.

This raises the natural question for whether a definition for directed graphs can be given that precludes the construction of malformed graphs. The answer is affirmative; this definition of *algebraic graphs* comprises four graph construction primitives. The simplest graph is the empty graph, denoted by $e = (\emptyset, \emptyset) \in G$. Similarly, the single vertex graph with one vertex v is denoted by $v = (v, \emptyset) \in G$. There are two primitive binary operations for constructing a larger graphs from smaller graphs. These two primitive operations are called *overlay* and *connect*, defined as

$$\begin{aligned} \text{Overlay}\left((V_1, E_1), (V_2, E_2)\right) &= (V_1 \cup V_2, E_1 \cup E_2) \\ \text{Connect}\left((V_1, E_1), (V_2, E_2)\right) &= (V_1 \cup V_2, E_1 \cup E_2 \cup (V_1 \times V_2)) . \end{aligned} \tag{43}$$

The overlay of two graphs comprises the union of their vertices and edges, and the connect operation takes the union of two graphs and creates new edges from each vertex in V_1 to each vertex in V_2. For example, the graph Connect $(1, \text{Overlay}\,(2, 3))$ can be illustrated as



The graphs constructed by the above four primitives are called *algebraic directed graphs* (ADGs). Graphs constructed in this way are sound and complete, in the sense that malformed graphs cannot be constructed and any graphs can be constructed in this way, proofs of these claims can be found in Mokhov [2017].

The four primitive ADG operations can be defined by a polynomial functor $\mathbf{F_A} = \mathbf{1}+\mathbf{A}+\mathbf{id}\times\mathbf{id}+\mathbf{id}\times\mathbf{id}$, the corresponding datatype in Haskell is:

```
data DiGraphF a x = NilF | VertF a | OverF x x | ConnF x x
```

Similarly, the explicit recursive definition can be rendered as

```
data DiGraph a = Empty
               | Vertex a
               | Overlay (Graph a) (Graph a)
               | Connect (Graph a) (Graph a)
```

Here `Empty` and `Vertex` construct the empty graph and single-vertex graph, respectively; `Overlay` composes two graphs by taking the union of their vertices and edges, and `Connect` is similar to `Overlay` but also creates edges between the vertices of the two graphs.

The directed graphs constructed in this way are safe and flexible. Additional axioms extend the definition of directed graphs to represent undirected, reflexive, and hypergraphs [Mokhov, 2017].

The algebraic directed graph functor on the morphism part is defined as $\mathbf{F_A} f = id_1 + id_A + f \times f + f \times f$. The corresponding `fmap` for the functor `DiGraphF a` can be implemented explicitly as

```
fmap :: (y -> x) -> DiGraph a y -> DiGraph b x
fmap _ L = L
fmap f (Vertex a) = Vertex (f a)
fmap f (Overlay x y) = Overlay (f x) (f y)
fmap f (Connect x y) = Connect (f x) (f y)
```

Binary tree In the definition of a *branch-labelled binary tree*, every node in the tree has an associated node element and two subtrees. Hence, the binary tree polynomial functor is $\mathbf{F_A} = \mathbf{1} + \mathbf{id} \times \mathbf{A} \times \mathbf{id}$. The datatype implement this functor is

```
data BtreeF a x = L | N x a x
```

The explicit recursive definition is rendered as

```
data Btree a = L | N (Btree a) a (Btree a)
```

The binary tree functor on the morphism part is defined as $\mathbf{F_A} f = id_1 + f \times id_A \times f$. The corresponding `fmap` based on `BtreeF a` functor can be rendered explicitly as

```
fmap :: (y -> x) -> BtreeF a y -> BtreeF b x
fmap _ L = L
fmap f (N x a y) = N (f x) a (f y)
```

II.2.3 Catamorphism combinatorial generation

This section, reformulates the generators introduced in Section II.1.2 of Part I, where various combinatorial generators are built using the **`foldl`** operator, which corresponds to the catamorphism based on the snoc-list datatype. To emphasize the datatype-generic nature of catamorphisms, we demonstrate how to construct catamorphism generators based on both cons-list datatype and join-list datatype. The modularity of the catamorphism generator is evident through this construction, as combinatorial generators based on cons-lists and join-lists are instantiated simply by defining the appropriate `ListFr a`-algebras and `ListFj a`-algebras.

Note that some of the generators described in this section do **not** achieve optimal efficiency in Haskell. We use Haskell primarily as a tool for explanation, while efficient implementations are written in an imperative language such as Python or C++. The goal here is to clarify each generator as much as possible to aid understanding, rather than to provide the most efficient Haskell code.

Additionally, after introducing the construction of generators for basic combinatorial structures, efficient generators for more complex combinatorial structures can be obtained by combining simpler ones. This is achieved by introducing three fundamental fusion laws: *filter fusion*, *product fusion*, and *cross product fusion*.

II.2.3.1 Cross product operator

Before introducing the various combinatorial generators based on catamorphisms, it is important to highlight an observation about the SDP combinatorial generators in the previous section. The use of `choice` functions can become cumbersome, as different `choice` functions must be defined whenever the decision functions involve new properties.

Nevertheless, these choice functions have similar natures and appear very similar to the Cartesian product over two lists. Indeed, both the Cartesian product operator and choice functions can be generalized using the *cross product* operator. With the help of this operator, we can construct generators in a more conceptually straightforward and succinct manner.

In Haskell, we can define the cross product as

```
crp f x y = [f a b | a <- x, b <- y]
```

The cross product function is a generalization of the Cartesian product, as it applies a binary function `f` to each element `a` in `x` and each element `b` in `y`, and stores them in the list. For instance, the Cartesian product operator can be defined as `cp x y = crp (,) x y`, so that evaluating `cp [1,2] [3,4]` will return `[(1,3),(1,4),(2,3),(2,4)]`. Alternatively, the `crp` operator can be defined in terms of `cp` as `crp f x y = map (uncurry f) (cp x y)`.

Two other, similar cross join operators `crpr` and `crpl` (short for "cross product, right" and "cross product, left") as follows

```
crpr f a y = [f a b | b <- y]
crpl f x b = [f a b | a <- x]
```

In combinatorial generation research, one frequently used operator is the *cross join* operator, which can be specified as

```
crj x y = crp (++) x y
```

Evaluating `crj [[1,2],[2,1]] [[3,4],[4,3]]` gives `[[1,2,3,4],[1,2,4,3],[2,1,3,4],[2,1,4,3]]`. Similarly, the "cross join, right" operator is `crjr = crpr (++)` and "cross join, left" operator is `crjl = crpl (++)`.

The cross product operator is frequently used in various combinatorial generation tasks, particularly for catamorphisms based on the join-list datatype. This is because the cross product encapsulates the essential idea in the principle of optimality: the solution to smaller "subproblems" can be combined to solve a larger "problem." When working with two sets or lists of subproblems, the cross product operator enables us to construct all possible combinations of solutions to these subproblems, which are then combined to obtain the solution to the final problem.

II.2.3.2 Catamorphism generators based on cons-list

Catamorphism-based combinatorial generators using cons-lists are essentially the same as the SDP generators described in Section II.1.2. However, as demonstrated below, expressing the same concepts using different languages at varying levels of abstraction can lead to significant differences. The combinatorial generators defined using cons-lists are much more succinct compared to the definitions provided in Section II.1.2.

Similar to the SDP generators, it is possible to fuse a filtering process into a catamorphism defined over cons-list algebra, if the predicate prefix-closed

$$\texttt{p (x ++ y)} \implies \texttt{p x},$$

for all `x` and `y`. Given a cons-list algebra `f`, the prefix-closed condition can also be rendered as `p f (Cons a x) = q f (Cons a x) && p x`, where `q` is modified predicate which is more efficient than `p`.

This Subsection redefines most of the generators illustrated in Section II.1.2 using catamorphisms based on cons-lists. Additionally, it introduces two new combinatorial generators that produce all possible initial or tail segments of a list. These new generators will enable the construction of a more efficient permutation generator compared to the previous definitions.

Sublists, K-sublists and K-sublists with revolving door ordering The sublist generator `subs` is defined as

```
subsAlg Nil = [[]]
subsAlg (Cons a xs) = crj fs xs
 where fs = [[], [a]]

subs = cata subsAlg
```

Similarly, since the max length predicate `maxlen k = (<= k) . length` is prefix-closed, the K-sublists `ksubs` generator can be obtained by fusing the max length predicate with `subsalg`

```
ksubsAlg k Nil = [[]]
ksubsAlg k (Cons a xs) = filter (maxlen k) $ crj fs xs
 where
  fs = [[], [a]]
  maxlen k = (<= k) . length

ksubs :: Int -> [a] -> [[a]]
ksubs k = filter (ksubs_p k) . cata (ksubsAlg k)
```

Evaluating `ksubs` generates all k-sublists for k smaller than K, `ksubs 2 [1,2,3] = [[2,3],[1,3],[1,2]]`.

Borrowing the definition of `upd_left`, `upd_right` and `sort_revol` from Section II.1.5, the definition of `ksubs_revol` is rendered as

```
ksubs_revolAlg :: Eq a => ListFr a [[[a]]] -> [[[a]]]
ksubs_revolAlg Nil = [[[]]]
ksubs_revolAlg (Cons a xs) = sort_revol [f x a | x <- xs, f <- [upd_left,
   upd_right]]

ksubs_revol :: Eq a => [a] -> [[[a]]]
ksubs_revol = cata ksubs_revolAlg
```

Evaluating these sublist generators will produce the same results as the SDP generators in Section II.1.2.

Sequence In the case of sequence generation, there is only one decision function. Thus, the sequence generator and its algebra can be defined as

```
seqnAlg :: ListFr a [[a]] -> [[a]]
seqnAlg Nil = [[]]
seqnAlg (Cons a xs) = crj fs xs
 where fs = [[a]]

seqn = cata seqnAlg
```

Initial segments and tail segments An *initial segment* of a list is also known as the *prefix* of the list. A list `y` is an initial segment of `x` if there exists a `z` such that `x = y ++ z`. The function `inits` returns the list of initial segments of a list, in *increasing order* of length. Conversely, a list `y` is a *tail segment* of `x` if there exists a `z` such that `x = z ++ y`. The function `tails` returns the list of tail segments of a list, in *decreasing order* of length. In Section 5.6 of Bird and De Moor [1996], two generators for producing initial and tail segments based on the cons-list operator are provided, which are defined as follows

```
initsAlg :: ListFr a [[a]] -> [[a]]
initsAlg Nil = [[]]
initsAlg (Cons a xs) = extend a xs
  where extend a xs = [[]] ++ (map (a:) xs)

inits = cata initsAlg

tailsAlg :: ListFr a [[a]] -> [[a]]
tailsAlg Nil = [[]]
tailsAlg (Cons a xs) = extend a xs
  where extend a (x:xs) = (a:x):x:xs

tails = cata tailsAlg
```

Binary and multiary assignments The binary and multiary assignment generators can be considered as sequences of cross join operations. For a binary assignment, the list `[[0], [1]]` is used, while for a multiary assignment, the list `[[i] | i <- [0..(m-1)]]`. These two generators and their algebras are given in Haskell as follows:

```
basgnsAlg :: ListFr a [[Int]] -> [[Int]]
basgnsAlg Nil = [[]]
basgnsAlg (Cons a xs) = crj fs xs
 where fs = [[0], [1]]
```

```
basgns = cata basgnsAlg

masgnsAlg :: Int -> ListFr a [[Int]] -> [[Int]]
masgnsAlg m Nil = [[]]
masgnsAlg m (Cons a xs) = crj fs xs
 where fs = [[i] | i <- [0..(m-1)]]

masgns m = cata (masgnsAlg m)
```

Permutations and K-permutations The permutation generator in Section II.1.2 was based on inserting a new element to the existing partial permutation, using an insertion function `insertKth` that inserts a new element `a` into the existing partial permutation.

Notice that in the definition of `insertKth`, the first `k` elements of list `x` are joined with the new element `a` and remaining list, for all `k` in `[0..len(x)]`. This is essentially the same as splitting `x` to a length `k` initial segment and a length `len(x)-k` tail segment. A split function `splits` that generates all possible splits of a list by zipping the results of an initial segment generator with a tail segment generator is given by

```
splits :: [a] -> [([a],[a])]
splits = (uncurry zip) . (pair inits tails)
```

and `insertKth` for all `k` in `[0..len(x)]` can be defined by the following `adds` function

```
adds a x = [y ++ [a] ++ z | (y,z) <- (splits x) ]
```

Thus a more efficient permutation generator based on `adds`, can be defined as

```
permsAlg :: ListFr a [[a]] -> [[a]]
permsAlg Nil = [[]]
permsAlg (Cons a xs) = concat [adds a x | x <- xs]

perms = cata permsAlg
```

Similarly, the algebra for the K-permutations generator can be defined more compactly as

```
kperms_p x = not (elem (head x) (tail x))

kpermsAlg :: Eq a => [a] -> ListFr a [[a]] -> [[a]]
kpermsAlg x Nil = [[]]
kpermsAlg x (Cons a ys) = filter kperms_p (prepends x ys)
 where prepends x ys = [i:y| i <- x, y <- ys]

kperms k x = cata (kpermsAlg x) (take k x)
```

List partitions The list partition, similarly, can be defined more compactly as

```
partsAlg :: Eq a => ListFr a [[[a]]] -> [[[a]]]
partsAlg Nil = [[[]]]
partsAlg (Cons a xs)
    | xs == [[[]]] = [[[a]]]
    | otherwise = [f x | x <- xs, f <- [appdlast a, extent a]]

parts :: Eq a => [a] -> [[[a]]]
parts = cata partsAlg
```

II.2.3.3 Catamorphism generators based on join-list

This section provides a comprehensive explanation of the definition of several `ListFj a`-algebras used for generating basic combinatorial structures, including sublists, K-sublists, permutations, list partitions, K-combinations, and K-permutations.

One immediate advantage of join-list algebras for generating combinatorial structures is that they facilitate the design of embarrassingly parallelizable programs. It is known in the literature that *associativity* is the key to designing embarrassingly parallelizable programs [Ladner and Fischer, 1980, Cole, 1989, Emoto et al., 2012]. Associativity enables a sequence of operations to be carried out without regard to the order of these operations. This is inherent in the list join operation, that is

$$\texttt{x ++ y ++ z = (x ++ y) ++ z = x ++ (y ++ z)} \ . \tag{44}$$

Therefore, with a join-list algebra that is associative, an embarrassingly parallelizable program can be obtained immediately. This capability is crucial for solving large-scale combinatorial optimization problems.

Incorporating the filtering process into the join-list algebra differs from the cons-list case. For the join-list datatype, fusing a prefix-closed predicate `p` within a join-list algebra can be achieved if the predicate `p` is *segment-closed*

$$\texttt{p (x ++ y)} \implies \texttt{p x \&\& p y}, \tag{45}$$

for all `x` and `y`.

Sublists, K-sublists Intuitively, the join-list sublists generator is constructed using the fact that the sublists of size `k` can only be constructed by joining possible sublists of size smaller than `k`. The sublist algebra is defined as follows

```
subsAlg :: ListFj a [[a]] -> [[a]]
subsAlg Nil = [[]]
subsAlg (Single a) = [[],[a]]
subsAlg (Join x y) = crj x y

subs = cata subsAlg
```

The associativity of the `crj` operator can be verified as follows

```
  crj (crj x y) z
≡ definition of crj
  crj [a + b | a <- x, b <- y] z
≡ definition of crj
  [a + b + c| a <- x, b <- y, c <- z]
≡  definition of crj, twice
  crj x (crj y z)
```

Similarly, it is straightforward to verify that the maximum length predicate `maxlen k x` is not only prefix-closed but also segment-closed. Thus the K-sublists algebra can be viewed as a sublists algebra combined with a maximum length filter, which is defined as follows

```
ksubsAlg :: Int -> ListFj a [[a]] -> [[a]]
ksubsAlg k Nil = [[]]
ksubsAlg k (Single a) = filter (maxlen k) [[],[a]]
ksubsAlg k (Join x y) = filter (maxlen k)(crj x y)

ksubs k = filter (ksubs_p k) . cata (ksubsAlg k)
```

where `maxlen k = (<= k) . length`.

Binary assignments and multiary assignments Analogously, the binary assignment algebra is essentially the same as the sublist algebra, which can be defined as

```
basgnsAlg :: ListFj a [[Int]] -> [[Int]]
basgnsAlg Nil = [[]]
basgnsAlg(Single a) = [[0],[1]]
basgnsAlg (Join x y) = crj x y

basgns = cata basgnsAlg
```

and the multiary assignment generator can be defined as

```
masgnsAlg :: Int -> ListFj a [[Int]] -> [[Int]]
masgnsAlg m Nil = [[]]
masgnsAlg m (Single a) = [[i]| i <- [0..(m-1)]]
masgnsAlg m (Join x y) = crj x y

masgns m = cata (masgnsAlg m)
```

Permutations To design a permutation generator based on the join-list datatype, the insertion or selection processes used in the cons-list generator are no longer suitable. Instead, a list of permutations

`xs` must be merged with another list of permutations `ys`, rather than inserting a new element `a` to a list of permutations `xs`.

To address this, Jeuring [1993] introduced a `merge` operator that can merge two permutations while preserving associativity over sets [Jeuring, 1993]. Since we do not care about the ordering of configurations, we can safely define it over lists, as given by

```
merge x [] = [x]
merge [] y = [y]
merge x@(a:x') y@(b:y') = (crpr (++) [a] (merge x' y)) ++
                          (crpr (++) [b] (merge x y'))
```

The operation `merge x y` means that, merging two partial permutations `x` and `y`, requires insert the element of `y` into all possible positions of `x` while preserving the original ordering of elements in `y`. For instance, evaluating `merge [1,2] [3,4]` gives `[[1,2,3,4],[1,3,2,4],[1,3,4,2],[3,1,2,4],[3,1,4,2],[3,4,1,2]]`. In this result, the merged permutation maintains the order of the original partial permutations; for instance, `1` is ahead of `2`, and `3` is ahead of `4`. Indeed, this merge function is the join-list version of the *interleave function* [Bird and Gibbons, 2020]. This function is discussed in greater detail in Chapter III.2 in Part III.

After defining this associative operation, the permutation algebra on the join-list can be constructed as follows

```
permsAlg :: ListFj a [[a]] -> [[a]]
permsAlg Nil = [[]]
permsAlg (Single a) = [[a]]
permsAlg (Join x y) = concat (crp merge x y)

perms = cata permsAlg
```

List partitions The associative operation for list partition algebra, as presented by Jeuring [1993], is defined as

```
segmake xs [] = [xs]
segmake [] xs = [xs]
segmake xs ys = [xs ++ ys, init xs ++ [last xs ++ head ys] ++ tail ys]
```

Hence the list partitions algebra based on join-list is rendered as

```
partsAlg :: ListFj a [[[a]]] -> [[[a]]]
partsAlg Nil = [[]]
partsAlg (Single a) = [[[a]]]
partsAlg (Join x y) = concat (crp segmake x y)

parts = cata partsAlg
```

K-combinations and K-permutations The algebras for generating K-combinations and K-sublists are designed to produce the same outcomes, but they differ in their algebraic structures, resulting in outputs of different types. To distinguish between them, one is called, `ksubsAlg`, which is defined above. The other is called `kcombsAlg`; its generator stores k-combinations of the same size k together in a single list, which has a different type compared with the join-list `ksubsAlg` algebra defined above.

This `kcombs` generator is constructed from the observation that K-combinations can only be constructed from two combinations with size sum up to K. In other words, we need not only K-combinations, but also combinations smaller than size K. Thus, the algebra for the K-combination generator can be defined as follows

```
kcombsAlg :: Int -> ListFj a [[[a]]] -> [[[a]]]
kcombsAlg k Nil = [ [] ] : replicate k []
kcombsAlg k (Single a) = [[]]:[[a]]:replicate (k-1) []
kcombsAlg k (Join xss yss) = [ concat (reverse (zipWith crj xss (reverse
    yss'))) | yss' <- tail (inits yss) ]
```

For the `Nil` case, there is only one way to construct a 0-combination and no way to form larger combinations. For the `Single` case, assuming `k > 0`, there is one way to create a 0-combination, one way to form a 1-combination, and no way to generate combinations of larger sizes. Finally, for the `Join` case, constructing `n`-combinations for all $0 \leq$ `n` $\leq$ `k` requires combining the `i`-combinations from `xss` with the `j`-combinations from `yss` such that `i+j=n`. This process is essentially a convolution[14] between `xss` and `yss`. Specifically, to construct `k`-combinations, we need only the first `k` elements of `xss` and `yss`, as combinations of sizes greater than `n` are irrelevant. The additional `reverse` after `zipWith` ensures that the combinations are listed in a more natural order. However, if ordering is not a concern, this step can be omitted.

Note that if we define `revInits = map reverse . inits`, where `inits` takes quadratic time, and applying `map reverse` does too, by the accumulation lemma Bird [1987], they can be fused into a single function with linear time complexity using an accumulating parameter

```
revInits :: [a] -> [[a]]
revInits = revInitsFrom [] where
  revInitsFrom ys []     = [ys]
  revInitsFrom ys (x:xs) = ys : revInitsFrom (x:ys) xs
```

Running `revInits [1,2,3]` produces `[[],[1],[2,1],[3,2,1]]`. We will provide a more detailed explanation of list accumulation and Gibbons [1991]' tree accumulation in Section III.3.7.

With the help of `revInits` function, we can redefine the combination algebra more efficiently as follows

```
kcombsAlg' :: Int -> ListFj a [[ [a] ]] -> [[ [a] ]]
kcombsAlg' k Nil = [ [] ] : replicate k []
kcombsAlg' k (Single a) = [ [] ] : [ [a] ] : replicate (k-1) []
```

[14] Due to the use of `zipWith`, the convolution is only well-defined for lists of the same length. Since we are generating `k`-combinations using a catamorphism, cases where the lists have different lengths do not need to be processed.

```
kcombsAlg' k (Join xss yss) = [concat (reverse (zipWith crj xss (reverse
   yss'))) | yss' <- tail (inits yss) ]
```

Analogues to the design of the K-combination generator, the K-permutation generator is nothing more than the *integration of K-combinations and permutations*. Therefore, to design a K-permutations generator, replace the `crj` operator that is used in `kcombsAlg` with a `crm` operator (short for "cross merge")

```
kpermsAlg :: Int -> ListFj a [[ [a] ]] -> [[ [a] ]]
kpermsAlg k Nil = [ [] ] : replicate k []
kpermsAlg k (Single a) = [ [] ] : [ [a] ] : replicate (k-1) []
kpermsAlg k (Join xss yss) = [concat (reverse (zipWith crm xss (reverse
   yss'))) | yss' <- tail (inits yss) ]
```

II.2.3.4 Building complex combinatorial structures from the simpler ones

In this thesis we identify three fusion laws that can aid in constructing efficient combinatorial generators for complex combinatorial structures. These three fusion laws are called *filter fusion, product fusion*, and *cross product fusion*.

In particular, a specific instance of cross-product fusion is *Cartesian product fusion*, which is highly useful in solving machine learning problems where there is a need to construct an efficient generator for enumerating the Cartesian product of two or more basic combinatorial structures.

Another application of the Cartesian product fusion law enables the evaluation of objective function values of configurations during the recursive generation process, this is known as *evaluation fusion*. In the combinatorial optimization literature, evaluation fusion is often employed by default without undergoing verification for correctness. The Cartesian product fusion law formalizes this.

Filter fusion Filtering is used above in constructing the K-sublists algebra, where configurations that do not satisfy the predicate `p = (<= k) . length` are filtered out. In principle, any desired predicate can be used to select configurations from the set of all candidate configurations in the search space $\mathcal{S}$.

However, if the goal is to create an efficient combinatorial generators, it is crucial to identify partial configurations that will not satisfy the predicate before extending them to complete configurations. Eliminating these infeasible configurations early on avoids wasting computational resources. This technique is known as *filter fusion*. In other words, we have

```
filter p .  cata alg = cata (alg_filt p)
```

where `alg_filt = filter p .  alg`, if predicate `p` is prefix-closed (`p (x ++ y)`$\Longrightarrow$ `p x`) for cons-list algebra and segment-closed (`p (x ++ y)` $\Longrightarrow$ `p x && p y`) for join-list algebra.

Product fusion (banana-split law) Consider a list of numbers for which the mean of these numbers is required. The mean of a list of numbers can be obtained by calculating the sum of the list and then dividing the sum by the length of the list. In Haskell, the sum of a list of numbers is obtained by the `sum` operator. Equivalently, it can be implemented as a catamorphism

```
sumalg :: Num a => ListFr a a -> a
sumalg Nil = 0
sumalg (Cons a acc) = a + acc

sum' :: Num a => [a] -> a
sum' = cata sumalg
```

Similarly, the built-in function `length` can also be defined by a catamorphism, rendered as

```
lenalg :: ListFr a Int -> Int
lenalg Nil = 0
lenalg (Cons a acc) = 1 + acc

length' :: [a] -> Int
length' = cata lenalg
```

Given above two functions, the mean is computed using,

```
average = uncurry div . (pair sum' length')
```

where the function `uncurry div` calculates the division of a pair of values.

This naive implementation `average` traverse the input list twice, once for catamorphism `sum'`, once for catamorphism `length'`. It is possible, however, to merge the two separate catamorphisms together to form a single catamorphism. An obvious question to ask is, can we merge any two catamorphisms? The answer is given in the lemma below.

Lemma 2. *Product fusion law.* Given any two algebras `alg1 :: Functor func =>func a -> a` and `alg2 :: Functor func =>func b -> b` with the same base functor `func`, the following equality holds

```
pair (cata alg1) (cata alg2) = cata (prodalg alg1 alg2),
```
(46)

where `prodalg` is defined as

```
prodalg alg1 alg2 = pair (alg1 . (fmap fst)) (alg2 . (fmap snd))
```

.

Proof. The proof is given in Bird and De Moor [1996], chapter 3. □

In the study of constructive algorithmics, this is technique known as the *banana-split law*. The name banana-split is because of the squiggly notation for catamorphisms. In literature, the symbol (|`alg`|) is often used represent catamorphism `cata alg`, the brackets (| |) look like bananas.

Applying product fusion to the above example, obtains the following average function by calling a single catamorphism

```
average_prodalg = uncurry div . (cata (prodalg sumalg lenalg))
```

The algebra `prodalg sumalg lenalg` can also be defined explicitly as

```
sumlenalg :: Num a => ListFr a (a, Int) -> (a, Int)
sumlenalg Nil = (0,0)
sumlenalg (Cons a (b,n)) = (a + b, n + 1)
```

and evaluating `uncurry div . (cata sumlenalg)` is equivalent to evaluating `uncurry div . (cata (prodalg sumalg lenalg))`.

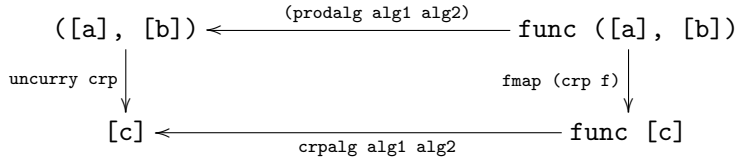
In the context of combinatorial generation, to generate combinatorial configurations for two (or more) different combinatorial structures, product fusion generates them by calling a single catamorphism. For instance, to generate all sublists and permutations of a given list `xs = [1,2]`, evaluating `cata (prodalg subsalg permsalg) xs` gives `([[],[2],[1],[1,2]],[[1,2],[2,1]])`, the first element `[[],[2],[1],[1,2]]` represents all sublists for list `xs`, and the second element `[[1,2],[2,1]]` represents are all permutations for list `xs`.

Cross product fusion The above two primitive constructions—filter fusion and product fusion—may seem limited in certain situations. In many cases, building more complex combinatorial structures demands fusing operations that go beyond simple filtering or pairing. It is worth exploring how to fuse the cross product of any two catamorphism generators. The solution is given below.

Lemma 3. *Cross product fusion law.* Given any two algebras `alg1 :: func a -> a` and `alg2 :: func b -> b` and the bijective function `f :: a -> b -> c`. Then the *cross product algebra* should satisfies the following fusion condition

```
uncurry (crp f) . (prodalg alg1 alg2) = (crpalg alg1 alg2) . fmap (crp f),      (47)
```

In other words, the following diagram commutes



Proof. The cross product algebra `crpalg` can be derived as follows

```
  uncurry (crp f) . (pair (cata alg1) (cata alg2)) = cata (crpalg alg1 alg2)
≡ product fusion law
  uncurry (crp f) . cata (prodalg alg1 alg2) = cata (crpalg alg1 alg2)
≡ fusion law (42)
  uncurry (crp f) . (prodalg alg1 alg2) = (crpalg alg1 alg2) . fmap (crp f)
```

□

Cartesian product fusion Unfortunately, the cross-product fusion above does not directly provide a concrete definition for the cross-product algebra. This is because `crp f` is generally not invertible, even if `f` is invertible. However, under certain strict conditions—such as assuming that the input to the cross product contains no duplicate elements—we can derive a concrete implementation.

The following is an analysis of cross product fusion in the simplest case: *Cartesian product fusion*, for which we are able to give a concrete definition.

By observing the construction order in the Haskell list comprehension, we present an ad-hoc implementation for the inverse of the Cartesian product as follows

```
invcp :: Eq a => [(a, a)] -> ([a], [a])
invcp xs = (nub (map fst xs), nub (map snd xs))

nub :: Eq a => [a] -> [a]
nub [] = []
nub (x:xs) = x : nub (filter (/= x) xs)
```

This definition is heuristic, relying on the construction order in Haskell list comprehensions. In other languages, the function `invcp` may require a different definition. Moreover, the use of a bijective function `f` does not guarantee that `crp f` is bijective. For instance, when `f = (,)` (i.e., the Cartesian product), we cannot always recover the original two lists from their Cartesian product if there are duplicates.

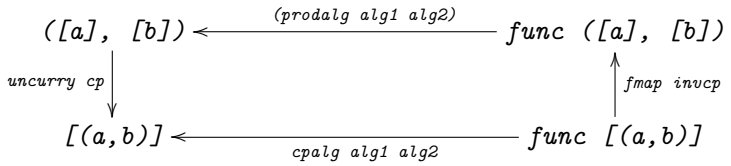
For example, given `cp [1,1] [2,2] = [(1,2),(1,2),(1,2),(1,2)]`, applying `invcrp` to `[(1,2),(1,2),(1,2),(1,2)]` returns `([1],[2])`, which does not fully reconstruct the original lists. However, we can still safely use this approach, since we only apply it within a catamorphism. By ensuring that input elements are distinct, we eliminate the possibility of duplicate elements within a list or across two lists.

Then we have following Lemma.

Lemma 4. *Cartesian product fusion law.* Assuming the input of a catamorphism contains no duplicates and non-empty. Given any two algebras `alg1 :: func a -> a` and `alg2 :: func b -> b` and the bijective function
`f :: a -> b -> c`. Then the following equality holds:

$$\texttt{uncurry (cp f) . (pair (cata alg1) (cata alg2)) = cata (cpalg alg1 alg2)}, \tag{48}$$

where
`cpalg alg1 alg2 = uncurry (cp f) . pair (alg1.(fmap fst)) (alg2.(fmap snd))) . (fmap (invcp f))`,
is called the *Cartesian product fusion algebra.* In other words, the following diagram commutes



Intuitively, the above diagram means that applying `crpalg alg1 alg2` to `[(a,b)]` is equivalent to first recovering `func [([a],[b])]` from their Cartesian product `([a],[b])`, then calculating the result of `alg1 (func [a])` and `alg2 (func [b])` separately by applying `prodalg alg1 alg2` to `func ([a], [b])`. Finally, apply the Cartesian product operator again to `([a],[b])` to obtain the updated `[c]`.

Proof. The Cartesian product algebra `cpalg` can be derived as follows

```
  uncurry (cp f) . (pair (cata alg1) (cata alg2)) = cata (cpalg alg1 alg2)
```

≡ cross product fusion law

```
  uncurry (cp f) . (prodalg alg1 alg2) = (cpalg alg1 alg2) . fmap (cp f)
```

≡ definition of `invcp f`

```
  (cpalg alg1 alg2) = uncurry (cp f) . (prodalg alg1 alg2). fmap (invcp f)
```

The function `invcp :: (a -> b -> c) -> [c] -> ([a], [b])` is the inverse of of the cross product function `cp`, it receives a function `f`, and a list `[c]`, which is the cross product of the pair `([a],[b])`, and returns a pair lists `([a],[b])`. □

In combinatorial optimization problems, there is often the need to evaluate the objective value for each configuration incrementally in a recursive program. This requires "tupling" each configuration with the input data sequence `dataseqn` to make evaluations feasible. In other words, there is the need to compute `cp dataseqn (gen dataseqn)`, where `gen dataseqn` generates all possible configurations of the problem.

However, computing `cp dataseqn . gen dataseqn` directly is inefficient, as the generator `gen dataseqn` may produce an exponential number of configurations. If the generator `gen` is defined by a catamorphism, and since the sequence can also be generated by a catamorphism, then Cartesian product fusion law can be applied to fuse these two catamorphism generators. This allows the application of the evaluation function `eval` to each configuration during the recursive generation process, as every configuration is now tupled with the data sequence. Therefore, evaluation is always feasible when using a catamorphism generator. For instance, evaluating `cata (cpalg subsalg seqnalg) [1,2]`, gives `[([],[1,2]),([2],[1,2]),([1],[1,2]),([1,2],[1,2])]`.

Similarly, evaluating `cata (cpalg subsalg permsalg) [1,2]` gives `[([],[1,2]),([],[2,1]),([2],[1,2]),([2],[2,1]),([1],[1,2]),([1],[2,1]),([1,2],[1,2]),([1,2],[2,1])]` the Cartesian product for all sublists of `[1,2]` and all permutations of `[1,2]`.

Furthermore, the `cpalg` operator can be implemented more efficiently as

```
unit x = [x]
pairlist (f, g) = (uncurry cp) . pair (f.(fmap (unit.fst))) (g.(fmap
   (unit.snd)))

cpalg' alg1 alg2 = alg where
    alg Nil          = pairlist (alg1, alg2) Nil
    alg (Single a)   = pairlist (alg1, alg2) (Single a)
    alg (Join x y)   = concat (cpf (pairlist (alg1, alg2).(uncurry Join))
       x y)
```

where `cpf = [f (a, b) | a <- x, b <- y]`, which is the cross product applied to an *uncurried* function.

The efficiency improvement for the new Cartesian product fusion algebra arises because reversing a single configuration, which is just a pair in a list of the Cartesian product pairs, is more efficient than

reversing the results of Cartesian product by using `invcp`. The equivalence between `cpalg'` and `cpalg` will be discussed in later section after introducing (52) and the foundations of relational algebra theory in Section II.2.5.

Time-space trade-off in fused cross product generators From the above examples, it can be seen that using cross product fusion enables the easy construction of efficient, complex combinatorial generators. Computational efficiency is significantly improved by the fusion law. For instance, if one combinatorial structure has M objects and another has L objects, where M and L are usually exponentially large, constructing two generators separately and then calculating their cross product will require at least $O(M + L + ML)$ operations. In contrast, a fused program using the cross product fusion law requires only $O(ML)$ operations.

The main disadvantage of using a fused cross product generator is that it often consumes $O(ML)$ space as well, which is memory-intensive for most combinatorial objects. Therefore, constructing the cross product of two combinatorial generators must carefully navigate the time-space trade-off. There are two scenarios to consider: if a filtering process is applied to the cross product of configurations, a significant proportion of configurations can be eliminated during the recursive generation process when constructing a fused cross product generator. By contrast, if no filtering process can be applied to the cross product configurations, it is better to run two generators separately, as two separate generators only consume $O(M + N)$ space, whereas a fused generator requires $O(ML)$ space. Therefore, identifying whether a filtering process can be applied to the fused configurations is crucial in deciding whether to use a fused cross product generator or to run two generators separately and then compute their cross product.

II.2.4 Structured recursion schemes

As introduced in Subsection I.2.3.3, there exists a "zoo" of recursive morphisms, each preserving certain properties that ordinary catamorphisms do not. Due to limited space, and because some of these morphisms require extensive knowledge of category theory to explain properly, it is infeasible to introduce all of them in this thesis. Instead, the thesis will focus on two of them that are most relevant to the contributions therein, namely *anamorphisms* and *hylomorphisms*.

II.2.4.1 Anamorphisms

Let $\mathbf{F}$ be an endofunctor from $\mathbf{F} : \mathcal{C} \to \mathcal{C}$. Dual to $\mathbf{F}$-algebras $alg : \mathbf{F}a \to a$ (`alg :: func a -> a` in Haskell), is an $\mathbf{F}$-*coalgebra* with type $coalg : a \to \mathbf{F}a$ (`coalg :: a -> func a` in Haskell). In other words, the $\mathbf{F}$-coalgebra wraps an object A in the context of $\mathbf{F}$. One example of $\mathbf{F}$-coalgebra occurs in the above discussion—the terminal coalgebra $out : \mu\mathbf{F} \to \mathbf{F}\mu\mathbf{F}$ (`out :: Mu func -> func (Mu func)` in Haskell).

Analogous to homomorphisms in $\mathbf{F}$-algebras, homomorphisms between two $\mathbf{F}$-coalgebras `coalg1 :: a -> func a` and `coalg2 :: b -> func b`, are defined by a morphism between their carrier `h :: a -> b` such that `coalg1 . h = (fmap h) . coalg2`.

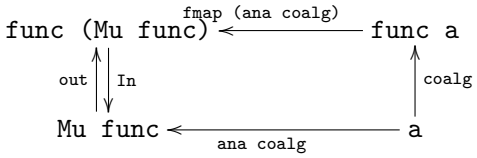
The catamorphism is a homomorphism from the initial $\mathbf{F}$-algebra to another $\mathbf{F}$-algebra, an $\mathbf{F}$-algebra `alg :: func a -> a`, can be thought as an *evaluation* step, which turns a data structure into a single value, just like a monoid operation that turns two values with the same type to a single value. Dually, a

F-coalgebra `coalg :: a -> func a` can be thought of as generating a data structure from a seed. The idea of a coalgebra is that a seed is used to create a *single level* of a recursive data structure. This single-level data structure is precisely described by a functor **F**, and the *co-recursive* data structures related to **F**-coalgebras are modeled by *the greatest fixed point*. Categorically, the greatest fixed point is denoted as $\nu\mathbf{F}$. In Haskell (or any SCPO category more generally), the least fixed point $\mu\mathbf{F}$ and the greatest fixed point $\nu\mathbf{F}$ are isomorphic to each other [Hinze, 2013]. To avoid confusion, we will use $\mu\mathbf{F}$ in the math-style formula (and `Mu` in the Haskell function) to denote both the least fixed point and the greatest fixed point.

Similar to the catamorphism case, **F**-coalgebras and homomorphisms between them form a category **CoAlg**(**F**). An **F**-coalgebra is said to be a terminal **F**-coalgebra if it is a *terminal object* in category **CoAlg**(**F**), denoted by $out : \mathbf{F}\mu\mathbf{F} \to \mu\mathbf{F}$. In Haskell it has a type `out :: Mu func -> func (Mu func)` (greatest fixed point $\nu\mathbf{F}$ is replaced with the least fixed point $\mu\mathbf{F}$ for consistency). Dual to the algebra case, the fixed point for the terminal **F**-coalgebra will correspond to co-recursive/co-inductive datatypes. Because the terminal **F**-coalgebra is the terminal object in category **CoAlg**(**F**), terminality implies that there exists a unique morphism from other coalgebras in **CoAlg**(**F**) to terminal **F**-coalgebra `out`. This unique homomorphism, called an *anamorphism*, is characterized by the universal property `out . (ana coalg) = fmap (ana coalg) . coalg`, because `out` has an inverse `In`, induced by the isomorphism between `Mu func` and `func (Mu func)`. We can characterize the anamorphism as

```
ana coalg = In . fmap (ana coalg) . coalg,
```
(49)

where the type information is summarized in the following diagram



Similar to the catamorphism reflection law, it is easy to verify the anamorphism reflection law, `ana out = id`.

In Haskell, anamorphisms can be implemented as follows:

```
ana :: Functor f =>  (a -> f a) -> a -> Mu f
ana coalg = In . fmap (ana coalg) . coalg
```

Perhaps the simplest co-recursive datatype is the `Stream` datatype, i.e. the *infinite list*. It is defined similarly to the finite list, but the nullary list constructor `Nil` is omitted. The `Nil` constructor, which enables recursion termination, is absent. Hence, streams will never terminate. Categorically, the stream functor is defined as $\mathbf{F}_\mathbf{A} = \mathbf{A} \times \mathbf{id}$. In Haskell, we can define stream functor as

```
data StreamF a x = Cons a x deriving (Functor, Show)
```

Because the least and the greatest fixed points coincide in Haskell, the recursive definition `Stream` can be defined by taking the fixed point of the `StreamF a` functor, then the `Stream` datatype can be defined by the following type synonyms

```
type Stream a = Mu (StreamF a)
```

which is equivalent to

```
data Stream a = StreamF {hd::a, tl::(Stream a)} deriving Show
```

where the stream consists of two parts: the *head*, denoted by an element of type `a`, and the tail, represented by another `Stream a`, which is infinitely long.

A simple example anamorphism is the stream generator. In Haskell, it is possible to generate an infinite list starting with `n` using the syntax `[n..]`. A similar function `intsFrom` generates a `Stream` of integers starting from `n`:

```
intfromalg :: Int -> (StreamF Int) Int
intfromalg a = Cons a (a+1)

intsFrom :: Int -> Stream Int
intsFrom n = ana intfromalg n
```

In order to print a stream, convert a `Stream` to an (infinite) list by using helper function

```
toList :: Stream a -> [a]
toList (In (Cons x xs)) = x : toList xs
```

so that when evaluating `toList (intsFrom 3)` (lazily) generates an infinite list `[3,4,5,6,7..]`.

II.2.4.2 Hylomorphisms

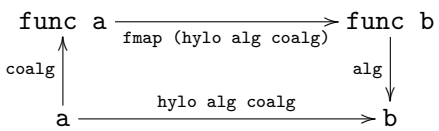
Very powerful and generic recursive morphism*, hylomorphisms* is, subsume *almost all* practical recursions, including both *structured* and *generative* recursions, and almost all structured recursive schemes are special cases of hylomorphisms.

The definition of hylomorphism is straightforward taking into consideration what anamorphisms and catamorphisms are. Indeed, a hylomorphism is simply a composition of a catamorphism with an anamorphism
`hylo alg coalg = (cata alg) . (ana coalg)`. In this definition, it is easy to verify that both catamorphism `cata alg` and anamorphism `ana coalg` is a special case of hylomorphism, because the catamorphism and anamorphism reflection law `cata In = ana out = id`.

A hylomorphism can be also characterized as a least fixed point [Bird and De Moor, 1996]. Given a coalgebra `coalg :: a -> func a` and an algebra `alg :: func b -> b`, a hylomorphism in Haskell is

```
hylo :: Functor func => (func b -> b) -> (a -> func a) -> (a -> b)
hylo alg coalg = alg . fmap (hylo alg coalg) . coalg
```

The hylomorphism is a morphism `hylo alg coalg :: a -> b` that maps a coalgebra `coalg :: a -> func a` to an algebra `alg :: func b -> b` such that



commutes, where functor `func` is the base functor **F** of hylomorphism.

The hylomorphism commutative diagram, as depicted in the above diagram, closely resembles that of the catamorphism. Indeed, hylomorphism can be understood as a *generalized form of catamorphism* in the sense that the terminal algebra `out` in the catamorphism is substituted with a **F**-coalgebra. This substitution affords an extensive degree of flexibility in program construction compared to catamorphisms, where the **F**-coalgebra are restricted to be the terminal algebra of the base functor.

However, the flexibility afforded by hylomorphism comes at a cost. Hylomorphism may not have a unique solution. Even when both `coalg` and algebra `alg` are total functions, the hylomorphism `hylo alg coalg` may not be total. By contrast, `cata alg` and `ana coalg` are always total if `alg` and `coalg` are. Indeed the hylomorphism `hylo alg coalg` has a unique solution if and only if the `coalg` is a *recursive coalgebra* Capretta et al. [2006]. Section II.2.4.4, clarifies the conditions under which coalgebras become recursive coalgebras. Hinze et al. [2015] propose a simple toolbox for constructing recursive coalgebras, which, by definition, guarantee that hylomorphisms have unique solutions, regardless of the algebra.

Hylomorphisms can be understood as a recursive process that allows for *arbitrary decomposition* of a problem. This definition of hylomorphisms clearly subsumes the classical definitions of dynamic programming (DP) and divide-and-conquer (D&C) algorithms. In particular, DP algorithm is is a degenerate case of the hylomorphism, as the original DP definition is grounded in SDP, corresponding to a *sequential decomposition* of the problem, consuming one data point with each recursive call. Similarly, this definition is also more general than the classical definition of the D&C method; the differences will be elaborated upon in the next subsection. Thus, this thesis refers to hylomorphisms as *generalized divide-and-conquer* (D&C) recursions.

II.2.4.3 Hylomorphisms and divide-and-conquer algorithms

The definition of the generalized D&C method given above differs from the classical definitions found in typical algorithm design textbooks [Kleinberg and Tardos, 2006, Cormen et al., 2022]. Traditionally, ordinary D&C algorithms are characterized by recursively decomposing problems into two equal halves, such that the merged solutions of disjoint subsets are also disjoint—that is, they do not share subsolutions. This disjoint property is often omitted in classical algorithm design books, yet it is an implicit feature present in all D&C algorithms. This property is made explicit here, as it is a defining feature distinguishing D&C from dynamic programming. Classical examples include the *mergesort* and *quicksort* algorithms. By using this binary subdivision strategy, D&C typically achieves a speed-up from $O\left(N^2\right)$ to $O\left(Nlog\left(N\right)\right)$.

This section further explores the connection between hylomorphisms and the divide-and-conquer method by analyzing two classical divide-and-conquer algorithms, *mergesort* and the *quicksort*, and characterizing them in terms of hylomorphisms.

Mergesort algorithm The *mergesort algorithm* is often considered the archetypal D&C algorithm, for sorting a list of elements into a specified order. Mergesort comprises two fundamental steps:

1. **Recursive splitting**: Divide the unsorted list recursively into two equal halves until each sublist consists of singleton lists.

2. **Recursive merging**: Merge the sublists recursively to generate new sorted sublists until only one sublist remains.

In Haskell, the mergesort algorithm can be implemented as

```
mergeSort' :: Ord a => [a] -> [a]
mergeSort' [] = []
mergeSort' [a] = [a]
mergeSort' x = merge (mergeSort' l) (mergeSort' r)
 where (l, r) = splitAt (length x `div` 2) x
```

where the function `splitAt (length x `div` 2)` splits the list `x` into two equal halves, and the `merge` operation is defined as

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] m = m
merge n [] = n
merge (x:xs) (y:ys)
    | x <= y    = x : merge xs (y:ys)
    | otherwise = y : merge (x:xs) ys
```

The mergesort algorithm has worst-case time complexity $O(N \log N)$ for a length N list. The definition of the mergesort algorithm matches on three patterns, the empty list, the singleton list and a third pattern that recursively splits the input list `x` into two parts, `l` and `r`. This immediately brings to mind the join-list datatype, as the definition of `mergeSort'` has three constructors. Indeed, the join-list functor `ListFj a` can serve as the base functor for the mergesort algorithm using a hylomorphism.

The `ListFj a`-coalgebra of the hylomorphism corresponds to a one-step decomposition process. Subdivide the unsorted list into two equal halves and store them in the intermediate datatype,

```
split :: [a] -> (ListFj a [a])
split []  = Nil
split [x] = Single x
split xs  = Join l r where
  (l, r) = splitAt (length xs `div` 2) xs
```

so that the two partitioned lists `l` and `r` are stored in the value constructor `Join` of the join-list functor, while the singletons and empties are stored in their corresponding value constructors.

The `ListFj a`-algebra for the hylomorphism corresponds to the recombination stage in the recursion which is implemented by

```
mcombine :: Ord a => ListFj a [a] -> [a]
mcombine Nil = []
mcombine (Single x) = [x]
mcombine (Join l r) = merge l r
```

where the third pattern of `ListFj a`-algebra matches to the to value `Join l r`, representing the operation that merges two ordered list `l` and `r`.

Combining the `ListFj a`-coalgebra and `ListFj a`-algebra above, we can redefine the mergesort algorithm as

```
mergeSort :: Ord a => [a] -> [a]
mergeSort = hylo mcombine split
```

It is straightforward to observe how the structure of computing mergesort is manifested in the definition of `mergeSort` in terms of the hylomorphism. The problem is initially decomposed by `ListFj a`-coalgebra `split`, then solved by `hylo`, and finally, the solutions are recombined by the `ListFj a`-algebra `mcombine`.

Quicksort algorithm The *quicksort algorithm* shares a nearly identical decomposition structure with mergesort and is widely understood as a classical D&C algorithm. The quicksort algorithm consists of three essential steps:

1. **Pivot selection and recursive partitioning**: Select an arbitrary pivot element and partition the unordered list into two sublists. One sublist contains elements smaller than the pivot element, while the other sublist contains elements greater than or equal to the pivot element. Repeatedly apply this process to each new sublist until the new sublists are ordered.

2. **Concatenation**: Concatenate all ordered sublists together to obtain the final ordered list solution.

In Haskell, `quickSort'` algorithm can be implemented as

```
quickSort' :: Ord a => [a] -> [a]
quickSort' [] = []
quickSort' (a:x) = (quickSort' l) ++ [a] ++ (quickSort' r)
 where
  l =[b | b <- x,b < a]
  r =[b | b <- x,b >= a]
```

In the quicksort algorithm, a pivot element is selected from the input list and the remaining elements are partitioned into two sublists. One way to understand this partition is that the pivot element becomes the root node of the tree, and there are two associated subtrees connected to the root node. This naturally creates a subdivision that is similar to the binary tree datatype that defined in the Section II.2.2.5, where the second value constructor `Node x a x` has the same structure as the partition process of the quicksort algorithm.

Indeed, the base functor for the hylomorphism implementation of the quicksort algorithm is precisely the binary tree functor `BtreeF a`. The `BtreeF a`-coalgebra corresponds to the subdivision stage, which can be defined as

```
partition :: Ord a => [a] -> BtreeF a [a]
partition [] = Empty
partition (a:x) = Node [b | b <- x, b < a] a [b | b <- x, b >= a]
```

At the same time, since the combining step in the quicksort algorithm is just the list concatenation operation. The `BtreeF a`-algebra for quicksort algorithm can be defined as

```
qcombine :: BtreeF a [a] -> [a]
qcombine Empty =[]
qcombine (Node l a r) = l ++ [a] ++ r
```

Combining the `BtreeF a`-algebra and the `BtreeF a`-coalgebra together, obtains the hylomorphism implementation for the quicksort algorithm is rendered as

```
qucikSort :: Ord a => [a] -> [a]
qucikSort = hylo qcombine partition
```

II.2.4.4 Recursive coalgebras

Hylomorphisms can be understood as a three phrase program. In the first phase, a problem is divided into sub-problems by the **F**-coalgebra, where **F** is the base functor. In the second phase, sub-problems are solved recursively and independently to form sub-solutions. In the final phase, these sub-solutions are combined together to form the complete solutions of the original problem.

The recursive structure of a hylomorphism is determined by the base functor **F**. Depending on the shape of **F**, hylomorphisms can capture various constructs such as while-loops, and divide-and-conquer schemes. In fact, *most* practical programs can be formulated as hylomorphisms [Hu et al., 1996]. An counter example would be *nested* recursions, such as *Ackermann* recursion.

As mentioned earlier, the generality of hylomorphisms can lead to solutions that are either non-unique or not well-defined. To ensure the well-definedness of a hylomorphism `hylo alg coalg`, one can either concentrate on devising *recursive coalgebras* to guarantee unique solutions for any *algebra*s, or alternatively, explore *co-recursive algebras* to ensure uniqueness for any *coalgebras*.

Definition 13. *Recursive coalgebras.* Given a base functor `func`, a coalgebra `coalg :: a -> func a` is called *recursive* if for every algebra `alg :: func b -> b` there is a unique hylomorphism `hylo alg coalg :: a -> b`.

The existence of recursive coalgebras provides another kind of initiality, and the category of recursive coalgebras $rec\,(\mathbf{F})$ forms a subcategory of the category of coalgebras $coalg\,(\mathbf{F})$. The notion of recursive coalgebras generalizes terminal algebras insofar as structured recursion is concerned. In many applications, we need more than initial algebras to solve a problem, as we can see in the above mergesort and quicksort algorithms where both `split` and `partition` are not terminal coalgebras. The position taken here is that the recursive coalgebras are a more useful tool in the study of structured recursion than initiality, and that most results for structured recursion and initial algebras can be recast in a clearer way in this more general framework.

Although Hinze et al. [2015] proposed a toolbox for assembling recursive coalgebras by identifying the *conjugate rule*, a full explanation for their theory involves many highly abstract categorical ideas such as *adjunctions* and *conjugates*, which are difficult to explain or give a concrete Haskell example in the limited space available here. Nevertheless, we found that the two propositions for constructing recursive coalgebras, presented by Capretta et al. [2006], are easier to grasp.

Proposition 1. If an endofunctor **F** has an initial algebra `in`, the inverse of the initial algebra is a terminal recursive coalgebra, for instance, for catamorphisms the terminal algebra `out` is also a recursive coalgebra.

Recall that an **F**-coalgebra is said to be a terminal **F**-coalgebra if it is a *terminal object* in category **CoAlg**(**F**), denoted by $out : \mu\mathbf{F} \to \mathbf{F}\mu\mathbf{F}$. This observation is easily comprehensible given the previous exposition of the fact that catamorphisms possess unique solutions. This aspect provides a clearer explanation as to why initiality guarantees the uniqueness of the catamorphism. Another useful proposition is presented below.

Proposition 2. If `m :: b -> a` is a *split monic* coalgebra morphism from a `coalg :: b -> func b` to a recursive coalgebra `rcoalg :: a -> func a`, then `coalg` is also a recursive coalgebra. A split morphism is defined as a morphism with a left-inverse.

For further details and more complicated constructions of recursive coalgebras, refer to Capretta et al. [2006].

II.2.5 Foundations for the algebra of programming

As mentioned earlier, the theory of the algebra of programming is a generalization of the Bird-Meertens formalism from *total functions* to *relations*. Before going into details of the theory, a few foundational concepts for understanding the algebra of programming theory are illustrated.

II.2.5.1 Motivations for using relational algebra

The classical Bird-Meertens formalism, only involves total functions [Bird, 1987, Meertens, 1986]. However, when it comes to program derivation, generalizing total functions to relations appears to be indispensable, offering significant advantages in the following aspects:

1. **A model of nondeterminism:** In many cases of program derivation, non-deterministic functions are required, and relying solely on total functions is too restrictive. For example, in optimization problems, there is often more than one optimal solution. Modeling this nondeterminism using relations is much simpler than using set-valued total functions.

2. **Ease of structuring certain proofs**: There are deterministic programming problems (functions) where it is helpful to consider non-deterministic programs (relations) in passing from specification to implementations. While not all functions have an *inverse*, *all* relations have a *converse*.

3. **Necessity in defining specifications**: All programs are *total functions* [Hoare and He, 1987], but programs are only a *proper subset* of specifications. This is because certain relations necessary for defining a specification cannot be implemented as computable programs. For instance, the complement relation is a valid and reasonable relation for specifications but is not executable. Similarly, in our context, sublist and permutation generators are total functions, but determining whether a list is a sublist or permutation of another list, is a relation[15].

II.2.5.2 Definition of relation

A relation can be interpreted as a *Boolean-valued function* or a *non-deterministic mapping*. Unlike functions, every relation has a *converse*, denoted by the superscript circle $^\circ$. For instance, the converse of

[15] A relation can be viewed as a binary Boolean-valued function.

relation `r a b` can be defined as `r° a b = r b a`.

A Boolean-valued function in Haskell can be defined as

```
type Rel a b = a -> b -> Bool
```

Compared with defining a relation as a non-deterministic mapping, interpreting a relation as a Boolean-valued function has a significant shortcomings in implementation: a Boolean-valued function cannot capture non-deterministic behavior. Specifically, there is the need to give an input `a` to relation `r`, and `r a` will return all `b`'s such that `r a b == True`.

It is not possible to define a non-deterministic function in Haskell. Instead, when necessary, non-deterministic functions will be represented using *pseudo-Haskell code*. The non-deterministic outputs of a relation are expressed using logical `or`. For instance, `r a = b or c` means relation `r` can output `b` or `c`. Moreover, certain relations, such as $\leq$, cannot be fully defined even using this pseudo-Haskell code style. Thus, whenever we define a relation in terms of a Boolean-valued function, we implicitly define its corresponding non-deterministic functions as well. The choice of definition should depend on the context, particularly the type of the function.

Two relations `r :: Rel a b` and `s :: Rel a b` with the same input and output type can be compared, the *inclusion relation* is analogous to set inclusion; thus,

$$\mathtt{r} \subseteq \mathtt{s} \iff (\forall \mathtt{a}, \mathtt{b} : \mathtt{r\ a\ b} \implies \mathtt{s\ a\ b})\,. \tag{50}$$

A *preorder* is a *reflexive* (`r a a` for all `a`) and *transitive* (`r a b && r b c` implies `r a c` for all `a`) relation. A *partial order* is a preorder with the *anti-symmetric* property (`r a b && r b a` implies `a = b`). A partial order is a *linear/total order* if `r a b` or `r b a` exists for all `a` and `b`.

For illustration, the well-known lesser, greater and equal relation can be defined as

```
leq :: Ord a => Rel a a
leq a b = a <= b

geq :: Ord a => Rel a a
geq a b = a >= b

eq :: Ord a => Rel a a
eq a b = a == b
```

II.2.5.3 Reformulate the combinatorial optimization problem specification

As explained in Section II.2.3, any combinatorial optimization problem can be specified through the exhaustive search paradigm, which can be re-formulated in Haskell as

```
sel r . filter p . (map eval) . (cpr dataseqn) . gen
```

where `cpr a x = [(a,b)| b <- x]`. There is an additional operation `cp dataseqn :: [Comb] -> [(Comb, Seqn)]`, the Cartesian product `cp dataseqn` tuples every combinatorial configuration with the input data sequence `dataseqn`. This process makes evaluation fusion feasible, so as to evaluate the objective function for each

tuple with respect to a sequence of data `dataseqn` by applying `eval :: (Comb, Seqn) -> Config` to each tuple in `[(Comb, Seqn)]`.

Typically, the generator `gen` is expressed as a catamorphism, various combinatorial generators based on the catamorphism in Section II.2.3. If a predicate `p` is prefixed-closed, the filtering and evaluation processes are integrated into the generator, and a combinatorial optimization problem can be specified using catamorphism as

```
sel r . cata alg
```

where algebra `alg` represents the algebra obtained from the fusion of the filtering and evaluation processes.

In more general settings, so as to decompose a problem using a coalgebra `coalg`, the generator `gen` can be specified as a hylomorphism. Consequently, the combinatorial optimization problem can be expressed in a sophisticated form through hylomorphism.

```
sel r . hylo alg coalg
```

As explained in the Section I.2.1 of Part I, a selector is typically used to choose a configuration with minimal objective error (cost). Without loss of generality, the selector can be considered as a selection process with respect to a relation `r`, which is usually defined as a total order of the form $\texttt{cost}^{\circ}\texttt{ . leq . cost}$, where the `cost` function calculates the objective values of a configuration. In Haskell, the selector over a (non-empty) list can be defined as

```
minlist :: (Rel a a) -> [a] -> a
minlist r [a] = a
minlist r (a:xs) = f a (minlist r xs)
  where f a b = if r a b then a else b
```

If we define the relation `r` as `leq`, then `minlist leq` will select the first candidate with the minimum objective value. In this thesis, the preorder used in selector `sel` is denoted by the lowercase letter `r`, referred to as the *selector relation*.

Note that selecting the first element with the smallest objective value is too restrictive for applying equational reasoning to the optimization problem; in most cases, we need only *implication* rather than *equivalence* Bird and Gibbons [2020]. Therefore, to apply equational reasoning, we need to generalize the function `minlist` into a relation `minlistR`, which selects one of the optimal solutions from a list of candidates. However, defining this relation formally would be cumbersome; the use of `minlistR` is simply to extend our powers of specification and will not appear in any final algorithm. Therefore, for the purposes of this thesis, we can safely use `minlist` to derive a program, as long as we remember that `minlist` returns one possible optimal configuration rather than the first.

II.2.5.4 Relational F-algebras

Relational F-algebras/coalgebras Before discussing more general results, there is the need to distinguish the difference between *relational* **F**-*algebras* (`algR`) and *functional* **F**-*algebras* (`alg`). All algebras discussed so far are functional **F**-algebras. Introducing relational **F**-algebras simplifies reasoning about programs. The identity between relational **F**-algebras and functional **F**-algebras was first exploited by Eilenberg and Wright [1967] to reason about the equivalence between deterministic and non-deterministic

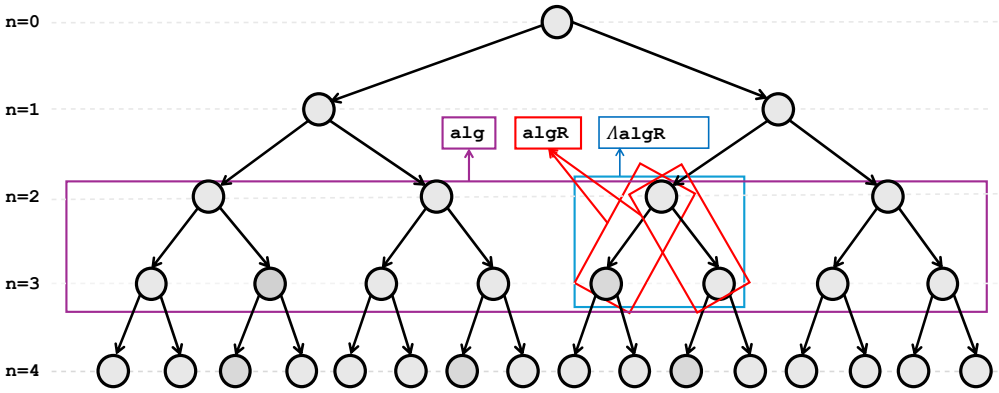


Figure 10: The difference between functional algebra `alg` (purple), relational algebra `algR` (red), and the power transpose of the relational algebra Λ`algR` (blue) in the context of the sequential decision process (catamorphism over cons-list datatype). The arrows in red boxes represent possible mappings of the relational algebra `algR :: func a -> a`. In this case, it has two possibilities (two decision functions). The power transpose of the relational algebra Λ`algR:: func a -> [a]` maps a configuration to all possible choices of `algR` and then stores them in a list. Subsequently, the functional algebra `alg` applies a list of decision functions to a list of partial configurations and stores the updated results.

automata in the context of set theory. For this reason, Bird and De Moor [1996] named the identity between functional **F**-algebras and relational **F**-algebra as the *Eilenberg-Wright lemma*.

In the discussion here, for the ease of understanding, the identity between functional algebra and relational algebra is left implicit, assuming the relational **F**-algebras and functional **F**-algebras are connected to each other by two functions τ and τ^{-1} such that $\tau\,(\texttt{algR}) = \texttt{alg}$ and $\tau^{-1}\,(\texttt{alg}) = \texttt{algR}$.

Definition 14. *Functional* **F***-algebra and relational* **F***-algebra.* Denote the base functor **F** as `func`. Given a functional algebra `alg :: func [a] -> [a]`, its corresponding relational algebra has type `algR :: func a -> a`, such that $\tau\,(\texttt{algR}) = \texttt{alg}$ and $\tau^{-1}\,(\texttt{alg}) = \texttt{algR}$.

Similarly, the functional **F**-coalgebra and relational **F**-coalgebra are given in the same style, as follows.

Definition 15. *Functional* **F***-coalgebra and relational* **F***-coalgebra.* Denote base functor **F** as `func`. Given a functional coalgebra
`coalg :: [a] -> func [a]`, its corresponding relational algebra is defined as `coalgR :: a -> func a` , such that $\xi\,(\texttt{coalgR}) = \texttt{coalg}$ and $\xi^{-1}\,(\texttt{coalg}) = \texttt{coalgR}$.

In the context of SDPs, the distinctions between relational algebra and functional algebra are depicted in Fig. 10. A *relational catamorphism* `cata algR` is a catamorphism based on a relational algebra `algR :: func Config -> Config`. In every recursive step, `algR` updates a single partial configuration `Config` by choosing *one* decision function from a list of possible decision functions. This results in only one partial configuration at each stage of recursion. In contrast, the corresponding functional algebra

`alg :: [Config] -> [Config]` applies *all* decision functions to the partial configurations generated in the previous stage.

Power transpose In the study of relational algebras, there exists a very important operator Λ, called *power transpose*, which takes a relation `r :: a -> b` and returns a function `Λr :: a -> [b]`. We can define it by list comprehension

$$\texttt{Λr a = [b | b <- bs, r a b == True]}, \tag{51}$$

where `bs` denotes a list of all possible elements with type `b`, so `Λr a` returns all `b` such that `r a b == True`. The list comprehension here is defined as a pseudo-Haskell code style, as the relation `r` here is a non-deterministic function. The implementation of the power transpose of a relation `Λr` in Haskell, is denote by `r_pt`.

There is a subtle difference to raise here. Given a relational algebra `algR`, the functional algebra `alg :: func [a] -> [a]` is not the same as `ΛalgR :: func a -> [a]`, despite both being functions. As depicted in Fig. 10,
`alg :: func [a] -> [a]` applies *all* decision functions to *all* configurations generated in the previous stage, whereas `ΛalgR :: func a -> [a]` applies *all* decision functions to *one* configuration generated in the previous stage. Note, because the power transpose Λ cannot be implemented in Haskell in practice, `algR_pt` (short for "relational algebra, power transpose") is used to denote `ΛalgR` , whenever we need to implement the power transpose of a relational algebra in Haskell.

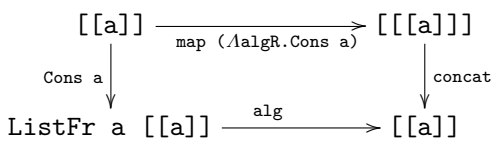
Indeed, these two functions (`alg` and `algR_pt`) are associated by equality

$$\texttt{concat . map (ΛalgR . (Cons a)) = alg . (Cons a)}, \tag{52}$$

over the cons-list catamorphism. Note that, notation `Cons a` has been abused, the data constructor `Cons :: a -> x -> ListFr a x` receives a value `a` and returns a shape `x -> ListFr a x`.

The intuition behind this equality lies in the fact that a functional algebra `alg . (Cons a):: [[a]] -> [[a]]` updates all configurations in a list of configurations `xs :: [a]` by parameterizing `alg :: func [[a]] -> [[a]]` with `Cons a :: x -> ListFr a x`. By contrast, the power transpose of a relational algebra `ΛalgR :: func a -> [a]` can only update a single configuration. To address this, parameterize it with `Cons a :: [a] -> ListFr a [a]` and apply it to all configurations in `xs` using the `map` function. This process has the type
`map (ΛalgR . (Cons a)) :: [a] -> [[a]]`. Finally, apply the `concat` function, which removes the inner brackets and transforms the list of lists into a single list.

In other words, equation 52 is same as the following commutative diagram



The correctness of equality (52) can be generalized to other base functors as well. The focus on the cons-list functor is for ease of explanation. The proof for equality (52) involves several definitions in relational algebras that will not be used elsewhere, so the proof is given in Corollary 6 in the appendix.

Defining a combinatorial optimization problem in terms of relations Earlier, combinatorial optimization problems were specified through hylomorphisms where the algebras and coalgebras are functional. Given a relational algebra `algR :: func a -> a`, and a relational coalgebra `algR :: func a -> a`, a combinatorial optimization problem can be specified through a relational hylomorphism as

$$\texttt{sel r . } \Lambda\texttt{(hylo algR coalgR)}, \tag{53}$$

where Λ`(hylo algR coalgR) = hylo alg coalg`.

Similarly, a combinatorial optimization problem can also be specified through a relational catamorphism as

$$\texttt{sel r . } \Lambda\texttt{(cata algR)}, \tag{54}$$

where Λ`(cata algR) = cata alg`.

II.2.5.5 Monotonic algebras

A monotonic algebra is an abstraction and generalization of Bellman's principle of optimality, and it is one of the most important properties in constructing efficient recursive optimization programs. By definition, an algebra `algR :: func a -> a` is *monotonic on a relation* `rel :: a -> a` if

$$\texttt{algR . (fmap rel)} \subseteq \texttt{rel . algR}. \tag{55}$$

Example 3. *Sequential decision processes.* For combinatorial optimization algorithms based on SDPs (catamorphisms),
`algR :: func Config -> Config` can be considered as a single decision function. The condition of (55) states that if `a` is a predecessor of configuration `x'`, i.e., `x' = algR (Cons a x)`, then `rel x y` implies that `rel x' y'`, where `y' = algR (Cons a y)`. Indeed, in the context of SDP, the monotonicity condition in (55) can be simplified to the following

$$\texttt{rel a b} \subseteq \texttt{rel a' b'}, \tag{56}$$

which agrees with the classical definition of monotonicity in existing literature [Ibaraki, 1977, Karp and Held, 1967]; the abstraction (55) takes a much more generic form.

Example 4. *Universal algebras.* Consider the base functor `Sqr`, which represents the squaring functor. Previously defined was the algebra `plus :: Double -> Double -> Double` (or equivalently `plus :: Sqr Double ->` This algebra is monotonic with respect to relation `leq`, which can be expressed as `plus . (Sqr leq)` $\subseteq$ `leq . plus`. At the point-wise level, this monotonicity can be understood as `c = a + b` $\wedge$ `a` $\leq$ `a'` $\wedge$ `b` $\leq$ `b'` $\implies$ `c` where `+` and $\leq$ are the infix notation for `plus` and `leq`.

When `algR` is a function, denoted by `algRf`, monotonicity has two useful facts.

Fact 1. Given a function `algRf`, then `algRf` is monotonic on `R` if and only if it is monotonic on `R`$^\circ$.

Fact 2. Given a function `algRf`, monotonicity is equivalent to *distributivity.* It is said that `algRf` *distributes over* `r` if

$$\texttt{algRf . sel r} \subseteq \texttt{sel r . alg}, \tag{57}$$

For instance, `plus` distributes over `leq`, which can be interpreted point-wise as
`minlist x + minlist y = minlist [ a + b | a <- x, b <- y ]`, assume `x` and `y` are non-empty lists.

II.2.6 Thinning

The *thinning algorithm* is equivalent to the exploitation of *dominance relations* in the algorithm design literature [Ibaraki, 1977, Galil and Giancarlo, 1989, Eppstein et al., 1992]. The use of thinning or dominance relations is concerned with improving the time complexity of naive dynamic programming algorithms and many other recursive algorithms. In the discussion the thinning algorithm is characterized by parameterizing it with a dominance relation.

The thinning technique exploits the fundamental fact that certain partial configurations are superior to others, and it is a waste of computational resources to extend these non-optimal partial configurations. When employing the thinning algorithm to accelerate a recursive optimization algorithm, two extremes exist. At one end of the spectrum, all non-optimal partial configurations are discarded, leaving only a single partial configuration to be maintained at each recursive stage. This procedure leads to a globally optimal solution if the *greedy condition* holds. This condition characterizes situations where a problem can be solved using a *greedy algorithm.*

At the opposite end, maintaining all possible partial configurations at each stage leads to a brute-force enumeration algorithm. Between the two extremes of one and all, there is a third possibility: in each recursive stage, a collection of representative partial configurations is selected, namely those that might eventually be extended to an optimal solution, while all other partial configurations that cannot lead to optimal solutions are deleted from the candidate list.

Discovering dominance relations is often challenging and requires special insights about a particular optimization problem. This thesis introduces two *generic* dominance relations: the *global upper bound* and the *finite dominance relation.* The first technique is frequently used in many BnB algorithms but has rarely been discussed formally. The latter is entirely novel, although the special case of the finite dominance relation has been used in the literature, Zhang et al. [2023], He and Little [2023a]. However, these applications are neither generic nor formal. These two dominance relations are widely applicable to many machine learning or combinatorial optimization problems where easily accessible approximate algorithms are available and the problem processes only finite data.

Bird and Gibbons [2020] also provide an excellent explanation of thinning algorithms using Haskell. However, their discussion focuses on applying thinning algorithms to solve various problems, whereas our focus is on demonstrating the concepts—how thinning relates to greedy algorithms (Subsection II.2.6.1), the different implementations of thinning algorithms (Subsection II.2.6.2), and two generic dominance relations that are frequently used in practice (Subsection II.2.6.3).

II.2.6.1 What is thinning?

This Subsection characterizes the theorems and corollaries of the thinning technique, formalized by Bird and De Moor [1996]. Following this, it expands upon the discussion regarding the connection between the greedy condition in the *thinning theorem* and the greedy condition in *matroid theory.* The latter is one widely accepted conventional characterization of the greedy algorithm.

Thin-introduction rule Given a dominance relation `domR :: a -> a` (short for *"dominance relation"*), and a candidate list `x :: [a]`, the thinning relation `thin domR :: [a] -> [a]` selects a sublist

y from lists x such that all elements of x have a lower bound under dominance relation domR in y. It is evident from the definition that the dominance relation domR should be a preorder such that domR ⊆ r, otherwise applying the thinning operation is the same as applying sel r directly.

The thinning relation can be introduced into an optimization problem with the following *thin-introduction rule*

$$\texttt{sel r = sel r . thin domR}, \tag{58}$$

This rule states that the optimal configuration is selected by thin domR first, and then min r should consist of the optimal solution selected by min r.

The dominance relation domR in thinning is required to be a preorder, which allows the use of transitivity. If a dominates b, and b dominates c, then a dominates c. In practice, this enables the consideration only of configurations that are "most likely" to dominate others. if a does not dominate c and a dominates b, then it is impossible for b to dominate c, so, comparing b with c is a waste of time.

The thinning theorem

Theorem 1. *Thinning theorem.* Assume a base functor func. If algR :: func a -> a is monotonic on domR°, then the following implication holds:

$$\texttt{sel r . (cata ((thin domR). alg))} \subseteq \texttt{sel r . } \Lambda\texttt{(cata algR)}. \tag{59}$$

Proof. See Bird and De Moor [1996] Section 8.1. □

The thinning theorem 1 states that solutions returned by the thinning algorithm (left-hand side of the inclusion (59)) are also solutions of the brute-force algorithm (right-hand side of the inclusion (59)). It immediately follows from the theorem that the thinning algorithm becomes a brute-force algorithm—the inclusion (59) becomes an identity—if domR == id.The greedy algorithm is then characterized by the following corollary.

Theorem 2. *Greedy theorem.* Given a combinatorial optimization problem specified in the form of (54). If algR is monotonic on r°. Then

$$\texttt{cata (sel r . } \Lambda\texttt{algR))} \subseteq \texttt{sel r . } \Lambda\texttt{(cata algR)}. \tag{60}$$

where cata (sel r . ΛalgR)) is the specification of the greedy algorithm.

Proof. See Bird and De Moor [1996] Section 8.1. □

It is straightforward to observe that Thm. 2 can be regarded as a special case of Thm. 1 when domR = r, according to the thin-introduction rule (58). However, applying this corollary in practice can be challenging, as it necessitates the prior definition of the relational algebra algR. In contrast, the subsequent theorem is generally more straightforward to apply.

Corollary 2. *Greedy theorem variant.* A combinatorial optimization problem specified in the form of (54), if f is monotonic on r°, such that f ⊆ sel r . ΛalgR. Then

$$\texttt{cata f} \subseteq \texttt{sel r . } \Lambda\texttt{(cata algR)} \tag{61}$$

In the context of SDPs, the function `f` in Corollary 2 can be understood as the *best decision function* with respect to the objective in choice function list Λ`algR`. Diagrammatically, in Fig. 10, the function `f` represents the *best* (in terms of objective values) arrow within a red box, located inside the blue box.

For the maximization problem involving `max` rather than `min`, just replace the monotonic condition on $\mathtt{r}^\circ$ with `r`. In practice, this greedy condition is very easy to verify when the combinatorial optimization problem is specified as an SDP, the only test is whether the *decision functions are monotonic on* $\mathtt{r}^\circ$, which is usually the preorder with respect to the objective function.

This test should be contrasted with the classical greedy condition which requires finding a *matroid* for the problem [Schrijver et al., 2003]. This is almost as hard as finding a greedy algorithm directly. By contrast, the characterization above is significantly simpler, more concise, and much easier to verify in practice. Below, the relationship between these two characterizations of the greedy condition is explored.

Greedy algorithm and matroid theory *Matroid theory* was introduced to generalize the idea of *linear independence* in linear algebra [Whitney, 1935], and develops a fruitful theory from certain axioms which it demands hold for a collection of *independent sets*. As Welsh [2010] note:" Matroid theory has exactly the same relationship to linear algebra as does point set topology to the theory of real variables."

A pair $M = (S, \mathcal{I})$ is called a matroid if S is a finite set and $\mathcal{I}$ is a nonempty collection of subsets of S satisfying:

1. Hereditary property: If $I \in \mathcal{I}$ and $J \subseteq I$, then $J \in \mathcal{I}$.

2. Exchange property: If $I, J \in \mathcal{I}$ and $|I| < |J|$, then $I + z \in \mathcal{I}$ for some $z \in J \backslash I$.

Here, I is called an *independent set* if $I \in \mathcal{I}$, and *dependent set* otherwise. In the context of matroid theory, the greedy algorithm is characterized by the following theorem.

Theorem 3. *Greedy theorem in terms of matroids.* Let $\mathcal{I}$ be a nonempty collection of subsets of a finite set S closed under taking subsets. For any weight function $w : S \to \mathbb{R}$ the aim is to select a set I in $\mathcal{I}$ minimizing $w(I)$. The greedy algorithm consists of setting $I \coloneqq \emptyset$, and next repeatedly choosing $y \in S \backslash I$ such that $I \cup y \in \mathcal{I}$ and $w(I \cup y)$ is as small as possible. The repetition stops if no such y exists. The greedy algorithm leads to a set I in $\mathcal{I}$ of minimal weight $w(I)$ if and only if $(S, \mathcal{I})$ is a matroid.

Proof. See Section 40.1 Schrijver et al. [2003]. □

Corollary 3. An exhaustive specification using a relational catamorphism (54) introduces a matroid. If the algebra `algR` of the catamorphism satisfies the greedy condition given in Thm. 2, then the resulting greedy algorithm of Thm. 2 is a valid matroid greedy algorithm.

Proof. The first part of the proof shows that any exhaustive specification using a catamorphism (54) introduces a matroid, and the second part proves that greedy algorithms given by Thm. 2 also satisfy the matroid greedy theorem.

Define $\mathcal{I}_{\text{SDP}} = \mathcal{S} \cup \mathcal{S}'$ as the union of the set of all possible configurations $\mathcal{S}$ and set all partial configurations $\mathcal{S}'$. By partial configurations, we mean, applying a finite number times, the algebra `algR` to a partial configuration `a'` $\in \mathcal{S}'$, then `a` $\in \mathcal{S}$ such that `a = algR (Cons i (.. algR (Cons i a')))`. Given $\mathcal{I}_{\text{SDP}}$ consists of all possible configurations $\mathcal{S}$ and set all partial configurations $\mathcal{S}'$, the first matroid

condition is satisfied. If `a'` $\in \mathcal{S}'$ is a partial configuration, assume `b' = algR (Cons i a')`, then `b'` is either a partial configuration or a complete configuration. Defining `|a'|` as the number of times needed to apply `algR` to obtain `a'` from the empty configuration, then `|b'|` $\geq$ `|a'|`. Therefore, the second condition of the matroid is satisfied. Hence $\mathcal{I}_{\text{SDP}}$ is an independent set of the matroid $M_{\text{SDP}} = (\mathcal{I}_{\text{SDP}}, \mathcal{I}_{\text{SDP}})$.

The greedy algorithm `cata (sel r . ΛalgR))` is the same as illustrating that the greedy algorithm consists of setting $I \coloneqq \emptyset$, and next repeatedly choosing $y \in S \backslash I$ with $I \cup y \in \mathcal{I}_{\text{SDP}}$ and with $w(I \cup y)$ as small as possible.

To demonstrate this, a symbolic way to describe the (repeated) selection process in the matroid greedy algorithm is

$$y = \underset{y \in S \backslash I}{\operatorname{argmin}}\, w(I \cup y)\,. \tag{62}$$

In our framework, the catamorphism `cata (sel r . ΛalgR))` recursively applies the algebra `(sel r . ΛalgR)` starting from the base case `Nil`, which corresponds to the empty set $I \coloneqq \emptyset$ in matroid theory. Similarly, the update function $I \cup y$ in matroid theory corresponds to the definition of the SDP decision function
`algR (Cons y I)` over the cons-list datatype. However, instead of selecting the smallest $I \cup y$ such that $y \in S \backslash I$, the selection is the best decision function with respect to a preorder `r` by `sel r . ΛalgR`. These two processes are equivalent because the cons-list catamorphism iterates over the input list from right to left, visiting each element exactly once.

Since $\mathcal{I}_{\text{SDP}}$ is an independent set of the matroid $M_{\text{SDP}} = (\mathcal{I}_{\text{SDP}}, \mathcal{I}_{\text{SDP}})$, then `cata (sel r . ΛalgR))` is a valid matroid greedy algorithm. □

The key distinctions between the classical greedy theorem in matroid theory and the novel characterization given here, lie in both the formulation of the greedy algorithm and the definition of "independence." In the approach given in this thesis, the greedy algorithm is unambiguously specified through an SDP (catamorphism), whereas in matroid theory, the greedy algorithm is characterized through an *algorithmic description*. Selecting the best decision function `f` is the same as the selection procedure of the greedy algorithm in matroid theory, where y is chosen so that $w(y)$ as small as possible. Additionally, in the matroid greedy theorem, "independence" is characterized by the structure of the matroid, while in the characterization given here, independence is an inherent property of the SDP formulation itself.

Given these distinctions, the position of this thesis is that the greedy theorem presented in Thm. 2 is based on a clearer and more rigorous problem specification. This increased clarity allows for a deeper understanding of the essential properties of the greedy algorithm, making its derivation easier and more systematic.

II.2.6.2 Different implementations of thinning

This Subsection, explores how to implement the thinning algorithm in practice. The naive *perfect* thinning algorithm requires comparing all pairs of configurations, thus $O(M^2)$ evaluations of dominance relation `rel` are required, where M is the number of partial configurations at each recursive stage which is usually exponentially/polynomially large. By perfect, it is meant that all *dominated partial configurations* are dropped.

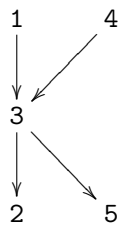
Certainly, the naive approach is impractical. Ideally, a thinning algorithm should be a linear-time program—linear with respect to the number of partial configurations, $O(M)$—that produces the smallest possible result by eliminating as many partial configurations as possible. However, a linear-time program cannot always guarantee the smallest result. Achieving both goals at the same time somewhat paradoxical because deleting more partial configurations means evaluating the dominance relation more frequently.

In practice, there is a need to balance the time taken to run the thinning algorithm and the number of configurations that can be deleted by invoking this algorithm. This Subsection provides various implementations of the thinning algorithm in Haskell, and the speed-up brought by running these different implementations will depend on the problem at hand.

As an example, given a list of elements `l = [1,2,3,4,5]` we define a preorder

```
relList = [(1,2),(3,2),(1,3),(1,5),(4,5),(4,2),(4,3)]
rel a b = (a, b) `elem` relList
```

If element `(a, b)` exists in list `relList`, then `rel a b == True`. In this case, the preorder `rel` is a partial order, i.e., it has the anti-symmetric property, otherwise thinning may have non-unique solutions. The value `4` is not comparable with other elements in `l`, and the optimal thinning of `l` with respect to preorder `rel a b` is `[1,4]`. This gives us a topological ordering rendered as



where every direct edge (or directed path) from `i` to vertex `j`, means `i` comes before `j` in the ordering. Thus the valid ordering for this partial order including `[1,4,3,2,5]`, `[4,1,3,2,5]`, `[1,4,3,5,2]`, `[4,1,3,5,2]`.

Exhaustive thinning *Exhaustive thinning* requires comparing all pairs of elements in the input list `x :: [a]`. One way to implement the exhaustive thinning function is by the following logic: use the first element `a` of the input list `l` to compare with all elements in the remaining list `x` (`l = [a] ++ x`); when `rel a b == True` delete `b` from `x`, and whenever `rel b a == True`, stop comparing and delete `a` from `l`. Then recursively apply this process to `x'`, where `x'` is the list obtained by deleting all `b`s from `x` such that `rel a b == True`. This idea is formalized in the definition

```
del :: Eq a => [a] -> [a] -> [a]
del x y = [a | a <- x, not (a `elem` y) ]

thin_exh :: Eq a => Rel a a -> [a] -> [a]
thin_exh rel []  = []
thin_exh rel [a] = [a]
thin_exh rel l@(a:x) = (h a x) ++ thin_exh rel x'
  where
    h a x
```

```
    | all (\b -> not (rel b a)) x = [a]
    | otherwise = []
  x' = x `del` [b | b <- x, rel a b]
```

The `@` symbols creates an alias `l` for pattern `(a:x)`, it meaning `l = [a] ++ x`, the function `h a x` checks if the head element `a` of list `l` is dominated –`rel b a == True` –by any elements `b` in `x`, if not, it retains `a` at the front of the list `l`, otherwise it is deleted. The list `x'` is strictly smaller than `x` since some elements from `x` are deleted to to create `x'`. This guarantees that `thin_exh` will terminate.

For instance, evaluating `thin_exh rel [1,2,3,4,5]` gives us list `[1,4]`. Another example is when the relation `r` is the total order `<`, run `thin_exh (<) [2,3,4,1,5]` obtains `[1]`.

In the worst case, exhaustive thinning compare all pairs of elements in `l`, thus the `thin_exh` function will evaluate `rel` $O(M^2)$ time, where M is the length of list `l`.

Thinning after sorting Although exhaustive thinning is perfect in the sense that all dominated configurations will be eliminated, there exists another way to achieve perfect thinning that could be more efficient than the exhaustive thinning strategy: *thinning after sorting*. The logic here is straightforward: sort a list with respect to preorder `rel` first, then scan the list from start to end. Find the first element `b` dominated by the first element `a`, then all elements after `b` should be dropped. It is clear that the time complexity of this strategy is dominated by the time spent on sorting, as scanning the entire list takes only $O(M)$ time, where M is the length of the list. It is well-known that sorting on total/linear order set has optimal worst-case *query complexity*[16] $O(M \log M)$, sorting a partially ordered set has a worst-case query complexity of $(M(W + \log M))$, where W is the maximal cardinality of the incomparable subsets, and this has been proved to be asymptotically optimal [Daskalakis et al., 2011].

The implementation of sorting algorithms is beyond the scope of this research, for convenience, assume a sorting function `sort` that sorts the list in order. The thinning algorithm for the sorted list is implemented as

```
thin_sort :: Rel a a -> [a] -> [a]
thin_sort rel [] = []
thin_sort rel [a] = [a]
thin_sort rel l@(a:x) = [a] ++ prefix
  where
    prefix = takeWhile (not . (rel a)) x
```

where the `takeWhile` function takes an *initial segment* of the list `x` for which all elements satisfy the predicate `not . (rel a)`. For instance, `takeWhile (\x -> x <= 3) [1, 2, 3, 4, 2,1] = [1,2,3]`. The function `takeWhile` will stop taking elements when it encounters the number 4. For instance, given the preorder `rel`, any of the valid orderings presented above will work. Evaluating the four valid orderings will return `[1,4]`, `[4,1]`, `[1,4]`, and `[4,1]`, respectively.

This method is effective due to the *transitivity* of the preorder: if `rel a b = True` and `rel b c = True` then `rel a c = True` follows. In a sorted list, it it only necessary to compare the "best" configuration

[16] Query complexity is the number of comparisons performed.

`a` with the "last" configuration `b` such that `rel a b = False`, with respect to the relation `rel`. Consequently, any configuration `c` that comes "after" `b` will automatically satisfy `rel a c = True`, as we have sorted the list based on a given preorder.

Thinning by bumping After introducing the perfect thinning algorithms, the section below introduces *linear-time* thinning algorithms. These implementations are only perfect in certain situations. The first linear-time method is called *bumping*, the Haskell implementation is rendered as

```
thin_bump :: Rel a a -> [a] -> [a]
thin_bump rel [] = []
thin_bump rel [a] = [a]
thin_bump rel l@(a:x) = (bump [a] x)
  where
    bump acc [] = acc
    bump acc x@(b:bs)
      | rel (head acc) b = bump acc bs
      | rel b (head acc) = bump ((tail acc) ++ [b]) bs
      | otherwise = bump (acc ++ [b]) bs
```

The `bump` function has an *accumulator* `acc` which is initialized to the first element `a` of list `l`, and it always uses the first element `head acc` of the accumulator to recursively compare with the element `b` in `x`. If `r (head acc) b == True`, it ignores `b` and compares the next elements in `bs`. If `r b (head acc) == True`, i.e. the first element in `acc` is dominated by some `b`, then it deletes `head acc` from the accumulator. In the worst case, this strategy begins deleting elements from the accumulator only after traversing to the last element of `l`. In this case, the number of evaluations of `rel` is $3M$, hence the complexity of `thin_bump` is $O(M)$.

This strategy is perfect if the relation `rel` is a *total order*. For instance, evaluating `thin_bump (<) [4,1,2,3,5]` will return `[1]`. However, in more general cases, when the dominance relation is a preorder, the effectiveness of this thinning algorithm will become quite unpredictable. For instance, consider again the above definition of dominance relation `rel`, `thin_bump (rel) [4,1,2,3,5] = [4,1,2,3,5]` but `thin_bump rel [5,3,2,1,4] = [2,1,4]`.

Thinning by squeezing In a previous study, De Moor [1995] proposed another method for implementing a linear-time thinning algorithm, he called *squeezing*. This method removes an element from a list if its neighbor is "smaller" under the preorder `rel`. In Haskell,

```
thin_squeeze :: Rel a a -> [a] -> [a]
thin_squeeze rel [] = []
thin_squeeze rel [a] = [a]
thin_squeeze rel l@(a:b:x)
  | rel a b = thin_squeeze rel ([a] ++ x)
  | rel b a = thin_squeeze rel ([b] ++ x)
  | otherwise = [a] ++ (thin_squeeze rel ([b] ++ x))
```

As with the `thin_bump` function, there is no guarantee that `thin_squeeze` is perfect in general. For instance, evaluating `thin_squeeze rel [1,4,5,3,2]` will give `[1,4,5,3]`. The `thin_squeeze` function is usually more efficient when the candidate list is sorted, such as `thin_squeeze rel [1,4,2,3,5]` will return `[1,4,2]`.

Therefore, to make the program more efficient, the `thin_squeeze` function is often combined with another operator `mmerge`, which is a generalized merge operation similar to that used in merge sort. It takes a preorder `rel`, along with two lists `x` and `y` both sorted with respect to `rel`, and merges them to produce a new sorted list containing exactly the elements of `x` and `y` [De Moor, 1995]. The implementation is as follows

```
mmerge :: Rel a a -> [a] -> [a] -> [a]
mmerge rel [] y = y
mmerge rel x [] = x
mmerge rel (a:x) (b:y)
  | rel a b   = [a] ++ mmerge rel x (b:y)
  | otherwise = [b] ++ mmerge rel (a:x) y
```

For instance, evaluating `mmerge (<) [1,3,5] [2,4,6] = [1,2,3,4,5,6]`. The composition of the `thin_squeeze` function and the `mmerge` function is referred to as `purge`. It takes two ordered lists, merges them, and then applies the squeezing operation to the result.

```
purge :: Rel a a -> Rel a a -> [a] -> [a] -> [a]
purge rel1 rel2 x y = thin_squeeze rel1 (mmerge rel2 x y)
```

In practice, the relation in `thin_squeeze` may be different to the one used with `mmerge`.

II.2.6.3 Dominance relations

Our discussion of dominance relations here is not novel; previous research, such as de Moor and Gibbons [1999], Bird and De Moor [1996], De Moor [1995], has examined this technique in depth. However, the dominance relations discussed in these studies focus on specific problems, such as the paragraph formatting problem [de Moor and Gibbons, 1999], the knapsack problem, and the bitonic tours problem, and others [De Moor, 1995]. Since these problems are not strongly connected—in the sense that their objective functions are too distinct—there are few insights to be gained from applying a dominance relation used for one problem to another.

In contrast, most machine learning problems define objective functions in a manner similar to that presented in (1), where the objective value is computed as the sum of loss terms, one for each data item. This ubiquitous pattern in objective function definitions allows us to study general dominance relations applicable across different problems.

In this section, we focus on introducing two generic and powerful dominance relations—the *Global Upper Bound* (GUB) and the *Finite Dominance Relation* (FDR)—rather than studying the applications of dominance relations in distinct problems.

The GUB dominance relation is used frequently in BnB studies but has rarely been discussed rigorously; the FDR, is, to the best of our knowledge, novel.

The discussion of Part III, explores how these two dominance relations are ubiquitous in machine learning and combinatorial optimization research. In these fields, approximate algorithms are easy to obtain, and the data is always finite. These characteristics simplify the design and implementation of the GUB and FDR relations.

Previously, the classical definition of monotonicity in the context of SDP was derived in Subsection II.2.5.5. Thm. 59 discusses how a relation `domR` satisfying monotonicity can be fused into the thinning algorithm. However, what we mean by a dominance relation has not yet been formally characterized. . A formal definition of dominance relations in the context of SDPs, follows.

Definition 16. *Dominance relation.* In a sequential decision process, given a relational algebra `algR :: func Config -> Config`, and two configuration `x :: Config` and `y :: Config`, a relation `domR :: Config -> Config` is called a dominance relation if

1. `domR` is a preorder.
2. `domR x y` $\implies$ `domR x' y'`, where `x' = algR (Cons a x)` and `y' = algR (Cons a y)`.
3. `domR` $\subseteq$ `r`.

In the above, relation `r :: Config -> Config` is the preorder with respect to the objective function value. The second condition is essentially the point-wise expression of (56). The preorder requirement is necessary because transitivity makes the thinning process more efficient.

Global upper bound dominance Approximate/heuristic algorithms are ubiquitous in the study of machine learning, operations research, and combinatorial optimization, where many problems involve intractable combinatorics. Solving these problems with exact algorithms becomes inefficient as the dataset grows larger. However, using approximate algorithms it is easy to obtain an approximate solution very cheaply, since these algorithms typically have a low-order polynomial time complexity.

In an optimization task, assume lower is better, the global upper bound dominance relation exploits the simple fact that the globally optimal solution will always be at least as good as any locally optimal solution. Hence, any approximate/heuristic algorithm can be used to obtain a global upper bound, and any partial configurations with an objective value greater than this global upper bound are guaranteed to be non-optimal. In practice, the use of GUB is extremely powerful as it can significantly shrink the search space. This is because the global upper bound obtained by approximate algorithms is usually very tight. As a result, exact algorithms can often improve their wall clock run time from several hours to just a few seconds after applying this technique.

To simplify notation, a triple in the form of `cnfg = (c,s,e) :: Config` is used to represent a configuration. This configuration triple consists of a combinatorial configuration, a data sequence, and the objective value for this configuration.

Definition 17. *Global upper bound dominance relation.* Suppose there is another *fictitious configuration* `fict = (_,_,ub)`[17]. The GUB relation is defined as `gubdomR fict cnfg = True` if `leq ub e = True`, i.e., `ub` $\leq$ `e`.

[17] In Haskell, symbol "_" denotes that the value is irrelevant.

Theorem 4. The global upper bound dominance relation *gubdomR* is a valid dominance relation if the objective function is non-decreasing with respect to the update. In other words, if `eval cnfg' >= eval cnfg`, where `cnfg' = algR (Cons a cnfg)` is the updated configuration for the partial configuration `cnfg`. Then *gubdomR* satisfies the following monotonicity condition

$$\texttt{gubdomR fict cnfg} \Longrightarrow \texttt{gubdomR fict cnfg'}, \tag{63}$$

Proof. Since `eval cnfg'` $\geq$ `eval cnfg` $\geq$ `ub`, it follows that `gubdomR fict cnfg'` must holds if `gubdomR fict cnfg` holds. □

Finite dominance The finite dominance relation is another generic and useful dominance relation which exploits the finiteness of the dataset. Since there is only a limited amount of data, in many applications it is possible to estimate how the objective function value of a partial configuration will change if extended to completion. There are two types of estimations: either estimate the "*largest*" objective value or the "*lowest*" objective value *before* extending a partial configuration to a complete configuration. These two approximations are referred to as the *pessimistic upper bound* or the *optimistic lower bound*.

The intuition for constructing a valid finite dominance relation is that if the pessimistic upper bound of a partial configuration `a` is greater than the optimistic lower bound of `b`, then `b` can be discarded. This is because the best possible extension of `b` cannot achieve a lower objective value than `a`. This concept is akin to the lower and upper bound techniques used in branch-and-bound (BnB) algorithms to reduce the combinatorial search space, although these techniques are rarely analyzed formally. Below is a formalization this idea and proof of its correctness in the context of SDPs.

Definition 18. *pessimistic upper bound* and *optimistic lower bound*. Assuming there are two configuration `a = (c_a, s_a, e_a)` and `b = (c_b, s_b, e_b)` along with two functions `pes_ub :: [Config] -> Loss` and `opt_lb :: [Config] -> Loss` to estimate the *pessimistic upper bound* or *optimistic lower bound* of a configuration. The pessimistic upper bound has the property

$$\texttt{pes\_ub algR (Cons a x) <= pes\_ub x},$$

for any partial configuration `x`. Similarly, the optimistic lower bound has the property

$$\texttt{opt\_lb algR (Cons a x) >= opt\_ub x},$$

These two functions can be utilized to formulate a specialized dominance relation, as established in the following theorem.

Theorem 5. Define the *finite dominance relation* as

$$\texttt{fdomR a b == True},$$

if `pes_ub x` $\leq$ `opt_lb y`. Then, the finite dominance relation `fdomR` is a valid dominance relation if the objective function is non-decreasing with respect to the update. In other words, we have

$$\texttt{fdomR x y} \Longrightarrow \texttt{fdomR x' y'}, \tag{64}$$

where `x' = algR (Cons a x)` and `y' = algR (Cons a y)` is the updated configuration for partial configuration `x` and `y`.

Proof. This implication holds because of the properties of the `pes_ub` and `opt_lb` functions. The `pes_ub` has the property that `pes_ub x'` $\leq$ `pes_ub x`, because function `pes_ub x` evaluates the "worst-case" objective value of `x`, so that any update `x'` of `x` will have smaller or equal worst-case objective value because since `x'` may not be the *worst update* of `x`. Similarly, `opt_lb y'` $\geq$ `opt_lb y`, because any update `y'` may not be the *best update* of `y`. Therefore, `pes_ub x'`$\leq$ `pes_ub x`$\leq$`opt_lb y`$\leq$`opt_lb y'`, the implication holds. □

Another way to use finite dominance is to combine it with the global upper bound technique. The optimistic lower bound function `opt_lb` calculates the best-case objective value of a partial configuration. One intuition to combine it with the GUB technique is that if the optimistic lower bound of a partial configuration `a` is greater than the global upper bound `ub`, then `a` can be freely discarded. To verify this intuition, define a dominance relation as `fubdomR fict a == True` if `ub` $\leq$ `opt_lb a`. It is necessary to verify that `fubdomR` satisfies

$$\texttt{fubdomR fict a} \Longrightarrow \texttt{fubdomR fict a}. \tag{65}$$

This implication is true because `opt_lb a'` $\geq$ `opt_lb a` $\geq$ `ub`.

The below two examples illustrate the applications of finite dominance in machine learning problems.

Example 5. *Classification problem.* In previous work on the 0-1 loss classification problem [He and Little, 2023a], finite dominance was applied to develop an algorithm called *incremental combinatorial purging* (ICG-purge). This algorithm exploits the fact that the optimistic lower bound for a partial configuration is the same as the 0-1 loss of this configuration, and the pessimistic upper bound for a partial configuration `a` –a binary assignment –is equivalent to the number of remaining recursive steps n. Hence, if a configuration `loss a + n <= loss b`, then configuration `b` will be non-optimal, so that `a` dominates `b` . This result can be easily generalized to the weighted 0-1 loss classification problem and other loss functions with objectives which are non-decreasing with increasing amount of data.

Example 6. *Regression tree problems.* Previous work of Zhang et al. [2023] designed a method called "the K-means lower bound" to calculate the optimistic lower bound for the sparse regression tree problem. Their method relies on the intuition that an optimal regression tree solution can never be better than the result of 1D K-means clustering on labels. This introduces an optimistic lower bound, which allows elimination of any partial configurations that have an optimistic lower bound greater than the global upper bound.

II.2.7 Recursive optimization framework

II.2.7.1 Hylomorphism recursive optimization framework

Given a combinatorial optimization problem that is specified in the form

$$\texttt{sel r . } \Lambda\texttt{(hylo algRf coalgR)}, \tag{66}$$

and consider only the case where the relational algebra is a function `algRf :: func a -> a` (function is a special case of relation), a *relational function-algebra*. Note that both relational function-algebra `algRf` and its functional correspondence `alg :: func [a] -> [a]` are functions, they are distinguished by their type information. For this specification, Bird and De Moor [1996] present the following theorem.

Theorem 6. *Dynamic programming theorem.* If `algRf` is monotonic on `r`, then the solution to specification (66) can be obtained by

$$\texttt{hyROF = sel r . (map (algRf . (fmap hyROF))) . ΛcoalgR}, \tag{67}$$

where the relational algebra has type `algRf :: func b -> b`, and the power transpose of the relational coalgebra has type `ΛcoalgR :: a -> [func a]`.

When a DP algorithm exists, it is precisely when the condition of Thm. 6 holds for the *cons-list datatype*. Considering the greater expressivity of hylomorphism, which extends beyond the cons-list datatype, Theorem 6 provides a more general and powerful framework for constructing recursive optimization programs.

We can implement (66) in Haskell as

```
hyROF :: Functor func => Rel b b->(func b-> b)->(a ->[func a])-> a -> b
hyROF r alg coalg = minlist r . (map (alg . fmap (hyROF r alg coalg))) .
   coalg
```

Because it is not possible to implement the power transpose in Haskell, the function `coalgR_pt` is used to represent the function after applying the power transpose to relational coalgebra `coalgR`. It is easy to verify that when relational function-coalgebra `coalgR` is the terminal algebra `out`, the program (67) becomes a catamorphism[18].

For some special relations, the selector relation `r` is difficult to implement in a functional language. To address this issue, instead replace selector `minlist r` with a function `polymin` of type `polymin :: [b] -> b`. The term "polymin" is an abbreviation for "polymorphic min," a name inspired by the semiring homomorphism connecting the min-plus semiring $(\mathbb{R}, \min, +, -\infty, 0)$ to other semirings in combinatorial optimization, such as the generator semiring $([[\mathbb{X}]], \cup, \circ, \emptyset, [[\,]])$.

Then `hyROF` can be reformulated as

```
hyROF_poly :: Functor func =>([b]-> b)->(func b->b)->(a->[func a])-> a-> b
hyROF_poly polymin alg coalg = polymin .
        (map (alg . fmap (hyROF_poly polymin alg coalg))) . coalg
```

Furthermore, consider the case where a dominance relation exists so that the thinning algorithm is applicable, then the following corollary applies.

Theorem 7. *Hylomorphism recursive optimization theorem.* If `algRf` is monotonic on `r`, given a dominance relation `domR :: Rel (func a) (func a)`, and if `domR` is a preorder satisfying

$$\texttt{algRf . (fmap (hylo algRf coalgR)) . domR}^\circ \subseteq \texttt{r}^\circ \texttt{. algRf . (fmap (hylo algRf coalgR))}, \tag{68}$$

then solution obtained by

$$\texttt{hyROF\_thin = sel r . (map (algRf . (fmap hyROF))) . thin domR . ΛcoalgR}, \tag{69}$$

is a solution of (66).

[18] To see this, apply the rule `min r . map X ⊆ X . ∈` and `∈ . Λ = id`. The inclusion becomes an equality when `X` is a function

Proof. See Bird and De Moor [1996], Chapter 8. □

The above function `hyROF_thin` can be implemented as

```
hyROF_thin :: Functor func =>Rel b b -> (func b -> b)->(a -> [func a])->
   a -> b
hyROF_thin r algRf coalgR_pt = minlist r .
        (map (alg . fmap (hyROF_thin r alg coalgR_pt))) . (thin domR) .
           coalgR_pt
```

Problems that satisfy the conditions given by Thm. 7 are problems that can be solved by the *hylomorphism recursive optimization framework* (Hy-ROF).

II.2.7.2 Catamorphism recursive optimization framework

In many cases, specifying a combinatorial optimization problem via a hylomorphism (53) is unnecessary because the coalgebra is not required. More commonly, the problem can only be decomposed sequentially, i.e. the problem can only be specified through a catamorphism (54). Compared with hylomorphisms, the sequential decomposition process of catamorphisms provides simplicity, making the program much more comprehensible. Therefore, when applicable, it is preferable to construct an efficient recursive optimization program based on catamorphisms rather than hylomorphisms.

Given a combinatorial optimization problem problem specified as (54), the following corollary provides a method for constructing an efficient program to solve it.

Corollary 4. *Catamorphism recursive optimization framework.* If `domR` $\subseteq$ `r` and `algR:: func a -> a` is monotonic on `domR`$^\circ$, then the solution of

$$\texttt{sel r . cata (thin domR . alg)}, \tag{70}$$

is also a solution of (54), where the functional algebra `alg :: func [a] -> [a]` is the functional correspondence of the relational algebra `algR`.

In particular, when the greedy condition is satisfied, i.e. `domR = r`, (70) becomes a greedy algorithm.

Any problem that satisfies the condition given by Corollary 4, are the problems that can be solved by the *catamorphism recursive optimization framework* (Cata-ROF).

II.2.8 Branch-and-bound method in constructive algorithmics

Branch-and-bound (BnB) algorithms are methods for global optimization in combinatorial optimization problems. It has been widely used to solve computation-intensive, typically NP-hard, problems, such as the traveling salesman problem [Balas and Toth, 1983], the vehicle routing problem [Christofides et al., 1981] in operation research, the hyperplane decision tree problem [Dunn, 2018], sparse decision tree problem [Lin et al., 2020], 0-1 loss linear classification problem [Nguyen and Sanner, 2013], Euclidean K-center problems [Fayed and Atiya, 2013], and K-means problem [Du Merle et al., 1999, Koontz et al., 1975] in machine learning.

The BnB method has also been studied within the functional programming community. For instance, Bird and Hughes [1987] demonstrated how to derive the well-known *alpha-beta pruning algorithm* for game

tree evaluation specification directly from its specification. Fokkinga [1991] showed how to incorporate a depth-first search strategy into a fold operator; their discussion is closely related to the exposition in this section. However, we extend this work by generalizing it to include various search strategies and thinning methods in a catamorphism.

The underlying strategy of BnB algorithms is to decompose a given problem, which is difficult to solve directly, into consecutive partial problems of smaller sizes. This decomposition is applied repeatedly until each subproblem is either solved or proven not to yield an optimal solution to the original problem. The assessment of a partial problem in BnB algorithms typically involves computing a lower bound on the minimum objective value, similar to the optimistic lower bound introduced above. If the computed lower bound exceeds the objective value of the best upper bound currently available, this indicates that the partial problem cannot provide an optimal solution to the original problem.

Backtracking is widely used in combinatorial optimization [Kreher and Stinson, 1999] and combinatorial generation [Ruskey, 2003] studies, and is often considered a key feature of BnB algorithms. In the study of combinatorial generation, it is required to generate all possible configurations that satisfy some predicates. A seed configuration is recursively updated according to some procedure, once the partial configuration becomes infeasible with respect to a predicate, then the algorithm *backtracks* to the previous feasible partial configuration, and continues updating from there.

On the other hand, in the study of BnB algorithms, the problem requires finding one optimal solution instead of generating all possible configurations. BnB algorithms systematically search for the optimal solution to a problem by repeatedly attempting to extend an approximate solution in all possible ways. If a particular extension fails, the algorithm "backtracks" to the last point where alternative options are still available.

Depth-first search in constructive algorithmics

Two common issues in the study of BnB algorithms are the *proof of exhaustiveness* and the *analysis of time complexity*. These issues are often conducted through tedious inductive proofs or, more frequently, presented as assertions or informal explanations. However, the lack of proof of exhaustiveness poses significant risks, as it leaves uncertainty about whether BnB algorithms are truly exact. If BnB algorithms are indeed exact, we should be able to derive them from an exhaustive search specification.

At the same time, when claiming that an algorithm can find the exact solution to a problem, it is essential to clarify what is meant by "can" first. In other words, the interpretation of "can" varies depending on whether we allow a time of 1 minute, 1 hour, or 1 year, or merely require that the time be finite. However, most BnB algorithms exhibit exponential complexity in the worst case, so claiming that these algorithms can solve the problem often amounts to a vacuous assertion.

All in all, the correctness and the terseness of the proof, as well as the worst-case time guarantee, are critical for exact algorithms. To address the aforementioned issues in BnB studies, this thesis explores how to formally characterize the BnB method within this framework. As noted in Section I.2.2, BnB algorithms consist of four main components: *branching rules*, *pruning*, *bounding rules*, and *search strategies*. Through the exposition in this thesis, it is immediately possible to connect these terms to decision functions, the thinning process, and dominance relations introduced above. However, search strategies and backtracking techniques have not yet been discussed. This section will analyze these components within

the framework of the thesis. It will become clear that the BnB method, when defined as a combinatorial generator incorporating backtracking techniques, can be derived from catamorphism generators. Since catamorphism generators are exhaustive, it follows that combinatorial generators involving backtracking are also exhaustive.

Section II.1.2 illustrated the combinatorial generation tree for nearly all the SDP generators introduced. Drawing a generation tree diagram for a catamorphism generator is the same as representing a *recursion* as an *iteration*. Indeed, transforming a recursion into an iteration is always possible for any recursion that is defined by a catamorphism. The key to incorporating the backtracking technique into catamorphisms lies in transforming a *recursive-style* program into an *iterative-style* program. A similar characterization can be found in [Fokkinga, 1991]. Since Fokkinga [1991]'s work was published in the early stages after the introduction of constructive algorithmics by Meertens [1986], his proof relies on inductive style arguments and spans nearly five pages. Below is a much simpler characterization based on the definition of catamorphisms.

Theorem 8. *Branch-and-bound characterization theorem.* Given a catamorphism `cata alg x` on cons-list `ListFr a`, where `x = [an...a1]` is a finite list, the following equality holds,

```
cata alg x = concat $ map (alg_pt_iter x 0 algR_pt) e,
```
(71)

where `e = alg_pt Nil`, `algR_pt :: ListFr a b -> [b]` is the power transpose of the relational algebra `algR`, and `alg_pt_iter` is defined as

```
algR_iter :: [a] -> Int -> (ListFr a b -> [b]) -> b -> [b]
algR_iter x i algR_pt y
    | i == length x = [y]
    | i < length x = concat $ map (algR_iter x (i+1) algR_pt) $ algR_pt
       (Cons (x!!i) y)
```

where `algR_pt_iter` represents an iteration of `algR_pt` with input `y` from index `i` to the end of list `x`. The right-hand side of (71) is referred to as an *iterative catamorphism* or *branch-and-bound algorithm* (catamorphism with backtracking).

Proof. A catamorphism over cons-list can be expanded informally as

```
cata alg [an..a1] = alg (Cons an (alg ... alg (Cons a1 (alg []))))),
```
(72)

and the following equational reasoning steps hold:

```
  alg (Cons an (alg ... alg (Cons a1 (alg []))))
```
≡ let `e = alg []`
```
  alg (Cons an (alg ... alg (Cons a1 e)))
```
≡ equivalence equation
```
  alg . (Cons an) (alg ... alg . (Cons a1) e)
```
≡ equation (52) `alg . (Cons a)= concat . map algR_pt . (Cons a)`
```
  concat $ map (algR_pt . (Cons an)) (concat ... concat (map (algR_pt . (Cons a1)) e))
```
≡ definition of `algR_pt_iter`
```
  concat $ map (algR_pt_iter x 0 algR_pt) e
```
recall that `Cons a` has type `Cons a :: x -> ListFr a x`. □

Indeed, a seed configuration `e` is repeatedly extended by `algR_pt` in a *depth-first way* because of the *laziness* of Haskell[19].

Bounding and different search strategies

As discussed, search strategies and bounding are two fundamental components of the BnB method. Having examined the depth-first technique, we now extend the discussion to incorporate alternative search strategies and bounding techniques.

The function `cata_iter` in (71) is performed in a depth-first way, as the `map` function always processes elements sequentially from the beginning of the list. The iterative catamorphism in (71) can be generalized to accommodate arbitrary search strategies by incorporating sorting based on a total ordering `s`. We can redefine `algR_iter` as follows

```
algR_iter' :: Eq b => Rel b b->[a]->Int->(ListFr a b->[b])->b->[b]
algR_iter' s x i algR_pt y
    | i == length x = [y]
    | i < length x = sort s $ concat $ map (algR_iter' s x (i+1) algR_pt)
       $ algR_pt (Cons (x!!i) y)
```

where `sort :: Rel a a -> [a] -> [a]` sorts a list according to a total order `s`. This sorting is implemented using the bubble sort algorithm

```
bubble r [] = []
bubble r [x] = [x]
bubble r (x:y:xs) = if r x y then x : bubble r (x:xs) else y : bubble r
   (y:xs)
```

[19] Haskell is a lazy language. This means that expressions are not evaluated when they are bound to variables, but their evaluation is deferred until their results are needed by other computations. In consequence, arguments are not evaluated before they are passed to a function, but only when their values are actually used.

```
sort r (x:xs) | bubble r (x:xs) == (x:xs) = x:xs
              | otherwise                 = sort r (bubble r (x:xs))
```

If `r` defines a total order based on the objective value, then `iter_cata` executes in a *best-first* order.

The bounding technique is a special case of the dominance relation. As established in Theorem (1), if the relational algebra `algR` is monotonic with respect to `domR`$^\circ$, the thinning process can be fused within a catamorphism. Similarly, if *`algR:: ListFr a b -> b`* is monotonic on r°, we can integrate the thinning process within `algR_iter'` by defining

```
algR_iter_thin :: Eq b =>Rel b b->Rel b b->[a]->Int->(ListFr a
   b->[b])->b->[b]
algR_iter_thin r s x i algR_pt y
    | i == length x = [y]
    | i < length x  = sort s $ thin r $ concat $ map (algR_iter_thin r s
       x (i+1) algR_pt) $ algR_pt (Cons (x!!i) y)
```

Finally, we have

```
cata_iter_thin ::Eq b => Rel b b->Rel b b->[a]->(ListFr a
   b->[b])->[b]->[b]
cata_iter_thin r s x algR_pt e = concat $ map (algR_iter_thin r s x 0
   algR_pt) e
```

The iterative catamorphism, `cata_iter_thin` demonstrates that *the BnB algorithm can be derived from an exhaustive search specification* defined by a cons-list catamorphism. Furthermore, `cata_iter_thin` provides a formal and generic definition for the BnB method, where different bounding methods and search strategies can be implemented by simply modifying relation `r` and `s`.

Comparison of different search strategies The use of backtracking techniques can improve best-case time and space efficiency. However, when generation trees are expanded using a depth-first approach —or other strategies such as best-first or iterative deepening—it is critical to recognize that, under certain objectives, backtracking may result in a greater number of partial configurations being visited. For instance, if the optimization objective is related to the *depth* of a configuration in the generation tree—defined as the number of update operations applied to reach the configuration from the initial seeds—then an ordinary catamorphism without backtracking is likely more efficient, as it explores solutions *layer-by-layer*. Thus, in combinatorial optimization, while backtracking can *improve* best-case time complexity, it may also *worsen* worst-case time complexity in some problems.

Furthermore, from an implementation perspective, managing an entire list of partial configurations is easier than handling each partial configuration one-by-one by using a for loop. Therefore, once the backtracking method is involved, parallelization becomes difficult due to the required communication between different processors. By contrast, the ordinary catamorphism generator based on join-lists is embarrassingly parallelizable, requiring no communication between processors. This lack of parallelization is evident in machine learning research, where BnB algorithms employing depth-first search [Zhang et al., 2023, Hu et al., 2019, Lin et al., 2020, Nguyen and Sanner, 2013] are rarely parallelized.

In terms of memory complexity, when a complete configuration with an objective value smaller than the global upper bound is found, the global upper bound can be replaced with this configuration's objective value. Consequently, the number of configurations generated by backtracking algorithms is always less than or equal to that produced by the ordinary catamorphism generator, as more configurations are pruned by a tighter global upper bound. In practice, using backtracking techniques can result in significant memory savings.

II.2.9 Reconciling combinatorial optimization methods

Inclusion relationships between different combinatorial optimization methods Now, after introducing the hylomorphism and catamorphism recursive optimization frameworks above, it is appropriate to return to the question of what is the recursive optimization framework, and how are classical combinatorial optimization methods related to it?

The recursive optimization framework is a way to construct efficient recursive programs for solving combinatorial optimization problems. It subsumes classical algorithm design strategies, such as the greedy method, dynamic programmings and divide-and-conquer, so that the following inclusion relations hold:

$$\text{SDP} \subseteq \text{Greedy algorithm} \subseteq \text{BnB} \subseteq \text{General SDP} \subseteq \text{ DP}, \text{Classical D\&C} \subseteq \text{General D\&C}, \tag{73}$$

where *general SDP* and *general D&C* refer to ordinary SDP (catamorphism) and ordinary D&C (hylomorphism) with additional *thinning processes*, *alternative search strategies*, and the *memoization technique*.

These inclusion relations are defined by the generality of their abstraction used to define these methods

$$\text{Catamorphism} \subseteq \text{Cata-ROF} \subseteq \text{Hy-ROF}. \tag{74}$$

In other words, the more structure the recursion requires, the more general the abstraction needed to describe it. Therefore, the sequential decision process has the least structure (sequential), and BnB requires more structure than the Greedy algorithm, as it incorporates backtracking techniques. However BnB is still fundamentally characterized by SDP in its essence. The DP method requires the addtional structure of overlapping subproblems and the classical D&C requires the structures of non-overlapping subproblems. The property of overlapping subproblems is partly what distinguishes the DP method from classical D&C method. The classical D&C method decomposes a problem into subproblems arbitrarily and **without overlap**. By contrast, dynamic programming always involves **overlapping** subproblems in its decomposition. Clearly, both DP and classical D&C methods can be represented by a hylomorphism, i.e. general D&C. Therefore, we consider them incomparable.

Dynamic programming and memoization The DP algorithms are characterized by the dynamic programming theorem 6, which can be further refined based on whether the DP problem's subproblems have a *layered* or *symmetric* structure [Bird, 2008]. However, Thm. 6 itself does not allow the use of the memoization technique which is often considered one of the main characteristics of the DP algorithm. The position of this thesis, based on the reasoning presented earlier, is that memoization is not a special "technique" for speeding up the DP algorithm but rather an inherent byproduct of DP recursion itself. A bottom-up recursion naturally avoids the recomputation of subproblems. This efficiency stems directly

from the recursive structure of the algorithm, not from an otherwise deliberately imposed additional code optimization strategy.

When implementing memoization, an alternative perspective can be helpful: viewing DP algorithms as a form of the *course-of-value recursion* [Uustalu and Vene, 1999, Uustalu et al., 2001]. This approach integrates the catamorphism with context-sensitive computations modeled by *comonads*. In this way, DP can be characterized as a *histomorphism* [Hinze and Wu, 2013].

However, there is more than one way to incorporate the memoization technique. Two common methods frequently used in the literature are tabulation and bottom-up recursion. Indeed, histomorphism can be understood as a generic tabulation technique due to its datatype genericness. The systematic approach to transforming a top-down recursion into a bottom-up recursion has also been studied by [Bird, 2008].

Branch-and-bound and dynamic programming Kohler and Steiglitz [1974], Ibaraki [1977] discuss how several dynamic programming algorithms can be formulated within the framework of BnB, which seems to imply that DP methods are a subclass of the BnB method. However, based on the reasoning presented above, there are two reasons to disagree with this conclusion.

First, almost all BnB algorithms [Balas and Toth, 1983, Aglin et al., 2020, Diehr, 1985, Fayed and Atiya, 2013, Fokkinga, 1991, Ibaraki, 1977, Zhang, 1996, Nguyen and Sanner, 2013] describe BnB as a sequential decomposition process, i.e., a catamorphism. Moreover, foundational theoretical research on the BnB method [Ibaraki, 1977, Kohler and Steiglitz, 1974, Zhang, 1996] defines branching rules as decomposing a complete problem into distinct subproblems. However, the definition of these subproblems is vaguely stated in early works. Nonetheless, in applications where the BnB method is combined with MIP solvers—particularly when integrated with the cutting plane algorithm—branching is further refined as splitting a problem according to a variable into disjoint subproblems [Huang, 2008].

Second, using alternative search strategies in BnB algorithms involving overlapping subproblems, becomes intricate. In dynamic programming recursion, a depth-first search strategy will inevitably revisit the same subproblem multiple times, as subproblems are shared. While recomputation can be avoided through *caching* or the use of a *memoization table*, the lookup time can be significant, especially for complex datatypes like trees or lists.

II.2.10 From theory to practice

To demonstrate the value of the novel framework presented above, below it is used to solve two common combinatorial optimization problems, the *maximum sublist sum problem* and the *sequence alignment problem*. The first problem is known to have a greedy solution, and the other is well-known to have a DP solution. This section shows how to design these two algorithm from scratch.

For each combinatorial optimization problem, it is sometimes required to solve either the *optimal configuration problem* or the *optimal value problem*. The optimal configuration problem involves finding the optimal configuration of a combinatorial optimization problem, whereas the optimal value problem requires determining the optimal objective value of a combinatorial optimization problem. In the framework of this thesis, these tasks can be achieved by modifying the corresponding algebra or coalgebra. In every case, the optimal value of the problem can be obtained from the optimal configuration. However, executing

a program that directly produces the optimal value is often more efficient in terms of actual wall-clock runtime, though only by a constant factor. This is because manipulating a value—typically a real number or an integer—is more efficient than handling a configuration, which is usually stored as a list.

The analysis of this section provides solutions for both the optimal configuration problem and the optimal value problem for both the maximum sublist sum problem and the sequence alignment problem.

II.2.10.1 Maximum sublist sum problem

The goal of the maximum sublist sum problem is to find the maximum sum of any sublist within a given list. This problem has a greedy solution. This section explains how to derive an efficient algorithm for this problem. Based on previous discussion in Section II.2.3, the sublists generator for the cons-list datatype can be defined as follows

```
subsAlg Nil = [[]]
subsAlg (Cons a xs) = map (a:) xs ++ xs


subs = cata subsAlg
```

For this problem, the cost of a configuration (sublists) is simply the sum of the values within the sublists. Thus, the selector relation is a total order that can be defined as `r = sum° . geq . sum`. Note that, since we never implement the defining component of `r` separately, for ease of explanation, `sum°` is used here as a symbolic representation to describe how the configuration can be recovered. This does not necessarily imply that we can invert a value to retrieve the original configuration. In Haskell, `r` can be defined as the following expression

```
r :: (Num a, Ord a) => Rel [a] [a]
r a b = sum a >= sum b
```

Thus the selector of the maximum sublist sum problem can be defined as

```
maxlist :: (Num a, Ord a) => [[a]] -> [a]
maxlist = minlist rel
```

Note that `maxlist` is a function rather than a relation. As explained in Subsection (II.2.5.3), it returns the first candidate with the maximum objective value. However, in the context of this thesis, it is safe to use `maxlist` to apply equational reasoning, as long as we assume `maxlist` selects one possible configuration with an optimal objective value.

The exhaustive search algorithm for this problem can hence be defined as

```
maxSub :: (Num a, Ord a) => [a] -> [a]
maxSub = maxlist . subs
```

This exhaustive search algorithm is undoubtedly inefficient, as the number of possible sublists generated by the `subs` is 2^N large. However, for this problem, selector `maxlist` can be integrated into the generator. Therefore, there exist a greedy solution to the maximum sublist sum problem.

Optimal configuration problem In order to prove there exists a greedy solution to the maximum sublists sum problem, it is necessary to show that the relational algebra of the generator satisfies the greedy condition given in Thm. 2, i.e. to prove `subsalgR` is monotonic with respect to `r°`. According to the Eilenberg-Wright lemma, the relational definition of the sublists algebra `subsalgR` is defined as

```
subsAlgR :: ListFr a [a] -> [a]
subsAlgR Nil = []
subsAlgR (Cons a x) = a:x or x
```

Again, the `or` symbol does not exist in Haskell; it is used to represent logical "or" (since relations cannot be defined in Haskell).

In the context of SDPs, the meaning of monotonicity is given by (56). The next step is to verify this is true for algebra `subsalgR` and relation `r°= sum . geq . sum°`. If two configurations (sublists) `x` and `y` satisfy `r° x y = True`, i.e. `x` has a smaller sum value compared with `y`, then it is easy to verify `r° x y ⊆ r° x' y'`, where `x' = subsAlgR (Cons a x)` and `y' = subsAlgR (Cons a y)`. More simply, we have

$$\texttt{sum x} \leq \texttt{sum y} \implies \texttt{sum a:x} \leq \texttt{sum a:y}$$

Following the result of the greedy theorem 2,

$$\texttt{cata (max . } \Lambda\texttt{subsalgR)} \subseteq \texttt{max . } \Lambda\texttt{(cata subsalgR)}$$

where the power transpose of the relational algebra Λ`subsalgR` is defined as

```
subsAlgR_pt :: ListFr a [a] -> [[a]]
subsAlgR_pt Nil = []
subsAlgR_pt (Cons a x) = [a:x] ++ [x]
```

Moreover, the selector `maxlist` can be fused into `subsAlgR_pt` by defining

```
maxSubsAlgR_pt :: (Num a, Ord a) => ListFr a [a] -> [a]
maxSubsAlgR_pt Nil = []
maxSubsAlgR_pt (Cons a x) = maxlist ([a:x] ++ [x])
```

Finally, the maximum sublists sum problem can be solved greedily using the following program

```
maxSubs' = cata maxSubsAlgR_pt
```

Optimal value problem In some applications, it may be only of interest to obtain the *optimal value* of the problem, rather than the *optimal configuration*. It turns out that if the algebra for the optimal configuration problem is monotonic with respect to a selector relation of the form `r = cost° . leq . cost`, then the algebra for the optimal value problem is also monotonic with respect to `r'° = leq`, and the algebra for the optimal value problem can be obtained by fusing the algebra for the optimal configuration problem with the *incremental update* of the `cost` function.

In the case of the maximum sublists sum problem, the algebra for the optimal value problem is defined as

```
subsValAlgR :: ListFr a Int -> Int
subsValAlgR Nil = 0
subsValAlgR (Cons a acc) = (a+acc) or acc
```

which is obtained by integrating `subsAlg` with the incremental update of the `sum` function, where the new value `a` is simply added to the accumulated value `acc`. It is trivial to verify that `subsValAlgR` is monotonic to `r'° = leq`, given `subsalgR` is monotonic to `r°`. Therefore, the optimal value of the maximum sublist sum problem can be obtained using the following greedy algorithm

```
maxSubsValAlgR_pt :: ListFr Int Int -> Int
maxSubsValAlgR_pt Nil = 0
maxSubsValAlgR_pt (Cons a acc) = max (a+acc) acc

maxSubsVal = cata maxSubsValAlgR_pt
```

where `maxSubsValAlgR_pt` is the fused function of `max :: a -> a -> a` and the power transpose of `subsValAlgR`. Furthermore, since `max` will always select large value, and `a + acc` will only be larger than `acc` if `a > 0`. Thus we can equivalently define the above program as

```
maxSubsValAlgR_pt' :: ListFr Int Int -> Int
maxSubsValAlgR_pt' Nil = 0
maxSubsValAlgR_pt' (Cons a acc) = if a>0 then a+acc else acc

maxSubsVal' = cata maxSubsValAlgR_pt'
```

II.2.10.2 Sequence alignment problem

The sequential alignment problem is a well-known problem in bioinformatics that can be solved using a dynamic programming algorithm. Consider two DNA or protein sequences, and the goal of inferring if they are homologous or not. To do this, one must measure the similarity of the two sequences `x` and `y`, and their similarity is measured by finding the *shortest edit sequence*. In the basic case, there are just three basic edit operations: *match*, *delete* and *insert*. The match operation matches a base pair `a` which is an element of both in x and `y`, the delete operation deletes a base pair `a` from `x`, and the insert operation inserts a base pair `a` in `y`. These three basic operations can be defined as the following type

```
data Op a = Mat a | Del a | Ins a deriving Show
```

Optimal configuration problem The sequence of editing operations above contains enough information to construct sequences `x` and `y` from scratch. Match `a` means append `a` to both sequences, delete `a` means append `a` to the left sequence, insert `a` means append `a` to the right sequence.

Given an editing sequence `[Op a]` it is straightforward to define a cons-list algebra such that the catamorphism defined by it receives an editing sequence and recursively constructs the original two strings:

```
editalg :: ListFr (Op a) ([a],[a]) ->  ([a],[a])
editalg Nil = ([],[])
```

```
editalg (Cons op (x,y)) = case op of (Mat a) -> ([a] ++ x, [a] ++ y)
                                      (Del a) -> ([a] ++ x, y)
                                      (Ins a) -> (x, [a] ++ y)

edit :: [Op a] -> ([a],[a])
edit = cata editalg
```

where `case op of` can be read as "if `op = Mat a`, then `([a]++x, [a]++y)`", "if `op = Del a`, then `([a]++x, y)`", and "if `op = Ins a`, then `([a]++x, [a]++y)`". For instance, evaluating `cata editalg [Del 2,Ins 1,` returns`([2,3],[1,3])`.

Constructing the `edit` function is clearly much easier compared with constructing the sequence alignment algorithm directly. With careful observation, this task can be accomplished by most individuals. This is, in fact, the most significant reason for using this relational framework. It cannot be achieved within a purely functional framework because, as we will see shortly, the converse of `edit` is not a function but a relation. Next, we will demonstrate that the sequence alignment dynamic programming algorithm can indeed be derived from the specification of the `edit` function.

The `edit` function recovers two sequences from a given editing sequence, and its converse `edit`$^\circ$ should be a relation that takes a pair of sequences and returns an editing sequence. In other words, `edit`$^\circ$ should possess type information `edit`$^\circ$ `:: ([a],[a]) -> [Op a]`. As a result, from the definition, Λ`edit`$^\circ$ should have a type Λ`edit`$^\circ$ `:: ([a],[a]) -> [[Op a]]`, i.e., a function that takes a pair of sequences and returns *all* corresponding edit sequences. Hence, the specification of the sequence alignment problem can be formally characterized as

$$\texttt{min r .}\ \ \Lambda(\texttt{edit}^\circ) \tag{75}$$

where `edit`$^\circ$ `= ana editalg`$^\circ$ because `edit` is defined as a catamorphism so its converse `edit`$^\circ$ should be an anamorphism. Although `editalg` is a function, its inverse `editalg`$^\circ$ will be a non-deterministic relation, since any pair of sequences has three possible choices of operations. Let `editalg`$^\circ$ `= uneditCoalgR` which can be defined as

```
uneditCoalgR :: Eq a => ([a],[a]) -> ListFr (Op a) ([a],[a])
uneditCoalgR ([],[]) = Nil
uneditCoalgR ((a:x),[]) = Cons (Del a) (x,[])
uneditCoalgR ([],(b:y)) = Cons (Ins b) ([],y)
uneditCoalgR ((a:x),(b:y)) = if a=b then Cons (Mat a) (x,y) else
                             or Cons (Del a) (x,b:y)
                             or Cons (Ins b) (a:x,y)
```

Recall that the use of `or` keyword implies that we are defining a relation in functional programming language. The third pattern can be interpreted as follows: if `a=b` then apply the operation `Mat a`; otherwise, either delete `a` using `Del a` or insert `b` using `Ins b`.

The derivation of `uneditCoalgR` is based on the fact that when `x` or `y` in pair `(x,y)` is non-empty, we always have three choices to generate new an operation: if the first elements of `x` and `y` matches, construct a new `Mat` operation, or either delete the first element from `x` or insert a new element to `y`. However, note

that when the `Mat` operation is applied, it can always lead to a shorter sequence, thus `uneditCoalgR` can be modified as

```
uneditCoalgR :: Eq a => ([a],[a]) -> ListFr (Op a) ([a],[a])
uneditCoalgR ([],[]) = Nil
uneditCoalgR ((a:x),[]) = Cons (Del a) (x,[])
uneditCoalgR ([],(b:y)) = Cons (Ins b) ([],y)
uneditCoalgR ((a:x),(b:y))
  | (a == b) = Cons (Mat a) (x,y)
  | otherwise = (Cons (Del a) (x,b:y)) or (Cons (Ins b) (a:x,y))
```

To design a DP solution for problem (75), it is required to verify that the algebra for (75) is monotonic to the selection relation `r`. However, at first glance, there exists no algebra for specification (75), since this is an anamorphism instead of a hylomorphism.

Nonetheless, based on previous discussions, the hylomorphism is equivalent to a catamorphism composed with anamorphism `hylo alg coalg = cata alg . ana coalg` and `cata in = id`. Thus `ana uneditCoalgR` can be considered as a hylomorphism for which the algebra is the initial algebra. In other words, we have equality

```
hylo in uneditcoalg = cata in . ana uneditcoalg = ana uneditcoalg
```

and the initial algebra `in` can be defined explicitly as

```
in :: ListFr a [a] -> [a]
in Nil = []
in (Cons a x) = a:x
```

Recall that, the original problem is to find a minimum-length edit sequence which aligns the two input sequences. By defining the selector relation as `r = length° . leq . length`, it is trivial to verify that initial algebra `in` (this keyward is already used in Haskell, so `in` needs to be renamed in the actual implementation) is monotonic with respect to `r`, as adding more operations can only increase the length of a configuration. Again, we assume the selector `minlist` will select one possible configuration with optimal objective value.

Therefore, according to Thm. 6, the problem (75) can be solved exactly by the following program

```
mes :: Eq a => ([a],[a]) -> [Op a]
mes = hyROF r mesAlg mesCoalg
 where
  r a b = length a <= length b
  mesAlg = in
  mesCoalg = uneditCoalgR_pt
```

where `uneditCoalgR_pt` is the power transpose of coalgebra `uneditCoalgR`, which is defined as

```
uneditCoalgR_pt :: Eq a => ([a],[a]) -> [ListFr (Op a) ([a],[a])]
uneditCoalgR_pt ([],[]) = [Nil]
uneditCoalgR_pt ((a:x),[]) = [Cons (Del a) (x,[])]
```

```
uneditCoalgR_pt ([],(b:y)) = [Cons (Ins b) ([],y)]
uneditCoalgR_pt ((a:x),(b:y))
  | (a == b) = [Cons (Mat a) (x,y)]
  | otherwise = [(Cons (Del a) (x,b:y)), (Cons (Ins b) (a:x,y))]
```

For instance, executing `mes ([1,2,3], [3,2,1])` gives us `[Del 1,Del 2,Mat 3,Ins 2,Ins 1]`.

Optimal value and all alignment sequences problems In order to solve the optimal value problem, we simply need to simply modify the algebra `mesAlg` as follows

```
mesValAlg :: ListFr (Op a) Int -> Int
mesValAlg Nil = 0
mesValAlg (Cons a acc) = 1 + acc
```

This is obtained by integrating `mesAlg` with the incremental update of the cost function `length`, where the update simply adds `1`, as each additional element increases the list's length by one.

Let `mesValCoalg = uneditCoalgR_pt`, the coalgebra is kept unmodified. We have the following program for solving the optimal value problem of the sequence alignment problem

```
mesVal :: Eq a => ([a],[a]) -> Int
mesVal = hyROF r mesValAlg mesValCoalg
 where
  r = (<=)
  mesValCoalg = uneditCoalgR_pt
```

Evaluating `mesVal ([1,2,3], [3,2,1])` gives us the length of the shortest editing sequence, which is `5`.

Similarly, we can also construct a program for the generation problem, which generates all possible edit sequences for a pair of input sequences, the algebra used in the generator is defined as

```
genesAlg :: Eq a => ListFr (Op a) [[Op a]] -> [[Op a]]
genesAlg Nil = [[]]
genesAlg (Cons a xs) = map (a:) xs
```

For the generation problem, it is not easy to define the selector relation `r` that is used to define the selector, as it involves introducing many concepts that will not be used elsewhere in the thesis. To simplify the discussion, define the selector for the generator problem as `sel r = concat`, the `concat :: [[a]] -> [a]`, which flattens a list of lists to a list by removing inner brackets,. This satisfies the type requirement in order to use program `hyROF_poly`. Using this, the generator for the sequence alignment problem is

```
genes :: Eq a => ([a],[a]) -> [[Op a]]
genes = hyROF_poly concat genesAlg  genesCoalg
 where genesCoalg = uneditCoalgR_pt
```

which generates all possible alignment sequences for a pair of sequences. For instance, evaluating `genes ([1,2,3], [3,2,1])` returns
`[[Del 1,Del 2,Mat 3,Ins 2,Ins 1],[Del 1,Ins 3,Mat 2,Del 3,Ins 1]..]`

II.2.11 Chapter discussion

This chapter presents a comprehensive exposition of several fundamental concepts in constructive algorithmics, as well as Bird and De Moor [1996]'s relational calculus of programs, with a particular emphasis on combinatorial optimization. Despite its conceptual simplicity and terseness, Bird and De Moor [1996]'s relational calculus is not widely used in practice, perhaps because most are more familiar with functions than relations. It is natural to ask whether this formalism can be expressed purely in functional terms. However, when it comes to program derivation, the relational formalism appears to be indispensable, as many specifications can only be accurately expressed through relations rather than functions.

Additionally, this chapter introduced various efficient catamorphism generators for both cons-list and join-list datatypes in Section II.2.3, covering various basic combinatorial structures. These generators constitute an extensive, readily applicable library for combinatorial optimization tasks involving such structures. The discussion of Part III demonstrates how these basic generators, combined with the principles for constructing complex combinatorial generators introduced in Subsection II.2.3.4, can aid in solving many NP-hard combinatorial machine learning problems. However, many of the combinatorial generators presented are based on existing work in the literature, and their derivations have largely been guided by observation and intuition. It remains an open question whether more sophisticated design principles can be formulated to derive these generators from relational specifications, as Bird and De Moor [1996] do for sublist, permutation, and partition generators over cons-list catamorphisms.

The position taken in this thesis is that the algorithm design framework proposed here is particularly important in the design of exact machine learning and other optimization algorithms, as these algorithms are typically applied to high-stakes problems where even small errors can lead to significant consequences. This framework allows for the derivation of algorithms through equational reasoning, ensuring programs are provably correct with straightforward and transparent reasoning steps. In other words, it not only enables the creation of correct programs but also provides simple proofs of their correctness. By contrast, formal proofs for, for example, classical BnB algorithms, often involve ad-hoc inductive steps, and proofs for MIP solvers require managing generative recursions, making termination proofs necessary and parallelization difficult.

II.3 Combinatorial geometry

This chapter examines geometric objects consisting of *finite hyperplanes* and *data points*. There are two key reasons motivating the study of these objects. Firstly, the combinatorics of these two kinds of geometric objects are well-studied and the combinatorial problems related to these two objects are central topics in combinatorial geometry. Secondly, most successful machine learning models, including ReLU neural networks and decision trees, are based on piecewise linear models (PWL). Understanding the geometry of finite hyperplanes will provide deeper insights into the essential combinatorial properties of these problems.

A dissection of $\mathbb{R}^D$ by a finite number of hyperplanes is called a *hyperplane arrangement*. Superficially, a hyperplane arrangement might seem to contain more information or structure than a set of data points (a point configuration). However, a valuable approach to studying geometric objects involving points and hyperplanes is to explore the transformations between these two objects. By studying the *dual transformation* between point configurations and hyperplane arrangements, it will later be seen that the superficial impression of the structural information contained in hyperplane arrangement and point configuration is incorrect. Both hyperplane arrangements and point configurations possess equally rich combinatorial structure.

The goal of this chapter is to demonstrate that applying geometric principles can provide new insights and interpretations into the combinatorics of various problems that involve finite hyperplanes and finite data points.

II.3.1 Foundations

Algebraic geometry is the study of the solutions of systems of polynomial equations and the geometric structures that these solutions form. It combines techniques from abstract algebra, particularly commutative algebra, in the context of geometry.

The primary objects of study in algebraic geometry are *algebraic varieties*, which are sets of solutions to polynomial equations. These polynomial equations can define simple curves or more complex structures like surfaces or higher-dimensional analogues. In the simplest case, the polynomial systems involve only *degree-one polynomials* (*linear functions*), which can be understood as hyperplanes, which will be the central focus of this discussion.

This section provides a brief introduction to the foundational geometric definitions that will be frequently used. Throughout the discussion, the focus is on Euclidean space over the real closed field $\mathbb{R}$.

II.3.1.1 Affine varieties and polynomials

Both the affine D-space, for $D \geq 1 \in \mathbb{N}$, and the vector D-space over $\mathbb{R}$ are denoted by $\mathbb{R}^D$. Both are the set of all D-tuples of elements of $\mathbb{R}$. To distinguish a vector in the vector space and a point in affine space, an element $\boldsymbol{x} = (x_1, x_2 \ldots, x_D) \in \mathbb{R}^D$ will be called a *point* in the affine space $\mathbb{R}^D$, an element $\boldsymbol{x} = (x_1, x_2 \ldots, x_D)^T \in \mathbb{R}^D$ is called a *vector* in the vector space $\mathbb{R}^D$, where x_i is called the *coordinate* of $\boldsymbol{x}$.

Definition 19. *Monomial.* A *monomial* with respect to a D-tuple $\boldsymbol{x} = (x_1, x_2 \ldots, x_D)$ is a product of

the form

$$M = \boldsymbol{x}^{\boldsymbol{\alpha}} = x_1^{\alpha_1} \cdot x_2^{\alpha_2} \ldots \cdot x_D^{\alpha_D}, \tag{76}$$

where $\boldsymbol{\alpha} = (\alpha_1, \alpha_2 \ldots, \alpha_D)$ and $\alpha_1, \alpha_2 \ldots, \alpha_D$ are nonnegative integers. The *total degree* of this monomial is the sum $|\boldsymbol{\alpha}| = \alpha_1 + \cdots + \alpha_n$. When $\boldsymbol{\alpha} = \mathbf{0} = (0, \ldots, 0)$, $\boldsymbol{x}^{\mathbf{0}} = 1$.

Definition 20. *Polynomial.* A polynomial P in $x_1, x_2 \ldots, x_D$ with coefficients in $\mathbb{R}$ is a finite linear combination (with coefficients in the field $\mathbb{R}$) of monomials. A polynomial $P(\boldsymbol{x})$, or P in short, will be given in the form

$$P(\boldsymbol{x}) = \sum_i w_i \boldsymbol{x}^{\boldsymbol{\alpha}_i}, w_i \in \mathbb{R}, \tag{77}$$

where i is finite. The set of all polynomials with variables $x_1, x_2 \ldots, x_D$ and coefficients in $\mathbb{R}$ is denoted by $\mathbb{R}[x_1, x_2 \ldots, x_D]$ or $\mathbb{R}[\boldsymbol{x}]$.

Let $P = \sum_i w_i \boldsymbol{x}^{\boldsymbol{\alpha}_i}$ be a polynomial in $\mathbb{R}[\boldsymbol{x}]$. Then $\boldsymbol{\alpha}_i$ is called the *coefficient* of the monomial $\boldsymbol{x}^{\boldsymbol{\alpha}_i}$. If $w_i \neq 0$, then $w_i \boldsymbol{x}^{\boldsymbol{\alpha}_i}$ is called a *term* of P. The *maximal degree* of P, denoted $deg(P)$, is the maximum $|\boldsymbol{\alpha}_i|$ such that the coefficient α_i is nonzero. For instance, the polynomial $P(\boldsymbol{x}) = 5x_1^2 + 3x_1x_2 + x_2^2 + x_1 + x_2 + 3$ for $\boldsymbol{x} \in \mathbb{R}^2$ has six terms and maximal degree two. Note that, since $|\boldsymbol{\alpha}_i|$ is defined as the sum of monomial degree, so polynomial $P'(\boldsymbol{x}) = 5x_1^2x_2^2 + 3x_1x_2$ has degree four.

The number of possible monomial terms of a degree K polynomial is equivalent to the number of ways of selecting K variables from the multisets of $D+1$ variables[20]. This is equivalent to the *size K combinations of $D+1$ elements taken with replacement*. In other words, selecting K variables from the variable set $(x_0, x_1, \ldots, x_D)$ in homogeneous coordinates with repetition, leads to the following fact.

Fact 3. If polynomial P in $\mathbb{R}[x_1, x_2 \ldots, x_D]$ has maximal degree K, then polynomial P has $\begin{pmatrix} D+K \\ D \end{pmatrix}$ monomial terms at most.

As an example, a polynomial in $\mathbb{R}[x_1, x_2]$ with maximal degree two (a two-dimensional conic section) has six terms at most (including the constant term).

Under addition and multiplication, $\mathbb{R}[x_1, x_2 \ldots, x_D]$ satisfies all of the field axioms except for the existence of multiplicative inverses, hence it is a *commutative ring*, called the *polynomial ring*.

Given polynomial $P \in \mathbb{R}[\boldsymbol{x}]$, the set of *zeros* of P is
$V(P) = \{(x_1, x_2 \ldots, x_D) \in \mathbb{R}^D : P(x_1, x_2 \ldots, x_D) = 0\}$. Geometrically, the zero set of a polynomial P determines a subset of $\mathbb{R}^D$, this subset is called a *surface* and it has dimensionality smaller than D and called a *hypersurface* if it has dimension $D-1$. Specifically, if this subset is an affine subspace defined by a *degree-one* polynomial, it is referred to as a *flat*. A $D-1$-dimensional affine flat is known as a *hyperplane*.

More generally, the *intersection* of the zero sets for a set of polynomials is equivalent to the intersection set of their corresponding surfaces in $\mathbb{R}^D$. This intersection set is called an *algebraic variety*. It is defined formally as follows.

Definition 21. *Algebraic variety.* Assume a set of polynomials $\mathcal{P} = \{P_i : i \in \mathcal{I}\}$ in $\mathbb{R}[x_1, x_2 \ldots, x_D]$, where $\mathcal{I} = \{1, 2, \ldots I\}$ is the index set for the polynomial. Then the set

[20] There are $D+1$ variables considering polynomials in homogeneous coordinates, i.e. *projective space* $\mathbb{P}^D$.

$$Var\left(\mathcal{P}\right)=\left\{\left(x_1,x_2\ldots,x_D\right)\in\mathbb{R}^D:P_i\left(x_1,x_2\ldots,x_D\right)=0,\,\forall i\in\mathcal{I}\right\},\tag{78}$$

is called the *affine variety* defined by $\mathcal{P}$.

In particular, the algebraic variety for one polynomial P_i is a surface denoted by $s_i=\left\{\boldsymbol{x}\in\mathbb{R}^D:P_i\left(\boldsymbol{x}\right)=0\right\}$, and the algebraic variety for a degree-one polynomial is a hyperplane, denoted as $h_i=\left\{\boldsymbol{x}\in\mathbb{R}^D:\boldsymbol{w}^T\boldsymbol{x}=c\right\}$.

II.3.1.2 Arrangements

This Subsection, reviews basic terminology and combinatorics of the arrangement of the *hypersurface* and *hyperplane*.

Definition 22. *Surface arrangement.* Let $\mathcal{P}=\{P_i:i\in\mathcal{I}\}$ by a system of polynomials. A finite *surface arrangement* $\mathcal{S}_{\mathcal{P}}=\{s_i:i\in\mathcal{I}\}$, where $s_i=\left\{\boldsymbol{x}\in\mathbb{R}^D:P_i\left(\boldsymbol{x}\right)=0\right\}$, is a finite set of surfaces defined by polynomial system $\mathcal{P}$. The system $\mathcal{P}$ is a *central* surface arrangement if the coefficient w for the zero degree term $w\boldsymbol{x}^{\mathbf{0}}$ equals zero.

In particular, for surfaces which are hyperplanes (with dimension $D-1$ and maximal degree-one) the definition of hyperplane arrangement applies.

Definition 23. *Hyperplane arrangement.* A finite *hyperplane arrangement* $\mathcal{H}=\{h_i:i\in\mathcal{I}\}$ is a finite set of hyperplanes $h_i=\left\{\boldsymbol{x}\in\mathbb{R}^D:\boldsymbol{w}_i^T\boldsymbol{x}=b_i\right\}$ in $\mathbb{R}^D$ for some constant $b_i\in\mathbb{R}$, where $\boldsymbol{w}$ is called the *normal vector* of hyperplane H_i. The the set $\mathcal{H}$ is a *central* hyperplane arrangement, if $b_i=0$ for all $i\in\mathcal{I}$.

In particular, a hyperplane $h=\left\{\boldsymbol{x}\in\mathbb{R}^D:\boldsymbol{w}^T\boldsymbol{x}=c\right\}$ is called an *affine hyperplane* if $c\neq 0$, and *linear hyperplane* if $c=0$.

A critically important concept in the study of arrangements is *general position*. This implies the geometric *general case situation*, as opposed to some more special or coincidental cases that are possible, referred to as *special position*.

A set of polynomials $\mathcal{P}=\{P_i\mid i\in\mathcal{I}\}$ in D variables is in *general position* if no $D+1$ polynomial has a common zero. In the special case of degree-one polynomials, the definition of general position for the hyperplane arrangement is given below.

Definition 24. *General position for hyperplanes.* A hyperplane arrangement $\mathcal{H}=\{h_i:i\in\mathcal{I}\}$ is in *general linear position* (or general position for short) if

$$\{h_1,...,h_k\}\in\mathcal{H},\;1\leq k\leq D\Longrightarrow\dim\left(h_1\cap\cdots\cap h_k\right)=D-k,\tag{79}$$

where $\dim\left(h_1\cap\cdots\cap h_k\right)$ is the *dimension* of the set $h_1\cap\cdots\cap h_k$ and

$$\{h_1\cap\cdots\cap h_k\}\in\mathcal{H},\;k>D\Longrightarrow h_1\cap\cdots\cap h_k=\{\emptyset\}.\tag{80}$$

A hyperplane arrangement $\mathcal{H}$ is in general position if the intersection of any k hyperplanes is contained in a $(D-k)$-dimensional *flat*, for $1\leq k\leq D$. For example, if $D=2$ then a set of lines is in general position if no two are parallel and no three meet at a point. A hyperplane arrangement in general position is also called a *simple* hyperplane arrangement. From a linear algebra perspective, a hyperplane arrangement $\mathcal{H}$ is simple if the normal vectors of any D hyperplanes in $\mathcal{H}$ are *linearly independent*.

Definition 25. *Sign vector and hyperplane arrangement.* Let $\mathcal{H} = \{h_i : i \in \mathcal{I}\}$ be a *hyperplane* arrangement. Denote by $h_i^+ = \left\{\boldsymbol{x} \in \mathbb{R}^D : \boldsymbol{w}_i^T \boldsymbol{x} > b_i\right\}$, $h_i^- = \left\{\boldsymbol{x} \in \mathbb{R}^D : \boldsymbol{w}_i^T \boldsymbol{x} < b_i\right\}$ the open sets "above" or "below" H_i. To each point $\boldsymbol{x} \in \mathbb{R}^D$ is associated the *sign vector* of data point $\boldsymbol{x}$ with respect to arrangement $\mathcal{H}$, denoted as $\text{sign}_{\mathcal{H}}(\boldsymbol{x}) = (\delta_1(\boldsymbol{x}), \delta_2(\boldsymbol{x}), \ldots, \delta_I(\boldsymbol{x}))$, where δ_i is given by

$$\delta_i(\boldsymbol{x}) = \begin{cases} +1 & \boldsymbol{x} \in h_i^+ \\ 0 & \boldsymbol{x} \in h_i \\ -1 & \boldsymbol{x} \in h_i^- \end{cases}, 1 \leq i \leq I. \tag{81}$$

Similarly, for the more general case, denote $s_i^+ = \left\{\boldsymbol{x} \in \mathbb{R}^D : P_i(\boldsymbol{x}) > 0\right\}$, $s_i^- = \left\{\boldsymbol{x} \in \mathbb{R}^D : P_i(\boldsymbol{x}) < 0\right\}$ as the open sets "above" or "below" hypersurface s_i. The sign vector of data point $\boldsymbol{x}$ with respect to hypersurface arrangement $\mathcal{S}_{\mathcal{P}}$ is denoted as $\text{sign}_{\mathcal{S}_{\mathcal{P}}}(\boldsymbol{x}) = (\delta_1(\boldsymbol{x}), \delta_2(\boldsymbol{x}), \ldots, \delta_I(\boldsymbol{x}))$, or just $\text{sign}_{\mathcal{S}}$ if the polynomial system $\mathcal{P}$ is clear from the context.

The main focus of this thesis is hyperplane arrangements. In particular, the thesis is interested in the *connected components* (regions) bounded by hyperplanes. These connected regions are the *faces* of an arrangement, defined as follows.

Definition 26. *Faces of a hyperplane arrangement.* Let $\mathcal{F}_{\mathcal{H}}$ be the set of all sign vectors $\text{sign}_{\mathcal{H}}(\boldsymbol{x})$ in $\mathbb{R}^D$ for arrangement $\mathcal{H}$, which is defined as

$$\mathcal{F}_{\mathcal{H}} = \left\{\text{sign}_{\mathcal{H}}(\boldsymbol{x}) : \boldsymbol{x} \in \mathbb{R}^D\right\}, \tag{82}$$

A *face* f (connected component) of an arrangement $f \subseteq \mathbb{R}^D$ is a maximal subset of $\mathbb{R}^D$, such that all $\boldsymbol{x} \in f$ have the same sign vector $\text{sign}_{\mathcal{H}}(\boldsymbol{x}) \in \mathcal{F}_{\mathcal{H}}$. Given a sign vector $\text{sign}_{\mathcal{H}}(\boldsymbol{x}) = (\delta_1(\boldsymbol{x}), \delta_2(\boldsymbol{x}), \ldots, \delta_I(\boldsymbol{x}))$, the connected region of f can be defined as $f = \bigcap_{i \in \mathcal{I}} h_i^{\delta_i(f)}$. In fact, f defines an *equivalence class* in $\mathbb{R}^D$. Since any point $\boldsymbol{x} \in f$ has the same sign vector, then $\text{sign}_{\mathcal{H}}(f)$ is *the* sign vector for any point in f.

Different vectors in $\mathcal{F}_{\mathcal{H}}$ define different faces of the arrangement $\mathcal{H}$. A face is said to be *k-dimensional* if it is contained in a *k-flat* for $-1 \leq k \leq D+1$. Some special faces are given the name *vertices* ($k = 0$), *edges* ($k = 1$), and *cells* ($k = D$). A k-face g and a $(k-1)$-face f are said to be *incident* if f is contained in the boundary of face g, for $1 \leq k \leq D$. In that case, face g is called a *superface* of f, and f is called a *subface* of g.

It is assumed that all flats or hyperplanes considered in this thesis are *non-vertical*; a flat is called *vertical* if it contains a line *parallel* to the *axis*.

II.3.1.3 The combinatorial complexity of arrangements

Before studying the theory of arrangements, it is necessary to investigate the combinatorial complexity of hyperplane and hypersurface arrangements and these results will become useful when analyzing the combinatorial complexity of the problem.

Theorem 9. *Face counting theorem for hyperplanes.* Let $\mathcal{H} = \{h_n : n \in \mathcal{N}\}$ be a hyperplane arrangement in $\mathbb{R}^D$. The number of k-dimensional faces in the arrangement $\mathcal{H}$, for $1 \leq k \leq D$, is

$$F_D(\mathcal{H}) \leq \sum_{i=0}^{k} \binom{D-i}{k-i} \binom{N}{D-i}. \tag{83}$$

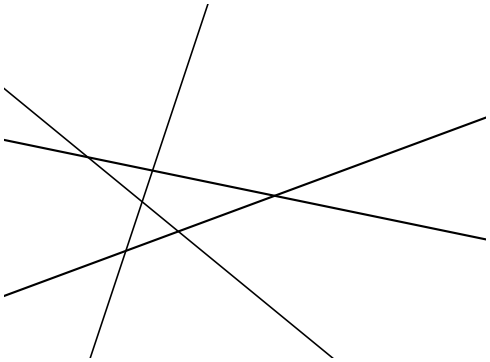


Figure 11: An arrangement of four lines in $\mathbb{R}^2$. Four lines intersect such that no three lines meet at a single point, so this arrangement is in general position. This arrangement divides the plane into three closed regions (bounded cells) and eight open regions (unbounded cells).

The maximum is obtained when $\mathcal{H}$ is simple [Toth et al., 2017].

A special case of (83) occurs when $k = D$, which obtains the following lemma for counting the number of cells of an arrangement.

Lemma 5. *Cell counting formula.* Let $\mathcal{H} = \{h_n \mid n \in \mathcal{N}\}$ be a *non-central* and simple arrangement in $\mathbb{R}^D$. The number of cells (D-dimensional faces) in the arrangement $\mathcal{H}$ is

$$C_D(\mathcal{H}) \leq \sum_{d=0}^{D} \binom{N}{d}. \tag{84}$$

For any *central* arrangement $\mathcal{H}$ of N hyperplanes in $\mathbb{R}^D$, the number of cells (D-dimensional faces) in the arrangement $\mathcal{H}$ is

$$C_D(\mathcal{H}) \leq 2\sum_{d=0}^{D-1} \binom{N-1}{d}. \tag{85}$$

The maximum is attained exactly when $\mathcal{H}$ is simple [Fukuda, 2016].

For instance, the non-central arrangement of four lines shown in Fig. 11 has 11 cells.

The cell counting formula for central arrangements (85) is, in fact, equivalent to *Cover's dichotomy counting formula* [Cover, 1965], which is well-known in machine learning communities. This formula states that, given a data set $\mathcal{D}$ of size N in general position in $\mathbb{R}^D$, the number of *dichotomies* (linearly separable predictions by **affine** hyperplanes) is given by:

$$\text{Cover}(N, D+1) = 2\sum_{d=0}^{D} \binom{N-1}{d} \tag{86}$$

Note that in this summation, the upper limit is D rather than $D-1$, as occurs in (85). This difference arises because a dataset in general position in $\mathbb{R}^D$ is isomorphic to a central arrangement in $\mathbb{R}^{D+1}$. The relationship between Cover's dichotomy counting formula and the cell counting formula will be revisited in the discussion on point-hyperplane duality.

The cells in an arrangement can be further split into two classes, the *bounded cells* and *unbounded cells*. Informally, a cell is bounded if it is a closed region surrounded by hyperplanes (the boundaries are not contained in cells), and unbounded otherwise. In particular, given a fixed number of hyperplanes in

$\mathbb{R}^D$, the number of bounded and unbounded cells is fixed, the number of bounded cells can be computed using the following Lemma [Stanley et al., 2004].

Lemma 6. Let $\mathcal{H} = \{h_n \mid n \in \mathcal{N}\}$ be a hyperplane arrangement in $\mathbb{R}^D$. The number of bounded cells in the arrangement $\mathcal{H}$ is

$$B_D(\mathcal{H}) \leq \begin{pmatrix} N-1 \\ D \end{pmatrix}. \tag{87}$$

The maximum is attained exactly when $\mathcal{H}$ is simple.

As depicted in Fig. 11, in a simple arrangement of four lines, the number of bounded cells is three.

Theorem 10. *Asymptotic complexity for hypersurface arrangements.* Given an arrangement of surfaces $\mathcal{S}_\mathcal{P} = \{s_n : n \in \mathcal{N}\}$ in $\mathbb{R}^D$ defined by polynomial $\mathcal{P} = \{P_n : n \in \mathcal{N}\}$, as defined above, the maximum combinatorial complexity of the arrangement $\mathcal{S}_\mathcal{P}$ is $O\left(N^D\right)$. There are such arrangements whose complexity is $\Theta\left(N^D\right)$. The constant of proportionality in these bounds depends on D and on the maximal degree of the polynomials $\mathcal{P}$[21]. The asymptotically highest complexity is obtained when the surfaces are in general positions [Sharir, 1994].

II.3.1.4 Points and hyperplane duality

The concept of duality in geometry establishes a profound relationship between points and hyperplanes, revealing that they are in one-to-one correspondence. By constructing a *dual transformation*, points can be systematically mapped to hyperplanes and vice versa, while preserving *incidence relations* within them.

In an affine space, a *point configuration* is a set of data points $\boldsymbol{p}_n \in \mathbb{R}^D$. When fixing an origin, a point configuration becomes a set of vectors, which is equivalent to the data set defined previously, thus a point configuration is denoted by $\mathcal{D} = \{\boldsymbol{p}_n : n \in \mathcal{N}\}$. A vector space is a set with linear structure, but point configuration is just a finite set.

A point configuration also has a definition for *general position*: a set of points in D-dimensional affine space is in general position if no k of them lie in a $(k-2)$-dimensional affine subspace of $\mathbb{R}^D$, for $k = 2, 3, ..., D+1$. As will be explained shortly, this definition is equivalent to that of general position for hyperplane arrangements.

The dual transformations between a hyperplane and a point are established in the following definition.

Definition 27. *Duality for affine arrangement.* The geometric *dual transformation* $\phi : \mathbb{R}^D \to \mathbb{R}^D$ maps a point $\boldsymbol{p}$ to a non-vertical affine hyperplane $\phi(\boldsymbol{p})$, defined by the equation

$$p_1 x_1 + p_2 x_2 + ... + p_{D-1} x_{D-1} - x_D = p_D, \tag{88}$$

and conversely, the function ϕ^{-1} transforms a (non-vertical) hyperplane h defined by polynomial $w_1 x_1 + w_2 x_2 + ... + w_{D-1} x_{D-1} - x_D = w_D$ to a point $\phi^{-1}(h) = (w_1, w_2, \ldots, w_D)^T$.

The terms *primal space*, and *dual space* refer to the spaces before and after transformation by ϕ and ϕ^{-1}. The dual transformation is naturally extended to a set of points $\phi(\mathcal{D})$ and a set of hyperplanes $\phi(\mathcal{H})$ by applying it to all points and hyperplanes in the set. Sometimes, the dual hyperplane arrangement of points set $\mathcal{D}$ is denoted as $\mathcal{H}_\mathcal{D}$ and the dual point configuration of $\mathcal{H}$ as $\mathcal{D}_\mathcal{H}$.

[21] However, if the maximal degree of the $\mathcal{P}$ is very high, there will be a very large constant term in the exponent hidden in the big O notation.

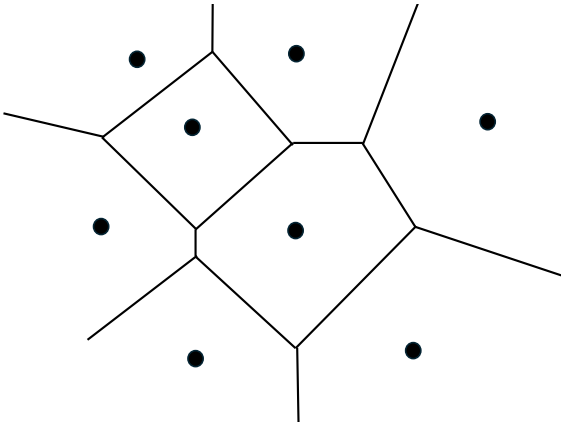


Figure 12: A (Euclidean) Voronoi diagram in $\mathbb{R}^2$. The black points represent the *centroids* of each Voronoi cell (connected regions in $\mathbb{R}^2$), and the black lines denote the boundaries of the connected regions.

Theorem 11. *Incidence relations of the dual transformation.* Let $\boldsymbol{p}$ be a point and a non-vertical affine hyperplane $h = \left\{\boldsymbol{x} : \boldsymbol{w}^T\boldsymbol{x} = 0\right\}$ in $\mathbb{R}^D$. Under the dual transformation ϕ, $\boldsymbol{p}$ and H satisfy the following properties:

1. *Incidence preservation*: Point $\boldsymbol{p}$ belongs to hyperplane h if and only if point $\phi^{-1}(h)$ belongs to hyperplane $\phi(\boldsymbol{p}) = p$,

2. *Order preservation*: Point $\boldsymbol{p}$ lies above (below) hyperplane h if and only if point $\phi^{-1}(h)$ lies above (below) hyperplane $\phi(\boldsymbol{p})$.

That the dual transformation preserves the incidence relations above can be proved by examining the relationship between the dual transformation ϕ and the *unit paraboloid* [Edelsbrunner, 1987].

The incidence preservation property described above implies a duality between the definitions of general position for point configurations and hyperplane arrangements. For instance, when $D = 2$, three points lying in the same 1-flat l (a line) correspond to three lines in the dual space intersecting at the same point $\phi(l)$, these three lines are mutually parallel if the line l is vertical.

II.3.1.5 Voronoi diagrams

The *Voronoi diagram* of a finite set of objects is another important geometric structure in machine learning, combinatorial geometry and many fields of computer science. Applications of Voronoi diagrams are concerned specifically with problems involving the "closeness" of points in a finite set. A number of seemingly unrelated problems involving the proximity of N points, such as finding a Euclidean minimum spanning tree, the smallest circle enclosing the set, K-nearest and farthest neighbors, the two closest points, and a proper straight-line triangulation [Shamos and Hoey, 1975], can be solved efficiently by using Voronoi diagrams.

The Voronoi diagram subdivides the embedding space into regions, each region consisting of the points that are closer to a given object than to the others. This closeness is defined by the following distance functions.

Definition 28. *Distance functions.* The distance between two points $\boldsymbol{x} = (x_1, x_2, ..., x_D)^T, \boldsymbol{y} = (y_1, y_2, ..., y_D)^T \in \mathbb{R}^D$, is denoted by $d(\boldsymbol{x}, \boldsymbol{y})$ and it must satisfy the following properties:

1. Coincidence: $d(\boldsymbol{x}, \boldsymbol{x}) = 0$,

2. Triangle inequality: $d(\boldsymbol{x}, \boldsymbol{y}) \leq d(\boldsymbol{x}, \boldsymbol{x}') + d(\boldsymbol{x}', \boldsymbol{y}), \forall \boldsymbol{x}, \boldsymbol{x}', \boldsymbol{y} \in \mathcal{D}$.

Note that, some textbooks might also include the *symmetry* and *non-negative* properties, but these two properties can be derived from the above two properties [Gower and Legendre, 1986]. The well-known L_p metric is defined as

$$d_p(\boldsymbol{x}, \boldsymbol{y}) = \left(\sum_{i=1}^{D} |y_i - x_i| \, |^p \right)^{1/p}. \tag{89}$$

A special case of this metric is the *Euclidean distance* or L_2 *distance* is defined as

$$d_2(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|_2^2 = \|\boldsymbol{y} - \boldsymbol{x}\|_2^2 = \sqrt{\sum_{i=1}^{D} (y_i - x_i)^2}, \tag{90}$$

and similarly the L_1 *distance*

$$d_1(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{y} - \boldsymbol{x}\|_1 = |\boldsymbol{y} - \boldsymbol{x}| = |\boldsymbol{y} - \boldsymbol{x}| = \sum_{i=1}^{D} |y_i - x_i|. \tag{91}$$

Definition 29. *Voronoi diagram.* Given a set of centroids $\mathcal{D} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$ in $\mathbb{R}^D$, the Voronoi diagram $\mathcal{V}(\mathcal{D})$ is defined by a set of regions $\mathcal{V}(\mathcal{D}) = \{V_1, V_2, \ldots, V_N\}$, where V_n, $n \in \mathcal{N}$ is a subspace of $\mathbb{R}^D$ which consists of all points closer to centroids $\boldsymbol{x}_n$ than any other centroids in $\mathcal{D}$. In other words, the data set $\mathcal{D}$ *partitions* the ambient space $\mathbb{R}^D$ into *Voronoi regions/polygons* $\mathcal{V}(\mathcal{D}) = \{V_1, V_2, \ldots, V_N\}$ defined by

$$V_n = \left\{ \boldsymbol{x} \in \mathbb{R}^D \mid d(\boldsymbol{x} - \boldsymbol{x}_n)^2 \leq d(\boldsymbol{x} - \boldsymbol{x}_j)^2, \forall n, j \in \mathcal{N} \wedge k \neq j \right\}. \tag{92}$$

It is safe to assume no data lies on the boundaries of any two adjacent Voronoi regions, and ignore the equality in the below discussion.

The Voronoi diagram/partition defined on Euclidean distance is called the *Euclidean Voronoi diagram*. An example of a Euclidean Voronoi diagram is depicted in Fig. 12. It is possible to define variants of Voronoi diagrams depending on the class of objects, the distance function and the embedding space. For instance, when the distance function in a Voronoi diagram is defined by a *Bregman divergence* [Banerjee et al., 2005], it is referred to as a *Bregman Voronoi diagram*. This thesis, however, will focus on investigating the Euclidean Voronoi diagram in more detail.

II.3.2 Classification problems and duality

The *classification* problem in machine learning, particularly the *linear classification problem*, is one central theme of machine learning. The effectiveness of a linear classifier depends upon the algorithm's ability to construct a good *linear model* (*linear decision boundary*) for prediction. Given a set of data, each linear model corresponds to a specific *linear dichotomy* (linearly separable predictions), determined by the *incidence relations* between the data points and the *decision hyperplane*. Importantly, this incidence relation can be analyzed through the dual transformation ϕ.

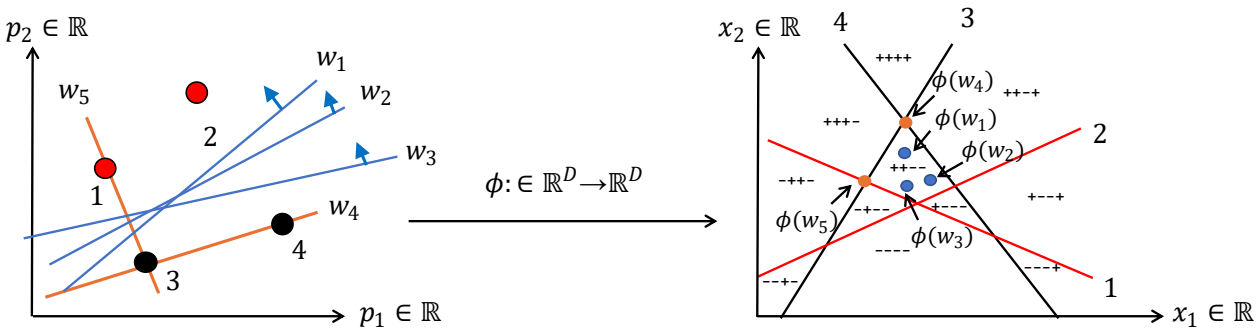


Figure 13: A point configuration $\mathcal{D}$ (left-panel) and its dual arrangement $\mathcal{H}_{\mathcal{D}}$ (right-panel). The yellow hyperplanes w_4, w_5 with two points lying on them in $\mathbb{R}^D$ correspond to the yellow points in the dual space, which are the intersection of corresponding dual hyperplanes $\phi(w_4)$, $\phi(w_5)$. For (blue) hyperplanes w_1, w_2, w_3 with the same prediction labels $(+,+,-,-)$, their corresponding dual points $\phi(w_1)$, $\phi(w_2)$, $\phi(w_3)$ lie in the same cell of dual arrangement $\phi(\mathcal{D})$.

This section will first explore the geometric relationships between points, hyperplanes, and dichotomies, focusing on their *combinatorial complexity* and *incidence relations*. By analyzing the incidence relations between data points and hyperplanes, a different perspective on the linear classification problem is attained. This perspective can facilitate the development of an efficient and general algorithm capable of solving linear classification problems with arbitrary objectives. Then, the dicussion will be generalized to *non-linear* (*polynomial hypersurface*) classification problems.

II.3.2.1 Linear classification and duality

As introduced above, the combinatorial complexity of linear dichotomies with respect to data set $\mathcal{D}$ is given by (86). However, Cover's theorem only provides insight into the combinatorial complexity of possible dichotomies. How to *enumerate* these dichotomies remains an open problem.

In previous studies by the author and others [He and Little, 2023b], it was shown that the possible dichotomies with respect to a given data set $\mathcal{D}$ can be equivalently obtained by enumerating the cells of the dual arrangement $\mathcal{H}_{\mathcal{D}}$. This result is a consequence of Thm. 11. The *order preservation property* of the dual transformation ϕ reveals a *topological equivalence* between the dual space and the primal space. It can be difficult to visualize how Cover's dichotomies form equivalence classes for decision hyperplanes, but the same decision hyperplanes in the dual space $\phi(\boldsymbol{p})$, $\forall \boldsymbol{p} \in \mathbb{R}^D$ partition the space into different cells, where each cell corresponds to an equivalence class of dichotomies (Fig. 13).

This Subsection begins by analyzing how the combinatorial complexity of cells—both bounded and unbounded—relates to dichotomies. This will offer insights into how these concepts are interconnected and also provide a counting argument for the enumeration algorithms designed later in the thesis.

It has been previously explained that the possible dichotomies for data set in $\mathbb{R}^D$ is equivalent to the number of cells of a *central arrangement* in $\mathbb{R}^{D+1}$. Next, the correspondence between Cover's dichotomies and the cells of a dual arrangement $\mathcal{H}_{\mathcal{D}}$ is explained, which offers an alternative approach to proving

Cover's counting theorem.

Lemma 7. For a set points $\mathcal{D} = \left\{x_n \in \mathbb{R}^D : n \in \mathcal{N}\right\}$ in general position, the total number of linear dichotomies in Cover's function counting theorem, is the same as the number of cells of the dual arrangement $\mathcal{H}_{\mathcal{D}}$, plus the number of bounded cells of $\mathcal{H}_{\mathcal{D}}$. In other words, the number of dichotomies is equivalent to the number of unbounded cells plus twice of the number of bounded cells.

Proof. Given a set of points $\mathcal{D} = \left\{x_n \in \mathbb{R}^D : n \in \mathcal{N}\right\}$ in general position. Cover, 1965's function counting theorem states that the number of linearly separable dichotomies given by affine hyperplanes is

$$\text{Cover}\left(N, D+1\right) = 2\sum_{d=0}^{D}\left(\begin{array}{c} N-1 \\ d \end{array}\right). \tag{93}$$

The original Cover's function counting theorem counts the number of linearly separable dichotomies given by *linear* hyperplanes. However, the dual arrangement $\mathcal{H}_{\mathcal{D}}$ consists of a set of *affine* hyperplanes. Nevertheless, the number of dichotomies given by affine hyperplanes in $\mathbb{R}^D$ for data set $\mathcal{D}$ is equivalent to the number of dichotomies given by linear hyperplanes for data set $\bar{\mathcal{D}}$ in $\mathbb{R}^{D+1}$ (where $\bar{\mathcal{D}}$ is the *homogeneous dataset*, which is obtained by embedding $\mathcal{D}$ in homogeneous space. Recall that, $\bar{\boldsymbol{x}} = (\boldsymbol{x}, 1)$ is the data in homogeneous coordinates).

Subsection II.3.1.3 shows that for a simple arrangement $\mathcal{H} = \{h_n : n \in \mathcal{N}\}$ in $\mathbb{R}^D$, the number of cells is $C_D\left(\mathcal{H}\right) = \sum_{d=0}^{D}\left(\begin{array}{c} N \\ d \end{array}\right)$, and the number of bounded regions is $B_D\left(\mathcal{H}\right) = \left(\begin{array}{c} N-1 \\ D \end{array}\right)$.

Putting these two pieces of information together, obtains

$$\begin{aligned}
B_D\left(\mathcal{H}_{\mathcal{D}}\right) + C_D\left(\mathcal{H}_{\mathcal{D}}\right) =& \left(\begin{array}{c} N-1 \\ D \end{array}\right) + \sum_{d=0}^{D}\left(\begin{array}{c} N \\ d \end{array}\right) \\
=& \left(\begin{array}{c} N-1 \\ D \end{array}\right) + \sum_{d=0}^{D}\left[\left(\begin{array}{c} N-1 \\ d \end{array}\right) + \left(\begin{array}{c} N-1 \\ d-1 \end{array}\right)\right] \\
=& \sum_{d=0}^{D}\left(\begin{array}{c} N-1 \\ d \end{array}\right) + \sum_{d=0}^{D}\left(\begin{array}{c} N-1 \\ d \end{array}\right) \\
=& 2\sum_{d=0}^{D}\left(\begin{array}{c} N-1 \\ d \end{array}\right) \\
=& \text{Cover}\left(N, D+1\right).
\end{aligned} \tag{94}$$

□

The equivalence between the number of dichotomies and the sum of the number of bounded cells and the number of cells may initially seem unclear. The intuition lies in the fact that not every dichotomy in the primal space corresponds to a cell in the dual space. Specifically, decision boundaries associated with unbounded cells correspond to two dichotomies, whereas those associated with bounded cells correspond to only one. This relationship is clarified by the following lemma.

Lemma 8. For a data set $\mathcal{D}$ in general position, each of Cover's dichotomies corresponds to a cell in the dual space, and dichotomies corresponding to bounded cells have no *complement cell* (cells with reverse sign vector). Dichotomies corresponding to the unbounded cells in the dual arrangements $\phi\left(\mathcal{D}\right)$ have a complement cell.

Proof. The first statement is true because of the order preservation property ⁻data item $\boldsymbol{x}$ lies above (below) hyperplane h if and only if point $\phi^{-1}(h)$ lies above (below) hyperplane $\phi(\boldsymbol{x})$. For a data set $\mathcal{D}$ and hyperplane h, assume h has a normal vector $\boldsymbol{w}$ (in homogeneous coordinates) and there is no data item lying on h. Then, hyperplane h will partition the set $\mathcal{D}$ into two subsets $\mathcal{D}_h^+ = \{\boldsymbol{x}_n : \boldsymbol{w}^T\boldsymbol{x} > 0\}$ and $\mathcal{D}_h^- = \{\boldsymbol{x}_n : \boldsymbol{w}^T\boldsymbol{x} < 0\}$, and according to the Thm. 11, $\mathcal{D}$ has a unique associated dual arrangement $\phi(\mathcal{D})$. Thus, the sign vector of the point $\phi^{-1}(h)$ with respect to arrangement $\phi(\mathcal{D})$ partitions the arrangement into two subsets $\phi_h(\mathcal{D})^+ = \left\{\phi(\boldsymbol{x}_n) : \boldsymbol{\nu}_{\phi(\boldsymbol{x}_n)}^T \phi^{-1}(h) > 0\right\}$ and $\phi_h(\mathcal{D})^- = \left\{\boldsymbol{\nu}_{\phi(\boldsymbol{x}_n)}^T \phi^{-1}(h) < 0\right\}$, where $\boldsymbol{\nu}_{\phi(\boldsymbol{x}_n)}^T$ is the normal vector to the dual hyperplane $\phi(\boldsymbol{x}_n)$, in other words, point $\phi^{-1}(h)$ lies in a cell of arrangement $\phi(\mathcal{D})$.

Next, it is necessary to prove that bounded cells have no complement cell. The reverse assignment of the bounded cells of the dual arrangements $\phi(\mathcal{D})$ cannot appear in the primal space since the transformation ϕ can only have normal vector $\boldsymbol{\nu}$ pointing in one direction, in other words, transformation ϕ: $x_D = p_1x_1 + p_2x_2 + ... + p_{D-1}x_{D-1} - p_D$ implies the Dth component of normal vector $\boldsymbol{\nu}$ is -1. For unbounded cells, in dual space, every unbounded cell f associates with another cell g, such that g has an opposite sign vector to f. This is because every hyperplane $\phi(\boldsymbol{x}_n)$ is cut by another $N-1$ hyperplanes into $N+1$ pieces (since in a simple arrangement no two hyperplanes are parallel), and each of the hyperplanes contains two rays, call them $\boldsymbol{r}_1$,$\boldsymbol{r}_2$. These two rays point in opposite directions, which means that the cell incident with $\boldsymbol{r}_1$ has an opposite sign vector to $\boldsymbol{r}_2$ with respect to all other $N-1$ hyperplanes. Therefore, it is only necessary to take the cell f incident with $\boldsymbol{r}_1$, and in the positive direction with respect to $\phi(\boldsymbol{x}_n)$, take g to be the cell incident with $\boldsymbol{r}_2$, and in the negative direction with respect to $\phi(\boldsymbol{x}_n)$. In this way, two unbounded cells f and g are obtained with opposite sign vectors. This means that, for point $\phi^{-1}(h)$ in these unbounded cells, this hyperplane h partitions the data set to $\mathcal{D}_h^+$ and $\mathcal{D}_h^-$, and it is possible to move the position of hyperplane h in the primal space. So, there exists a new hyperplane h' obtained by moving h, and it partitions the data set to $\mathcal{D}_{h'}^+ = \mathcal{D}_h^-$ and $\mathcal{D}_{h'}^- = \mathcal{D}_h^+$. In other words, h' has opposite assignment compared to hyperplane h. This corresponds, in the dual space, to moving a point $\phi^{-1}(h)$ inside the cell f, to cell g. For instance, in the simplest case, a hyperplane can be moved from left-most to the right-most to obtain an opposite assignment without changing the direction of the normal vector. □

Since each of Cover's dichotomies corresponds to a cell in the dual space, and dichotomies corresponding to bounded cells have no complement cell (cells with reverse sign vector), lemma 8 demonstrates that all possible *Cover's dichotomies* of a given data set $\mathcal{D}$ can be obtained by enumerating the cells of an arrangement and the complemented cells of the bounded cells. The enumeration of the complements of the bounded cells requires an additional process, as the bounded cells within the arrangement do not have complementary cells. This result directly leads to the following theorem.

Theorem 12. *Linear classification theorem.* Let $\mathcal{D}$ be a data set in general position in $\mathbb{R}^D$. If an $O\left(N^D\right)$-time cell enumeration algorithm exists, then exact solutions for the linear classification problem with an arbitrary objective function can be obtained in at most $O\left(t_{\text{eval}} \times N^D\right)$ time by exhaustively enumerating the cells of the dual arrangement $\mathcal{H}_{\mathcal{D}}$, where t_{eval} represents the time required to evaluate the classification objective.

The next lemma explains not only that Cover's dichotomies have corresponding dual cells for the dual hyperplane arrangement, but also that hyperplanes containing $0 \leq k \leq D$ data points have corresponding

dual faces.

Lemma 9. For a data set $\mathcal{D}$ in general position, a hyperplane with k data items lying on it, $0 \leq k \leq D$ correspond to a $(D-k)$-face in the dual arrangement $\mathcal{H}_{\mathcal{D}}$. Hyperplanes with D points lying on it, correspond to vertices in the dual arrangement.

Proof. According to the incidence preservation property, k data items lying on a hyperplane will intersect with k hyperplanes, and the intersection of k hyperplanes will create a $(D-k)$-dimensional space, which is a $(D-k)$-face, and the 0-faces are the *vertices* of the arrangement. □

Definition 30. *Separation set.* Given a hyperplane arrangement $\mathcal{H} = \{h_n : n \in \mathcal{N}\}$. The separation set $sep(f, g)$ for two faces f, g is defined by

$$\text{sep}(f, g) = \{n \in \mathcal{N} : \delta_n(f) = -\delta_n(g) \neq 0\}, \tag{95}$$

using which, we say that the two faces f, g are *conformal* if $\text{sep}(f, g) = \emptyset$.

That two faces that are conformal is essentially the same thing as saying that two faces have consistent classification assignments.

Lemma 10. Given a hyperplane arrangement $\mathcal{H} = \{h_n : n \in \mathcal{N}\}$, two faces f, g are conformal if and only if f and g are subfaces of a common face or one face is a subface of the other.

A similar result is described in *oriented matroid* theory [Björner, 1999].

The following lemma will be instrumental in the analysis, presented later, of the linear classification problem with the *0-1 loss* objective. It suggests that the optimal cell, with respect to 0-1 loss, is conformal to the optimal vertex.

Lemma 11. Given a hyperplane arrangement $\mathcal{H} = \{h_n : n \in \mathcal{N}\}$, for an arbitrary maximal face (cell) f, the sign vector of f is $\text{sign}_{\mathcal{H}}(f)$. For an arbitrary $(D-d)$-dimension face g, $0 < d \leq D$, the number of different signs of $\text{sign}_{\mathcal{H}}(g)$ with respect to $\text{sign}_{\mathcal{H}}(f)$ is larger than or equal to d, where equality holds only when g is conformal to f (g is a subface of f).

Proof. Denote the number of different signs of $\text{sign}_{\mathcal{H}}(g)$ with respect to $\text{sign}_{\mathcal{H}}(f)$ by $E_{0\text{-}1}(g)$. In a simple arrangement, the sign vector $\text{sign}_{\mathcal{H}}(f)$ of a cell f has no zero signs, and a $(D-d)$-dimension face has d zero signs. Thus the number of different signs of $\text{sign}_{\mathcal{H}}(g)$ with respect to $\text{sign}_{\mathcal{H}}(f)$ must be larger than or equal to d, i.e., $E_{0\text{-}1}(f) \geq d$. If $\text{sep}(f, g) = \emptyset$, then $E_{0\text{-}1}(g) = d$ according to the definition of $\text{sep}(f, g) = \emptyset$. In this case, f, g are conformal. By contrast, if f, g are not conformal, i.e. $\text{sep}(f, g) \neq \emptyset$, and assuming $|\text{sep}(f, g)| = C$, then according to the definition of the objective function and conformal faces, $E_{0\text{-}1}(\text{sign}_{\mathcal{H}}(f)) = d + C$. Hence, $E_{0\text{-}1}(\text{sign}_{\mathcal{H}}(g)) \geq d$, and equality holds only when g is conformal to f. □

In the linear classification problem with the 0-1 loss objective, Lemma 11 will later be used to prove Thm. 15. This theorem states that the optimal decision boundary is adjacent to the decision boundary that includes D data points, and that the optimal classification predictions are consistent with the predictions of this optimal boundary with D points lying on it. Since there are only $\begin{pmatrix} N \\ D \end{pmatrix}$ such hyperplanes, it is possible to solve the 0-1 loss linear classification problem in polynomial time.

II.3.2.2 Growth function and the complexity of the classification problem

In the study of *foundational machine learning theory*, known as *statistical learning theory*, a critically important concept is the *growth function*, which measures the complexity of a given *hypothesis set*. Recall that a hypothesis set is a set of functions mapping data set $\mathcal{D}$ to the set of predicted labels; for linear classification problem, the hypothesis set is just the set of all decision hyperplanes in the feature space.

Definition 31. *Growth function.* The growth function $\Pi_{\mathbb{H}} : \mathbb{N} \to \mathbb{N}$ for a hypothesis set $\mathbb{H}$ is defined by

$$\forall N \in \mathbb{N}, \Pi_{\mathbb{H}}(N) = \max_{\{\boldsymbol{x}_n : n \in \mathcal{N}\}} \left| \left\{ \operatorname{sign}\left(\boldsymbol{w}^T \bar{\boldsymbol{x}}_n\right) : \boldsymbol{w} \in \mathbb{R}^{D+1} \right\} \right|, \tag{96}$$

In other words, $\Pi_{\mathbb{H}}(N)$ is the maximum number of distinct ways in which N points can be classified using hypotheses in $\mathbb{H}$, i.e. the *number of dichotomies* which can be realized by the hypothesis set.

The next result, known as *Sauer's lemma* [Sauer, 1972], clarifies the connection between the notions of growth function and *VC-dimension* [Mohri et al., 2018]. The VC-dimension is a widely-used measure of the complexity of a classification model. The simple D-dimensional *linear hyperplane* classification model, which is discussed in detail below, has VC-dimension $D+1$. This is lower than that of other widely used models, such as the decision tree model (axis-parallel hyper-rectangles), which has a VC-dimension $2D$; the K-degree polynomial has a VC-dimension $O\left(D^K\right)$; and the L-layer, W-weight piecewise linear deep neural network, has VC-dimension of $O(WL\log(W))$ [Blumer et al., 1989, Bartlett et al., 2019, Vapnik, 1999].

Lemma 12. *Sauer's lemma.* Let $\mathbb{H}$ be a hypothesis set with $\operatorname{VCdim}(\mathbb{H}) = D$. Then, for all $N \in \mathbb{N}$, the following inequality holds

$$\Pi_{\mathbb{H}}(N) \leq \sum_{d=0}^{D} \binom{N}{d}. \tag{97}$$

It is clear that when $\operatorname{VCdim}(\mathbb{H}) = D+1$, for N data points in D-dimensional space, Cover's functional counting theorem satisfies the above inequality

$$\begin{aligned} \operatorname{Cover}(N, D+1) =& 2\sum_{d=0}^{D} \binom{N-1}{d} \\ =& \sum_{d=0}^{D} \binom{N-1}{d} + \sum_{d=0}^{D} \binom{N-1}{d} \\ \leq& \sum_{d=0}^{D} \binom{N-1}{d+1} + \sum_{d=0}^{D} \binom{N-1}{d} \\ =& \sum_{d=0}^{D+1} \binom{N}{d+1}. \end{aligned} \tag{98}$$

Since the right-hand side of the (97) is always *polynomially* large when the VC-dimension of the hypothesis set is finite, this says that it is always possible to construct a polynomial-time, exact classification algorithm for any hypothesis set with finite VC-dimension.

In the next section, it will be shown that a polynomial hypersurface is isomorphic to a hyperplane in a higher-dimensional space. This will enable a more precise analysis of the combinatorial complexity

of hypersurfaces. Furthermore, the isomorphism between hyperplanes and hypersurfaces allows extension of Thm. 12 from linear classification problems to polynomial hypersurface classification problems. This extension enables the construction of an algorithm that can find an optimal non-linear (polynomial hypersurface) classifier in polynomial time.

II.3.2.3 Non-linear (polynomial) classification and the Veronese embedding

Based on the point-hyperplane duality, equivalence relations for linear classifiers on finite sets of data were established above. However, a linear classifier is often too restrictive in practice, as many problems require more complex decision boundaries. It is natural to ask whether it is possible to extend the theory to non-linear classification. This section explore a well-known concept in algebraic geometry, the *W-tuple Veronese embedding*, which allows the generalization of the previous strategy for solving classification problem with *hyperplane classifier* to problems involving *hypersurface classifiers*, with a worst-case time complexity $O\left(N^G\right)$, where $G = \begin{pmatrix} D+W \\ D \end{pmatrix} - 1$, and W is the degree of the polynomial defining the hypersurface. If both W and D are fixed constants, this again gives a polynomial-time algorithm for solving the 0-1 loss hypersurface classification problem.

In algebraic geometry, an *embedding* is a *morphism* ρ of an algebraic variety V, such that the variety V is isomorphic to its image $\rho(V)$. In Exercise 2.12 and Exercise 3.4, Chapter I of Hartshorne [1977], the following embedding in projective space is demonstrated.

Definition 32. *The W-tuple Veronese embedding.* Given variables $x_0, x_1, \ldots x_D$ in projective space $\mathbb{P}^D$ (which is isomorphic to the affine space $\mathbb{R}^D$ when ignoring the points at infinity [Cox et al., 1997]), let $M_0, M_1, \ldots M_G$ be all monomials of degree W with variables $x_0, x_1, \ldots x_D$, where $G = \begin{pmatrix} D+W \\ D \end{pmatrix} - 1$. Define a mapping $\rho_W : \mathbb{P}^D \to \mathbb{P}^G$ which sends the point $\bar{\boldsymbol{p}} = (p_0, p_1, \ldots p_D) \in \mathbb{P}^D$ to the point $\rho_W(\bar{\boldsymbol{p}}) = (M_0(\bar{\boldsymbol{p}}), M_1(\bar{\boldsymbol{p}}), \ldots M_G(\bar{\boldsymbol{p}}))$. This is called the *W-tuple Veronese embedding* of $\mathbb{P}^D$ in $\mathbb{P}^G$.

This embedding introduces the following isomorphism. Although there are $\begin{pmatrix} D+W \\ D \end{pmatrix}$ coefficients in $\mathbb{P}^G$, there must exist one monomial with coefficients that are all zero, in other words, it is a constant, because of the homogeneity, so scaling removes one dimension; concretely, setting one of the coefficients to one accomplishes this.

Lemma 13. The W-tuple of $\mathbb{P}^D$ is an injective isomorphism onto its image.

Proof. see exercise 3.4, chapter I [Hartshorne, 1977]. □

A consequence of this lemma is that, a hyperplane $h \in \mathbb{P}^G$ defined as $w_0 y_0 + w_1 y_1 + \ldots + w_G y_G = 0$, will contain points $\bar{\boldsymbol{x}} = (x_0, x_1, \ldots x_D) \in \mathbb{P}^D$ such that $w_0 M_0(\bar{\boldsymbol{x}}) + w_1 M_1(\bar{\boldsymbol{x}}) + \ldots + w_G M_G(\bar{\boldsymbol{x}}) = 0$. In other words, a W-degree polynomial in dimension $\mathbb{P}^D$ is isomorphic to a hyperplane in $\mathbb{P}^G$.

Example 7. Take a conic section in affine space $\mathbb{R}^2$ (with variables x_1, x_2), defined by polynomial equation $w_0 x_1^2 + w_2 x_2^2 + w_3 x_1 x_2 + w_3 x_1 + w_4 x_2 + w_5 = 0$. This polynomial is equivalent to the polynomial $w_0 x_1^2 + w_2 x_2^2 + w_3 x_1 x_2 + w_3 x_1 x_0 + w_4 x_2 x_0 + w_5 x_0^2 = 0$ in projective space $\mathbb{P}^2$ (with variables x_0, x_1, x_2).

This conic section is isomorphic to a hyperplane hh in $\mathbb{P}^5$ $(y_0, y_1, y_2, y_3, y_4, y_5)$, namely the hyperplane defined by equation $w_0y_0 + w_1y_1 + \ldots + w_4y_4 + w_5y_5 = 0$.

This example explains why five points can determine a conic section in $\mathbb{R}^2$, because a conic section (which corresponds to polynomial of degree two) in $\mathbb{R}^2$ is isomorphic to a hyperplane in $\mathbb{R}^5$.

This shows that the degree W-polynomial in D-dimensional space is isomorphic to a hyperplane in $G = \begin{pmatrix} D+W \\ D \end{pmatrix} - 1$ space. Since these hyperplanes have duality, the hypersurfaces corresponding to it also have duality. Therefore, it is possible to generalize the linear classification theorem 12 to the following, hypersurface case.

Theorem 13. *Hypersurface classification theorem.* Let $\mathcal{D}$ be a data set in general position in $\mathbb{R}^D$. Assume t_{eval} is the time required to evaluate the classification objective function value. Exact solutions for hypersurface classification problem with arbitrary objective function, such that the hypersurface is defined by a degree W polynomial, can be obtained in at most $O\left(t_{\text{eval}} \times N^G\right)$ time by enumerating the cells of the dual arrangement $\mathcal{H}_{\tilde{\mathcal{D}}}$, where $\tilde{\mathcal{D}} = \rho_W\left(\overline{\mathcal{D}}\right)$ denotes the dataset obtained by transforming each data $\bar{\boldsymbol{x}} \in \mathbb{P}^D$ to its W-tuple Veronese embedding $\rho_W\left(\bar{\boldsymbol{x}}\right) = \left(M_0\left(\bar{\boldsymbol{x}}\right), M_1\left(\bar{\boldsymbol{x}}\right), \ldots M_G\left(\bar{\boldsymbol{x}}\right)\right) \in \mathbb{P}^G$ (which is equivalent to a point $\left(1, M_1\left(\bar{\boldsymbol{x}}\right), \ldots M_G\left(\bar{\boldsymbol{x}}\right)\right) \in \mathbb{R}^{G+1}$).

II.3.3 Methods for cell enumeration

Above it was shown that key to solving classification problems exactly lies in finding the optimal dichotomy, which can be achieved by enumerating all possible cells of the dual arrangement. Beyond the linear classification problem, it turns out that many otherwise intractable combinatorial optimization problems can also be solved exactly by enumerating the cells of an arrangement. For instance, it has been shown that both the *integer quadratic programming problem* [Ferrez et al., 2005] and the 0-1 loss linear classification problem [He and Little, 2023b], can be solved exactly through cell enumeration. The applications of integer quadratic programming are extensive, with the well-known *sparse regression problem* [Bertsimas et al., 2020] being a notable example. Furthermore, many machine learning problems discussed in this thesis, such as K-means clustering and linear classification, can also be solved exactly by enumerating the cells of a hyperplane arrangement.

However, it has not yet been explained how to *enumerate* all possible cells of an arrangement. Denote the set of all cells of $\mathcal{H}$ by $\mathcal{S}_{\text{cell}}\left(\mathcal{H}\right)$. When the arrangement $\mathcal{H}$ is clear from the context, it is denote by $\mathcal{S}_{\text{cell}}$ directly. From Lemma 5, it is established that the number of possible cells for an arrangement $\mathcal{H}$ is given by $\left|\mathcal{S}_{\text{cell}}\left(\mathcal{H}\right)\right| = C_D\left(\mathcal{H}\right) = \sum_{d=0}^{D} \begin{pmatrix} N \\ d \end{pmatrix}$, which is polynomially large with fixed D. Therefore, developing an efficient algorithm for enumerating the cells of an arrangement, would also obtain a polynomial-time algorithm for solving all the aforementioned problems related to cell enumeration.

This section provides a comprehensive review of methods for enumerating the cells of a hyperplane arrangement. Based on the combinatorial properties of different cell enumeration algorithms, they are classified into two major classes: *linear programming-based* (LP-based) generation and *hyperplane-based* (H-based) generation. Each class of algorithms has its own advantages and limitations when solving intractable combinatorial optimization problems, making it difficult to fully replace one with the other. These differences are explored in the following discussion.

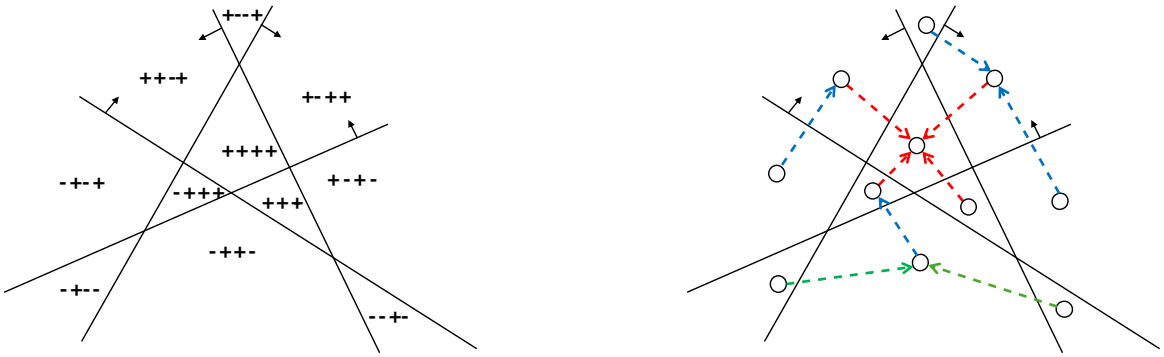


Figure 14: A hyperplane arrangement (left panel), consisting of four hyperplanes, with black arrows indicating their directions. The positive and negative labels represent the underlying sign vector of each cell. The right panel illustrates the generation process of the most well-known cell enumeration algorithm—the *reverse search* algorithm. Different arrow colors represent the recursive stages of this algorithm: red arrows indicate cells generated in the first recursive step, blue in the second, and green in the third.

II.3.3.1 Linear programming-based methods for cell enumeration

Given an arbitrary simple arrangement $\mathcal{H}$, the objective of a cell enumeration algorithm is to generate all possible cells $\mathcal{S}_{\text{cell}}(\mathcal{H})$ of $\mathcal{H}$. Consider a central arrangement $\mathcal{H} = \{h_n : n \in \mathcal{N}\}$ defined by hyperplanes $h_n = \left\{\boldsymbol{x} \in \mathbb{R}^D : \boldsymbol{w}_n^T \boldsymbol{x} = 0\right\}$ in $\mathbb{R}^D$. Each cell can be represented by an N-tuple $\{+,-\}^N$, where each $+$ or $-$ sign indicates whether a point lies on the positive or negative side of the hyperplane h_n, $\forall n \in \mathcal{N}$. One way to enumerate cells of an arrangement $\mathcal{H}$ is by generating the possible sign vectors that $\mathcal{H}$ can represent. Note, the sign vectors are equivalent to *binary assignments*, and there are 2^N possible length-N binary assignments, but only $C_D(\mathcal{H}) = \sum_{d=0}^{D} \binom{N}{d}$ possible cells.

To determine whether a sign vector $c = (c_1, c_2, \ldots, c_N) \in \{+,-\}^N$ indeed represents a cell in $\mathcal{H}$, solve the following linear programming problem,

$$\begin{aligned} &\min_{\boldsymbol{x}} \ \mathbf{0}^T \boldsymbol{x} \\ &\text{s.t. } \operatorname{diag}(c)^T \overline{\boldsymbol{W}} \boldsymbol{x} \geq \mathbf{1}, \end{aligned} \tag{99}$$

where $\mathbf{0}$ is the all zero vector in $\mathbb{R}^{D+1}$, $\mathbf{1}$ is the all one vector in $\mathbb{R}^N$. The matrix $\operatorname{diag}(c)$ is an $N \times N$ diagonal matrix, with the sign vector c placed along the diagonal. The matrix$\boldsymbol{W}$ is the normal vector matrix, where each row corresponds to a normal vector $\boldsymbol{w}_n \in \mathbb{R}^{D+1}$, $n \in \mathcal{N}$. Denote c^+ and c^- as the positive and negative index sets of the sign vector c respectively. For instance, if $c = \{+,+,-,-\}$, then $c^+ = \{1,2\}$ and $c^- = \{3,4\}$.

There exists a class of cell enumeration algorithms based on the *cell feasibility predicate* (99), which will be the focus of this Subsection. These algorithms will be referred to as *linear programming-based cell generation* (LP-CG) algorithms, p_{cell} being the cell feasibility predicate in the following discussions.

Reverse search algorithm The reverse search algorithm is the first and perhaps also the most well-known algorithm for enumerating the cells of an arrangement. It starts by setting an initial cell c^*, where all prediction labels are $+$. This can be done by selecting an arbitrary cell of the arrangement and

reversing the orientation of any hyperplanes with negative labels by replacing their expression $\boldsymbol{w}_n^T x = b_n$ with $-\boldsymbol{w}_n^T x = -b_n$. This operation does not change the arrangement itself.

Starting from the initial cell c^*, adjacent cells are generated recursively by flipping the sign of one prediction label c_n^* from $+$ to $-$ for all index n. Since there are N indices in total, each flipping operation involves at most $O(N)$ operations. To determine if a flipped sign vector c' is indeed a valid cell, solve the linear program (99). The algorithm will return all possible cells after recursively executing the flipping operation up to N times. This is because each assignment has only N possible ways to flip the prediction labels.

Two cells c' and c are *adjacent* if their sign vectors are different in exactly one component. Specifically, c' is considered the *parent* of c if they are different in index j, and $c'_j = -$, $c_j = +$. It is important to note that there may be multiple distinct cells adjacent to the same cell. Consequently, the strategy described above may generate the same cell multiple times.

A critical step in avoiding the generation of duplicate cells is to design a unique child predicate q, such that each cell c' has *exactly one associated child* c. In other words, $q(c', c) = True$, if and only if c is the *unique* child of its parent c' and return $q(c'', c) = False$ for all other parents c'' of c. The cell c will be called the *unique child* of c'. The definition of q is not unique, for the detailed definition of q refer to Ferrez et al. [2005].

This generation process can be described as the following sequential decision process

$$\begin{aligned} gen_{\text{rev}}(0) &= \text{adjcells}(c^*) \\ gen_{\text{rev}}(N) &= gen(N-1) \cup filter_q(map_{\text{adjcells}}(gen(N-1))), \end{aligned} \tag{100}$$

where $map_{\text{adjcells}}(cs)$ maps the adjcells function to each cell c in cs, and $\text{adjcells}(c)$ generate all parent cells of c. This function is defined as

$$\text{adjcells}(c) = \left[\text{flip}_n(c) \,|\forall n \in c^+, \text{if } p_{\text{cell}}(\text{flip}_n(c))\right], \tag{101}$$

where p_{cell} is the cell feasibility test defined in (99), and $\text{flip}_n(c)$ function flip the n-the index of sign vector c. The $filter_q$ function here is symbolic, on generating a list of parent cells $\text{adjcells}(c)$ from its child c, it is necessary to filter out all c' in $\text{adjcells}(c)$ for which c is not its unique child, i.e., $q(c', c) = False$.

The original characterization of the reverse search algorithm given by [Avis and Fukuda, 1996] is usually executed in a depth-first way, which can be described by the algorithmic process in Algorithm 1. All possible cells of an arrangement can be obtained by running $\text{RevSearch}(\mathcal{H}, c^*)$. Let $LP(n, d)$ denote the time to solve a linear program with n inequalities in d variables. Then, the time complexity of this algorithm is $O(N \times \text{LP}(N, D) \times C_D(\mathcal{H}))$ because in the worst case, it is necessary to run a linear program $O(N)$ times to detect the adjacent cells of each cell. When $\mathcal{H}$ is a central arrangement, the number of cells of $\mathcal{H}$ is $C_D(\mathcal{H}) = \sum_{d=0}^{D} \binom{N}{d}$.

The generation process of the reverse search algorithm is depicted in the right-panel of Fig. 14.

Incremental sign construction algorithm In previous analysis for solving the 0-1 loss linear classification problem, the author and others developed a generic cell enumeration generator based on the *binary assignment generator* introduced in Section II.2.3 [He and Little, 2023a]. This is referred to as

Algorithm 1 Abstraction of the *reverse search algorithm* for enumerating the cells of a hyperplane arrangement.

1. **Algorithm:** RevSearch $(\mathcal{H}, c)$
2. **Inputs**: An arbitrary cell $c \in \{1, -1\}^N$, and an arrangement $\mathcal{H} = \{h_n : n \in \mathcal{N}\}$ defined by hyperplanes $h_n = \left\{\boldsymbol{x} \in \mathbb{R}^D : \boldsymbol{w}_n^T \boldsymbol{x} = b_n\right\}$ in $\mathbb{R}^D$.
3. **begin**
4. $\quad \mathcal{S}_{\text{cell}} = \{c\}$
5. $\quad cs = \text{adjcells}\,(c)$
6. $\quad$ **for** $c' \in cs$ **do**
7. $\quad\quad$ **if** $q\,(c, c') = \text{True}$ **then**
8. $\quad\quad\quad \mathcal{S}_{\text{cell}} = \mathcal{S}_{\text{cell}} \cup \{c'\}$
9. $\quad\quad\quad \text{RevSearch}\,(c')$
10. $\quad\quad$ **end**
11. $\quad$ **end**
12. **end**
13. **Outputs**: $\mathcal{S}_{\text{cell}}$

the *incremental sign construction algorithm*, because it constructs sign vectors incrementally. Rada and Cerny [2018] independently discovered the depth-first search version of this algorithm, though five years earlier.

The idea behind this cell enumerator is based on the fact that the cell feasibility test is segment-closed with respect to the binary assignment generator. Recall that a predicate p is segment-closed if $p(x \cup y) \implies (x) \wedge p(y)$. This property holds for this problem because if a partial sign vector s that is not a cell in the *partial arrangement* $\mathcal{H}'$ (i.e., $\mathcal{H}'$ is obtained by deleting some hyperplanes from $\mathcal{H}$), then any cell c extended from s will not be a feasible cell with respect to $\mathcal{H}$. Dually, this means that if a partial binary assignment s is not linearly separable with respect to the point set $\phi(\mathcal{H}')$, then any assignment c extended from s will not be linearly separable with respect to $\phi(\mathcal{H})$.

Therefore, an efficient cell enumeration generator can be constructed by simply incorporating the cell feasibility predicate p_{cell} insides the `basgn` generator, which can be defined as

$$\begin{aligned} gen_{\text{cell}}(0) &= [[\,]] \\ gen_{\text{cell}}(N) &= filter_{p_{\text{cell}}}([1] \circ gen_{\text{cell}}(N-1) \cup [-1] \circ gen_{\text{cell}}(N-1)), \end{aligned} \tag{102}$$

where $\circ$ is the cross-join operator. In the n-th recursive step of program (102), there are at most $\text{Cover}(n, D+1) = 2\sum_{d=0}^{D} \binom{n-1}{d}$ cells, which differs from the number of cells in the reverse search algorithm, as this generator also accounts for the dual cells. Thus program (102) will have a complexity of
$O\left(\sum_{n=0}^{N} (\text{LP}(n, D) \times \text{Cover}(n, D+1))\right)$ in order to enumerate all possible cells of an central arrangement $\mathcal{H}$ in $\mathbb{R}^D$.

In the actual implementation of (102), the number of linear programs that need to be run can be reduced by half by storing an interior point for each cell during recursive generation. The underlying logic is as follows: during recursion, whenever a new hyperplane h_n is added, the existing cell c is related to h_n in two ways:

1. The whole connected component of c belongs to h_n^+ or h_n^- .
2. The hyperplane h_n subdivides the connected component of c into two new components, thus forming two new cells.

Therefore, if an interior point $\boldsymbol{x}_c \in \mathbb{R}^D$ of cell $c \in \{1, -1\}^{n-1}$ belongs to h_n^+ or h_n^- then there must exist a new cell $c' = (1, c) \in \{1, -1\}^n$ or $c' = (-1, c) \in \{1, -1\}^n$ after introducing the new hyperplane h_n. Once the sign vector of c' is determined, assume $c' = (1, c)$. The two cases described above can be distinguished by checking if the sign vector $(-1, c)$ is a cell by running the cell feasibility predicate (99). Thus, only half of the sign vectors need to be tested by the linear program (99), as the cells represented by interior points $\boldsymbol{x}_c$ do not require the feasibility test. Therefore, the actual number of linear programs that need to be run in the n-th recursive step of the program (102) is

$$\text{Cover}(n, D+1) - \text{Cover}(n-1, D+1) = 2\sum_{d=0}^{D} \binom{n-1}{d} - 2\sum_{d=0}^{D} \binom{n-2}{d} = \text{Cover}(n-1, D). \tag{103}$$

Thus, the actual complexity of the algorithm will be

$$O\left(\sum_{n=0}^{N}\left(\mathrm{LP}\left(n,D\right)\times\mathrm{Cover}\left(n-1,D\right)\right)\right)=O\left(\sum_{n=0}^{N}\left(\mathrm{LP}\left(n,D\right)\times n^{D-1}\right)\right). \tag{104}$$

It is well known that linear programming algorithms, such as the interior point method, can terminate after a finite number of iterations, depending on the desired solution precision [Little, 2019]. A linear programming algorithm with worst-case complexity of $O\left(N^3\right)$ [Vaidya, 1989], leads to the cell enumerator's complexity (104), being upper-bounded by $O\left(N^{D+3}\right)$.

Compared with the reverse search algorithm (100), the incremental sign construction algorithm (102) solves more but smaller linear programming problems. In the previous analysis by Rada and Cerny [2018], the incremental sign construction algorithm (102) appears more efficient, but it is difficult to analyze this precisely based on different implementations.

II.3.3.2 Hyperplane-based method for cell enumeration

The term "hyperplane-based" method refers to approaches designed to explicitly construct hyperplanes, unlike the LP-based method discussed earlier, which characterizes hyperplanes by enumerating all possible sign vectors of an arrangement. In $\mathbb{R}^D$, a hyperplane is uniquely defined by D points, so that hyperplane-based methods, which represent a hyperplane by identifying the data points that lie on it, provides a different perspective on hyperplanes and eventually results in a new class of algorithms that have a different combinatorial structure.

Vertex enumeration and the obvious cell enumeration algorithm Consider a *non-central* and *simple* arrangement $\mathcal{H}$, consists of N hyperplane in $\mathbb{R}^D$. The obvious strategy for enumerating cells of an arrangement is well-known in its *dual form* in machine learning studies [Murthy et al., 1994, Dunn, 2018]. This states that, given a set of data points $\mathcal{D}$ of size N in $\mathbb{R}^D$, there are $2^D\begin{pmatrix}N\\D\end{pmatrix}$ possible ways to partition $\mathcal{D}$ by a linear hyperplane. This is obtained from the intuition that every set of D points can determine a hyperplane, and for each hyperplane, there are 2^D distinguished ways to assign different labels to D data points that lie on the hyperplane. As depicted in the left-panel of Fig. 13, in $\mathbb{R}^D$, each yellow hyperplane can be determined by an arbitrary two data items, which can be shifted infinitesimally to obtain 2^D hyperplanes without affecting the classification prediction of the other data points.

In the space of hyperplane arrangements, this corresponds to the arbitrary D hyperplane having intersection points (vertices). Each vertex in the dual space will be adjacent to exactly 2^D *cells*, because the data points are assumed to be in general position. By enumerating all possible vertices first and then enumerating the adjacent cells of each vertices, all possible cells can be enumerated in time $2^D\begin{pmatrix}N\\D\end{pmatrix}$. This strategy can be described by the algorithmic process given in Algorithm 2. Note the adjacent cells of a vertex $v\in\{1,0,-1\}^N$ can be obtained more efficiently compared with obtaining the adjacent cells of a cell $c\in\{1,-1\}$. Thus, the adjacent cells of v are obtained by simply replacing the 0 signs with arbitrary 1 or -1 signs.

However, as can be read from the Fig. 13, there are many *overlapping cells* using the obvious strategy described above. For instance, according to Lemma 5, when $N=100$ and $D=3$, the num-

Algorithm 2 An obvious algorithm for cell enumaration of hyperplane arrangements.

1. **Algorithm:** CellEnm $(\mathcal{H})$
2. **Inputs**: An arrangement $\mathcal{H} = \{h_n : n \in \mathcal{N}\}$ defined by hyperplanes $h_n = \left\{\boldsymbol{x} \in \mathbb{R}^D : \boldsymbol{w}_n^T \boldsymbol{x} = b_n\right\}$ in $\mathbb{R}^D$.
3. **begin**
4. $\quad \mathcal{S}_{\text{cell}} = \emptyset$
5. $\quad$ Generate all possible K-combinations of hyperplanes, and stored in the set $\mathcal{S}_{\text{vert}}$.
6. $\quad$ **for** $v \in \mathcal{S}_{\textbf{vert}}$ **do**
7. $\quad\quad cs = \text{adjcells}\,(v)$
8. $\quad\quad \mathcal{S}_{\text{cell}} = \mathcal{S}_{\text{cell}} \cup cs$
9. $\quad\quad$ **end**
10. $\quad$ **end**
11. **Outputs**: $\mathcal{S}_{\text{cell}}$

ber of cells is $\sum_{d=0}^{3} \begin{pmatrix} 100 \\ d \end{pmatrix} = 166751$, whereas the combinatorial complexity of the obvious strategy is $2^3 \begin{pmatrix} 100 \\ 3 \end{pmatrix} = 1293600$, the combinatorial complexity of this obvious strategy wastes almost $\left(2^D - 1\right) \begin{pmatrix} N \\ D \end{pmatrix}$ computations for enumerating duplicates cells!

In the following, a generic cell enumeration algorithm for enumerating the cells of a non-central arrangement $\mathcal{H}$ is introduced, and each cell will be visited only once, thus the resulting algorithms will have a complexity of $\sum_{d=0}^{D} \begin{pmatrix} N \\ d \end{pmatrix}$.

Efficient cell enumeration based on the K-combinations generator Consider a *non-central* and *simple* arrangement $\mathcal{H}$, consisting of N hyperplane in $\mathbb{R}^D$. This Section, presents a novel cell enumeration algorithm based on enumerating the vertices of arrangements. The algorithms constructed in this section exploit an important lemma found in Gerstner and Holtz [2006].

Let the D hyperplanes E_{t_d}, $1 \leq d \leq D$, be the *axis-parallel* hyperplanes defined by $\boldsymbol{e}_d^T \boldsymbol{x} = t_d$, where the d-th unit vector $\boldsymbol{e}_d$ is the vector with all components equal to zero except for the d-th component which is equal to one. The constant t_d is chosen so large such that all vertices of the hyperplane arrangement $\mathcal{H}$ are below E_{t_d}, $\forall 1 \leq d \leq D$. For instance, in Fig. 15, two axis-parallel hyperplanes E_{t_1} and E_{t_2} in $\mathbb{R}^2$ are indicated by dashed lines, and are chosen such that all vertices (black dots in Fig.) of $\mathcal{H}$ are *below* E_{t_d}, $\forall 1 \leq d \leq D$. Then the following lemma holds, which establish the one-to-one correspondence between the cells of arrangement $\mathcal{H}$ with vertices of $\mathcal{H}$ and the intersection vertices of hyperplanes in $\mathcal{H}$ with E_{t_d}s.

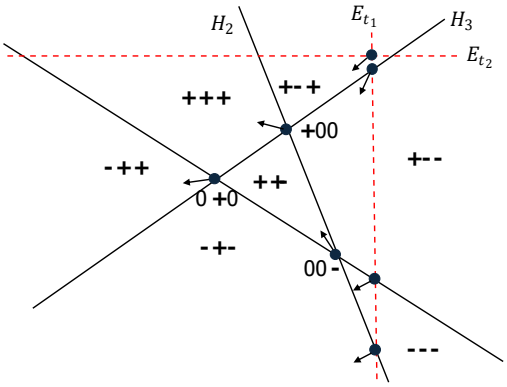


Figure 15: Intersection of axis-parallel hyperplanes E_{t_d}, $1 \leq d \leq 2$ with an arrangement consisting of three lines in $\mathbb{R}^2$

Methods for cell enumeration	Worst-case time complexity	Best-case time complexity	Worst-case memory usage	Best-case memory usage
LP-based	Always worse	Usually better	Always worse	Usually better
H-based	Always better	Usually worse	Always better	Usually worse

Table 1: Qualitative efficiency comparison between linear programming (LP) based methods and direct hyperplane construction (H) based methods for cell enumeration in terms of worst/best case time/space complexity.

Lemma 14. Let $\mathcal{H}$ be a simple arrangement. Let the vertex set $\mathcal{V}_{\mathcal{H}}$ consist of the intersection points of any d different hyperplanes h_n, $n \in \{N\}$, with the first $D-d$ hyperplanes E_{t_d}, $d = 1, \ldots, D-d$, where $d = 0, \ldots, D$. The vertex set $\mathcal{V}_{\mathcal{H}}$ is in one-to-one correspondence with the cells of arrangement $\mathcal{H}$. Each vertex uniquely determines a cell by moving this vertex along the backwards direction of edges that intersect it. The backwards direction is defined as the direction that decreases the value of the first $D-d$ coordinates.

The correctness of this lemma can be proved through an inductive sweep hyperplane, the proof is beyond the scope of this thesis. The algorithmic process described in Gerstner and Holtz [2006] is shown in Algorithm 3. It is evident that generating the vertex set $\mathcal{V}_{\mathcal{H}}$ (line 4) is well-suited to being accomplished using the K-combinations generator `kcombs` designed in Section II.2.3.

The remaining evaluation steps (lines 6-10) can be considered special cases and thus can be easily incorporated into the recursive generation steps of `kcombs`. However, since there is no publicly available code for the algorithm from Gerstner and Holtz [2006], it remains unclear how to efficiently select sufficiently large t_d values. One way to resolve this issue is to determine t_d after all vertices of an arrangement are enumerated. Then, t_d for all $d \in \mathcal{D}$ can be determined by calculating the maximum coordinates with respect to all vertices.

Algorithm 3 Efficient cell enumeration algorithm

1. **Algorithm:** ECellEnm $(\mathcal{H})$
2. **Inputs**: An arrangement $\mathcal{H} = \{h_n : n \in \mathcal{N}\}$ defined by hyperplanes $h_n = \left\{\boldsymbol{x} \in \mathbb{R}^D : \boldsymbol{w}_n^T \boldsymbol{x} = b_n\right\}$ in $\mathbb{R}^D$.
3. **begin**
4. $\quad \mathcal{S}_{\text{cell}} = \emptyset$
5. $\quad$ Compute all vertices set $\mathcal{V}_{\mathcal{H}}$
6. $\quad$ **for** $v \in \mathcal{V}_{\mathcal{H}}$ **do**
7. $\quad\quad$ let H_i denote the D hyperplanes intersecting v for $i = 1, \ldots, D$
8. $\quad\quad$ find a backward point $\boldsymbol{x}_v$ on the edge through v which does not lie on any hyperplane H_i
9. $\quad\quad$ compute the sign vector of $\boldsymbol{x}_v$, $sv = \text{sign}_{\mathcal{H}}(\boldsymbol{x}_v)$
10. $\quad\quad$ $\mathcal{S}_{\text{cell}} = \mathcal{S}_{\text{cell}} \cup sv$
11. $\quad$ **end**
12. **Outputs**: $\mathcal{S}_{\text{cell}}$

II.3.3.3 Efficiency of cell enumeration methods in combinatorial optimization

In terms of enumerating cells, H-based methods are much more efficient than LP-based methods, because calculating the inverse of a matrix, which takes $O(D^3)$ time in the worst-case, is much more efficient than calculating a linear program with N inequalities in D variables. A qualitative efficiency comparison between these two classes of methods is given in Table 1, considering solely their performance on the cell enumeration task.

However, in the context of combinatorial optimization, these different cell enumeration methods each have unique advantages that cannot be fully replaced by the others. This is because they characterize the combinatorics of each cell differently. LP-based methods uniquely characterize each cell by the sign vector it represents. By contrast, H-based methods characterize each cell by vertices (or dually, hyperplanes), which are uniquely determined by the intersection of D hyperplanes (or dually, a D-combination of data points). Thus, H-based methods are more memory efficient, since storing a sign vector requires $O(N)$ space but a hyperplane requires only $O(D)$ space.

Furthermore, for some combinatorial optimization problems, such as the 0-1 linear classification problem, characterizing cells (or hyperplanes) as sign vectors offers significantly better best-case complexity. This is because any partial sign vector that can be proven to be non-optimal can be safely discarded without full extension. By contrast, H-based methods characterize hyperplanes as *D-combinations of data points*. During recursion, d-combinations of data points for $0 \leq d < D$ are challenging to prove as non-optimal, as these combinations are insufficient to construct a hyperplane. Moreover, these d-combinations must be stored in order to construct complete D-combinations. Therefore, combinatorial optimization al-

gorithms based on H-based methods typically have better worst-case time complexity but worse best-case time complexity in combinatorial optimization tasks.

II.3.4 Euclidean Voronoi diagrams and the K-means problem

II.3.4.1 The K-means Euclidean Voronoi partition

Clustering in machine learning is the grouping of similar objects and the clustering of a set is a set partition of elements that is chosen to minimize some measure of dissimilarity of which there are numerous kinds. The K-clustering problem involves finding a set of *centroids* $\mathcal{U} = \{\boldsymbol{\mu}_k : k \in \mathcal{K}\}$, where $\mathcal{K} = \{1, \ldots, K\}$ in $\mathbb{R}^D$. Put all centroids in $\mathbb{R}^D$ together into a DK-tuple, denoted by $\overrightarrow{\boldsymbol{\mu}} = (\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_K) \in \mathbb{R}^{DK}$. Each centroid $\boldsymbol{\mu}_k$ is associated with a unique subset of data points, these data points are closer to this centroid than other centroids (in terms of distance). This subset is called a *cluster*, denoted by C_k. Together, they form a K-cluster set $\mathcal{C} = \{C_1, C_2, \ldots, C_K\}$. Thus $\mathcal{K}$ are then called the *cluster labels.*

The most commonly used dissimilarity measure is *Euclidean distance.* The K-clustering problem defined on Euclidean distance is called the K-means problem, which can be defined over the continuous variable $\overrightarrow{\boldsymbol{\mu}}$ as

$$\overrightarrow{\boldsymbol{\mu}}^* = \underset{\overrightarrow{\boldsymbol{\mu}} \in \mathbb{R}^{DK}}{\operatorname{argmin}} \, E_{\text{K-means}}\left(\overrightarrow{\boldsymbol{\mu}}\right) = \sum_{\boldsymbol{\mu}_k \in \mathcal{U}} \sum_{\boldsymbol{x}_n \in C_k} d_2\left(\boldsymbol{x}_n, \boldsymbol{\mu}_k\right)^2 \quad . \tag{105}$$

The objective function $\sum_{k \in \mathcal{K}} \sum_{\boldsymbol{x}_n \in C_k} d_2\left(\boldsymbol{x}_n, \boldsymbol{\mu}_k\right)^2$ is convex in $\boldsymbol{\mu}_k$; if the K-cluster set $\mathcal{C} = \{C_1, C_2, \ldots, C_K\}$ is fixed, the optimal centroids $\mathcal{U}$ which solve the (105) are uniquely determined, so the map between a set of class labels and a set of centroids is given analytically by

$$\boldsymbol{\mu}_k = \frac{1}{\mid C_k \mid} \sum_{\boldsymbol{x}_n \in C_k} \boldsymbol{x}_n, \quad k \in \mathcal{K}. \tag{106}$$

Given the correspondence between class labels and centroids, it is possible to define a combinatorial parameter $s = (\alpha_1, \alpha_2, \ldots, \alpha_N) \in \mathcal{S}_{\text{kasgns}} = \mathcal{K}^N$ which is the *K-class assignment* for the K-clustering problem such that $\alpha_n = k$ if data item $\boldsymbol{x}_n$ is assigned cluster k. Then, the K-means problem can be reformulated as

$$s^* = \underset{s \in \mathcal{S}_{\text{kasgns}}}{\operatorname{argmin}} \, E_{\text{K-means}}\left(s\right) = \sum_{k \in \mathcal{K}} \sum_{n \in \mathcal{N}} \mathbf{1}\left[s_n == k\right] d_2\left(\boldsymbol{x}_n, \boldsymbol{\mu}_k\right)^2 \quad , \tag{107}$$

where function $\mathbf{1}\left[s_n == k\right]$ returns 1 if the Boolean argument $s_n == k$ is true, and 0 if false.

Similarly, with a set of centroids $\mathcal{U}$, the assignment with respect to data set $\mathcal{D}$ can be determined uniquely. Indeed, a set of centroids $\mathcal{U} = \{\boldsymbol{\mu}_k : k \in \mathcal{K}\}$ forms a Euclidean Voronoi diagram that partitions the space into K Voronoi regions. Let $\mathcal{D}$ be a data set and $\mathcal{U} = \{\mu_1, \mu_2, \ldots, \mu_K\}$ be a set of centroids. Every set of centroids $\mathcal{U} = \{\mu_1, \mu_2, \ldots, \mu_K\}$ has an associated Voronoi diagram $\mathcal{V}\left(\mathcal{U}\right) = \{V_1, V_2, \ldots, V_K\}$, so that each centroid $\boldsymbol{\mu}_k$ partitions the ambient space $\mathbb{R}^D$ into regions defined by

$$V_k = \left\{\boldsymbol{x} \in \mathbb{R}^D \mid d_2\left(\boldsymbol{x} - \boldsymbol{\mu}_k\right)^2 \leq d_2\left(\boldsymbol{x} - \boldsymbol{\mu}_j\right)^2, \forall k, j \in \mathcal{K}\right\}, \tag{108}$$

which naturally partitions the data set $\mathcal{D}$ into K clusters $\mathcal{C} = \{C_1, C_2, \ldots, C_K\}$ defined by

$$C_k = \left\{\boldsymbol{x} \in \mathcal{D} \mid d_2\left(\boldsymbol{x} - \boldsymbol{\mu}_k\right)^2 \leq d_2\left(\boldsymbol{x} - \boldsymbol{\mu}_j\right)^2, \forall k, j \in \mathcal{K}\right\}, \tag{109}$$

The data points lying in region V_k are assigned to the cluster C_k. Below, the cluster set $\mathcal{C}$ is called the *Voronoi partition* to distinguish it from the Voronoi diagram $\mathcal{V}(\mathcal{U}) = \{V_1, V_2, \ldots, V_K\}$, the K-class assignment c associated with each Voronoi partition $\mathcal{C}$ is then called *Voronoi partition assignment.*

II.3.4.2 The optimality of the K-means problem

To solve the K-means clustering problem, the most obvious strategy is to enumerate all possible K-class assignments in the search space $\mathcal{S}_{\text{kasgns}}$. However, the number of possible K-class assignments for a size N data set is K^N. Every K-class assignment c in $\mathcal{S}_{\text{kasgns}}$ will introduce a unique partition $\{C_1, C_2, \ldots, C_K\}$. However, not all partitions are useful. In fact, it can be proved that only a *Voronoi partition* can be the global optimal partition for the K-means problems [Tîrnăucă et al., 2018, Inaba et al., 1994, Hasegawa et al., 1993].

Lemma 15. The optimum partition for the K-clustering problem must be a Voronoi partition.

Similar results can be extended to *Bregman Voronoi diagrams*, where distance functions are generalized to *Bregman divergences*.

Although the number of possible K-class assignments is $O\left(K^N\right)$, the number of Voronoi partitions is only polynomial large when D and K are fixed. Inaba et al. [1994] have shown that the number of Voronoi partitions is *at most* $O\left(N^{DK}\right)$. This arises from the fact that the Voronoi diagram can be represented by a hyperplane arrangement consisting of $NK(K-1)/2$ hyperplanes in DK-dimensional space, and the number of cells for this arrangement is at most $O\left(N^{DK}\right)$.

This result may look strange in the first place, as the Voronoi diagrams are defined by quadratic polynomials. The reason behind this is that we only care about the sign value of each quadratic term in (109), instead of the actual value of $d_2(\boldsymbol{x} - \boldsymbol{\mu}_k)^2$. In the following discussion, we will explain the reasons behind it, and the result will naturally yield algorithms for solving the K-means problem.

II.3.4.3 The sign vector of the Euclidean Voronoi diagram

Reformulate the expression $d_2(\boldsymbol{x}_n - \boldsymbol{\mu}_k)^2 \leq d_2\left(\boldsymbol{x}_n - \boldsymbol{\mu}_j\right)^2$ to the equivalent form

$$\boldsymbol{\mu}_k^2 - \boldsymbol{\mu}_j^2 - 2\boldsymbol{x}_n^T\left(\boldsymbol{\mu}_k - \boldsymbol{\mu}_j\right) \leq 0, \tag{110}$$

if the data items x_n is fixed, this formula becomes a polynomial with variable $\boldsymbol{\mu}_k$, $\boldsymbol{\mu}_j$.

Now, define $d_2(\boldsymbol{x}_n - \boldsymbol{\mu}_k)^2 - d_2\left(\boldsymbol{x}_n - \boldsymbol{\mu}_j\right)^2$ as a polynomial parameterized by $\boldsymbol{x}_n$ and with variable $\boldsymbol{\mu}_k$, $\boldsymbol{\mu}_j$

$$P_{\boldsymbol{x}_n,k,j} = \sum_{d=1}^{D}\mu_{kd}^2 - \sum_{d=1}^{D}\mu_{jd}^2 - 2\sum_{d=1}^{D}x_{nd}\left(\mu_{kd} - \mu_{jd}\right), \tag{111}$$

where $P_{\boldsymbol{x}_n,k,j}$ is a polynomial in $\mathbb{R}\left[\overrightarrow{\boldsymbol{\mu}}\right]$, and $\overrightarrow{\boldsymbol{\mu}} = (\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_K) \in \mathbb{R}^{DK}$ is the K-tuples of K centroids in $\mathbb{R}^D$.

For each pair of centroids and a data items, define a polynomial (111). Therefore, the total number of polynomial equations for a size N data set with K centroids in $\mathbb{R}^D$ is $M = N\begin{pmatrix} K \\ 2 \end{pmatrix} = NK(K-1)/2$. These polynomials form an arrangement $\mathcal{S}_{\mathcal{P}} = \{s_{\boldsymbol{x}_n,k,j} : \forall n \in \mathcal{N}, \forall k, j \in \mathcal{K} \wedge j \neq k\}$, where

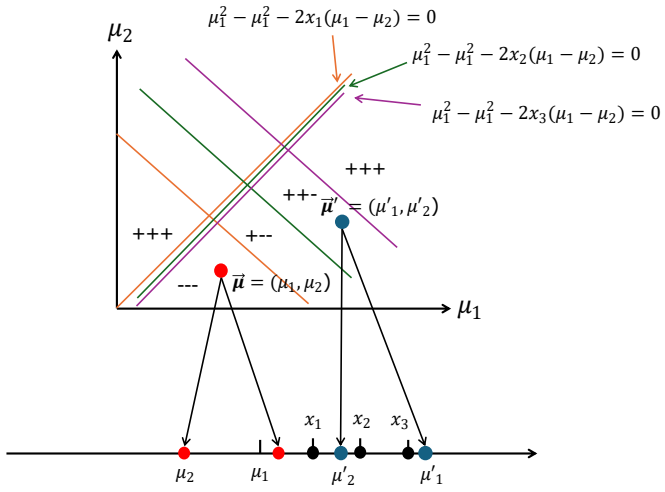


Figure 16: The hypersurface arrangement introduced by the Voronoi arrangement of three data points x_1, x_2 and x_3 in $\mathbb{R}$. The centroid vector $\overrightarrow{\boldsymbol{\mu}}$ lying within a given cell produces the same partition of data points. For example, two centroid vectors (represented by the red and blue points) lying in different cells will result in different partitions of the data. The centroid vector represented by the red point classifies all x_1, x_2, x_3 to centroid μ_1, while the centroid vector represented by the blue point classifies x_1 and x_2 to μ'_2, and x_3 to μ'_1.

$s_{\boldsymbol{x}_n,k,j} = \left\{\boldsymbol{x}_n \in \mathbb{R}^D : \sum_{d=1}^D \mu_{kd}^2 - \sum_{d=1}^D \mu_{jd}^2 - 2\sum_{d=1}^D x_{nd}\left(\mu_{kd} - \mu_{jd}\right)\right\}$. The arrangement $\mathcal{S}_\mathcal{P}$ is called the *Voronoi arrangement* defined by data set $\mathcal{D}$.

Lemma 16. Assume there are no data lying on the boundary of each partition. A K-class partition $\{C_1, C_2, \ldots, C_K\}$ is a Voronoi partition if and only if

$$C_k = \left\{\boldsymbol{x}_n \in \mathcal{D} : P_{\boldsymbol{x}_n,k,j} < 0 \wedge P_{\boldsymbol{x}_n,j,k} > 0\right\}, \forall k, j \in \mathcal{K} \wedge j \neq k. \tag{112}$$

In fact, the actual value of $P_{\boldsymbol{x}_n,k,j}$ is not of interest, since it is only necessary to know if $\boldsymbol{x}_n$ is closer to $\boldsymbol{\mu}_k$ or $\boldsymbol{\mu}_j$. This information can be obtained by knowing the sign vector of $\overrightarrow{\boldsymbol{\mu}} \in \mathbb{R}^{DK}$ of the polynomial system $\mathcal{P} = \{P_{\boldsymbol{x}_n,k,j} \mid \forall n \in \mathcal{N}, \forall k, j \in \mathcal{K} \wedge j \neq k\}$.

Thus (112) can be reformulated using the sign vector of the polynomial system $\mathcal{P}$ as stated in the following lemma.

Lemma 17. Let $sign_{\mathcal{S}_\mathcal{P}}\left(\overrightarrow{\boldsymbol{\mu}}\right)$ be the sign vector of $\overrightarrow{\boldsymbol{\mu}}$ with respect to surface arrangement $\mathcal{S}_\mathcal{P}$ defined by polynomial system $\mathcal{P} = \{P_{\boldsymbol{x}_n,k,j} \mid \forall n \in \mathcal{N}, \forall k, j \in \mathcal{K} \wedge j \neq k\}$, and assume no data points lie on the boundary of each partition. Then a cluster set $\{C_1, C_2, \ldots, C_K\}$ is a Voronoi partition if and only if

$$C_k = \left\{\boldsymbol{x}_n \in \mathcal{D} : \delta_{\boldsymbol{x}_n,k,j}\left(\overrightarrow{\boldsymbol{\mu}}\right) = -1 \wedge \delta_{\boldsymbol{x}_n,j,k}\left(\overrightarrow{\boldsymbol{\mu}}\right) = 1\right\}, \forall k \in \mathcal{K}. \tag{113}$$

Geometrically, $\delta_{\boldsymbol{x}_n,k,j}\left(\overrightarrow{\boldsymbol{\mu}}_{k,j}\right) = -1$ means that $d_2\left(\boldsymbol{x}_n - \boldsymbol{\mu}_k\right)^2 \leq d_2\left(\boldsymbol{x}_n - \boldsymbol{\mu}_j\right)^2$, i.e. $\boldsymbol{x}_n$ is closer to $\boldsymbol{\mu}_k$ than $\boldsymbol{\mu}_j$.

Conversely, given a centroid vector $\overrightarrow{\boldsymbol{\mu}}$, the Voronoi partition $\{C_1, C_2, \ldots, C_K\}$ associated to $\overrightarrow{\boldsymbol{\mu}}$ is defined as

$$C_k = \left\{\boldsymbol{x}_n \in \mathcal{D} : \delta_{\boldsymbol{x}_n,k,j}\left(\overrightarrow{\boldsymbol{\mu}}\right) = -1 \wedge \delta_{\boldsymbol{x}_n,j,k}\left(\overrightarrow{\boldsymbol{\mu}}\right) = 1\right\}, \forall k \in \mathcal{K}. \tag{114}$$

The K-clustering problems now becomes the problem that finding an optimal sign vector $\delta_{\mathcal{P}}\left(\overrightarrow{\boldsymbol{\mu}}\right)$ for the arrangement $\mathcal{S}_{\mathcal{P}} = \{s_{\boldsymbol{x}_n,k,j} : \forall n \in \mathcal{N}, \forall k,j \in \mathcal{K} \wedge j \neq k\}$ determined by polynomial system $\mathcal{P}$. The optimal sign vector $\delta_{\mathcal{P}}\left(\overrightarrow{\boldsymbol{\mu}}\right)$ can be obtained by enumerating all possible cells of the arrangement $\mathcal{S}_{\mathcal{P}}$.

According to Lemma 10, the number of cells in arrangement $\mathcal{S}_{\mathcal{P}}$ is at most $O\left(N^{DK}\right)$. However, since $\mathcal{S}_{\mathcal{P}}$ is ***not*** a simple arrangement, the actual number of cells in arrangement $\mathcal{S}_{\mathcal{P}}$ is significantly smaller than the worst-case bound $O\left(N^{DK}\right)$. For instance, assume there are three points $x_1 = 1$, $x_2 = 2$, and $x_3 = 3$ in $\mathbb{R}$ and $K = 2$ clusters; the corresponding arrangement $\mathcal{S}_{\mathcal{P}}$ for the polynomial system defined by (111) consists of three two-dimensional conic sections shown in Fig. 16. This is clearly not a simple arrangement, as the three conic sections defined by three different data items have common intersections.

Similar to the linear case, the space of $\mathbb{R}^{DK}$ is partitioned into different connected components by arrangement $\mathcal{S}_{\mathcal{P}}$. The points in each connected region represent equivalence classes of centroids, such that centroids within the same region will have identical sign vectors. For instance, consider data in $\mathbb{R}$ and the number of clusters is two, then the centroid vector $\overrightarrow{\boldsymbol{\mu}}$ lies in $\mathbb{R}^2$. In Fig. 16, it can be seen that the space of $\mathbb{R}^2$ is partitioned into eight disjoint regions. Arbitrary points in region with $(-,-,-)$ sign (the red point in the figure) correspond to centroids with the property that $d_2\left(\mu_1, x_i\right) < d_2\left(\mu_2, x_i\right)$, for $i \in \{1,2,3\}$. In other words, μ_1 is closer to all three data points than μ_2. On the other hand, the blue point in region $(+,+,-)$ represents that centroids μ_1' are closer to x_1, x_2 and μ_2' is closer to x_3.

There exist various studies on how to enumerate all cells for an arbitrary polynomial surface arrangement [Caviness and Johnson, 2012]. However, enumerating the cells of a hypersurface arrangement is much harder than enumerating the cells of a hyperplane arrangement. Fortunately, because of the special geometry of the K-clustering problem, it is possible to enumerate the cells of the arrangement $\mathcal{S}_{\mathcal{P}}$ in a much simpler way.

In the next section, it will be shown that the hypersurface arrangement $\mathcal{S}_{\mathcal{P}}$ defined by quadratic polynomials (111) can be transformed into a hyperplane arrangement.

II.3.4.4 Variable replacement and optimal K-means clustering

The discussion here is similar to the discussion of Tîrnăucă et al. [2018], it shows that the system of polynomials in (111) can be solved by a system of linear equations by applying a change of variables.

Consider polynomials defined in (111) with KD variables, an important observation is that $P_{\boldsymbol{x}_n,k,j} = P_{\boldsymbol{x}_n,1,j} - P_{\boldsymbol{x}_n,1,k}$, $\forall 2 \leq k < j \leq K$. Replace the variable in (111) with equation $\sum_{d=1}^{D} \mu_{kd}^2 = Y_i, \forall i \in \mathcal{K}$ and $\mu_{1d} - \mu_{kd} = Z_{(k-1)d}$, $\forall k \in \{2,\ldots,K\}$ and $d \in \{1,\ldots,D\}$. The polynomial $P_{\boldsymbol{x}_n,k,j}$ can be reformulated as

$$\begin{aligned} P'_{\boldsymbol{x}_n,1,k} &= Y_1 - Y_k - 2\sum_{d=1}^{D} x_{nd} Z_{(k-1)d},\ \forall k \in \{2,\ldots,K\} \\ P'_{\boldsymbol{x}_n,k,j} &= P'_{\boldsymbol{x}_n,1,j} - P'_{\boldsymbol{x}_n,1,k},\ \forall 2 \leq k < j \leq K, \end{aligned} \tag{115}$$

which forms a new linear system $\mathcal{P}' = \left\{P'_{\boldsymbol{x}_n,j,k} : \forall n \in \mathcal{N}, k \neq j\right\}$ in $\mathbb{R}\left[Y_1 \ldots Y_K, Z_1, \ldots, Z_{(K-1)D}\right] = \mathbb{R}\left[\overrightarrow{\boldsymbol{\mu}}'\right]$, where $\overrightarrow{\boldsymbol{\mu}}' \in \mathbb{R}^{K+(K-1)D}$ is the modified variable form of $\overrightarrow{\boldsymbol{\mu}}$ through the above variable replacement. Denote the hyperplane arrangement defined by the polynomial system $\mathcal{P}'$ as $\mathcal{H}_{\mathcal{P}'}$.

After changing the variables, the new system $P'_{\boldsymbol{x}_n,1,k}$ becomes a *linear* system. Moreover, the number of variables in the new polynomial system $\mathcal{P}'$ involves only $K + D\left(K-1\right)$ variables whereas the old system involves KD variables. Moreover, polynomials $P'_{\boldsymbol{x}_n,k,j}$, $2 \leq k < j \leq K$ are all determined by polynomials

$P'_{\boldsymbol{x}_n,1,k}$, $k \in \{2,\ldots,K\}$. Thus all polynomials $P'_{\boldsymbol{x}_n,k,j}$, $2 \leq k < j \leq K$ can be ignored when enumerating the cells of the arrangement $\mathcal{H}_{\mathcal{P}'}$. Therefore, although there are $NK\left(K-1\right)/2$ polynomial equations in system $\mathcal{P}'$, the number of linearly independent equations is $\left(K-1\right)N$. By contrast, the old system has $NK\left(K-1\right)/2$ independent equations.

Example 8. Given a dataset $\mathcal{D}$ of size N in $\mathbb{R}^2$ and $K=3$. The coefficients for linear system (115) can be represented by the following coefficient matrix:

$$\begin{bmatrix}
1 & -1 & & & -2x_{11} & -2x_{12} & & & & \\
\vdots & \vdots & & & \vdots & \vdots & & & & \\
1 & -1 & & & -2x_{N1} & -2x_{N2} & & & & \\
1 & & -1 & & & & -2x_{11} & -2x_{12} & & \\
\vdots & & \vdots & & & & \vdots & \vdots & & \\
1 & & -1 & & & & -2x_{N1} & -2x_{N2} & & \\
1 & & & -1 & & & & & -2x_{12} & -2x_{12} \\
\vdots & & & \vdots & & & & & \vdots & \vdots \\
1 & & & -1 & & & & & -2x_{N2} & -2x_{N2}
\end{bmatrix}, \forall n \in \mathcal{N}. \tag{116}$$

By applying the variable replacement above, it is easy to show the following lemma holds.

Lemma 18. Given a polynomial sub-system $\mathcal{P} = \{P_{\boldsymbol{x}_n,k,j} \mid \forall n \in \mathcal{N}, \forall k,j \in \mathcal{K} \wedge j \neq k\}$, where $P_{\boldsymbol{x}_n,k,j}$ is defined by (111) in $\mathbb{R}\left[\overrightarrow{\boldsymbol{\mu}}\right]$. *Given a sign vector* $\alpha = \left\{\alpha_1,\ldots,\alpha_{|\mathcal{I}|}\right\} \in \{-1,1\}^{|\mathcal{I}|}$, *if*

$$\operatorname{sign}_{\mathcal{S}_\mathcal{P}}\left(\overrightarrow{\boldsymbol{\mu}}\right) = \alpha_i, \forall i \in \mathcal{I}, \tag{117}$$

has a solution, then

$$\operatorname{sign}_{\mathcal{H}_{\mathcal{P}'}}\left(\overrightarrow{\boldsymbol{\mu}}'\right) = \alpha_i, \forall i \in \mathcal{I}, \tag{118}$$

also has.

A consequence of Lemma 18 is that enumerating the cells for arrangement $\mathcal{S}_\mathcal{P}$ defined by polynomial $\mathcal{P} = \{P_{\boldsymbol{x}_n,k,j} \mid \forall n \in \mathcal{N}, \forall k,j \in \mathcal{K} \wedge j \neq k\}$ is equivalent to enumerating cells for hyperplane arrangement $\mathcal{H}_{\mathcal{P}'}$ defined by linear system $\mathcal{P}' = \left\{P'_{\boldsymbol{x}_n,j,k} : \forall n \in \mathcal{N}, k \neq j\right\}$. Since $\mathcal{H}_{\mathcal{P}'}$ is not in general position, the cells of this arrangement can not be enumerated by using the cell enumeration method we introduced in previous section. The following theorem for enumerating cells in arrangement $\mathcal{H}_{\mathcal{P}'}$ follows from the result of Tîrnăucă et al. [2018].

Theorem 14. The enumeration of all possible sign vectors (cells) of arrangement $\mathcal{P}'$, can be achieved by solving polynomial equations $P'_j\left(\overrightarrow{\boldsymbol{\mu}}\right) = \alpha_j$, $j \in \mathcal{J}$, where $\alpha = \left(\alpha_1,\ldots,\alpha_{|\mathcal{J}|}\right)$ represents all possible binary assignments in $\{1,-1\}^{|\mathcal{J}|}$, and $\mathcal{J}$ is a subset of $\mathcal{I}$ such that $|\mathcal{J}| \leq K + \left(K-1\right)D$.

II.3.4.5 Duality and 2-means problem

Indeed, Thm. 14 is similar to the obvious hyperplane-based method (111) for enumerating the cells of an arrangement. As a result, this approach is much less efficient than the ideal algorithm that enumerates

each cell only once. However, since the subspace defined by polynomials in $\mathcal{P}'$ are not even hyperplanes, the algorithms introduced in (II.3.3.1) cannot be directly applied to solve this problem. Developing a more efficient cell enumeration method tailored to this specific problem could be a promising direction for future research.

Nevertheless, intuitively, the 2-means clustering problem in its simplest form closely resembles a classification problem. In the 2-means problem, the partition of the data is determined by a hyperplane, due to the linear separation property inherent to all Bregman divergences [Banerjee et al., 2005]. This section will validate this intuition by showing that for the 2-means clustering problem, the arrangement introduced by the polynomials $\mathcal{P}'$ is indeed a hyperplane arrangement. Consequently, the 2-means problem can be solved exactly using any cell enumeration algorithm.

Consider the case where $K = 2$, the new polynomial system $\mathcal{P}'$ introduces a linear system with following coefficient matrix

$$\begin{bmatrix} 1 & -1 & -2x_{11} & -2x_{12} & & -2x_{1D} \\ 1 & -1 & -2x_{21} & -2x_{22} & & -2x_{2D} \\ & & & \vdots & \cdots & \vdots \\ 1 & -1 & -2x_{n1} & -2x_{n2} & & -2x_{nD} \\ & & & \vdots & & \vdots \\ 1 & -1 & -2x_{N1} & -2x_{N2} & & -2x_{ND} \end{bmatrix}. \tag{119}$$

At first glance, the affine flats represented by the matrix (119) form an arrangement in $\mathbb{R}^{D+2}$. However, the linear system (hyperplane arrangement) in (119) has a very special form, it consists of a set of hyperplanes with normal vector $\boldsymbol{w}_n = (1, -1, -2\boldsymbol{x}_1, \ldots, -2\boldsymbol{x}_D), \forall n \in \mathcal{N}$, the first two coordinates are fixed. Applying the inverse of the dual transformation ϕ^{-1} to the hyperplane system (119) obtains a set of data points $\left\{\phi^{-1}(\boldsymbol{w}_n) = (1, -1, -2\boldsymbol{x}_1, \ldots, -2\boldsymbol{x}_D) \in \mathbb{R}^{D+2} \mid \forall n \in \mathcal{N}\right\}$. The data points in this form are defined by the following.

Definition 33. *Double-homogeneous coordinate.* The double-homogeneous coordinate for a data item $\boldsymbol{x}_n = (x_{n1}, x_{n2}, \ldots, x_{nD}) \in \mathbb{R}^D$ is defined as

$$\bar{\bar{\boldsymbol{x}}}_n = (c_1, c_2, x_{n1}, x_{n2}, \ldots, x_{nD}) \in \mathbb{R}^{D+2}, \tag{120}$$

where c_1, c_2 are some constants.

In the case of data set $\left\{\phi^{-1}(\boldsymbol{w}_n) = (1, -1, -2\boldsymbol{x}_n) \in \mathbb{R}^{D+2} \mid \forall n \in \mathcal{N}\right\}$, imagine a set of datasets $\{-2\boldsymbol{x}_n : \forall n \in \mathcal{N}\}$ being moved to homogeneous coordinates twice. Geometrically, this means that there are two hyperplanes $Y_1 = 1$ and $Y_2 = -1$ (subspace with dimension $D + 2 - 1$, remembering that the dimension for the linear system (119) is $D + 2$), their intersection forms a $D + 2 - 2 = D$ dimensional subspace which is equivalent to the dimension of the data.

Therefore, the 2-means problem with respect to the dataset $\mathcal{D}$ can be solved exactly by enumerating the cells of the arrangement introduced by (119). This is equivalent to finding the optimal dichotomies for the dataset $\{-2\boldsymbol{x}_n : \forall n \in \mathcal{N}\}$. This confirms the intuition that the 2-means problem is equivalent to a linear classification problem and can thus be solved exactly by enumerating $O\left(N^D\right)$ cells.

II.3.5 Chapter discussion

This chapter has presented a comprehensive analysis of the geometric and combinatorial properties of Voronoi diagrams and hyperplanes. One important finding is that the way the `kcombs` generator constructs combinations perfectly aligns with the cell enumeration strategy presented by Gerstner and Holtz [2006], who developed a cell enumeration algorithm based on a naive one-by-one enumeration method—an approach that is, as one might expect, highly inefficient compared to the `kcombs` generator. Furthermore, the `kcombs`-based cell enumeration algorithm is expected to be significantly more efficient than the most well-known cell enumeration algorithm, the reverse search algorithm, as it eliminates the need to solve linear programs, relying instead on matrix inversion. It is difficult to imagine that any future algorithm will outperform cell enumeration generators based on `kcombs`, unless a more efficient representation of hyperplanes is discovered—one that enables hyperplane construction more efficiently than matrix inversion does.

We also present a reformulation of the reverse search algorithm, which is typically expressed in a depth-first way in the literature. These cell enumeration algorithms hold potential interest for both research in combinatorial geometry and optimization. For geometers, the construction of efficient cell enumeration algorithms is one of the most fundamental problems, and existing algorithms are often difficult to parallelize. In particular, although Avis and Fukuda [1996] provide a brief explanation of how to parallelize the reverse search algorithm, their depth-first approach, as discussed in Section II.2.8, often requires extensive communication between processors due to backtracking. By contrast, the novel cell enumeration algorithms in this chapter, including the reformulated reverse search algorithm, are much easier to parallelize and can be efficiently implemented on both CPUs and GPUs. This is because they can be constructed using the `kcombs` and `basgns` generators over join-lists, which are specifically designed for developing parallel algorithms. These cell enumeration algorithms will also be of interest to optimization research, as many intractable combinatorial optimization problems [Ferrez et al., 2005, He and Little, 2023a, Bertsimas et al., 2020] can be solved in polynomial time by applying them.

The earlier work on constructive algorithmics appears only weakly related to this chapter, as none of its theoretical foundations were directly applied here. The only connection lies in the generators introduced in Subsection II.2.3.3 which can be employed to construct cell enumeration algorithms. This choice stems from the fact that many combinatorial structures used to represent geometric objects—such as combinations and binary assignments—are already in their simplest forms, leaving no room for speed-up from a purely algorithmic perspective. Thus, the only viable optimization is to design a generator tailored for modern hardware, which motivated the development of the `kcombs` generator. Nevertheless, there remains an urgent need to apply constructive algorithmics to develop a calculational proof for the results in this chapter, as some of the presented proofs are notably lengthy.

Additionally, this chapter examined the geometric foundations of many essential machine learning models, such as K-means clustering and linear (or polynomial) classification models. These insights will lead to polynomial time algorithms for solving these combinatorially intractable machine learning problems, given the combinatorial generators introduced in Section II.1.2 and Section II.2.3. Part III will demonstrate how the geometric insights gained here can be applied to solve various fundamental machine learning problems.

Part III

Specialized theory: Designing tractable algorithms for fundamental problems in machine learning

This Part contains the **specialized theory** of this thesis which examines the essential combinatorial properties of several fundamental problems in machine learning: *classification problems* with linear or polynomial hypersurface decision boundaries, *K-clustering* (including K-means and K-medoids), the *empirical risk minimization* (ERM) problem for *feedforward neural networks* with ReLU activation functions (ReLU network), and *decision tree problems* with axis-parallel hyperplane, hyperplane, and hypersurface splitting rules.

Although all the problems examined in this part have been proven to be NP-hard in the past, it will be shown that all these problems can be solved in polynomial time when certain parameters, such as the dimensionality or the number of hyperplanes of the model, are fixed. Additionally, by analyzing their combinatorial properties, it will be demonstrated that algorithms for solving these problems can be easily derived using the catamorphism generators introduced in earlier sections.

Each chapter within this part is dedicated to a specific problem. The discussion in each chapter is organized into four sections: review of related studies, problem definition, the essential combinatorial properties of the problem (which will be further split up to discuss the combinatorial complexity of this problem and the design of the combinatorial generator), and further discussion. The further discussion addresses acceleration techniques tailored to each problem, which can result in algorithms that explore a provably smaller number of configurations than in the worst case, potentially achieving near-linear time efficiency in the best case. Additionally, advantages, drawbacks, and possible extensions to related problems are examined, providing a comprehensive understanding of each problem's challenges.

The theory of constructive algorithmics discussed in Chapter II.2 is applied exclusively to the algorithms developed in Chapters III.2 and III.3. In contrast, Chapters III.1 and III.4 focus solely on analyzing the combinatorial and geometric aspects of these problems. As mentioned, our framework integrates combinatorial generation, combinatorial geometry, and constructive algorithmics. For the problems in Chapters III.1 and III.4, our contributions lie in the generation aspect—applying the generic join-list combinatorial generator we designed—and the geometric aspect—identifying equivalence classes using geometric techniques—rather than in the algorithmic aspect.

This distinction arises because the combinatorial structures in the problems discussed in Chapters III.1 and III.4, once simplified using the geometric techniques introduced in this part, reduce to the simplest combinatorial objects such as *combinations* or *0-1 loss binary assignments*. Consequently, these problems can be solved directly using the combinatorial generators introduced in II.2.3.3. However, the simplicity of these combinatorial objects leaves no room for potential speed-up from a purely algorithmic perspective. In contrast, the problems discussed in Chapters III.2 and III.3 involve more complex combinatorial structures, such as *binary trees* and *combinations of combinations* (which we refer to as nested combinations), which

require the use of new datatypes to represent the configurations and the development of new algorithms through equational reasoning.

By analyzing the essential combinatorial aspects of these widely-used machine learning problems, we will demonstrate the significant potential of using the algorithm design framework presented in this thesis to creating efficient and exact algorithms tailored to some of the most difficult problems in machine learning.

III.1 The classification problem

Algorithms for solving the linear classification problem have a long history, dating back at least to 1936 with linear discriminant analysis (LDA). The ultimate objective of the linear classification problem is to find a hyperplane decision boundary that minimizes the number of misclassified data points. In other words, the aim is to solve the linear classification problem with *0-1 loss objective*.

For linearly separable data, many algorithms can obtain the exact solution to the corresponding 0-1 loss classification problem efficiently, but for data which is not linearly separable, it has been shown that this problem, in full generality, is NP-hard. Alternative approaches all involve approximations of some kind, including the use of *surrogates* for the 0-1 loss (for example, the *hinge* [Cortes and Vapnik, 1995] or *logistic* loss [Cox, 1958, 1966]) or approximate combinatorial search, none of which can be guaranteed to solve the problem exactly. Designing efficient algorithms to obtain an exact i.e. globally optimal solution for the 0-1 loss linear classification problem with fixed dimension, remains an open problem.

This chapter provides a detailed analysis of the combinatorial properties of the 0-1 loss linear classification problem by examining two representations of hyperplanes: one representation characterizes a hyperplane in $\mathbb{R}^D$ as *D-combinations of data points*, while the other characterizes them through the *prediction labels* (*assignment*) of a hyperplane with respect to a set of data. Each representation offers unique advantages that cannot be fully replaced by the other.

Moreover, this chapter will discuss the generalization of the novel algorithms presented here, to addressing both the polynomial hypersurface classification problem and the linear classification problem beyond the 0-1 loss. This includes tackling problems with intractable combinatorial objectives, such as the *margin-loss linear classification problem*. The exact margin-loss linear classification algorithm allows for the use of adjustable hyperparameters, potentially leading to more robust solutions for predictions.

III.1.1 Related studies

Classification algorithms have a long history. The first classification algorithms date back to the early 20th century, perhaps most importantly *logistic regression* (LR) which is regarded as one of the most useful algorithms for the linear classification problem. The logistic function first appeared in the early 19th century [Quetelet et al., 1826], and was rediscovered a few more times throughout the late 19th century and early 20th century. However, Wilson and Worcester were the first to use the logistic model in bioassay research [Wilson and Worcester, 1943], and Cox was the first to construct the log-linear model for the linear classification problem [Cox, 1958, 1966]. In 1972, Nelder and Wedderburn first proposed a generalization of the logistic model for linear-nonlinear classification. The support vector machine (SVM) is probably the most famous algorithm for the linear classification problem [Cortes and Vapnik, 1995]; it optimizes the *regularized* hinge loss to obtain a feasible decision hyperplane with *maximal margin*. Most of these algorithms can be considered as optimizing over convex surrogate losses for the 0-1 loss function. Recent studies have shown that optimizing surrogate losses, such as the hinge loss, lacks robustness to outliers [Long and Servedio, 2008, Liu and Wu, 2007]. The objectives of these surrogate losses, while leading to computationally efficient algorithms, fail to be robust compared with exact algorithms.

Little work appears to have been devoted to exact algorithms for the 0-1 loss classification problem. Tang et al. [2014] implemented an mixed-integer programming (MIP) approach to obtain the maximal

margin boundary for the optimal 0-1 loss, and Brooks [2011] optimized SVM with "*ramp loss*" and the hard-margin loss using a *quadratic mixed-integer program* (QMIP), where the ramp loss is a continuous function mixed with the 0-1 loss. Problems which involve optimizing the integer coefficient linear classifier —the normal vector used to define linear classifier involves only integer values—have also drawn some attention [Chevaleyre et al., 2013, Carrizosa et al., 2016], again exact solutions have only been obtained using inefficient MIP. *Scoring system research* is related to linear classification with integer coefficients, but many scoring systems are built using traditional heuristic/approximate classification methods, and Ustun [2017]'s empirical results show that the loss is substantial if optimizing convex surrogates. Therefore, Ustun and Rudin [2019] presented a *cutting-plane algorithm* to learn an optimal risk score, or solving it by formulating it as a MIP problem[Ustun and Rudin, 2016].

Perhaps closest to our work, Nguyen and Sanner [2013] developed a branch-and-bound algorithm (BnB) for solving (123). Nguyen and Sanner [2013] also constructed a polynomial-time combinatorial search algorithm which is similar to an algorithm in this thesis, but gave no proof of correctness of their algorithm. Hence, previous work on this problem of solving (123) is either computationally intractable, i.e. worse case exponential run-time complexity, or uses inefficient, off-the-shelf MIP solvers, or is not proven correct yet [Nguyen and Sanner, 2013].

Previously, the author and others implemented in Python, three other novel, algorithms (E01-ICG, E01-ICG-purge and E01-CE) for the 0-1 loss linear classification problem [He and Little, 2023a], and another algorithm called "E01-ICE" [He and Little, 2023b]. Although the discussions above did not include correctness proofs and formal algorithm derivations for the E01-ICG and the E01-ICG-purge algorithm, small-scale experiments have been conducted to compare the wall-clock run-time of these three algorithms against an earlier BnB algorithm. A detailed discussion on the correctness of these three algorithms will be presented in the discussion here.

III.1.2 Problem specification

The definition of the 0-1 loss linear classification problem is very simple: given data set $\mathcal{D} = \{\boldsymbol{x}_1, \dots \boldsymbol{x}_N\}$ and its associated binary label vector $\boldsymbol{t} = (t_1, \dots, t_N)^T$, find a linear hyperplane defined by normal vector $\boldsymbol{w}$ that minimizes the number of misclassified data points[22]. In other words, solve an optimization problem that minimizes the following objective

$$E_{0\text{-}1}(\boldsymbol{w}) = \sum_{n \in \mathcal{N}} \mathbf{1}\left[\text{sign}\left(\boldsymbol{w}^T \bar{\boldsymbol{x}}_n\right) \neq t_n\right], \tag{121}$$

which is a sum of 0-1 loss functions $\mathbf{1}[\,]$, each taking the value 1 if the Boolean argument is true, and 0 if false. The function sign returns 1 if the argument is positive, and 0 if negative. The linear decision function $\boldsymbol{w}^T \bar{\boldsymbol{x}}$ with parameters $\boldsymbol{w} \in \mathbb{R}^{D+1}$ ($\bar{\boldsymbol{x}} = \left(\boldsymbol{x}^T, 1\right)^T$ is the data in *homogeneous* coordinates) is highly interpretable since it represents a simple hyperplane boundary in feature space separating the two classes.

According to Vapnik [1999]'s *generalization bound theorem*, for the hyperplane classifier defined by $\boldsymbol{w}$, with high probability,

[22] Misclassified data points refer to instances in a dataset that a model (classifier) incorrectly predicts. In other words, data $\boldsymbol{x}_n$ is misclassified if sign $\left(\boldsymbol{w}^T \bar{\boldsymbol{x}}_n\right) \neq t_n$.

$$E_{\text{test}} \leq E_{\text{0-1}}(\boldsymbol{w}) + O\left(\sqrt{\frac{\log(N/(D+1))}{N/(D+1)}}\right), \tag{122}$$

where E_{test}, $E_{\text{0-1}}(\boldsymbol{w})$ are the *test 0-1 loss* (the number of misclassification on unseen data (the test dataset)) and the *empirical 0-1 loss* (the number of misclassification on training data (the training dataset)) of the linear model on the training data set, respectively [Mohri et al., 2018]. Equation (122) motivates finding the exact 0-1 loss on the training data, since, among all possible linear hyperplane classifiers, with high probability, none has better worst-case test 0-1 loss than the exact classifier.

The 0-1 loss linear classification problem requires finding an optimal $\boldsymbol{w}^*$ that minimizes (121):

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}\in\mathbb{R}^{D+1}}{\operatorname{argmin}} E_{\text{0-1}}(\boldsymbol{w}). \tag{123}$$

Although apparently simple, this is a surprisingly challenging optimization problem. Considered as a continuous optimization problem in $\boldsymbol{w}$, the standard ML optimization technique, gradient descent, is not applicable (since the gradients of $E_{\text{0-1}}(\boldsymbol{w})$ with respect to $\boldsymbol{w}$ are zero everywhere they exist), and the problem is non-convex so there are a potentially very large number of local minima in which gradient descent can become trapped. Heuristics exist, in particular the classic *perceptron training algorithm* and variants which cannot guaranteed to find the global minima. By replacing the loss function $\mathbf{1}[\,]$ with more manageable *surrogates* such as the *hinge loss* $E_{hinge}(\boldsymbol{w}) = \sum_{n\in\mathcal{N}} \max\left(0, 1 - l_n \boldsymbol{w}^T \bar{\boldsymbol{x}}_n\right)$ which is convex and differentiable nearly everywhere, the corresponding optimization is a linear problem solvable by general-purpose algorithms such as *interior-point primal-dual* optimization[Little, 2019]. However, this can only find a sub-optimal decision function corresponding to an upper bound on the globally optimal value of the objective $E_{\text{0-1}}(\boldsymbol{w})$.

An alternative is to recast the problem as a combinatorial one, in the following way. Every choice of parameters $\boldsymbol{w}$ determines a particular classification prediction or *binary assignment*, $s = \left(\text{sign}\left(\boldsymbol{w}^T\boldsymbol{x}_1\right), \text{sign}\left(\boldsymbol{w}^T\boldsymbol{x}_2\right), \ldots, \text{sign}\left(\boldsymbol{w}^T\boldsymbol{x}_N\right)\right) \in \mathcal{S}_{\text{basgn}}$, where $\mathcal{S}_{\text{basgn}} = \{1,-1\}^N$ is the *discrete search* or *solution space*, which consists of all possible 2^N *assignments/configurations*. Every assignment entails a particular total loss $E_{\text{0-1}}(\boldsymbol{w})$, and it is required to find the assignment that can obtain the optimal 0-1 loss $\hat{E}_{\text{0-1}}$. A naive way to solve the 0-1 loss linear classification problem is to use the binary assignment SDP generator introduced in Section II.2.3 to generate all possible binary assignments, and then select the best one, but this naive approach is inefficient since there are 2^Npossible binary assignments, and this problem is not a greedy problem, so there is no obvious way to fuse the selector `sel` inside the catamorphism generator.

Nevertheless, the number of data points N is finite and these points occupy zero volume of the real feature space $\mathbb{R}^{D+1}$. This implies that there are a finite number of *equivalence classes* of decision boundaries which share the same assignment. In fact, the geometry of the problem implies that not all binary assignments correspond to one of these equivalence classes; the only ones which do are the (linear) dichotomies, and while there are 2^N possible assignments, as we discussed in II.3.2, there are only $O\left(N^D\right)$ dichotomies for data set in D dimensions . Thus, the problem (123) can instead be treated as the combinatorial optimization problem of finding the best such dichotomies and their implied assignments, from which an optimal parameter $\boldsymbol{w}^*$ can be obtained.

III.1.3 The essential combinatorial properties of the linear classification problem

Section II.3.3 introduced two ways of characterizing the cells of an arrangement: a cell can be represented by its *sign vector* with respect to an arrangement $\mathcal{H}$, or by the *intersection of D hyperplanes*, i.e. vertices in set $\mathcal{V}_{\mathcal{H}}$. Similarly, due to point-line duality, these two representations are also valid for hyperplanes: a hyperplane can be characterized by *predicted labels* of a decision hyperplane with respect to $\mathcal{D}$, or by a *D-combination* of data items.

This section will discuss how to construct an efficient algorithm for solving the linear classification problem with 0-1 loss by analyzing the combinatorial properties of the two hyperplane representations mentioned above.

III.1.3.1 Hyperplane-based (H-based) algorithm

The exposition in Section II.3.2 introduced the relationships between dichotomies and separating hyperplanes for a given dataset $\mathcal{D}$, along with their connections in the dual space in terms of the cells and vertices of the dual arrangement $\mathcal{H}_{\mathcal{D}}$. The key to solving a linear classification problem is to enumerate all possible dichotomies, which is equivalent to enumerating the cells of the dual arrangement. It also presented two cell enumeration algorithms based on vertex enumeration in Subsection II.3.3.2, both with a worst-case complexity of $O\left(N^D\right)$. According to the linear classification theorem 12, it follows immediately that the 0-1 loss linear classification problem can be solved in $O\left(N^{D+1}\right)$ time, given $O\left(N\right)$ time for evaluating the 0-1 loss objective.

On the one hand, applying the cell enumeration algorithms aims to solve linear classification problems in general, i.e. linear classification problems with arbitrary objective functions. Regardless of the objective function chosen, all linear classification problems can be solved exactly by exhaustively enumerating all possible dichotomies.

On the other, each objective function possesses unique geometric, algebraic, or even combinatorial properties, which may enable more efficient methods for solving the linear classification problem if these properties are exploited. This section will demonstrate that this is indeed the case for the linear classification problem with 0-1 loss objective. It will show that the 0-1 loss linear classification problem can be solved more efficiently by enumerating only the vertices of the dual arrangement $\mathcal{H}_{\mathcal{D}}$, which correspond to the decision hyperplanes with D points lying on them in the primal space.

The informal intuition behind this claim goes as follows. It is possible to construct these solutions by selecting all D out of the N data points, finding the hyperplane that passes exactly through these points, and computing the corresponding assignments for the entire dataset along with their associated 0-1 loss. Each such hyperplane has two possible orientations, leading to two corresponding assignments. However, the D points used to construct these two hyperplanes, have undecided class assignments because the boundary goes exactly through them (so the classification model evaluates to 0 for these points). There are 2^D possible assignments of class labels to the D points on the boundary, and each of these 2^D assignments is a unique dichotomy. The best such dichotomy is the one with the smallest 0-1 loss, and this is guaranteed by selecting the labels of the D points such that they agree with their labels in the training data. The following theorem proves the above claim.

Theorem 15. Consider a dataset $\mathcal{D}$ of N data points of dimension D in general position, along with their associated labels, denote $\mathcal{S}_{\text{kcombs}}$ as the set of all D-combinations with respect to dataset $\mathcal{D}$. Then we have following inequality

$$\underset{s\in\mathcal{S}_{\text{kcombs}}}{\operatorname{argmin}} \min\left(E_{\text{0-1}}\left(\boldsymbol{w}_s\right), E_{\text{0-1}}\left(-\boldsymbol{w}_s\right)\right) \subseteq \underset{\boldsymbol{w}\in\mathbb{R}^{D+1}}{\operatorname{argmin}} E_{\text{0-1}}\left(\boldsymbol{w}\right) \tag{124}$$

where $\boldsymbol{w}_s$ represents the normal vector of the hyperplane that pass through the D-combination of data s, and $-\boldsymbol{w}_s$ is the negation of $\boldsymbol{w}_s$. The inner min on the left-hand side ensures that for each $s \in \mathcal{S}_{\text{kcombs}}$, we take the smaller of $E_{\text{0-1}}\left(\boldsymbol{w}_s\right)$ and $E_{\text{0-1}}\left(-\boldsymbol{w}_s\right)$, the outer argmin finds *one* of the value of that minimizes this quantity over all $s \in \mathcal{S}_{\text{kcombs}}$.

In other words, (124) means that all globally optimal solutions to problem (123), are equivalent (in terms of 0-1 loss) to the optimal solutions contained in the set of solutions of all positive and negatively-oriented linear classification decision hyperplanes (vertices in the dual space) which go through D out of N data points in the dataset $\mathcal{D}$.

Proof. First, transform a dataset$\mathcal{D}$ to its dual arrangement. According to Lemma 8 and Lemma 9, each dichotomy has a corresponding dual cell and if the sign vectors for all possible cells in the dual arrangement and their reverse signs are evaluated, the optimal solution for the 0-1 loss classification problem can be obtained. Assume the optimal cell is f, it is required to prove that, one of the adjacent vertices for this cell is also the optimal vertex. Then, finding an optimal vertex is equivalent to finding an optimal cell since the optimal cell is one of the adjacent cells of this vertex. According to Lemma 11, any vertices that are non-conformal have corresponding 0-1 loss with respect to $\text{sign}_{\mathcal{H}}\left(f\right)$ which is strictly greater than D. Since f is optimal, any sign vectors with larger sign difference (with respect to $\text{sign}_{\mathcal{H}}\left(f\right)$) will have larger 0-1 loss value (with respect to true label $\boldsymbol{t}$). Therefore, vertices that are conformal to f will have smaller 0-1 loss value, thus one can evaluate all vertices (and the reverse sign vector for these vertices) and choose the best one, which, according to Lemma 9, is equivalent to evaluating all possible positive and negatively-oriented linear classification decision hyperplanes and choosing one linear decision boundary with the smallest 0-1 loss value. □

Therefore, the 0-1 loss linear classification problem can be solved exactly by running two separate combination generators to enumerate the D-combinations of data points, and this can be done efficiently using any combinatorial generator introduced in Section II.2.3 with complexity $O\left(2 \times \binom{N}{d} \times N \times D^3\right) = O\left(N^{D+1}\right)$, where $O\left(D^3\right)$ time is required for obtaining the hyperplanes with D points lying on it.

Moreover, in the next section, it will be explained how the generators for enumerating positive and negative-oriented hyperplanes can be fused into a single process, thereby reducing the computational time by half.

III.1.3.2 Linear programming-based (LP-based) algorithm

The linear programming-based method characterizes a hyperplane as a binary assignment (prediction labels for data items in $\mathcal{D}$), which is the same as characterizing it as the sign-vector of an arrangement $\mathcal{H}_{\mathcal{D}} = \phi\left(\mathcal{D}\right)$. In Subsection II.3.3.1, two cell enumeration algorithms were introduced, the *reverse search algorithm* (100) and the *incremental sign construction algorithm* (102). Both algorithms can be applied to

solve linear classification problems in general. However, for solving the linear classification problem with the 0-1 loss objective, the cell generator (102) is more suitable than the reverse search algorithm (100). It will be explained why this is the case in the discussion of this section.

Consider a data set $\mathcal{D} = \{\boldsymbol{x}_n : n \in \mathcal{N}\}$ in $\mathbb{R}^D$ with true label vector $\boldsymbol{t} = (t_1, \ldots, t_N)$. Each data point can either have a positive prediction label 1 or negative prediction label -1, the prediction labels for all data points consists of a length N binary assignment $\boldsymbol{y} \in \{1, -1\}^N$. In previous research [He and Little, 2023a], the author and collaborators introduced the E01-ICG (short for, exact 0-1 incremental combinatorial generation) algorithm, which is the dual version of the (102). The E01-ICG generates binary assignments by recursively appending new prediction labels (positive and negative) to each partial assignment, and the 0-1 losses for these candidate configurations are updated in each recursive step. During iteration, infeasible configurations (that is, ones which are not linearly separable or have 0-1 loss which is larger than the given global upper bound) are filtered out. The feasibility of a binary assignment $\boldsymbol{y}$, for $1 \leq n \leq N$ is tested by the following linear program

$$\begin{aligned} &\min_{\boldsymbol{w}} \ \mathbf{0}^T \boldsymbol{w} \\ &\text{s.t. } \operatorname{diag}(\boldsymbol{y})^T \overline{\boldsymbol{X}} \boldsymbol{w} \geq \mathbf{1}. \end{aligned} \tag{125}$$

Dually, the linear program (125) is essentially the same as the cell feasibility test (99).

The E01-ICG algorithm is essentially the same as the recursion (102), except for the additional recursive update process for the 0-1 loss. Consider how the 0-1 loss should be updated directly in the recursion (102). In each recursion of (102), the two possible choices for a new class label $y_{n+1} \in \{1, -1\}$ are appended to all partial configurations generated so far. The objective values do not decrease through the recursive update process of the 0-1 loss because, for a partial binary assignment $\boldsymbol{y}' \in \{1, -1\}^n$ where $1 \leq n < N$, the 0-1 loss of $\boldsymbol{y}'$ either increases by one or remains the same after appending a new label y_{n+1} to it. Therefore, the objective values of the configurations are updated in a *monotonically non-decreasing* manner. It follows immediately that the dominance relations introduced in Subsection II.2.6.3, such as the global upper bound technique can be easily incorporated into the recursion (102), allowing any partial assignments with a 0-1 loss worse than the global upper bound to be discarded without further extension.

By contrast, the reverse search algorithm starts with a cell (a dichotomy in the primal space) where the sign vector has all positive labels, and then recursively flips positive signs $+$ to negative $-$. It is clear that this process does not have the property of being monotonically non-decreasing with respect to the objective, as long as the true label vector $\boldsymbol{t}$ is not all positive.

The monotonically non-decreasing property is crucial in optimization problems, as it enables the incorporation of various powerful dominance relations. In the worst case, assuming no acceleration techniques are applied, the E01-ICG algorithm will have complexity (104) in order to enumerate all possible dichotomies (cells in the dual space) in $\mathbb{R}^D$ (a data set in $\mathbb{R}^D$ is isomorphic to a central arrangement in $\mathbb{R}^{D+1}$). Additionally, $O(N \times \operatorname{Cover}(N, D))$ operations are needed to evaluate the 0-1 loss for each dichotomy. Thus, the E01-ICG algorithm has a complexity of

$$O\left(\sum_{n=0}^{N} \left(\operatorname{LP}(n, D) \times \operatorname{Cover}(n-1, D)\right) + N \times \operatorname{Cover}(N, D+1)\right) = O\left(\operatorname{LP}(N, D) \times N^D + N^{D+1}\right). \tag{126}$$

In the best case, where the upper bound is tightest (i.e., the data is linearly separable), the E01-ICG algorithm will terminate in at most $O\left(\sum_{n=0}^{N} \mathrm{LP}\left(n, D\right)\right)$ time. Similarly, if a linear program can be solved in $O\left(N^3\right)$ time, then the complexity of the assignment generation, as described in (126) is upper-bounded by $O\left(N^{D+3}\right)$.

III.1.4 Further discussions

III.1.4.1 Difference between H-based algorithm and LP-based algorithm

As previously discussed, the H-based and LP-based algorithms represent the combinatorial aspects of the problem in fundamentally different ways. Each algorithm introduces unique optimization features that cannot be replaced by the other. This Subsection, will explore their differences in terms of the acceleration techniques applicable to each, as well as their respective advantages and limitations.

Acceleration for H-based algorithms Due to the *symmetry* of the 0-1 loss, where a data item is assigned a label of either 1 or -1, the 0-1 loss for the negative orientation of a hyperplane can be directly derived from the positive orientation of the same hyperplane without calculating it explicitly. The following lemma formalizes this relationship.

Theorem 16. *Symmetry fusion theorem.* Consider a dataset $\mathcal{D}$ of N data points of dimension D in general position, along with their associated labels. Let h be a hyperplane which goes through D out of N data points in the dataset $\mathcal{D}$, separating the dataset into two disjoint sets $\mathcal{D}^+$ and $\mathcal{D}^-$. If the 0-1 loss for the positive orientation of this hyperplane is l, then the 0-1 loss for the negative orientation of this hyperplane is $N - l - D$.

Proof. Assume there are m^+ and m^- data points misclassified in $\mathcal{D}^+$ and $\mathcal{D}^-$, then the 0-1 loss for h equals $l = m^+ + m^-$. Denote the hyperplane h with negative orientation as h^-. In the partition introduced by h^-, all correctly classified data by h will be misclassified in h^-. Thus the 0-1 loss of h^- is $|\mathcal{D}^+| - m^+ + |\mathcal{D}^-| - m^-$. Since $|\mathcal{D}^+| + |\mathcal{D}^-| = N - D$, we obtain the 0-1 loss for h^- which is $N - D - l$. □

Acceleration in LP-based algorithms The two dominance relations, *finite dominance* (65) and *global upper bound* (63), are readily applicable in this context. As demonstrated in Subsection II.2.6.3, using these dominance relations to eliminate non-optimal partial configurations maintains exactness, provided that the update function is *monotonic* with respect to the objective. The monotonicity of the update function in the LP-based algorithm was confirmed in the preceding discussion on the LP-based algorithm.

The informal intuition behind the correctness of these dominance relations is as follows: for the global upper bound dominance relation, any *partial configuration* with an objective worse than an approximate solution is provably non-optimal and can therefore be safely discarded. In the case of the finite dominance relation, the *optimistic upper bound* of a partial configuration is calculated by assuming that all unobserved extensions of prediction labels are correct. If this optimistic upper bound is worse than either the current global upper bound or the *pessimistic lower bound*—obtained by assuming that all unobserved extensions of prediction labels are incorrect—the configuration can be discarded without compromising optimality.

The use of global upper bound techniques [He and Little, 2023b, Lin et al., 2020] has been empirically shown to be extremely powerful, as it can yield nearly linear time complexity in the best-case.

Algorithms	Configurations need to explores in the worst-case	Required operations to obtain the representation of hyperplane	The effectiveness of upper-bound	Recursively generates hyperplanes	Memory usage for each configuration
H-based	$\begin{pmatrix} N \\ D \end{pmatrix}$	Matrix inversion (efficient)	Limited effectiveness	Yes	$O(D)$
LP-based	$2\sum_{d=0}^{D}\begin{pmatrix} N-1 \\ d \end{pmatrix}$	Linear programming (much less efficient)	High effectiveness	No	O(N)

Table 2: Comparison between H-based methods and LP-based methods for the exact linear classification problem.

Complexity comparisons between H-based algorithm and LP-based algorithms The key distinctions between the H-based method and the LP-based method for the 0-1 loss linear classification problem are presented in Table 2. In summary, the H-based algorithm offers unique advantages in two key aspects: its *exceptional performance on low-dimensional, small-scale problems* and its *incremental hyperplane generation process.*

H-based algorithms often outperform LP-based methods in low-dimensional, small-scale problems for two main reasons: firstly, H-based algorithms can obtain the exact solution by exploring all possible $\begin{pmatrix} N \\ D \end{pmatrix}$ combinations, whereas LP-based methods must consider all possible dichotomies, which are provably larger in number than the D-combinations of data items for data set $\mathcal{D}$. Secondly, H-based algorithms generate hyperplanes through matrix inversion, which has a worst-case time complexity of $O(D^3)$. By contrast, LP-based methods require solving a large number of linear programs, and solving a linear program with D variables typically takes more time than matrix inversion when using classical LP solvers, such as the simplex method.

Furthermore, the incremental hyperplane generation process is critical for applications that require a balance between accuracy and computational time. By contrast, LP-based algorithms are unable to achieve this balance because a hyperplane represented by a partial assignment does not suffice for constructing a feasible solution. Additionally, LP-based methods consume more memory to represent a hyperplane; characterizing a hyperplane through its prediction labels requires $O(N)$ space for each configuration, whereas representing a hyperplane by the data points that lie on it requires only $O(D)$ space.

Despite the disadvantages of LP-based algorithms, they possess unique advantages that cannot be easily replaced by H-based algorithms. First, LP-based methods are significantly more effective for high-dimensional datasets compared to H-based algorithms. High-dimensional data is often easier to classify than low-dimensional data, thus the approximate solutions for high-dimensional problems tend to be highly accurate. Therefore, the use of a global upper bound allows the algorithm to eliminate a very

large fraction of candidate solutions before extending them to completion. By contrast, the global upper bound technique is only partially applicable to H-based algorithms. This limitation arises from relaxing the fixed-length predicate (*non-prefix-closed*) into a max-length predicate (*prefix-closed*), which restricts the generator's ability to fully exploit the global upper bound technique.

Consequently, d-combinations, where $0 \leq d < D$ are insufficient to construct a hyperplane, preventing the evaluation of the objective for d-combinations. Moreover, these d-combinations must be stored until all possible complete D-combinations are generated.

III.1.4.2 Non-linear (polynomial hypersurface) classification

Following the earlier discussion on the *Veronese embedding* in Subsection II.3.2.3, it is straightforward to generalize the novel algorithms for the linear classification problem to the polynomial hypersurface classification problem, since a polynomial hypersurface is isomorphic to a hyperplane in higher-dimensional embedding space $\mathbb{R}^G$, where $G = \binom{D+W}{D} - 1$. The hypersurface classification theorem (II.3.2.3) states that an $O\left(t_{\text{eval}} \times N^G\right)$-time algorithm for solving the hypersurface classification problem in general exists.

Similar to Thm. 15, the special combinatorial property of the 0-1 loss allows solving the problem more efficiently, which is described in the following theorem.

Theorem 17. *0-1 loss hypersurface classification theorem.* Consider a dataset $\mathcal{D}$ of N data points in general position in $\mathbb{R}^D$, along with their associated labels. All globally optimal solutions to problem (123) with objective

$$E_{\text{0-1}}\left(\boldsymbol{w}\right) = \sum_{n \in \mathcal{N}} \mathbf{1}\left[\text{sign}\left(\boldsymbol{w}^T \tilde{\boldsymbol{x}}_n\right) \neq t_n\right], \tag{127}$$

where $\tilde{\boldsymbol{x}}_n = \rho_W\left(\bar{\boldsymbol{x}}_n\right)$ denotes the W*-tuple Veronese embedding* of the homogeneous data $\bar{\boldsymbol{x}}_n = (1, x_{n1}, \ldots, x_{nD})$, and $\mathbf{w} \in \mathbb{R}^G$ where $G = \binom{D+W}{D} - 1$ is the number of monomials of a degree W-polynomial, are equivalent (in terms of 0-1 loss) to the optimal solutions contained in the set of solutions of all positive and negatively-oriented linear classification decision hyperplanes which go through G out of N data points in the dataset $\mathcal{D}$.

However, the dimension of the space that the isomorphic hyperplane lies in will increase exponentially with increasing D and W.

III.1.4.3 Margin loss linear classifier

The linear classification theorem 12 states that an $O\left(t_{\text{eval}} \times N^G\right)$-time algorithm for solving the linear classification problem can be constructed by enumerating all possible cells of the dual arrangement $\mathcal{H}_{\mathcal{D}}$. This holds true regardless of the loss function used. To illustrate this, it will be briefly explained how the same algorithmic process of exhaustively generating cells can be applied to the *margin loss* linear classification problem, thereby allowing the incorporation of a hyperparameter to mitigate overfitting.

The ρ*-margin loss* is a *discrete* variant of the *hinge loss* used in the well-known support vector machine (SVM) algorithm, which is defined as follows.

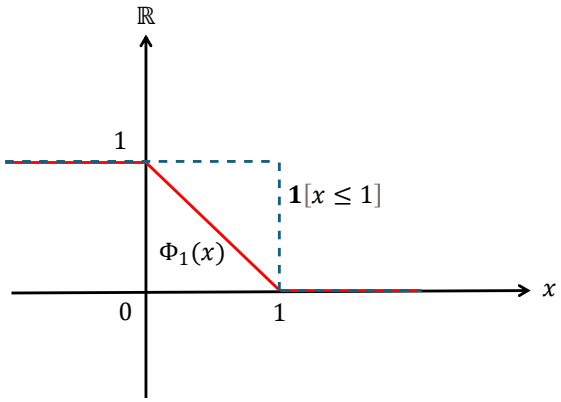


Figure 17: The margin loss $\Phi_\rho(x)$ (red), defined with respect to margin parameter $\rho = 1$ is upper bounded by the shifted 0-1 loss $\mathbf{1}[x \leq \rho]$ (blue dotted line), i.e., $\Phi_\rho(x) \leq \mathbf{1}[x \leq \rho]$, where $x = y_n h(n)$ in the classification problem.

Definition 34. For any $\rho > 0$, the the ρ-margin loss is the function $\Phi : \mathbb{R} \times \mathbb{R} \to \mathbb{R}^+$ defined by

$$\Phi_\rho(x) = \min\left(1, \max\left(0, 1 - \frac{x}{\rho}\right)\right) = \begin{cases} 1 & \text{if } x < 0 \\ 1 - \frac{x}{\rho} & \text{if } 0 \leq x \leq \rho \\ 0 & \text{if } \rho < x. \end{cases} \tag{128}$$

As shown in Fig. 17, the ρ-margin loss is upper bound by the *shifted 0-1 loss*, i.e.,

$$\Phi_\rho(y_n h(n)) \leq \mathbf{1}[y_n h(n) \leq \rho], \forall n \in \mathcal{N}, \tag{129}$$

where $h(n) = \boldsymbol{w}^T \boldsymbol{x}_n + b_n$ and y_n is the prediction label.

The value $y_n\left(\boldsymbol{w}^T \boldsymbol{x}_n + b_n\right)$, which is always positive $y_n\left(\boldsymbol{w}^T \boldsymbol{x}_n + b_n\right) > 0$ for all $n \in \mathcal{N}$, is known as the *functional margin* or *confidence margins*. The *normalized functional margin* $\frac{y_n\left(\boldsymbol{w}^T \boldsymbol{x}_n + b_n\right)}{|\boldsymbol{w}|}$ is known as the *geometric margin*, i.e., the *unit distance* of $\boldsymbol{x}_n$ to the decision boundary.

The objective function for the margin loss linear classification problem can be defined as the summation of the margin loss of each data item

$$E_{\text{margin}}(\boldsymbol{w}) = \sum_{n \in \mathcal{N}} \Phi_\rho\left(y_n\left(\boldsymbol{w}^T \bar{\boldsymbol{x}}_n\right)\right). \tag{130}$$

To state a result in learning theory [Mohri et al., 2018], let $\mathcal{D}$ be a dataset such that $\|\boldsymbol{x}_n\| \leq r$, $\forall \boldsymbol{x}_n \in \mathcal{D}$, and let $\mathcal{H} = \left\{\boldsymbol{x} \mapsto \boldsymbol{w}^T \boldsymbol{x} : \|\boldsymbol{w}\| \leq \Lambda\right\}$. Then, with probability at least $1 - \delta$, the generalization (test) error E_{test} for the binary linear classification task with margin loss objective is upper bounded by

$$E_{\text{test}} \leq E_{\text{margin}}(\boldsymbol{w}) + 2\sqrt{\frac{r^2 \Lambda^2 / \rho^2}{N}} + \sqrt{\frac{\log \frac{2}{\delta}}{2N}}. \tag{131}$$

Compared to the VC-dimension bound (122), which is defined in terms of the 0-1 loss, the generalization bound based on the ρ-margin loss offers two significant advantages. First, the generalization bound (131)

provides a trade-off: a larger value of ρ decreases the complexity term $\mathfrak{R}_N(\mathcal{H})$ (second term), but tends to increase the empirical margin-loss $E_{\text{margin}}(\boldsymbol{w})$ (first term) by requiring from a hypothesis a higher confidence margin. Second, the generalization bound (131) is remarkable, because it does not directly depend on the dimension of the feature space but only on the margin. This contrasts with the VC-dimension lower bound, which is dimension-dependent.

The MIP specification for the *margin loss linear classification problem* is defined as

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}\in\mathcal{H}}{\operatorname{argmin}}\ E_{\text{margin}}(\boldsymbol{w}) \tag{132}$$

Since the margin loss is discrete, the margin loss linear classification problem (132) cannot be directly optimized using convex optimization methods.

From the definition of the margin loss objective (130), for data points that are correctly classified, denote $\boldsymbol{y}^+$ as the labels corresponding to correctly classified instances with index I^+ and $\boldsymbol{y}^-$ as the labels corresponding to incorrectly classified instances with index I^-. There are two cases for correctly classified data points: firstly, if the data points lie *within* the confidence margin, i.e., the data points $\boldsymbol{x}$ have a distance $0 \leq \boldsymbol{w}^T\bar{\boldsymbol{x}} \leq \rho$, each of them contributes a margin loss of $1 - \frac{y_i\left(\boldsymbol{w}^T\bar{\boldsymbol{x}}\right)}{\rho}$; secondly, if the data points lie *outside* the confidence margin, i.e., when $\boldsymbol{w}^T\bar{\boldsymbol{x}} > \rho$ they result in zero margin loss. For data points in $\boldsymbol{y}^-$, which are misclassified, each contributes a margin loss of one.

Therefore, given a fixed dichotomy (linearly-separable assignment) $\boldsymbol{y} = (y_1, y_2, \dots y_N)$, optimizing the margin loss, can be achieved by solving the following problem

$$\begin{aligned}
&\min_{w} \sum_{i\in I^+} \frac{\xi_i}{\rho} + \left|I^-\right| \\
&\text{s.t. } y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i + b_i\right) \geq \rho - \xi_i, \forall i \in I^+ \\
&\qquad y_n\left(\boldsymbol{w}^T\boldsymbol{x}_n + b_n\right) \geq 0, , \forall n \in \mathcal{N} \\
&\qquad 0 \leq \xi_i \leq \rho,
\end{aligned} \tag{133}$$

where $y_n\left(\boldsymbol{w}^T\boldsymbol{x}_n + b_n\right) \geq 0$ is the linear feasibility constraints which ensure $\boldsymbol{y}$ is a dichotomy, and ξ_i, $i \in I^+$ are the slack variables. When $\xi_i = 0$, this represents that $\boldsymbol{x}_i$ is correctly classified and its confidence $y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i + b_i\right)$ is greater than ρ. On the other hand, when $0 < \xi_i < \rho$, $\boldsymbol{x}_i$ is correctly classified but its confidence margin is smaller than ρ, it thus has a margin loss $\frac{\xi_i}{\rho} = 1 - \frac{y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i + b_i\right)}{\rho}$. Finally, $\xi_i = \rho$ for data points lying on the decision boundary.

Problem (133) is a linear programming problem that can be optimally solved using standard linear programming solvers. However, this linear program (133) is based on a *functional margin*, thus the size of $\boldsymbol{w}_i$ and b_i can always be scaled to make constraints $y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i + b_i\right) \geq \rho - \xi_i$ feasible. Therefore, it is required to modify (133) to optimize the *geometric margin*

$$\begin{aligned}
&\min_{w,b,\xi_i} \frac{1}{2}\|w\|^2 + \sum_{i\in I^+} \xi_i \\
&s.t.\ y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i + b_i\right) \geq \rho - \xi_i, \forall i \in I^+. \\
&\qquad y_n\left(\boldsymbol{w}^T\boldsymbol{x}_n + b_n\right) \geq 0, , \forall n \in \mathcal{N} \\
&\qquad 0 \leq \xi_i \leq \rho.
\end{aligned} \tag{134}$$

The margin loss linear classification problem can therefore be solved exactly by enumerating all possible dichotomies (cells of the dual arrangements) and the evaluation of the objective function takes $t_{\text{eval}} = qp\left(N, D\right)$ time. According to the linear classification theorem 12, margin loss linear classification problem can be solved exactly in $O\left(qp\left(N, D\right) N^{D}\right)$ time.

III.2 Empirical risk minimization for the deep ReLU network

In recent years, neural networks have emerged as an extremely useful supervised learning technique, developed from early origins in the *perceptron learning algorithm* for classification problems. This model has influenced nearly every scientific field involving data analysis and has become one of the most widely used machine learning techniques today.

This section addresses the problem of *empirical risk minimization for 2-layer feedforward neural network models* (also known as multilayer perceptron, MLP, models) *with rectified linear units* (i.e. ReLU activation functions). Exactly optimizing even a 2-layer network with ReLU activation is known to be NP-hard. The combinatorial properties of this problem are analyzed, and a practical approach for solving it exactly in polynomial time is given, assuming that the dimension and the number of hidden nodes are fixed. Results from complexity theory demonstrate that this new approach achieves optimal efficiency in terms of worst-case complexity.

III.2.1 Related studies

It is well known that deep neural networks with ReLU activation functions are piecewise linear (PWL) classifiers [Arora et al., 2016, Maragos et al., 2021]. Training a PWL model exactly is extremely difficult because, even in the simplest case involving only one hyperplane, the problem remains NP-hard. The exact linear classification algorithm takes $O\left(N^{D+1}\right)$-time to complete in the worst case, which is exponential with respect to the dimension D. Indeed, Bertschinger et al. [2022] have shown that training fully connected neural networks is $\exists\mathbb{R}$-complete. Additionally, Goel et al. [2020] demonstrated that determining whether a 0-1 loss prediction is achievable by a network with K neurons is polynomially solvable when $K = 1$, i.e. in the linear classification case. This can be verified by solving a linear program or by running the LP-based algorithm for the linear classification problem presented earlier in this thesis. For the more general case where $K > 1$, the problem is NP-hard.

To the best of our knowledge, only Arora et al. [2016] have proposed a *one-by-one* enumeration strategy to train a two-layer ReLU neural network to a global optimum for *convex objective functions*. Later, Hertrich [2022] demonstrated that an exhaustive combinatorial search across all $O\left(N^{KD}\right)$ possible partitions introduced by K hyperplanes is essentially the **best** approach available.

However, Arora et al. [2016] only provided pseudo-code and a time complexity analysis, with no publicly available code or empirical analysis to support their claims. Additionally, they did not demonstrate how to enumerate the hyperplanes. From the description of their pseudo-code, it seems they assume these hyperplanes already exist. As such, their discussion appears more like a conjecture, suggesting that a polynomial-time algorithm exists for this problem, rather than presenting an immediately executable solution.

Moreover, the one-by-one enumeration strategy proposed by Arora et al. [2016] is impractical. To initiate the algorithm, it requires all possible representations of hyperplanes to be pre-stored, which is both memory-demanding and inefficient. As we have demonstrated, the possible partitions of hyperplanes have a size of $O\left(N^{D}\right)$.

Nevertheless, this chapter proposes an algorithm to train a two-layer neural network exactly for *arbitrary* loss functions, whereas previous complexity analyses have focused on either squared or convex

loss functions. This approach can be easily generalized to the multilayer case by training it greedily, with the worst-case complexity remaining the same. Unlike prior methods, this algorithm does not require all possible hyperplanes to be pre-stored. Instead, the feasible ReLU networks are generated recursively by the generator introduced to solve it.

III.2.2 Problem specification

Extend the ReLU activation function $\mathrm{ReLU}(x) = \max(0, x)$ to vectors $\boldsymbol{x} \in \mathbb{R}^D$ through entry-wise operation:
$\sigma(\boldsymbol{x}) = (\max(0, x_1), \max(0, x_2), \ldots, \max(0, x_D))$.

Now, consider a 2-layer feedforward ReLU neural network model with K hidden nodes. Each hidden node corresponds to an affine transformation $f_{\boldsymbol{w}_k} : \mathbb{R}^{D+1} \to \mathbb{R}$, which is defined by the distance to the homogeneous hyperplane H_k with normal vector $\boldsymbol{w}_k \in \mathbb{R}^{D+1}, \forall k \in \mathcal{K}$. These K affine transformations can be represented by a single affine transformation $f_{\boldsymbol{W}_1} : \mathbb{R}^{D+1} \to \mathbb{R}^K$, such that $\boldsymbol{W}_1^T = (\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_K)$. In other words, $\boldsymbol{W}_1 \in \mathbb{R}^{K \times (D+1)}$ is a $K \times (D+1)$ matrix defined by putting each vector $\boldsymbol{w}_k$ in the kth row of $\boldsymbol{W}_1$. Similarly, a linear transformation $f_{\boldsymbol{W}_2} : \mathbb{R}^K \to \mathbb{R}$ can be defined, such that $\boldsymbol{W}_2 = (\alpha_1, \alpha_2, \ldots, \alpha_K)$ corresponds to weights of the hidden layer. Thus, the decision function f for a 2-layer feedforward ReLU neural network can be defined as

$$f_{\boldsymbol{W}_1, \boldsymbol{W}_2} = f_{\boldsymbol{W}_2} \cdot \sigma \cdot f_{\boldsymbol{W}_1}. \tag{135}$$

Although the two parameters $\boldsymbol{W}_1, \boldsymbol{W}_2$ in (135) are hyperplanes in a continuous space, it should be familiar from previous arguments in this thesis that hyperplanes can be characterized combinatorially. Denote the combinatorial search space for $\boldsymbol{W}_1, \boldsymbol{W}_2$ as $\mathcal{S}_{\mathrm{ReLUNN}}$. The empirical risk minimization problem for the 2-layer feedforward ReLU neural network model can be defined as the following optimization problem

$$(\boldsymbol{W}_1^*, \boldsymbol{W}_2^*) = \underset{\boldsymbol{W}_1, \boldsymbol{W}_2 \in \mathcal{S}_{\mathrm{ReLUNN}}}{\operatorname{argmin}} E_{0\text{-}1}(\boldsymbol{W}_1, \boldsymbol{W}_2), \tag{136}$$

where $E_{0\text{-}1}(\boldsymbol{W}_1, \boldsymbol{W}_2) = \sum_{n \in \mathcal{N}} \mathbf{1}[\mathrm{sign}(f_{\boldsymbol{W}_1, \boldsymbol{W}_2}(\bar{\boldsymbol{x}}_n)) \neq t_n]$.

III.2.3 The essential combinatorial properties of the ReLU network

A 2-layer ReLU neural network is a piecewise linear (PWL) model, which can be represented using multiple hyperplanes. Analogous to the linear classification problem, where two algorithms that characterize the combinatorics of hyperplanes based on either data point combinations or prediction labels were introduced, the combinatorics of a ReLU neural network can also be characterized in these two ways. These two different characterizations led to the development of two new algorithms for constructing an exact ReLU network model. These two characterizations will be discussed in detail in this section.

III.2.3.1 Hyperplane-based method

Combinatorial complexity Due to the homogeneity of the ReLU activation function ($\max(0, ab) = a \max(0, b)$, for $a \geq 0$), the decision function introduced by the 2-layer ReLU network (135) can be written explicitly as

$$f_{\boldsymbol{W}_1, \boldsymbol{W}_2}(\boldsymbol{x}) = \sum_{k \in \mathcal{K}} \alpha_k \max(0, \boldsymbol{w}_k \bar{\boldsymbol{x}}) = \sum_{k \in \mathcal{K}} z_k \max(0, \tilde{\boldsymbol{w}}_k \bar{\boldsymbol{x}}), \tag{137}$$

where $\tilde{\boldsymbol{w}}_k = |\alpha_k| \, \boldsymbol{w}_k$, and $z_k \in \{1, -1\}$.

Equation (137) implies that the decision boundaries of a 2-layer neural network are essentially K-combinations of hyperplanes. The decision regions of a 2-layer network are controlled by the directions of the normal vectors to these hyperplanes, which are determined by a length-K binary assignment $(z_1, \ldots z_K) \in \{1, -1\}^K$.

Thus, the combinatorial search space of the two-layer ReLU neural network $\mathcal{S}_{\text{ReLU}}(N, K, D)$ consists of the *Cartesian product* of K-combinations of hyperplanes and length-K binary assignments with respect to a size N dataset in general position. The size of this space is given by

$$\left|\mathcal{S}_{\text{ReLU}}(N, D)\right| = 2^K \times \left(\binom{N}{D} \right) = O\left(N^{DK}\right). \tag{138}$$

The generator for the Cartesian product K-combinations and binary assignment can be easily constructed by directly applying the Cartesian product fusion theorem described in Subsection II.2.3.4. As explained in the previous Chapter, it is necessary to solve a matrix inversion for each D-combination of data items in order to get the representation for each hyperplane, this will take $O\left(D^3 \times \binom{N}{D}\right)$ time. Therefore, the time complexity for the exact ReLU neural network algorithm will have a complexity

$$O\left(2^K \times \left(\binom{N}{D} \right) + D^3 \times \binom{N}{D}\right) = O\left(N^{DK}\right). \tag{139}$$

Combination-combination nested generator in Haskell The difficulty here is to construct an algorithm that enumerates the *K-combinations of hyperplanes*, rather than K-combinations of data points. As previously mentioned, classical approaches for enumerating K-combinations of hyperplanes include the one-by-one enumeration method proposed by Arora et al. [2016]. Through the discussion in Chapter II.3, it is established that hyperplanes in $\mathbb{R}^D$ can be represented by D-combinations of data items. Therefore, the K-combinations of hyperplanes are simply K-combinations of D-combinations of data items.

Instead of using a one-by-one enumeration approach, the `kcombs` generator, introduced in Subsection II.2.3.3 of Part II, can alternatively be used as an algorithmic foundation. A *nested combination-combination* generator is then constructed as the composition of two combination generators. In Haskell, this generator can be specified as

```
dcombsKcombs :: Int -> Int -> [Int] -> (Css, NCss)
dcombsKcombs d k = <se d.fst,snd> . <id, kcombs k.(!!d)> . kcombs d
```

where we use `<f,g>` to denote product function `pair f g` for the brevity. This generator is parameterized by two integers: the dimension of the data `d` (assuming `d >= 2`) and the number of hyperplanes (hidden neurons) `k`. The types `Css` and `NCss` represent the output types of the combinations generated by `kcombs` and the nested combinations generated by `dcombsKcombs d k`, defined in Haskell as

```
type Comb = [Int]
type Ncomb = [Comb]
```

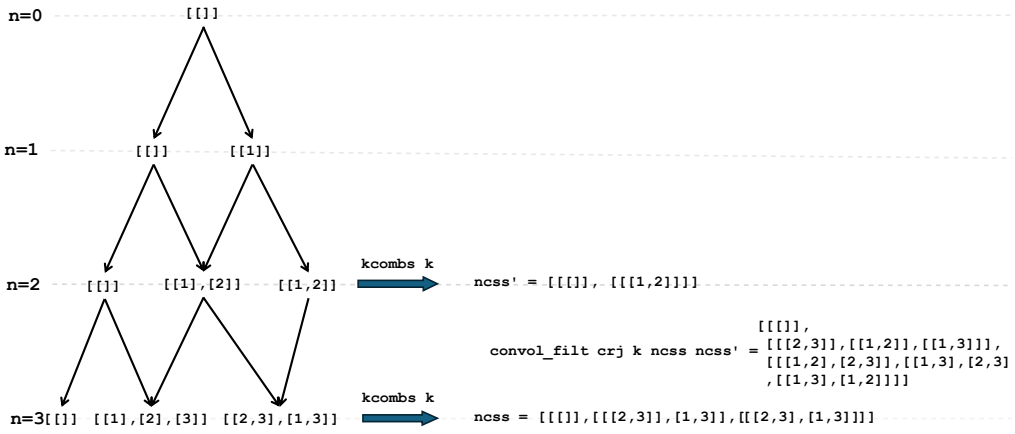


Figure 18: A combination-combination nested generator, where hyperplanes are represented as 2-combinations of data items, and the 2-layer ReLU networks with three hidden neurons are defined as 2-combinations of hyperplanes, i.e., `k=2`, `d=2`. The generation tree on the left of the figure illustrates the incremental generation process of the hyperplanes. In each recursive step, the newly generated D-combinations of data points (hyperplanes) `[[2,3],[1,3]]` are used to generate K-combinations of hyperplanes (ReLU network with K hidden neurons) `ncss` by running a `kcombs k` function, a process implicitly indicated by the large blue arrow. These newly generated K-combinations of hyperplanes `ncss` are merged with the previous K-combination of hyperplanes `ncss'` by `convol_filt crj k` function (which is defined in Subsection II.2.3.3 of Part II). This produces the *complete* list of lists at this recursive step, where each inner list contains ReLU network of a specific size.

```
type NCss = [[Ncomb]]
type Css = [[Comb]]
```

The nested combination generator first generate all combinations with respect to a given input list by using `kcombs d`, and then `<id,kcombs k.(!!d)>` returns a pair that the first element is the output of `kcombs d` and the second element are `k`-combinations of `d`-combinations (hyperplanes).

The `se d` function (short for "set empty") is defined as follows

```
se :: Int -> [[a]] -> [[a]]
se _ [] = []
se 0 (_:xs) = [] : xs
se d (x:xs) = x : se (d-1) xs
```

which set the `d`th element of a list to empty, once these `d`-combinations are used to generate new `k`-combinations, they immediately become redundant, so they are eliminated to save memory.

We can simplify the definition of `dcombsKcombs d k` by following equational reasoning

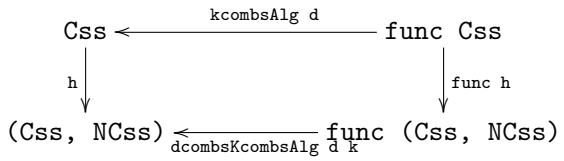
```
dcombsKcombs d k
  definition of dcombsKcombs d k
<se d.fst,snd> . <id, kcombs k.(!!d)> . kcombs d
≡ product fusion law <h,k>.m=<f,g>
<se d.fst,snd> . <kcombs d, kcombs k.(!!d).kcombs d>
≡ product fusion law <h,k>.m=<f,g>
<se d, kcombs k.(!!d)>. kcombs d
```

Clearly, `dcombsKcombs d k` is not efficient and it requires storing all possible hyperplanes before enumerating the K-combinations of hyperplanes. This is a non-trivial task, as the number of hyperplanes in D-dimensional space is $O\left(N^D\right)$, which is extremely large. Storing all these hyperplanes in advance is both memory-intensive and inefficient, making such methods impractical for real-world applications. Instead, the aim is to *fuse* the second combination generator within the first combination generator, then the nested combination generator can be expressed as a single catamorphism. This approach allows *incremental generation* of K-combinations of hyperplanes, eliminating the need to store all hyperplanes in advance.

According to the *catamorphism fusion law* (42), we need to develop an algebra `dcombsKcombsAlg`, that satisfies the following fusion condition

$$\texttt{h . kcombsAlg d = dcombsKcombsAlg d k. func h}, \tag{140}$$

where `h = <se d, kcombs k.(!!d)>`. In other words, the following diagram commutes:



Consider the case where `func` is the join-list algebra. We show that that the `Join` pattern of `dcombsKcombsAlg d k` can be defined as

```
<se d.kcsA d.func fst, kcsA k.(uncurry Join).<kcs k.!!d.kcsA d.func fst,kcsA k.func snd>>,
```
(141)

where `kcsA` and `kcs` are short for `kcombsAlg` and `kcombs`.

In order to prove the fusion condition, it requires an expansion of the above commutative diagram. To maintain the readability, we put the proof in Appendix B.2. The `Nil` case and `Single` case are easy to prove according to the definition, as we assume `d>=2`, thus there is no way to constructed any nested combinations for these two cases.

Informally, `dcombsKcombsAlg d k` first receives `func (Css,NCss)` returned by `func h` as input, and `se d.kcsA d.func fst` update the combinations (first elements in the tuple) and set the `d`-combination as empty. At the same time, then function `<kcs k.!!d.kcsA d.func fst,kcsA k.func snd>::func (Css,NCss)->(NCss,NCss)` update the combinations and nested combinations in the tuple, respectively, and the newly generated `d`-combinations are then used to generate new nested combinations. After that, the two nested combinations in the tuple are combined by using `kcsA k.(uncurry Join)::(NCss,NCss)->NCss`.

In Haskell, we can define `dcombsKcombsAlg d k` as

```
dcombsKcombsAlg :: Int->Int-> ListFj Int (Css, NCss)-> (Css, NCss)
dcombsKcombsAlg d k Nil = ([[[]]], [[[]]])
dcombsKcombsAlg d k (Single a) = ([[[]],[[a]]],[[[]]])
dcombsKcombsAlg d k (Join (css1, ncss1) (css2, ncss2)) = (se d css, ncss)
 where
  css = cvcrj d css1 css2
  ncss
   | null (css!!d) = [[[]]]
   | otherwise = cvcrj k (cvcrj k ncss1 ncss2) (kcombs k (css!!d))
```

where `cvcrj` is short for the join pattern of `kcombsAlg`.

Running `(snd (cata (dcombsKcombsAlg 2 2) [1,2,3]))!!d` produces the output `[[[1,3],[1,2]],[[2,3],[1,2]],[[2,3],[1,3]]]`, which consists of all possible 2-combinations of hyperplanes in $\mathbb{R}^2$ with respect to the input sequence `[1,2,3]`.

Intuitively, the algorithm is exhaustive because when two partial configurations are merged, `(css1 , ncss1)` and `(css2, ncss2)`, it is first needed to merge the *nested* combinations stored in the second element of the tuple, which represent K-combinations of D-combinations, using `cvcrj k ncss1 ncss2`. Next, the combinations stored in the first element of the tuple are merged by `css = cvcrj d css1 css2`. Since `css` generates a new list of D-combinations, a new list of K-combinations is created using `kcombs k (css!!d)`, which is merged with `cvcrj k ncss1 ncss2`. The *incremental* generation process for this combination-combination nested generator is illustrated in Fig. 18.

III.2.3.2 Linear programming-based method

Combinatorial complexity Section III.1.3 discussed the fact that hyperplanes can be represented by their prediction labels (assignments). Similarly, K hyperplanes can be represented by K assignments $\boldsymbol{y}_k$, $k \in \mathcal{K}$. Equation (135) implies that a data item $\boldsymbol{x}$ is predicted to the negative class by $f_{\boldsymbol{W}_1,\boldsymbol{W}_2}(\boldsymbol{x})$ if and only it lies in the negative sides of all hyperplanes H_k, $\forall k \in \mathcal{K}$, because $f_{\boldsymbol{W}_1,\boldsymbol{W}_2}(\boldsymbol{x})$ will return positive as long as there exists a k such that $\tilde{\boldsymbol{w}}_k \bar{\boldsymbol{x}} > 0$.

Therefore, the prediction labels of the 2-layer neural network $\boldsymbol{y}_{\text{ReLU}}$ consists of the union of positive prediction labels for each hyperplane H_k, and the remaining data item, which lies in the negative side with respect to all K hyperplanes will be assigned to the negative class. In other words, denoting $\boldsymbol{y}^+$ and $\boldsymbol{y}^-$ as the positive and negative predictions of $\boldsymbol{y}$ respectively, then

$$\begin{aligned} \boldsymbol{y}^+_{\text{ReLU}} &= \bigcup_{k \in \mathcal{K}} \boldsymbol{y}^+_k \\ \boldsymbol{y}^-_{\text{ReLU}} &= \mathcal{D} \backslash \boldsymbol{y}^+_{\text{ReLU}}, \end{aligned} \tag{142}$$

where $\backslash$ is the set difference and $\bigcup_{k \in \mathcal{K}} \boldsymbol{y}^+_k$ denotes the union of $\boldsymbol{y}^+_k$, $k \in \mathcal{K}$. For instance, if $\boldsymbol{y}_1 = (1, 1, -1, -1)$ and $\boldsymbol{y}_2 = (-1, 1, 1, -1)$, then $\boldsymbol{y}^+_1 = \{1, 2\}$ and $\boldsymbol{y}^+_2 = \{2, 3\}$, thus $\boldsymbol{y}^+_1 \cup \boldsymbol{y}^+_2 = \{1, 2, 3\}$.

By characterizing hyperplanes in terms of prediction labels, it is not necessary to compute the Cartesian product of combinations of hyperplanes and length K binary assignments $(z_1, \ldots z_K)$. because the prediction labels for both positive and negative orientation of hyperplanes are all included in the $2\sum_{d=0}^{D} \begin{pmatrix} N-1 \\ d \end{pmatrix}$ possible dichotomies, then the prediction labels for ReLU network $\boldsymbol{y}_{\text{ReLU}}$ can be calculated from (142). This operation will take $O(N \times K)$ time for each K-combination of assignments. Thus the total complexity of the LP-based method will be

$$O\left(\sum_{n=0}^{N} \left(\text{LP}(n, D) \times \text{Cover}(n-1, D)\right) + N \times K \times \begin{pmatrix} \text{Cover}(N, D+1) \\ K \end{pmatrix}\right) = O\left(N^{DK}\right). \tag{143}$$

III.2.4 Further discussion

III.2.4.1 Acceleration methods

Symmetry fusion in the hyperplane-based method In previous discussions on the hyperplane-based algorithm for solving the 0-1 loss linear classification problem, the symmetry fusion theorem 16 was established, which exploits the symmetry inherent in the 0-1 loss linear classification problems. This theorem states that the 0-1 loss for the negative orientation of a hyperplane can be computed from the 0-1 loss of the same hyperplane in the positive orientation. This result can indeed be extended to the ReLU network problem.

For a two-layer neural network, the data points can be classified into three categories based on their relationship to the K hyperplanes defined by the K hidden neurons:

1. Data points that lie in the region where all K hyperplanes are on the positive side.
2. Data points that lie in the region where all K hyperplanes are on the negative side.
3. Data points that lie in the region where some hyperplanes are on the positive side and others are on the negative side.

On reversing the orientation of all K hyperplanes, the classification for data points that fall into the class of the first two cases will be reversed, because the prediction labels of these data be reversed if the orientation for all hyperplanes is reversed (because of the symmetry). However, the classification of data points in the third category will remain unchanged. According to (142), the prediction labels of the 2-layer neural network, $\boldsymbol{y}_{\text{ReLU}}$, consist of the union of positive prediction labels for each hyperplane H_k. Therefore, a data point that lies in the positive region of any hyperplane H_k will be classified as positive. Consequently, for data points in the third category, there will always be some hyperplanes that classify these points as negative. On reversing the orientation of all hyperplanes, these data points will still be classified as positive.

Therefore, it is only necessary to consider the Cartesian product of K-combinations of hyperplanes with $2^K/2 = 2^{K-1}$ possible binary assignments. This approach effectively reduces the algorithm's running time by half.

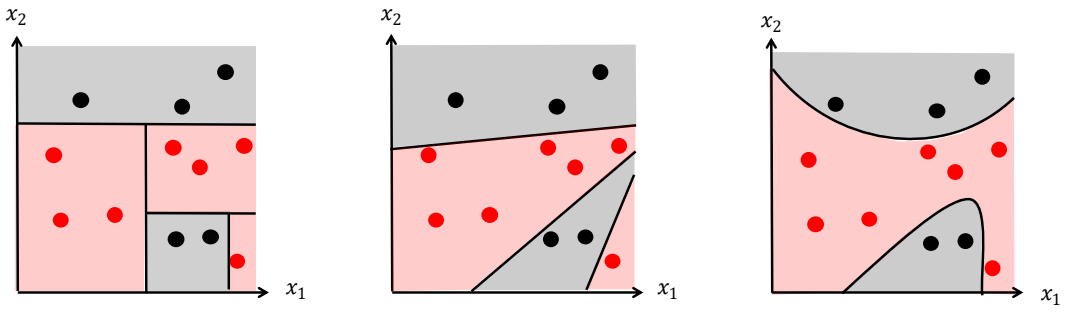


Figure 19: An axis-parallel decision tree model (left), a hyperplane (oblique) decision tree model (middle), and a hypersurface (defined by degree-2 polynomials) decision tree model (right). As the complexity of the splitting functions increase, the tree's complexity decreases (involving fewer splitting nodes), while achieving higher accuracy.

III.3 Decision tree problems

A decision tree is a supervised machine learning model to which makes predictions using a tree-based subdivision of the feature space. Imagine a flowchart or a series of "yes" or "no" questions that guide towards a final decision. Geometrically, each question or condition at a node splits the feature data into two groups based on a feature's value, these splits are parallel to the axis of the feature space. For instance, at each node, the tree asks a question about a single feature: "Is feature x_d greater than some value v?" This question divides the feature space into two regions, $x_d \leq v$ and $x_d > v$, through hyperplanes parallel to the axis, $x_d = 0$. Due to the unparalleled simplicity and interpretability of the decision tree model, algorithms that can learn an accurate decision tree model—for instance, classification and regression trees (CART) [Breiman et al., 1984], C4.5 [Quinlan, 2014] and random forests [Breiman, 2001a]—have achieved significant success across various fields. Breiman [2001b] aptly noted, "On interpretability, trees rate an A+."

This Chapter shows that the **axis-parallel decision tree problem** (DTree)**, hyperplane decision tree problem** (HDTree) and **polynomial hypersurface decision tree problem** (PHSDTree**)** can be solved exactly by using the same algorithmic processes developed in this chapter.

III.3.1 Related studies

The classical approximate/heuristic algorithms for creating decision trees, such as CART and C4.5, usually use a top-down, greedy approach. Therefore, these approximate algorithms cannot guarantee solutions that are optimal.

To improve the accuracy of the decision tree model, there are two common approaches. One is to find the globally optimal axis-parallel decision tree for the data. Alternatively, instead of constructing models that create axis-parallel hyperplanes, more complex splitting rules can be applied. For example, a generalization of the classical decision tree problem is the *hyperplane* (or *oblique*) *decision tree*, which uses hyperplane decision boundaries to potentially simplify boundary structures. The axis-parallel tree model

is very restrictive in many situations. Indeed, it is easy to show that axis-parallel methods will have to approximate the correct model with a "staircase-like" structure. By contrast, the tree generated using hyperplane splits is often smaller and more accurate than the axis-parallel tree. It should be intuitively clear that when the underlying decision region is defined by a *polygonal space partition*, it is preferable to use hyperplane decision trees for classification. By contrast, the axis-parallel tree model can only produce *hyper-rectangular* decision regions.

To illustrate, Fig. 19, depicts three different decision tree models—the axis-parallel decision tree, the hyperplane decision tree, and the hypersurface decision tree (defined by a degree-two polynomial)—to classify the same dataset. As the complexity of the splitting rule increases, the resulting decision tree becomes simpler and more accurate.

However, both generalizations of the classical axis-parallel tree (the hyperplane and hypersurface tree) are very difficult to solve. It is well-known that the problem of finding the smallest axis-parallel decision tree is NP-hard [Laurent and Rivest, 1976]. Similarly, for the hyperplane decision tree problem, even the top-down, inductive greedy optimization approach—similar to the CART algorithm—is NP-hard to solve exactly. This result comes from the fact that the 0-1 loss linear classification problem is considered as NP-hard. As Murthy et al. [1994] explained,"But when the tree uses oblique splits, it is not clear, even for a fixed number of attributes, how to generate an optimal (e.g., smallest) decision tree in polynomial time."Bertsimas and Dunn [2019]

Due to the formidable combinatorial complexity of the tasks, studies on decision tree problems focus on either designing good approximate algorithms [Wickramarachchi et al., 2016, Barros et al., 2011, Cai et al., 2020] or developing exact algorithms that produce trees with specific structures, such as the complete binary tree of a given depth, known as BinOCT Verwer and Zhang [2019]. In particular, a substantial number of studies have focused on optimal decision tree algorithms for datasets with binary features [Lin et al., 2020, Hu et al., 2020, Aglin et al., 2021, 2020, Nijssen and Fromont, 2010, Zhang et al., 2023]. This problem has a combinatorial complexity that is independent of the input data size, as the number of splitting rules depends solely on the number of possible features, thus polynomially solvable. On the other hand, exact algorithms addressing the axis-parallel decision tree problem in full generality primarily use mixed-integer programming (MIP) solvers [Aghaei et al., 2019, Mazumder et al., 2022, Aghaei et al., 2021, Bertsimas and Dunn, 2017, Günlük et al., 2021]. Bertsimas and Dunn [2017] employed MIP solvers for the hyperplane decision tree problem. However, it is well-known that MIP solvers have exponential (or worse) complexity in the worst case. Bertsimas and Dunn [2019] reported that their algorithm, based on MIP solvers, quickly becomes intractable when increasing the number of branch nodes. Additionally, Hu et al. [2019] observed that one solution presented in the figures of Bertsimas and Dunn [2017]'s is sub-optimal. However, since Bertsimas and Dunn [2017] did not release their code publicly, it remains unclear why their algorithm returns a sub-optimal solution. This suggests that Bertsimas and Dunn [2017]'s algorithm may not be exact, and our algorithm may be the only one capable of solving the hyperplane decision tree problem exactly.

III.3.2 Novel axioms for decision trees

As the name suggests, a decision tree is a tree-based model. In a *directed* graph, if there is a directed edge from one node to another, the node at the destination of the edge is called the *child* node of the source

node, while the source node is referred to as the *parent* node. If there is a directed *path* connecting two nodes, the source node is called the ancestor node.

A tree can be viewed as a special case of a directed graph where each node has multiple child nodes but only *one* parent node. The topmost node is referred to as the *root*, and a tree contains no *cycles*—defined as a path where the source and destination nodes coincide. The nodes farthest from the root are called *leaf nodes*, while all other nodes are referred to as *branch* or *internal* nodes. A sequence of nodes along the edges from the root to the leaf of a tree is called *path*.

To the best of our knowledge, the concept of a decision tree has not been defined rigorously and varies significantly across different fields. Even within the same field, such as machine learning, various definitions of decision trees have been proposed [Lin et al., 2020, Bennett, 1992, Bennett and Blue, 1996, Hu et al., 2020, Narodytska et al., 2018, Aglin et al., 2021, 2020, Avellaneda, 2020, Verwer and Zhang, 2019, Jia et al., 2019, Zhang et al., 2023, Mazumder et al., 2022]. These algorithms are all named "optimal decision tree algorithms", suggesting they aim to solve the decision tree problem, but their definitions differ to some extent.

The various definitions of decision trees typically share several common features:

- Each branch node of a decision tree contains only *one splitting rule*, which divides the ambient space[23] into two *disjoint* and *continuous* subspaces.
- Each leaf specifies a region defined by the path from the root to the leaf.
- A new splitting rule can only be generated from the subspace defined by its ancestor rules.

We will now formalize these concepts with more rigorous definitions.

Decision tree, complete graph and ancestry relation matrix

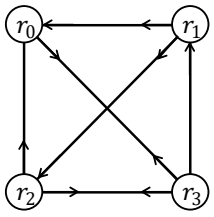
An important concept that we introduce is *rule feasibility*: this recognizes the fact that the decision tree model not only divides the ambient space into two regions but also constrains the space in which new splitting rules apply. If a new splitting rule applies only within the subspace defined by its ancestor, we refer to this rule as a feasible rule.

Feasibility essentially establishes an *ancestry relation* between splitting rules, which must satisfy *transitivity*. Specifically, if a splitting rule applies to the subspaces defined by its ancestor, it must also apply to the subspace defined by its ancestor's ancestor rules. We can formalize this ancestry relation with the following definition.

Definition 35. *Ancestry relations*. Given a list of K rules $rs = [r_1, r_2, \dots r_K]$, the ancestry relation between two hyperplanes is denoted by two arrows $\searrow$ and $\swarrow$. Define $r_i \swarrow r_j$ if r_j is in the left branch of r_i, and $r_i \searrow r_j$ if r_j is in the right branch of r_i, and $r_i \,(\swarrow \vee \searrow)\, r_j$ if r_j is in the left *or* right branch of r_i.

The notation $\searrow$ and $\swarrow$ can be read from left to right because $h_i \swarrow h_j$ and $h_i \searrow h_j$ do not imply $h_j \searrow h_i$ and $h_j \swarrow h_i$, unless h_i and h_j are mutual ancestors of each other. In other words, $\searrow$ and $\swarrow$ are not *commutative* relations.

[23] An ambient space is the space surrounding a mathematical (geometric or topological) object along with the object itself.



$$\begin{array}{c} \\ r_0 \\ r_1 \\ r_2 \\ r_3 \end{array} \begin{bmatrix} 0 & -1 & 0 & 1 \\ 1 & 0 & 1 & -1 \\ 1 & -1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Figure 20: The *ancestry relation graph* (left) captures all ancestry relations between four splitting rules $[r_0, r_1, r_2, r_3]$. In this graph, nodes represent rules, and arrows represent ancestral relations. An incoming arrow from r_j to a node r_i indicates that r_j is the right-children of r_i. The absence of an arrow indicates no ancestral relation. An outgoing arrows from r_i to a node r_j indicates that r_j is the left-children of r_i. The ancestral relation matrix (right) $\boldsymbol{K}$, where the elements $\boldsymbol{K}_{ij} = 1$, $\boldsymbol{K}_{ij} = -1$, and $\boldsymbol{K}_{ij} = 0$ indicate that r_j lies on the positive side, negative side of r_i, or that there is no ancestry relation between them, respectively.

These ancestry relations can be characterized as *homogeneous binary relations*. Relations and graphs are closely related, and homogeneous binary relations over a set can be represented as directed graphs [Schmidt and Ströhlein, 2012]. Therefore, the ancestry relations between hyperplanes can be encoded as a *complete graph*, where the branch nodes (hyperplanes) are the nodes in the graph, and the ancestry relations $\searrow$ and $\swarrow$ are represented as incoming and outgoing arrows in the graph. Note that the *adjacent* arrows to each node correspond to the ancestry relations. We refer to it as the *ancestry relation graph*. The reason this graph is complete is that every hyperplanes is related to any other hyperplanes in some way, either through an ancestry relation or by being unrelated. The left panel in Fig. 20 illustrates the corresponding complete graph for a given set of splitting rules, defined by hyperplanes, as shown in Fig. 24.

Moreover, binary relations can also be characterized as *Boolean matrices*. However, to encode two binary relations, $\searrow$ and $\swarrow$ in one matrix, the values 1 and -1 are used to distinguish them. We define the ancestry relation matrix as follows.

Definition 36. *Ancestry relation matrix.* Given a list of K rules $rs = [r_1, r_2, \dots r_K]$, he ancestry relations between any pair of rules can be characterized as a $K \times K$ square matrix $\boldsymbol{K}$, with elements defined as follows:

- $\boldsymbol{K}_{ij} = 1$ if $r_i \swarrow r_j$ (i.e.,r_j is in the left subtree of r_i),
- $\boldsymbol{K}_{ij} = -1$ if $r_i \searrow r_j$ (i.e.,r_j is in the right subtree of r_i),
- $\boldsymbol{K}_{ij} = 0$ if $r_i \overline{(\swarrow \vee \searrow)} r_j$, where $\overline{R}$ represent the complement relation of R. According to De Morgan's law $r_i \overline{(\swarrow \vee \searrow)} r_j = \left(r_i \overline{\swarrow} r_j\right) \wedge \left(r_i \overline{\searrow} r_j\right)$. In other words, $\boldsymbol{K}_{ij} = 0$ if and only if h_j is not a branch node in both the left and right subtree of h_i and $\boldsymbol{K}_{ij} \neq 0$, $i \neq j$, if r_i is the ancestor of r_j.

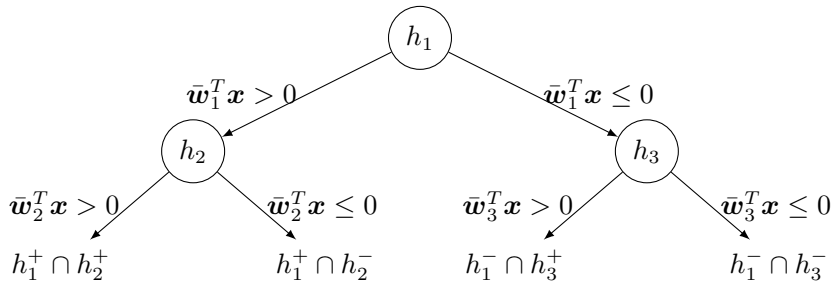
We are now ready to formalize the axioms of the decision tree.

Definition 37. *Axioms for proper decision trees.* We call a decision tree that satisfies the following axioms, a *proper* decision tree:

1. Each branch node is defined by a single splitting rule $r : \mathcal{R}$, and each splitting rule subdivides the ambient space into two disjoint and connected subspaces, r^+ and r^-,
2. Each leaf L is defined by the intersection of subspaces $\bigcap_{p \in P_L} r_p^{\pm}$ for all the splitting rules $\{r_p \mid p \in P_L\}$ in the path P_L from the root to leaf L. The connected region (subspace) defined by $\bigcap_{p \in P_L} r_p^{\pm}$ is referred to as the *decision region*,
3. The ancestry relation between any pair of splitting rules $r_i \, (\swarrow \vee \searrow) \, r_j$ is transitive; in other words, if $r_i \, (\swarrow \vee \searrow) \, r_j$ and $r_j \, (\swarrow \vee \searrow) \, r_k$ then $r_i \, (\swarrow \vee \searrow) \, r_k$,
4. For any pair of splitting rules r_i and r_j, only one of the following three cases is true: $r_i \swarrow r_j$, $r_i \searrow r_j$, and $r_i \overline{(\swarrow \vee \searrow)} r_j$; additionally, $r_i \overline{(\swarrow \vee \searrow)} r_i$ is always true. In other words, $\boldsymbol{K}_{ij} \in \{1, 0, -1\}$, and $\boldsymbol{K}_{ii} = 0$, $\forall i, j \in \mathcal{K}$.

Although the ancestry relation satisfies transitivity, it is *not* a *preorder*, as it fails to satisfy the reflexive property due to Axiom 4—no rule can be the ancestor of itself in a decision tree.

For instance, consider a tree with decision hyperplanes h_1, h_2, and h_3 defined by normal vector $\boldsymbol{w}_1$, $\boldsymbol{w}_2$, and $\boldsymbol{w}_3$, respectively. A tree of depth two can be illustrated as follows,



where $h_i^+ = \left\{\boldsymbol{x} : \boldsymbol{x} \in \mathcal{N}, \bar{\boldsymbol{w}}_i^T \boldsymbol{x} > 0\right\}$ and $h_i^- = \left\{\boldsymbol{x} : \boldsymbol{x} \in \mathcal{N}, \bar{\boldsymbol{w}}_i^T \boldsymbol{x} \leq 0\right\}$ denote the positive and negative predictions of the hyperplane with respect to the dataset $\mathcal{N}$. The following discussion omits the labels for the branch arrows $\boldsymbol{w}^T x \leq 0$ and $\boldsymbol{w}^T x > 0$, assuming by default that the left branch corresponds to "less than" and the right branch to "greater than".

These axioms encompass a wide range of decision trees, including decision tree models in machine learning, where splitting rules can be axis-parallel hyperplanes or hypersurfaces, as well as binary space partition trees [Motwani and Raghavan, 1996] and K-D trees [Bentley, 1975] in computational geometry.

In this study, we focus exclusively on the ODT problem for such proper decision trees. There is potential to extend this framework by modifying or introducing additional axioms to those given above, such as those for the axis-parallel decision tree problem *over binary feature data*. For the first problem, Axiom 4 no longer holds but we can modify it to permit $r_i \swarrow r_j$ and $r_i \searrow r_j$ to hold simultaneously for any pair of splitting rules.

III.3.2.1 When can decision trees be characterized by K-permutations?

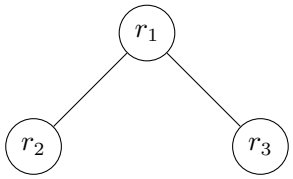


Figure 21: A decision tree with three splitting rules, corresponds to 3-permutation $[r_1, r_2, r_3]$.

The formalization of proper decision trees enables the analysis of their algorithmic and combinatorial properties. One of the most important combinatorial properties discussed in this paper is that any proper decision tree can be *uniquely* characterized as a K-permutation through a *level-order traversal* of the tree.

Tree traversal refers to the process of visiting or accessing each node of the tree *exactly once* in a specific order. Level-order traversal visits all nodes at the same level before moving on to the next level. The main idea of level-order traversal is to visit all nodes at higher levels before accessing any nodes at lower levels, thereby establishing a hierarchy of nodes between levels. For example, the level-order traversal for the binary tree in figure 21 has two possible corresponding 3-permutations, $[r_1, r_2, r_3]$ or $[r_1, r_3, r_2]$. If we fix a traversal order such that the left subtree is visited before the right subtree, only one arrangement of rules can exist. Based on the axioms of the proper decision tree, we can state the following theorem about the level-order traversal of a proper decision tree.

Theorem 18. Given a level-order traversal of a decision tree $[\ldots, r_i, \ldots, r_j, \ldots, r_k, \ldots]$, if r_j precedes r_k in the traversal, and r_i is the ancestor of r_j and r_k, then either:

1. r_j and r_k are at the same level, or
2. r_k is a descendant of r_j.

Only one of the two cases can occur. If the first case holds, they are the left or right children of another node, and their positions cannot be exchanged.

Proof. We prove this by contradiction. Assume, by contradiction, r_j is in the same level as r_k and r_k can be a descendant of r_j. Suppose we have a pair of rules r_k and r_j, where r_j precedes r_k in the level-order traversal.

Case 1: Assume r_j and r_k are at the same level, we prove that r_k cannot be the descendant of r_j. Because of Axiom 3, if r_j is in the same level of r_k, then r_j and r_k are generated from different branches of some ancestor r_i, which means that they lie in two disjoint regions defined by r_i. If r_k is the descendant of r_j then it is also a *left-descendant* of r_i due to associativity. According to Axiom 4, either r_j is a left child of r_i or right children of r_i it can not be both. This leads to a contradiction, as it would imply r_k belongs to both disjoint subregions defined by r_i.

Case 2: Assume r_k is a descendant of r_j, we prove that r_j and r_k cannot be at the same level. By the transitivity of the ancestry relation, both r_k and r_j are descendants of the parent node (*immediate ancestor*) of r_j, which we call r_i. Since, r_k cannot be both the right- and left-child of r_i at the same time, as it must either be the left-child or right-child according to Axiom 4. So r_k and r_j can not be in the same level if r_k is a descendant of r_j.

Thus, if r_j precedes r_k in the level-order traversal, this either places them at the same level *or* establishes an ancestor-descendant relationship between them, but not both. □

An immediate consequence of the above theorem is that any K-permutation of rules corresponds to the level-order traversal of *at most one* proper decision tree. The ordering between any two adjacent rules corresponds to only one structure: either they are on the same level, or one is the ancestor of the other. For instance, in figure 21, if r_2 and r_3 are in the same level and r_2 is the left-child of r_1, then r_3 cannot be the child of r_2, because it cannot be the left-child of r_1. Hence, only a proper decision tree corresponds to the permutation $[r_1, r_2, r_3]$. Therefore, once a proper decision tree is given, we can obtain its K-permutation representation easily by using a level-order traversal.

Moreover, the one-to-one correspondence between valid K-permutations and proper decision trees implies that the number of K-permutations is strictly greater than the number of possible proper decision trees. For instance, if there is only one proper decision tree which corresponds to K-permutation $[r_1, r_3, r_2]$, then all other K-permutations of the set $\{r_1, r_2, r_3\}$, are invalid.

Corollary 5. A decision tree consisting of K splitting rules corresponds to a unique K-permutation permutation if and only if it is proper. In other words, there exist an injective mapping from proper decision trees and K-permutations.

Proof. **Sufficiency**: If a decision tree is proper, its level-order traversal yields a unique valid K-permutation by level-order traversal, then it implies it is a proper decision tree。

Part 1: existence of mapping.

Because the level-order traversal algorithm is deterministic and the tree's structure is fixed (each branch node has a fixed position, left or right child of its parent), thus any binary tree can be transformed into a K-permutations, by a level-order traversal.

Part 2: the mapping is injective.

To prove injectivity, we show that distinct proper decision trees $T_1 \neq T_2$ produce distinct permutations $\sigma_1 \neq \sigma_2$. Given two different proper decision trees T_1 and T_2, constructed by using rules $rs_K = \{r_1, r_2, \ldots, r_K\}$. Let σ_1 and σ_2 be their level-order traversals. Assume, for contradiction, that $\sigma_1 = \sigma_2 = \sigma = [r_{j_1}, \ldots, r_{j_K}]$. Since $T_1 \neq T_2$, they differ in some structural property (e.g., r_i's parent, sibling order, or level, for any $r_i \in rs_K$). We examine three cases, showing each alters σ :

1. Different sibling order: assume in T_1, $r_2 \swarrow r_3$ (left child), but in T_2, $r_3 \swarrow r_2$. In T_1, σ might be $[r_1, \ldots, r_2, r_3 \ldots]$, and in T_1, σ might be $[r_1, \ldots, r_2, r_3 \ldots]$, this contradicts $\sigma_1 = \sigma_2$ as sibling order is fixed by Theorem 18.

2. Different levels: Suppose in T_1, r_2 is a child of r_1, but in T_2, r_2 is a grandchild. Assume r_3 is the child of r_1 and father the of r_2 in T_2, then we have $\sigma_1 = [r_1, \ldots, r_2 \ldots]$, $\sigma_1 = [r_1, \ldots r_3 \ldots, r_2 \ldots]$. we show that r_3 can not precedes r_2 in σ_1. If r_3 precedes r_2 in σ_1, then r_3 is in the same level of r_1, so r_3 and r_1 they are in different branches of another nodes. So r_3 cannot be the children of r_1 because of the Theorem 18.

3. Different parents: Suppose r_1's parent is r_2 in T_1 but r_3 in T_2. Assume $\sigma_1 = \sigma_2 = \sigma = [\ldots r_2, \ldots r_3 \ldots, r_1 \ldots]$, because r_2 precedes r_3 in σ, so in T_1, r_3 must be a node in the same level with r_2. In T_2, it can have two possible situations: (1) r_2 be the ancestor of r_3; (2) r_2 is in the same level of r_3's ancestor. However, as Theorem 18 explained T_2 cannot coexist with T_1, because r_2 and r_3 (and r_2 with

r_3's ancestor in the second case) must be the descendants of another node, if this node changes in T_1 and T_2, we resulting a different permutations anyway.

Thus $\sigma_1 \neq \sigma_2$, we have a contradiction, and the mapping is injective.

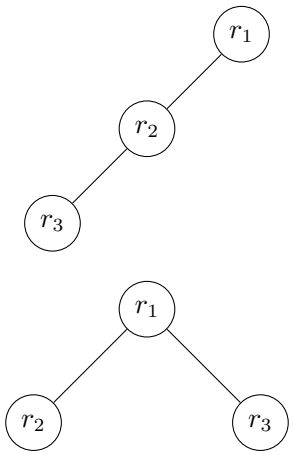
Necessity: The mapping from decision trees to K-permutations is not injective if the trees are non-proper. Specifically, distinct non-proper decision trees can correspond to identical permutations, rendering the mapping non-unique and thus non-injective.

Consider a permutation $[\ldots, r_i, \ldots, r_j, \ldots, r_k, \ldots]$. In the absence of Axioms 3 and 4, the structure of the tree is not sufficiently constrained: r_j and r_k may reside at the same level, or r_j is the ancestor of r_k, within the same permutation. Consequently, this permutation can be realized by at least two distinct non-proper trees T_1 and T_2. Since $T_1 \neq T_2$, this gives us an non-injective map. □

III.3.2.2 Incorrect claims in the literature

The study of exact algorithms for ODT problems in machine learning is dominated by the use of ad-hoc *branch-and-bound* (BnB) methods. Researchers often design algorithms or propose speed-up techniques based on intuition rather than rigorous proof, leading to logical or implementation errors. For instance, in the context of ODT algorithms, the decision tree problem over binary feature data has received the most attention—primarily because its combinatorial complexity is independent of input data size. Several fundamental errors have frequently appeared in the extensive literature on this topic.

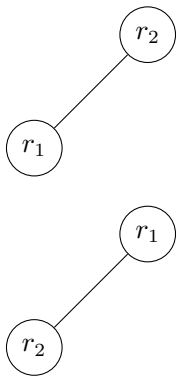
The first error, as we note in the introduction, is that the ODT problem over binary feature data does not satisfy the Axiom 4 characterizing proper decision trees. For the decision tree problem over binary feature data, each splitting rule is defined as *selecting a feature or not*, thus any splitting rule can be both the left-child or right-child of another. So previous research, such as Hu et al. [2019], has wrongly characterized their tree as K-permutations, because the decision tree over binary feature data does not satisfy the proper decision tree axioms and hence cannot be characterized by K-permutations. For example, in the decision tree over binary feature data, considering following two trees



Clearly, these two trees are distinct, yet both yield the same permutation $[r_1, r_2, r_3]$ in a level-order traversal. under level-order traversal. These two trees cannot exist at the same time if they are proper, but they can exist in the decision tree over binary feature data.

Since the implementation by Hu et al. [2019] differs from their pseudo-code, it remains unclear how their algorithm was actually implemented. Without the required rigorous proof, their algorithm is likely incorrect.

The second error is fundamental: a direct consequence of proper decision tree Axiom 1, is that trees defined by the same set of rules and having the same shape, but organized with different labels, will result in distinct trees. Otherwise, they would be equivalent to *unlabeled* trees, which correspond to the combinatorial objects counted by Catalan numbers. For instance, consider two trees that share the same topological shape and the same set of splitting rules



These two trees are distinct according to Axiom 1: in the first tree, the decision region defined by the right subtree of r_2 remains intact after introducing r_1. Conversely, in the second tree, the decision region defined by the right subtree of r_1 remains intact after introducing r_2. Consequently, the first tree creates three decision regions: $r_2^+ \cap r_1^+$, $r_2^+ \cap r_1^-$, and r_2^-. The second tree, on the other hand, generates a different set of regions: $r_2^+ \cap r_1^+$, $r_2^+ \cap r_1^-$, and r_1^-, which is fundamentally different.

This oversight is a common mistake in the literature studying the ODT problem over binary feature data [Lin et al., 2020, Hu et al., 2020, Aglin et al., 2021, 2020, Nijssen and Fromont, 2010, Zhang et al., 2023]. Many of these reports fail to distinguish trees with the same shape but different labels. For instance, some explicitly count the possible trees using some explicitly count the possible trees using Catalan numbers [Hu et al., 2019], while others employ Catalan number-style recursion [Demirović et al., 2022, Hu et al., 2019, Lin et al., 2020, Zhang et al., 2023, Aglin et al., 2021, 2020, Nijssen and Fromont, 2010].

Finally, another fundamental error found in the literature is the improper application of memoization techniques for decision tree problems, a problem which will be explained in detail in section III.3.9.1.

III.3.3 The combinatorial complexity of various optimal decision tree problems

Axis-parallel decision tree problem The complexity of the axis-parallel decision tree algorithms is well-known in the literature [Murthy et al., 1994, Mazumder et al., 2022]. For decision tree models with axis-parallel splits, there are only $N \times D$ distinct possibilities for each split. Due to the relatively low complexity of each split, heuristic methods such as C4.5 and CART can be trained greedily by selecting the best split. Specifically, C4.5 maximizes *information gain*, while CART optimize *Gini impurity* (also known as the *Gini index*) to determine the best split at each step. This selection process involves an

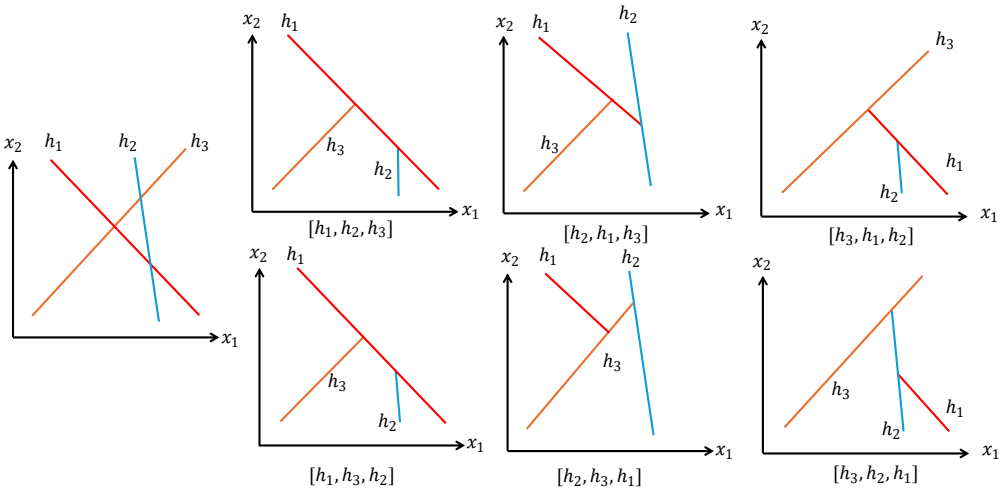


Figure 22: This example illustrates how the ordering (permutation) of hyperplanes affects the partitioning in a decision tree model. The six possible permutations of three hyperplanes h_1, h_2, h_3 result in different partitions of the space. Once a hyperplane is placed in the first position, the subsequent hyperplanes can only occupy the sub-regions defined by the first hyperplane. This pattern holds true for the other hyperplanes as well.

exhaustive search across all $N \times D$ splits. However, the exhaustive search approach for solving the axis-parallel decision tree problem is generally considered intractable. This is because the number of possible axis-parallel trees in $\mathbb{R}^D$ with K branch nodes is $\begin{pmatrix} ND \\ K \end{pmatrix}$, which corresponds to a complexity of $\mathrm{O}\left(N^K\right)$. Mazumder et al. [2022] reported that exhaustive search algorithms for solving the decision tree problem become intractable when $N = 10^4 \sim 10^5$, and $K = 2, 3$.

There follows an analysis of the combinatorial complexity of decision tree problems involving hyperplane or polynomial hypersurface splits.

Hyperplane and hypersurface decision tree problems Similar to the deep ReLU network model, the decision tree model also consists of piecewise linear (PWL) functions. For a tree with K leaves, exactly $K-1$ hyperplanes (splitting rules) are required to partition the feature space.

Following from the 0-1 loss linear classification theorem 15, since decision tree models are essentially PWL functions, Thm. 15 should immediately generalize to the hyperplane decision tree problem. Each split at a branch node partitions $\mathbb{R}^D$ into smaller connected regions, and the point-line duality remains valid in these smaller components. Therefore, it is safe to consider that the hyperplanes used to generate decision tree models consist of all hyperplanes passing through D out of N points, and there exists $\begin{pmatrix} N \\ D \end{pmatrix}$ of them.

Define $\mathcal{S}_{\mathrm{KPH}}\left(N, K, D\right)$ as the combinatorial search space of all possible K-permutations of all possible

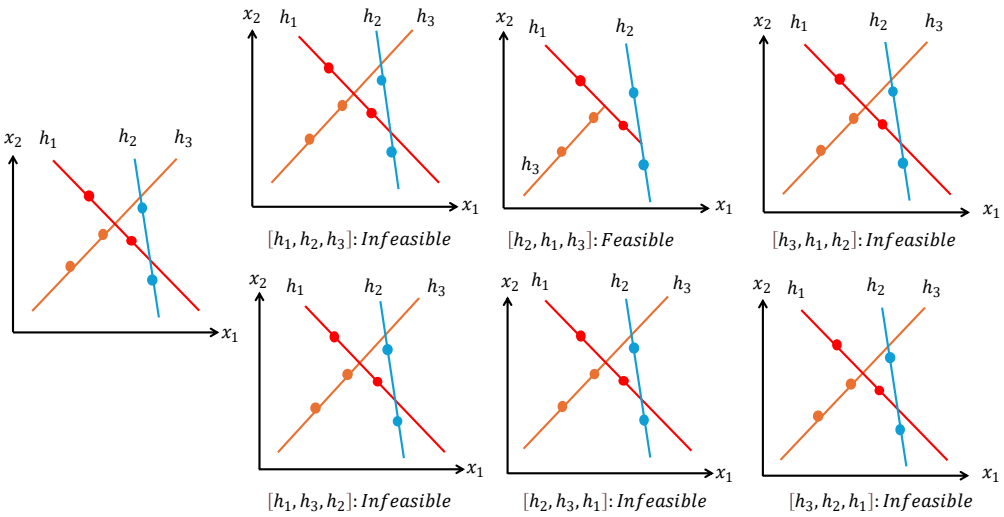


Figure 23: When representing hyperplanes as combinations of data items, the data points used to construct subsequent hyperplanes can only be selected from the sub-regions defined by the previous hyperplane. For instance, if the data points used to construct the second hyperplane h_2 come from two different regions divided by the initial hyperplane h_1, then any decision tree that uses h_1 followed by h_2 is infeasible. Among the six permutations of the three hyperplanes, only one feasible permutation exists $[h_2, h_1, h_3]$.

hyperplanes in $\mathbb{R}^D$, with respect to a dataset of size N in general position. The size of $\mathcal{S}_{\text{KPH}}(K, D)$ is

$$|\mathcal{S}_{\text{KPH}}(N, D)| = K! \times \left(\begin{pmatrix} N \\ D \end{pmatrix} \right). \tag{144}$$

Denote by $\mathcal{S}_{\text{HDtree}}(K, D)$ the combinatorial search space of all possible hyperplane decision trees with K branch nodes in $\mathbb{R}^D$. Given that there are 2^{K-1} possibilities for each K-permutation of hyperplanes, the combinatorial complexity of a hyperplane decision tree problem with K split ($K+1$ leaves) in D-dimensional space is

$$|\mathcal{S}_{\text{HDtree}}(D, K)| = 2^{K-1} \times |\mathcal{S}_{\text{KPH}}(N, D)| = O\left(N^{DK}\right), \tag{145}$$

In Subsection II.3.2.3, we demonstrate that hypersurface decision boundaries are isomorphic to hyperplanes in a higher-dimensional embedding space. To calculate the combinatorial complexity of a hypersurface decision tree, all that is required is to count the number of possible hyperplanes in the embedding space $G = \begin{pmatrix} N+W \\ D \end{pmatrix}$, where W is the degree of the polynomial for defining hypersurface G. Hence the combinatorial complexity of the hypersurface decision tree problem is $O\left(N^{GK}\right)$.

The feasibility of K-permutation of hyperplanes Equation (145) shows that the hyperplane decision tree exhibits the same asymptotic complexity as the 2-layer neural network problem. This is not surprising, as both models consist of K splitting hyperplanes.

At first glance, the big O notation in (145) may seem to obscure a significantly larger constant compared to the combinatorics of the 2-layer ReLU neural network problem. However, the true combinatorial

complexity of this problem is much smaller than given in (145). This is because each hyperplane splits the space into two disjoint regions, and subsequent hyperplanes need only be constructed from data points within these smaller regions.

Consequently, most of the configurations in $\mathcal{S}_{\text{HDtree}}(N, D, K)$ are infeasible and can be discarded. For instance, if the data points used to construct the second hyperplane h_2 come from two different regions divided by the initial hyperplane h_1, then any decision tree that uses h_1 followed by h_2 is infeasible. As illustrated in Fig. 23, although six 3-permutations of hyperplanes are shown, the only feasible configuration is $[h_2, h_1, h_3]$.

The feasibility test is a non-trivial operation. For each hyperplane, it must be verified whether the data points used to define it lie within the same region as determined by its ancestor hyperplanes. Testing whether D data points are on the same side of a hyperplane requires $O(D^2)$ computations. Given that a decision tree is defined by K hyperplanes, and in the worst case, a hyperplane may have K ancestors, the feasibility test incurs a worst-case time complexity of $O(K^2D^2)$.

III.3.4 Specifying the optimal proper decision tree problem through K-permutations

The secret to solving the ODT problem lies in the fact that the original ODT problem—find the optimal decision tree with K splitting rules out of a list of input rules $rs = [r_1, r_2, \ldots, r_M](M \geq K)$ that minimize the number of misclassification—is difficult to solve directly. However, after applying a sequence of equational reasoning steps, we are able to transform this difficult problem into a *simplified* ODT problem —the ODT problem with respect K *fixed splitting rules*, we show that this simplified problem exists a *dynamic programming recursion*, and then the algorithm for solving the simplified problem can be used to solve the original problem exactly.

The first step toward an alternative definition for the decision tree generator is to define the datatype for the decision tree problem. As discussed at the beginning of the chapter, a decision tree can be viewed as a binary tree where the branch nodes are defined by decision boundaries, and the leaf nodes are defined by the resulting prediction labels. Therefore, the decision tree can be defined as a binary tree in which the branch nodes and leave nodes have different types. In Haskell, we can define it as follows

```
data Dtree a b = DL b | DN (Dtree a b) a (Dtree a b)
```

This definition differs from the binary tree given in Section II.2.2.5, as it distinguishes between two different types for branch nodes and leaf nodes.

The map function over decision tree datatype is defined as

```
mapD :: (b -> c) -> Dtree a b -> Dtree a c
mapD f (DL a) = DL (f a)
mapD f (DN u a v) = DN (mapD f u) a (mapD f v)
```

which applies the function `f : b -> c` to every *leaf* of the tree.

By defining rules and input data using their indexes, the types for the rule and input data lists are defined in Haskell as follows

```
type D = [Int]
```

```
type R = Int
```

Similarly, we can define the type of the hyperplane splitting rule as follows

```
type H = Int
```

The goal of the ODT problem is to construct a function `odt :: Int -> [R] -> Dtree R D` that outputs the optimal decision tree with K splitting rules. This can be specified as

```
odt :: Int -> [R] -> Dtree R D
odt k = min r . genDTKs k
```

The function `genDTKs k`, short for "generate decision trees with K splitting rules", generates all possible decision trees of size $K \geq 1$ from a given input of splitting rules `rs` of size greater than K. Recall that the function `min r` selects *one* of the optimal decision trees from a non-empty list of candidates with respect to a relation `r`, since we need to use equational reasoning. Remember that `min r` is simply introduced to extend our powers of specification and will not appear in any final algorithm. We will still use `minlist r` in the implementation of our algorithm.

As discussed, proper decision trees can be characterized as K-permutations of rules. Thus one possible definition of the `genDTKs k` function is

```
genDTKs :: Int -> [R] -> [[R]]
genDTKs k = filter p . kperms k
```

The program begins by generating all possible K-permutations using `kperms k`, and then filters out, using `filter p`, those that cannot be used to construct proper decision trees. This two-step process ensures that only *valid permutations*—those that satisfy the predicate `p`, i.e. meet the structural and combinatorial requirements of proper decision trees—are retained in further computations. We can substitute `genDTKs k` in the definition of `odt k` , and thus we have

```
odt :: Int -> [D] -> [[R]]
odt k = min r . filter p . kperms k
```

As we have discussed, one possible definition of a K-permutation generator is defined through a K-*combination* generator—K-permutations is simply the all possible rearrangement (permutation) of each K-combinations. In other words, we can define `kperms k` as following `kperms k = concatMap perms . kcombs k`. By substituting the definition of `kperms k` into `odt k`, we obtain,

```
odt :: Int -> [D] -> [[R]]
odt k = min r . filter p . concatMap perms . kcombs k
```

Since `kcombs k` produces only $\begin{pmatrix} M \\ K \end{pmatrix} = O\left(M^K\right)$ K-combinations, and K-permutations are all possible permutations of each K-combinations, thus we have $|\mathcal{S}_{\text{kperms}}| = K! \times \begin{pmatrix} M \\ K \end{pmatrix} = O\left(M^K\right)$. This already gives a polynomial-time algorithm for solving the ODT problem, assuming that the predicate `p` has polynomial complexity and `min r` is linear in the candidate list size.

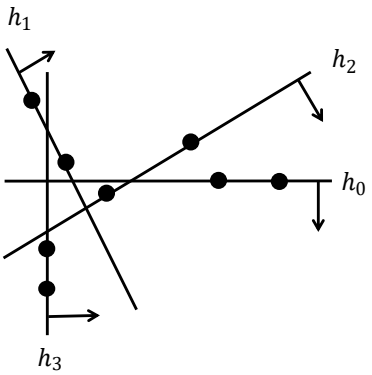


Figure 24: Four hyperplanes in $\mathbb{R}^2$. The black circles represent data points used to define these hyperplanes, and the black arrows indicate the direction of the hyperplanes.

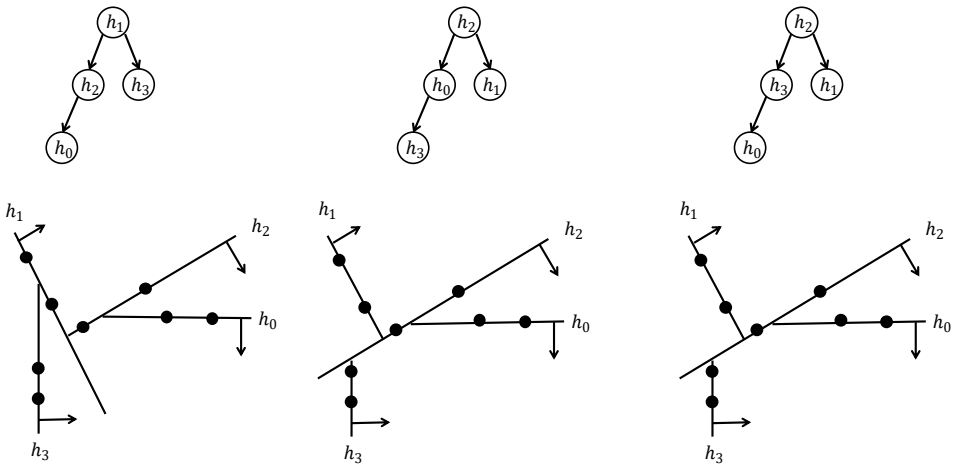


Figure 25: Three possible decision trees and their corresponding partitions of space in $\mathbb{R}^2$ for the four hyperplanes depicted in Fig. 24. The above three decision trees are the only feasible decision of a given combination of hyperplanes given in Fig , the three figures below describe the partition of space $\mathbb{R}^2$ resulting from the corresponding decision tree.

The redundancy of K-permutations

The definition above, based on K-permutations, remains unsatisfactory. The number of proper decision trees for a given set of rules is typically much smaller than the total number of permutations, and determining whether a permutation is feasible is often non-trivial. To quantify the redundancy in generated permutations and the complexity of feasibility test, we examine a specific case—the hyperplane decision tree problem—where splitting rules are defined by hyperplanes.

As we have discussed in Chapter III.1, we showed that hyperplanes in $\mathbb{R}^D$ can be constructed using D-combinations of data points. Consequently, within a decision region, hyperplanes can only be generated from the data points contained in the decision region defined by their ancestors. This implies that the feasibility test p in the hyperplane decision tree problem must ensure that all data points defining the hyperplanes in a subtree remain within the decision region specified by its ancestors. This requirement imposes a highly restrictive constraint.

To assess the impact of this constraint, we introduce simple probabilistic assumptions. Suppose each hyperplane classifies a data point into the positive or negative class with equal probability, i.e. 1/2 for each class. If a hyperplane can serve as the root of a decision tree, the probability of this occurring is $\left(\frac{1}{2}\right)^{D\times(K-1)}$, since each hyperplane is defined by D-combinations of data points. Furthermore, the probability of constructing a proper decision tree with a chain-like structure (each branch node has at most one children) is given by

$$\left(\frac{1}{2}\right)^{D\times(K-1)} \times \left(\frac{1}{2}\right)^{D\times(K-2)} \times \ldots \times \left(\frac{1}{2}\right)^{D} = O\left(\left(\frac{1}{2}\right)^{D\times K^2}\right).$$

In Section III.3.10, we will show that when a decision tree has a chain-like structure, it exhibits the highest combinatorial complexity. This naive probabilistic assumption provides insight into the rarity of proper decision trees with K splitting rules compared to the total number of K-permutations of splitting rules.

For example, the 4-combination of hyperplanes shown in Fig 24 produces only *three* decision trees, as illustrated in Fig. 25, while the total number of possible permutations is 4!=24. Interestingly, although there are three possible trees, there are only two possible partitions. This suggests an interesting direction for further speeding up the algorithm by eliminating such cases, as our current algorithm cannot remove these duplicate partitions.

Moreover, the feasibility test is a non-trivial operation. As we discussed in previous section, verifying whether the data points used to define it lie within the same region as determined by its ancestor hyperplanes requires $O\left(K^2D^2\right)$ time in the worst-case. This will waste a lot of computations since it must be applied to all possible K-permutations.

Therefore, in order to achieve the optimal efficiency, it is essential to design a tree generator that directly generates only proper decision trees, eliminating the need for post-generation filtering, and even better, if this generator is amenable to fusion. Next, we will explain the design of such a proper decision tree generator and then demonstrate that the recursive generator is fusable with the `min r` function.

III.3.5 A simplified decision tree problem: the decision tree problem with K fixed splitting rules (branch nodes)

In the expanded specification `odt k`, where we have defined

```
genDTKs k = filter p . concatMap perms . kcombs k
```

This program suggests two potential approaches for fusion. The first approach, as described in the previous section, involves fusing `concatMap perms` with `kcombs k` to obtain a single `kperms k` function. However, a major drawback of this method is that the number of proper decision trees for a given set of rules is significantly smaller than the total number of permutations.

Alternatively, we can try to fuse the composed function `filter p . concatMap perms` by following equational reasoning

```
  filter p . concatMap perms . kcombs k
≡ filter fusion filter p . concat = concat . map (filter p)
  concat . map (filter p) . map perms . kcombs k
≡ map composition map f . map g = map (f.g)
  concatMap (filter p . perms) . kcombs k
≡ define genDTs = filter p . perms
  concatMap genDTs . kcombs k
```

where the laws used in above derivation can all be found in Bird [1987], since these laws are easy to verify and intuitively obvious, we do not repeat the proof here.

Now we can redefine the `genDTKs k` as

```
genDTKs k = concatMap genDTs . kcombs k
```

where `genDTs = filter p . perms`. The `genDTs` first generate all permutations of a given K-combination that then select those satisfies the feasibility test `p`. In other words, `genDTs` returns all possible valid permutation of a given K-combination, and this function is applied to each K-combination generated by `kcombs k`. If we can fuse `filter p . perms` into a single program, this would eliminate the need for a feasibility test, as all decision trees produced by the generator are inherently proper by design. We could then potentially obtain an efficient definition for `genDTs` and consequently, an efficient definition for `genDTKs k` as well. Indeed, one of the main contribution of this chapter is to derive an efficient definition for `genDTs`, which will be explored in subsection III.3.8.

Assuming we have an efficient definition for `genDTs`, the optimal decision tree problem can be reformulated as

```
odt k :: [R] -> Dtree R D
odt k = min r . concatMap genDTs . kcombs
```

In this definition, `odt k` receives a list of all possible rules `rs`, such that `length rs >= k` and generates all possible K-combinations of them, and then the `genDTs` function is applied to each K-combination of rules generated by `kcombs k`. It is important to distinguish `genDTs` from `genDTKs k`. The latter is parameterized by `k`, whereas `genDTs` operates on the output of `kcombs k`, without being explicitly parameterized by `k`.

Moreover, the above specification of `odt k` suggest another potential fusion

```
  min r . concat . map genDTs . kcombs k
≡ reduction promotion law min r . concat = min r . map (min r)
  min r . map (min r . genDTs) . kcombs
≡ define sodt = min r . genDTs
  min r . map sodt . kcombs
```

Again, the reduce promotion law used in the derivation can be found in Bird [1987]. Then we have following new definition for the optimal decision tree problem

```
odt' :: Int -> [R] -> Dtree R D
odt' k = min r . map sodt . kcombs k
```

where `sodt` is short for "*simplified optimal decision tree problem*", which is defined as

```
sodt :: [R] -> Dtree R D
sodt rs = min r . genDTs
```

The `sodt rs` find the optimal decision tree with respect to a K-combination of rules `rs` (`length rs = k`). Then, if we apply `sodt rs` to each K-combination of rules generated by `kcombs k`, we thereby obtain the optimal solution to the optimal decision tree problem `odt' k`.

The program `odt' k` is potentially more efficient than `odt k`, for two reasons: Firstly, program `sodt` is defined based on `genDTs`, which generate directly the proper decision trees instead of K-permutations. As we have discussed, the number of possible K-permutations is much large than the number of proper decision trees. Secondly, because for each K-combination of `rs`, there is only ***one*** optimal decision tree returned by `sodt`. The `min r` function in `odt' k` only needs to select the optimal decision tree from the set of optimal decision trees generated by `map sodt . kcombs k`, which is significantly smaller than the set of all possible decision trees with `k` splitting rules.

Therefore, the focus can be shifted to solving the following problem:

> *What is the optimal proper decision tree with respect to K fixed splitting rules, where the splitting rules are hyperplanes characterized as combinations of data points? Does there exist a greedy or dynamic programming (DP) solution to this problem?*

In other words, we seek an efficient definition for the function `genDTs`, which takes a fixed sequence (K-combination) of rules and generates all possible decision trees with respect to input rule list `rs`. Moreover, if we can fuse `min r` with `genDTs`, it could lead to a greedy or dynamic programming solution for `sodt`. The following discussion addresses these two questions.

III.3.6 An efficient proper decision tree generator

As discussed in background section, the *structure* of the decision tree is completely determined by the branch nodes, as the leaf nodes are defined by the intersections of the subspaces for all splitting rules in the path from the root to the leaf.

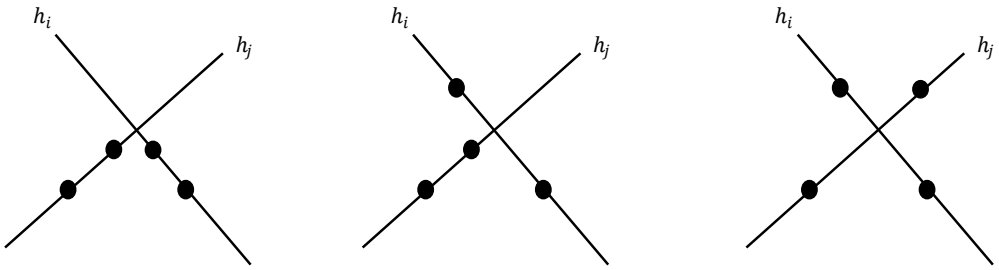


Figure 26: Three possible ancestry relations between two hyperplanes in $\mathbb{R}^2$, the black black circles represent data points used to define these hyperplanes.

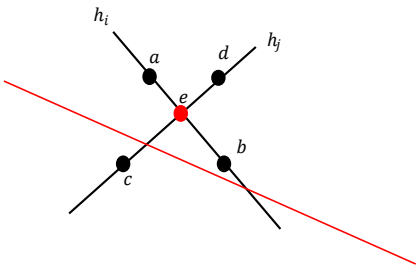


Figure 27: An example illustrating the proof of Fact 19. Demonstrating that the data items a, b, which define h_i, and c, d, which define h_j cannot be classified into the disjoint regions defined by a third hyperplane (red).

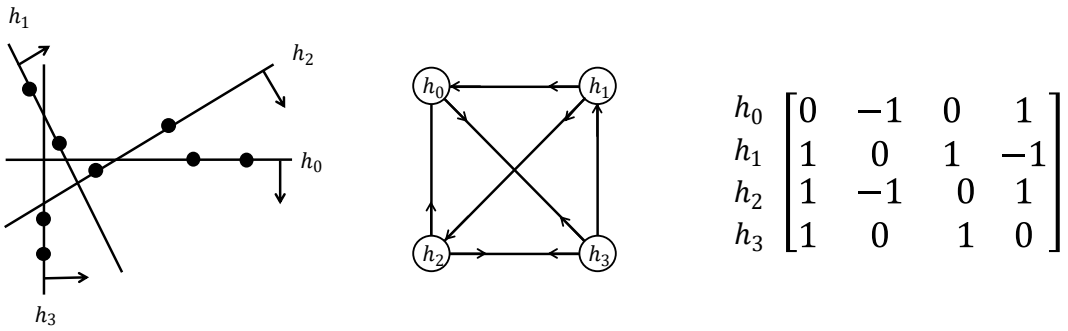


Figure 28: Three equivalent representations describing the ancestral relations between hyperplanes. A 4-combination of lines (left), each defined by two data points (black points) in $\mathbb{R}^2$, where black arrows represent the normal vectors to the corresponding hyperplanes. The *ancestry relation graph* (middle) captures all ancestry relations between hyperplanes. In this graph, nodes represent hyperplanes, and arrows represent ancestral relations. An incoming arrow to a node h_i indicates that the defining data of the corresponding hyperplane lies on the negative side of h_i. The absence of an arrow indicates no ancestral relation. Outgoing arrows represent hyperplanes whose defining data lies on the positive side of h_i. The ancestral relation matrix (right) $\boldsymbol{K}$, where the elements $\boldsymbol{K}_{ij} = 1$, $\boldsymbol{K}_{ij} = -1$, and $\boldsymbol{K}_{ij} = 0$ indicate that h_j lies on the positive side, negative side of h_i, or that there is no ancestry relation between them, respectively.

Therefore, to present a step-by-step derivation for `genDTs`, we first describe the construction of a "*partial decision tree generator*" `genBTs`, using the `Btree R` datatype. Then, we progressively extend the discussion to develop a "*complete decision tree generator*" by using Gibbons [1991]'s downward accumulation technique in next section.

Ancestry relation between hyperplanes

In the case of where splitting rules are defined by hyperplanes, the *ancestry relations* between two hyperplanes can be characterized in the following three ways:

1. **Mutual ancestry**: Both h_i and h_j can be the ancestors of each other.
2. **Asymmetrical ancestry**: Only one of them can be the ancestor of the other.
3. **No ancestry**: Neither h_i nor h_j can be the ancestor of the other. In this case, h_i and h_j are said to *cross each other*, and they are referred to as a pair of *crossed hyperplanes*.

Fig. 26, illustrates an example of these three cases.

Theorem 19. When two hyperplanes h_i and h_j cross each other, there exists no ancestry relation between h_i and h_j, and no hyperplane exists that can separate h_i and h_j into different branches. Thus any combination of hyperplanes that contains these crossed hyperplanes cannot form a proper decision tree.

Proof. To prove the first claim, it is shown that there exists no hyperplane that can split the data points defining h_i and h_j into two disjoint regions. This is a proof by contradiction. Assume a hyperplane is defined by data a and b, and h_j is defined by hyperplane c and d, and there exists third hyperplane h_k

that classifies a and b to the positive side of the hyperplane h_k and c, d to the negative side. In Euclidean space, a hyperplane divides the space into two disjoint convex regions. The convex combination of any data points in the positive region will also lie in the positive region of the hyperplane. For data points $\boldsymbol{x}_n$ on the positive side of a hyperplane, $\boldsymbol{w}^T\boldsymbol{x}_n > 0$. From the definition of convex combinations, for any set of data points $\{\boldsymbol{x}_1, \boldsymbol{x}_2 \ldots \boldsymbol{x}_m\}$, the convex combination $\boldsymbol{y} = \sum_{i=1}^{m} \lambda_i \boldsymbol{x}_i$ will satisfy $\boldsymbol{w}^T\boldsymbol{y} = \boldsymbol{w}^T \sum_{i=1}^{m} \lambda_i \boldsymbol{x}_i \geq 0$, where $\lambda_i \geq 0$. Now, consider the points a and b, which lie on the positive side of hyperplane h_k. Any points lying between a and b b will also be on the positive side of h_k , including the intersection point e of the segment ab with the segment cd. Since h_k classifies a and b to the positive side, it must classify e, which lies on the segment ab, to the positive side as well. This implies that any hyperplane containing e that intersects the positive region defined by h_k will also lie on the positive side. Hyperplane h_j is one such hyperplane, which leads to the contradiction that d lies in the negative region of h_k. Thus, we have a contradiction, proving that no hyperplane can split the data points defining h_i and h_j into two disjoint regions.Fig. 27 illustrates this argument geometrically in $\mathbb{R}^2$. The reasoning holds in higher-dimensional spaces, where lines are generalized to $D-1$-dimensional flats and intersection points are generalized to $D-2$-dimensional affine flats. For the second claim, given a set of hyperplanes which contains a pair of crossed hyperplanes h_i and h_j, any path in this decision tree contains h_i and h_j is not a feasible decision tree, because both h_i and h_j can not be the ancestor of each other. Therefore, any combination of hyperplanes that contains these the crossed hyperplanes cannot build any decision tree, because h_i and h_j cannot be in the same path, and there exists no third hyperplane to classify them into different branches. □

Since crossed hyperplanes cannot construct decision trees, it will be assumed that the set of hyperplanes used does not contain any crossed hyperplanes in the following discussion.

Decision tree generator over binary tree datatype

In order to construct a tree generator, there is the need to address an important question: which splitting rules can be the root of the tree? Because of axiom three, not every splitting rule can be the ancestor of another, and the root of the tree is the only thing we need to consider, because all splitting rules are the root of some subtrees.

According to the definition, a branch node can be the root if and only if it can be the ancestor of all descendant hyperplanes (except itself) in hs. In other words, $h_i \, (\swarrow \vee \searrow) \, h_j$, for all $h_j \in hs$, $j \neq i$. Geometrically, this means that all data points used to define the descendant hyperplanes must lie on the same (either positive or negative) side of the root hyperplane. In the ancestry relation graph, if h_i is the root, all edges connected to h_i and $h_j \in hs$ $(j \neq i)$ must have arrows, either incoming or outgoing. For example, in Fig. 28, h_0 cannot be the root of $hs = \{h_0, h_1, h_2, h_3\}$, because the head closer to h_0 in the edge between h_0 to h_2 does not contain an arrow. More simply, since $\boldsymbol{K}_{ij} \neq 0$, if h_i can be the ancestor of h_j, then a hyperplane h_i can be the root, if and only if $\boldsymbol{K}_{ij} \neq 0$ for all $h_j \in hs$. Therefore, a "split" function can be constructed that identifies which hyperplanes, within a given set of hyperplanes, are viable candidates to become the root of a decision tree with respect a set of hyperplane hs. This function can be defined in Haskell as

```
feasible_split :: [H] -> [H]
```

```
feasible_split hs = [ hi | hi <- hs, all (ind_mat hi) (filter (/= hi) hs)]
  where ind_mat hi hj = ((anc_rel_mat!!hi)!!hj)/=0
```

where the `anc_rel_mat` is the ancestry relation matrix, which is defined as a list of lists in Haskell.

Once the set of hyperplanes that are viable candidates for the root of the tree are determined, an important observation follows: when a hyperplane becomes the root, it splits the remaining hyperplanes into either the left or right branch (subtree) based on their values in the ancestry relation matrix. Specifically, if h_i is the root hyperplane and $\boldsymbol{K}_{ij} = 1$, then h_j goes to the left branch of h_i; otherwise, it goes to the right branch. This observation leads to the definition of the following function

```
split :: H -> [H] -> ([H], H, [H])
split r [hj] = ([], r, [])
split r [hj,hk]
 |r == hj = if (anc_rel_mat!!r)!!hk == 1 then ([hk], r, [])
                                         else ([], r, [hk])
 |r == hk = if (anc_rel_mat!!r)!!hj == 1 then ([hj], r, [])
                                         else ([], r, [hj])
split r hs = (hs_pos, r, hs_neg)
 where
  hs_pos = [hj | hj<-hs, (anc_rel_mat!!r)!!hj == 1]
  hs_neg = [hj | hj<-hs, (anc_rel_mat!!r)!!hj == -1]
```

The `split` function divides a set of hyperplanes `hs` into a triple—the hyperplanes classified into the left subtree (`hs_pos`), the root hyperplane `r`, and the hyperplanes classified into the right subtree `hs_neg`. The `split` function is also used in studying the optimal decision tree problem over binary feature data, as explored in several papers [Lin et al., 2020, Hu et al., 2020, Narodytska et al., 2018, Aglin et al., 2021, 2020, Avellaneda, 2020, Verwer and Zhang, 2019, Jia et al., 2019, Zhang et al., 2023, Mazumder et al., 2022]. However, none of these studies explicitly define the split function in their pseudo code, it remains obscure how their algorithm works in actual implementation.

With the help of the `split` function, which makes sure only *feasible splitting rules*—those that can serve as the root of a tree or subtree—are selected as the root. We can define an efficient decision tree generator as follows

```
genBTs :: [H] -> [Btree H]
genBTs [] = [L]
genBTs [a] = [N L a L]
genBTs hs = [N u r v | (hs_pos, r, hs_neg) <- splits, u <- genBTs hs_pos,
   v <- genBTs hs_neg]
 where splits = [split r hs | r <- feasible_split hs]
```

The `genBTs` generator recursively construct larger proper decision trees `N u r v` from smaller proper decision trees `genBTs hs_pos` and `genBTs hs_neg`, the `split` function ensures that only feasible splitting rules can become the root of a subtree during recursion. Note that the definition of `genBTs` (and the `genDTs` as well) does not require the input sequence `rs` to have a fixed size, as it can process input

sequences of arbitrary length. However, when used within the `sodt k` function, it must be constrained to ensure that the generated tree has a fixed size of `k`.

The complexity of `genBTs` depends on the number of possible proper decision trees. Since this number is determined by the distribution of the data, it is challenging to analyze the complexity precisely. However, we will provide a worst-case combinatorial complexity analysis in a later discussion, which shows that the algorithm has a complexity of $O(K! \times N)$ in the worst case, where K is the number of branch nodes.

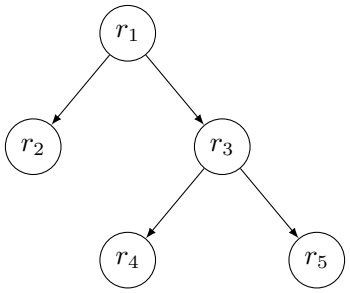
III.3.7 Downward accumulation

We are now half way towards our goal. The `genBTs` function provides an efficient way of generating the *structure of the decision tree*, namely binary tree representations of a decision tree. However, this is just a "partial decision tree generator." Since a decision tree is not a binary tree, we need to figure out how to "pass information down a tree" from the root towards the leafs during the recursive construction of the tree. In other words, we need to *accumulate* all the information for each path of the tree from the root to each leaf.

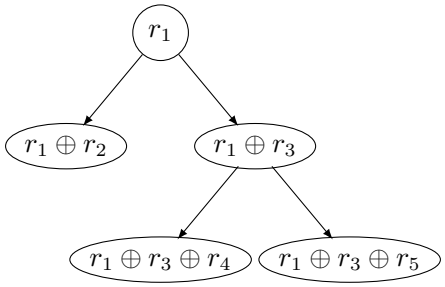
In this section we introduce a technique called *downwards accumulation* [Gibbons, 1991], which will helps us to construct the "complete decision tree generator." Accumulations are higher-order operations on structured objects that leave the shape of an object unchanged, but replace every element of that object with some accumulated information about other elements. For instance, the prefix sums (with binary operator $\oplus$) over a list is an example of accumulation over list $[a_1, a_2, \ldots a_n]$,

$$[a_1, a_1 \oplus a_2, \ldots, a_1 \oplus a_2 + \ldots \oplus a_n] .$$

Downward accumulation over binary trees is similar to list accumulation, as it replaces every element of a tree with some function of that element' s ancestors. For example, given a tree



Applying downwards accumulation with binary operator $\oplus$ to the above tree results in



The information in each leaf is determined by the path from the root to that leaf. Therefore, Gibbons [1991]'s *downward accumulation* method can be adopted for constructing the decision tree generator.

To formalize downward accumulation, as usual, we first need to define a *path generator* and a *path datatype*. The ancestry relation can be viewed as a path of length one, thus we can abstract ↙ and ↘ as constructors of the datatype. We define the path datatype recursively as

```
data Path a = S a
            | Path a / Path a
            | Path a \ Path a deriving Show
```

In other words, a path is either a single node `S a`, or two paths connected by ↙ or ↘. The Unicode symbols ↙ and ↘ are not rendered in the program listing environment, we replace them with "/" and "\", respectively. Note that, "\" should not be misinterpreted as the symbol used in the definition of a lambda expression. Since the lambda expression are not used in the discussion here, it is safe to use "\" to represent ↘.

The *path reduction* (also known as a *path homomorphism*), applies to a path and reduces it to a single value:

```
pathRed ::(b->c)->(b->c->c)->(b->c->c)->Path b->c
pathRed f op ot (S xs)     = f xs
pathRed f op ot (a / b) = pathRed (flip op (pathRed f op ot b)) op ot a
pathRed f op ot (a \ b) = pathRed (flip ot (pathRed f op ot b)) op ot a
```

where `filp :: (a -> b -> c) -> b -> a -> c` function flipping the order of arguments.

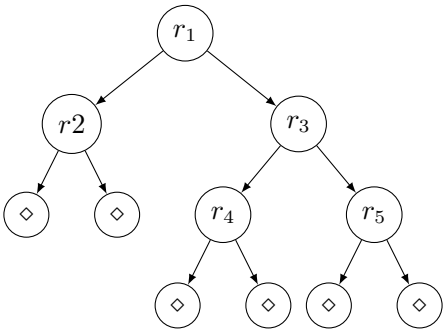
The next step toward defining downward accumulation requires a function that generates all possible paths of a tree. For this purpose, Gibbons [1996] introduced a definition of *paths* over a binary tree, which replaces *each* node with the path from the root to that node. However, the accumulation required for the decision tree problem differs from the classical formulation. In Gibbons [1996]'s downward accumulation algorithm, information is propagated to every node, treating both branch and leaf nodes uniformly. By contrast, the decision tree problem requires passing information only to the leaf nodes, leaving the branch nodes unchanged.

Analogous to Gibbons [1996]'s definition of *paths* for binary tree datatypes, we can alternatively define the path generator as

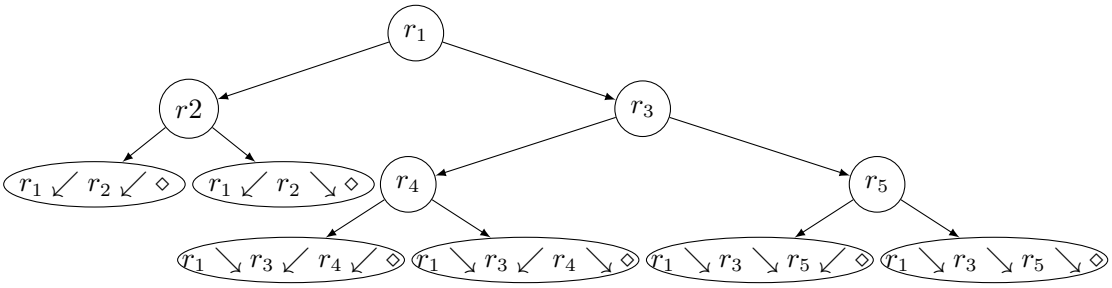
```
paths :: (a -> b) -> (a -> b) -> Dtree a b -> Dtree a (Path b)
paths f g (DL xs) = DL (S xs)
paths f g (DN u r v) = DN (mapD (S (f r)/) (paths f g u)) r (mapD (S (g
   r)\) (paths f g v))
```

where `Path` type receives only one type `b`, two functions `f` and `g` are used to transform `r::a` into type `b` , while also distinguishing between "left turn" (↙) and "right turn" (↘) .

To see how this works, consider the decision tree T given below, where for simplicity, the singleton path constructor `S` is left implicit, and we denote the leaf value using symbol ⋄



Running *paths* on decision tree T, we obtain



Here, only the leaf nodes are replaced with the path from the root to the ancestors, while the structure and branch nodes of the tree remain unchanged. Our required *downward accumulation* over proper decision tree datatypes, passes all information to the leaf nodes, leaving the splitting rules (branch nodes) unchanged. This can be formally defined as

```
dacc h op ot f g = mapD (pathRed h op ot) . paths f g
```

Every downward accumulation has the following property

```
dacc (DN u r v) = DN (mapD (op (f r)) (dacc u)) r (mapD (ot (g r)) (dacc' v))    (146)
```

where the parameters for `dacc h op ot f g` is ignored for simplicity.

Equation (146) can be proved by following equational reasoning

```
  dacc h op ot f g (DN u r v)
```
≡ definition of `dacc`
```
  mapD (pathRed h op ot) $ paths f g (DN u r v)
```
≡ definition of `paths`
```
  mapD pathRed (DN (mapD (S (f r)↙) (paths u)) r (mapD (S (g r)↘) (paths v)))
```
≡ definition of `mapD`
```
  DN (mapD pathRed (mapD (S (f r)↙) (paths u))) r (mapD pathRed (mapD (S (g r)↘) (paths v)))
```
≡ map composition
```
  DN (mapD (pathRed.(S (f r)↙)) (paths u)) r (mapD (pathRed.(S (g r)↘)) (paths v))
```
≡ definition of `pathRed`
```
  DN (mapD ((op (f r)).pathRed) (paths u)) r (mapD ((ot (g r)).pathRed) (paths v))
```
≡ map composition
```
  DN (mapD (op (f r)) $ mapD pathRed (paths u)) r (mapD (ot (g r)) $ pathRed (paths v))
```
≡ definition of `dacc`
```
  DN (mapD (op (f r)) $ dacc u) r (mapD (ot (g r)) $ dacc v)
```

we have ignored all parameters for `pathRed h op ot`, `paths f g`, and `dacc g op ot f g` for simplicity.

The downwards accumulation `dacc` is both *efficient* and *homomorphic*. This homomorphic downward accumulation can be computed in parallel functional time proportional to the product of the depth of the tree and the time taken by the individual operations [Gibbons, 1996], and thus is amenable for fusion.

III.3.8 An efficient definition for proper decision tree generator

In section III.3.6, we described the construction of a decision tree generator based on the binary tree data type. However, the `genBTs` function generates only the *structure* of the decision tree, which contains information solely about the branch nodes. While this structure is sufficient for evaluating the tree, constructing a *complete decision tree*—one that incorporates both branch nodes and leaf nodes—is essential for improving the algorithm's efficiency.

Before we moving towards in deriving complete decision tree generator, we need to generalize `genBTs` to define it over `Dtree H D` datatype

```
genBTs'':: [R] -> D -> [Dtree R D]
genBTs' [] xs  = [DL xs]
genBTs' [a] xs = [DN (DL xs) a (DL xs)]
genBTs' hs xs  = [DN u r v | (hs_pos, r, hs_neg) <- splits, u <- genBTs'
   hs_pos xs, v <- genBTs' hs_neg xs]
 where splits = [split r hs | r <- feasible_split hs]
```

where we simply replace all occurrences of `L` in the definition of `genBTs` with the input data sequence `xs`, and splitting rules are defined by hyperplanes `hs::[H]`.

The difference between complete and partial decision tree generator lies in the fact that the complete one contains accumulated information in leaves and the partial one has no information just leaf label `L`. This reasoning allows us to associated the complete decision tree generator `genDTs` with `genBTs'` as following

```
genDTs :: [H] -> D -> [Dtree H D]
genDTs rs xs = map daccDT $ genBTs' rs xs
 where daccDT = dacc id si si sl sr
```

where `si` is the standard set intersection operation over two lists, defined as

```
si :: Eq a => [a] -> [a] -> [a]
si xs ys = nub [x | x <- xs, x `elem` ys]
```

which computes the set intersection of two lists, `xs` and `ys`.

Functions `sl` and `sl` are abbreviations for "split left" and "split right", respectively, defined as

```
sl = fst . split_hr
sr = snd . split_hr
```

The `split_hr` function simply splits a given dataset (list of indices) into two smaller datasets h_r^+ and h_r^- determined by hyperplane h_r, recall that $h_r^+ = \{\boldsymbol{x} : \boldsymbol{x} \in \mathcal{N}, \bar{\boldsymbol{w}}_r^T \boldsymbol{x} < 0\}$ and $h_r^- = \{\boldsymbol{x} : \boldsymbol{x} \in \mathcal{N}, \bar{\boldsymbol{w}}_r^T \boldsymbol{x} \geq 0\}$. Therefore, `hr_pos` consists of data items with a prediction label equal to 1, and `hr_neg` consists of data items with a prediction label to -1. Thus `split_hr` function is defined as

```
split_hr :: H -> D -> (D, D)
split_hr r  = (hr_pos, hr_neg)
 where
  asgn = asgns!!r
  hr_pos = [i | i<-[0..n-1], (asgn!!i) == 1]
  hr_neg = [i | i<-[0..n-1], (asgn!!i) == -1]
```

where `asgns` is a list of assignments, and `asgn=asgns!!r` is the assignment (prediction labels) of the hyperplane indexed by `r`. The `split_hr r` function always split the *input data sequence* into two subsequences based on the predictions in `asgn`.

Based on the specification of `genDTs`, we can derive an efficient definition of `genDTs` by following equational reasoning

```
  map daccDT $ genBT' hs xs
```
≡ definition of `daccDT`
```
  map (mapD (pathRed id si si . paths sl sr)) $ genBTs' hs xs
```
≡ definition of `genBTs'`
```
  map (mapD (pathRed id si si . paths sl sr)) [DN u r v|
  (hs_pos, r, hs_neg) <- splits, u <- genBTs' hs_pos xs, v <- genBTs' hs_neg xs]
```
≡ definition of `map`
```
  [mapD (pathRed id si si . paths sl sr) (DN u r v)|
    (hs_pos, r, hs_neg) <- splits, u <- genBTs' hs_pos xs, v <- genBTs' hs_neg xs]
```
≡ property of downward accumulation
```
  [DN (mapD (si (sl r) . pathRed id si si) (paths sl sr u)) r
     (mapD (si (sr r) . pathRed id si si) (paths sl sr v)) |
     (hs_pos, r, hs_neg) <- splits, u <- genBTs' hs_pos xs, v <- genBTs' hs_neg xs]
```
≡ map composition, definition of list comprehension, and definition of `daccDT`
```
  [DN (mapD (si (sl r)) u) r (mapD (si (sr r)) v) |
     (hs_pos, r, hs_neg) <- splits, u <- map daccDT (genBTs' hs_pos xs),
     v <- map daccDT (genBTs' hs_neg xs)]
```
≡ definition of `genDTs`
```
  [DN (mapD (si (sl r)) u) r (mapD (si (sr r)) v) |
     (hs_pos, r, hs_neg) <- splits, u <- genDTs hs_pos xs),
     v <- genDTs hs_neg xs)]
```

The singleton and empty cases are very easy to prove, we only prove the singleton case here, the empty case is left as an interesting exercise

```
  map daccDT $ genBTs' [r] xs
```
≡ definition of `genBTs'`
```
  map daccDT [DN (DL xs) r (DL xs)]
```
≡ definition of `map`
```
  [daccDT (DN (DL xs) r (DL xs))]
```
≡ definition of `daccDT`
```
  [DN (DL (si xs (sl r))) r (DL (si xs (sr r)))]
```

Finally, an efficient definition for *genDTs* is rendered as

```
genDTs :: [H] -> D -> [Dtree H D]
genDTs [] xs  = [DL xs]
```

```
genDTs [r] xs = [DN (DL (si xs (sl r)) r (DL (si xs (sr r))))]
genDTs hs xs = [DN (mapD (si xs (sl r)) u) r (mapD (si xs (sr r)) v) |
                    (hs_pos, r, hs_neg) <- splits,
                    u<-genDTs hs_pos xs, v<-genDTs hs_neg xs]
                  where splits = [split r hs | r <- feasible_split hs]
```

Running `genDTs hs xs` will generate all proper decision tree with respect to a list of rules `hs`, and leaf nodes of each tree contains all downward accumulated information with respect to input data sequence `xs`.

The difference of `genDTs` and `genBTs'` is that `genDTs` accumulate information every time it create a root `r` to every proper decision subtree generated by `genDTs hs_pos xs` and `genDTs hs_neg xs` using `mapD (si (sl r))` and `mapD (si (sl r))`, respectively.

III.3.9 A generic dynamic programming algorithm for the proper decision tree problem

A dynamic programming recursion

The key fusion step in designing a DP algorithm is to fuse the `min r` function with the generator, thereby preventing the generation of partial configurations that cannot be extended to optimal solutions, i.e., optimal solutions to problems can be expressed purely in terms of optimal solutions to subproblems. This is also known as the *principle of optimality*, which is originally investigated by [Bellman, 1954]. Since 1967[Karp and Held, 1967], extensive study [Karp and Held, 1967, Ibaraki, 1977, Bird and De Moor, 1996, Bird and Gibbons, 2020] shows that the essence of the principle of optimality is *monotonicity*. In this section, we will explain the role of monotonicity in the decision tree problem and demonstrate how it leads to the derivation of the dynamic programming algorithm.

We have previously specified the simplified decision tree problem in (III.3.5). However this specification concerns decision tree problem in general, which may not involve any input data. Since now we tried to derive a dynamic programming algorithm for solving an optimization problem, we now redefine (we can use the same reasoning to derive parameterized `soct` from a parameterized `odt k` specification) the`sodt :: [R] -> D -> Dtree R D` problem by parameterized it with an input sequence `xs`

```
sodt :: [H] -> D -> Dtree H D
sodt hs xs = min r $ genDTs hs xs
```

where `min r` selects one of the optimal tree returned by `genDTs hs xs`.

The objective function `cost :: Dtree R D -> Obj` for any decision tree problems with objective function that conform to the following general scheme

```
type Obj = Double

cost :: (D->Obj)->(Obj->Obj->Obj)->Dtree R D->Obj
cost f g (DL xs) = f xs
cost f g (DN u r v) = g (cost f g (upl u)) (cost f g (upr v))
 where upl u = mapD (si (sl r)) u
```

```
upr v = mapD (si (sr r)) v
```

such that `g a b >= cost f g a` $\wedge$ `cost f g b`.

For example, consider the decision tree model for the classification problem, the ultimate goal is to find a proper decision tree that minimize the number of misclassification [Breiman et al., 1984]. Assume the data sequence `xs` is associated with a corresponding label sequence `ys`, where element `y` in `ys` represents a label, encoded as an integer ranging from `0` to `m-1`. We can define this objective as

```
cost_cmn = cost cmn (+)

cmn :: D -> Obj
cmn xs =   sum [1 | y<-ys', y'/= y]
 where y' = mostFreq m ys'
       ys' = [y | (i, y) <- zip [0..] ys, i `elem` xs]

mostFreq m ys = fst $ maximumBy f [(i, count i ys) | i <- [0..m-1]]
 where count i ys = sum [1 | y <- ys, y == i]
```

where `cmn` is short for "count misclassification number" and `mostFreq` function count the most frequent label `y'` in the label sequence `ys`. This is the most common decision trees objective function used in machine learning; alternative objective functions can also be used.

Based on this definition of the objective function, the following lemma trivially holds.

Theorem 20. *Monotonicity in the decision tree problem.* Given left subtrees u and u', and the right subtrees v and v' rooted at r,the implication

$$\texttt{cost u `leq` cost u'} \wedge \texttt{cost v `leq` cost v'} \Longrightarrow \texttt{cost (DN u r v) `leq` cost (DN u' r v')} \tag{147}$$

only holds, in general, if $r = r'$.

Proof. Assume `cost u `leq` cost u'` $\wedge$ `cost v `leq` cost v'`. According to the definition of the objective function, `cost (DN u r v) `leq` cost (DN u' r v')` only holds, in general if $r = r'$. □

Note that the monotonicity described above does not rule out the possibility that `cost (DN u r v) `leq` cost (DN u' r' v')` for objective functions with special f and g. Note again, we use `min r` is simply introduced to extend our powers of specification and will not appear in any final algorithm. It is safe to use as long as we remember that `min r` returns *one possible* optimal solution [Bird and Gibbons, 2020, Bird and De Moor, 1996].

Due to the monotonicity of the problem, we can now derive the program by following equational

reasoning

```
min r $ genDTs hs xs
```

≡definition of `genDTs`

```
min r [DN (mapD (si xs (sl r)) u) r (mapD (si xs (sr r)) v)|
  (hs_pos, r, hs_neg)<-splits, u<-genDTs hs_pos xs, v<-genDTs hs_neg xs)]
```

⊆ monotonicity

```
min r [DN (min r [mapD (si xs (sl r)) u| u <- genDTs hs_pos xs]) r
  (min r [mapD (si xs (sr r)) v | v <- genDTs hs_neg xs])|(hs_pos, r, hs_neg) <- splits]
```

≡ definition of `map`

```
min r [DN (min r $ map (mapD (si xs (sl r))) $ genDTs hs_pos xs) r
  (min r $ map (mapD (si xs (sr r))) $ genDTs hs_neg xs))|(hs_pos, r, hs_neg) <- splits]
```

≡ definition of `mapD`

```
min r [DN (min r $ genDTs hs_pos (si xs (sl r))) r
  (min r $ genDTs hs_neg (si xs (sr r))))|(hs_pos, r, hs_neg) <- splits]
```

≡ definition of `sodt`

```
min r [DN (sodt hs_pos (si xs (sl r))) r
  (sodt hs_neg (si xs (sr r))))|(hs_pos, r, hs_neg) <- splits]
```

≡ let `xs_pos, xs_neg = si xs (sl r), si xs (sr r)`

```
min r [DN (sodt hs_pos xs_pos) r (sodt hs_neg xs_neg)|(hs_pos, r, hs_neg) <- splits]
```

Again, the proof for singleton and empty cases are trivial to verify, we ignore the proof for them. Therefore, the ODT problem with K fixed splitting rules can be solved exactly using

```
sodt :: [H]-> D -> Dtree H D
sodt [] xs  = DL xs
sodt [r] xs = DN (DL xs_pos) r (DL xs_neg) where (xs_pos, xs_neg) = sai r
   xs
sodt hs xs =  minlist r [DN (sodt hs_pos xs_pos) r (sodt hs_neg xs_neg) |
          (hs_pos, r, hs_neg) <- splits, let (xs_pos, xs_neg) = sai r xs]
  where splits = [split r hs | r <- feasible_split hs]
        r a b = cost_cmt a <= cost_cmn b
```

where `sai` function is short for "split and intersect", defined as `sai r xs = (si xs (sl r), si xs (sr r))`.

III.3.9.1 Filtering process

In machine learning research, to prevent *overfitting*, a common approach is to impose a constraint that the number of data points in each leaf node must exceed a fixed size, $N_{\min}$, to avoid situations where a leaf contains only a small number of data points. One straightforward method to apply this constraint is to incorporate a filtering process by defining

```
genDTFs m hs xs = filter (minData m) $ genDTs hs xs
```

where the predicate `minData` checks whether the number of data points in *all* leaves is greater than `m`, which is implemented as

```
minData :: Int -> Dtree a [b] -> Bool
minData m (DL t) = (length t) >= m
minData m (DN u r v) = (minData m u) && (minData m v)
```

However, this direct specification is not ideal, as `genDTs` can potentially generate an extremely large number of trees, making post-generation filtering computationally inefficient. To make this program efficient, it is necessary to fuse the post-filtering process inside the generating function. As we explained in Section II.1.2, if the one-step update function in a recursion "*reflects*" a predicate `p`, then the filtering process can be incorporated directly into the recursion. This approach allows for the elimination of infeasible configurations before they are fully generated.

In the tree datatype, we say that "`f` reflects `p`" if `p (f (DN u r v))` $\Longrightarrow$ `p u` $\wedge$ `p v`. For decision tree problem, the update function `f` and the predicate `p` are defined as
`f (DN u r v) = DN (mapD (si (sl r)) u) r (mapD (si (sr r)) v)` and `minData` respectively. Since the number of data points in each leaf decreases as more splitting rules are introduced, it is trivial to verify that the implication holds.

As a result, the filtering process can be integrated into the generator, and the new generator, after fusion, is defined as

```
genDTFs :: Int -> [H] -> D ->[Dtree H D]
genDTFs m [] xs = [DL [0..n-1]]
genDTFs m [r] xs = [DN (DL (si xs (sl r))) r (DL (si xs (sr r)))]
genDTFs m hs xs = filter (minData m)
  [DN (mapD (si (sl r)) u) r (mapD (si (sr r)) v)|
       (hs_pos, r, hs_neg) <- splits,
       u<-genDTFs m hs_pos xs,
       v<-genDTFs m hs_neg xs]
 where splits = [split r hs | r <- feasible_split hs]
```

Substituting definition `genDTFs m` in the derivation of `sodt` could potential gives us a more efficient definition for `sodt`, as `genDTFs m` generate provably less configurations than `genDTs`.

Applicability of the memoization technique

In the computer science community, dynamic programming is widely recognized as recursion with overlapping subproblems, combined with memoization to avoid computations about repeated subproblems. both conditions are satisfied. We say, a dynamic programming solution exists if both conditions are satisfied.

At first glance, the optimal decision tree problem involves shared subproblems, suggesting that a DP solution is possible. However, we will explain in this section that, despite the existence of these shared subproblems, *memoization* is impractical for most of the decision tree problems.

Below, we analyze why this is the case, using a counterexample where the memoization technique **is applicable**—the matrix chain multiplication problem (MCMP)—and discuss the key differences.

In dynamic programming algorithms, a key requirement often overlooked in literature is that the optimal solution to one subproblem must be *equivalent* to the optimal solution to another. For example, in the matrix chain multiplication problem, the goal is to determine the most efficient way to multiply a sequence of matrices. Consider multiplying four matrices A, B, C, and D. $((AB)\,C)\,D = (AB)\,(CD)$ describes two ways of multiplying the matrices will yield the same result.

Because the computations involved may differ due to varying matrix sizes, the computation on one side may be more efficient than the other. Nonetheless, our discussion here is not focused on the computational complexity of this problem. One of the key components of the DP algorithm for MCMP is that the computational result (AB) can be reused. This is evident as (AB) appears twice in the computations for $((AB)\,C)\,D$ and $(AB)\,(CD)$. It is therefore possible to compute the result for the subproblem (AB) first, and then directly use it in the subsequent computations of $((AB)\,C)\,D$ and $(AB)\,(CD)$, thereby avoiding the recomputation of (AB).

However, in the decision tree problem, due to Lemma 20, the implication only holds true if $r = r'$. Therefore, to use the memoization technique, we need to store not only the optimal solution of a subtree generated by a given set of rules rs, but also the root of each subtree. This requires at least $O\left(|\mathcal{S}_{Dtree}| \times |\mathcal{S}_{\mathcal{H}}|\right)$ space, where $|\mathcal{S}_{Dtree}|$ is the number of possible decision trees, and $|\mathcal{S}_{\mathcal{H}}|$ is the number of possible roots. Thus, storing all this information during the algorithm's runtime is impractical for most decision tree problems considered in machine learning.

For example, a hyperplane decision tree problem involves $O\left(N^D\right)$ possible splitting rules and $O\left(N^{DK}\right)$ possible subtrees in the worst case. Therefore, the use of the memoization technique in many well-established studies [Hu et al., 2019, Aglin et al., 2021, Zhang et al., 2023, Lin et al., 2020, Demirović et al., 2022, Nijssen and Fromont, 2010, Aghaei et al., 2019] is wrong, as they only store the optimal solution of the subtree for a particular root. However, for different roots, the optimal subtree may differ.

III.3.10 A finer combinatorial complexity analysis

It is difficult to precisely analyze the average (or best) combinatorial complexity of the decision tree problem because it is highly dependent on the data, unless certain assumptions are made about the distribution of the data. In this section, we will analyze the worst-case complexity of this problem, which is related to the following lemma.

Lemma 19. The decision tree problem with K fixed rules achieves maximum combinatorial complexity when any rule can serve as the root and each branch node has exactly one child. Formally, for any $r_i \in rs$, we have $\boldsymbol{K}_{ij} = 1$ or $\boldsymbol{K}_{ij} = -1$ for all $r_j \in rs$, $i \neq j$, and $\left|\sum_{j \in rs'} \boldsymbol{K}_{ij}\right| = |rs'| - 1$ and for each subtree defined by splitting rule subset $rs' \subseteq rs$. Then the decision tree problem has the largest combinatorial complexity.

Proof. Consider the case where for any $r_i \in rs$, we have $\boldsymbol{K}_{ij} = 1$ or $\boldsymbol{K}_{ij} = -1$ for all $h_j \in hs$, $i \neq j$, and $\left|\sum_{j \in rs'} \boldsymbol{K}_{ij}\right| = |rs'| - 1$. Under these conditions, each subtree has exactly one child, resulting in a tree with a single path (excluding leaf nodes). Since the structure is fully determined by the branch nodes, we can disregard the leaf nodes. This configuration permits any permutation of branch nodes, yielding

maximum combinatorial complexity. We demonstrate this by proving that placing pairs of splitting rules at the same tree level reduces the problem's complexity.

For a K-permutation p, consider first the case of a chain of decision rules where each node has exactly one child. Given our assumption that any splitting rule can serve as the root, all permutations of the decision tree are valid, resulting in $K!$ possible chains. For the alternative case, consider a permutation $p = [\ldots r_j, r_k, \ldots]$ where rules r_j and r_k occupy the same level with immediate ancestor r_i. By Theorem 18, these rules must be the left and right children of r_i, respectively, and their positions are immutable. When r_i precedes both r_k and r_j in the permutation, r_j and r_k will always be separated into different branches. In the worst case, with r_k and r_j at the tail of a permutation list, i.e. $p' = [\ldots r_j, r_k]$. Thus, when the permutation $[\ldots r_k, r_j]$ is not allowed, all permutations where $(K-2)$ rules precede both r_k and r_j become invalid, eliminating $(K-2)!$ possible permutations. As additional pairs of rules become constrained to the same level, the number of invalid permutations increases monotonically. Therefore, the decision tree attains maximal combinatorial complexity when it assumes a "chain" structure, where each non-leaf node has exactly one child node. □

This fact implies that the decision tree generator given above, for K splitting rules, has a worst-case time complexity of $O(K!)$. Therefore, assume the predictions of all splitting rules are pre-computed and can be indexed in $O(1)$ time, and denote by $T(K)$ the worst-case complexity of *sodt* with respect to K splitting rules, so the following recurrence for the time complexity applies,

$$\begin{aligned} T(1) &= O(1) \\ T(K) &= K \times (T(K-1) + T(1)) + O(N), \end{aligned}$$

with solution $T(K) = O(K! \times N)$. While this complexity is factorial in K, it is important to note that the worst-case scenario occurs only when the tree consists of a ***single*** path of length K. However, such a tree is generally considered the least useful solution in practical decision tree problems, as it represents an extremely deep and narrow structure.

In most cases, decision trees that are as shallow as possible are preferred, as shallow trees are typically more interpretable. Deeper trees tend to become less interpretable, particularly when the number of nodes increases. Therefore, while the worst-case complexity is factorial, it does not necessarily represent the typical behavior of decision tree generation in practical scenarios, where the goal is often to minimize tree depth for improved clarity and efficiency.

Complexity of the hyperplane decision tree problem The complexity for the optimal hyperplane decision tree problem is relatively easy to analyze. Given that there are $O\left(N^{DK}\right)$ combination of hyperplanes in total,

$$O\left(N^{DK} + K! \times N \times N^{DK}\right) = O\left(N^{DK}\right). \tag{148}$$

Although this analysis presents the same upper-bound complexity as the previous one, it provides a better understanding of the problem. In practice, the average complexity will be much smaller than the theoretical upper bound established here, as we have shown that the worst-case complexity of the `sodt` program is attained only when the tree exhibits a "chain" structure, and we also show that, in Section (III.3.4), this happens with very low probability.

Complexity of the axis-parallel decision tree problem Similarly, for the axis-parallel decision tree problem, there are $\begin{pmatrix} ND \\ K \end{pmatrix} = O\left(N^K\right)$ possible splits of hyperplanes. Thus, the combinatorial complexity of the problem is

$$O\left(N \times D + K! \times N \times \times (N \times D)^K\right) = O\left(N^K\right). \tag{149}$$

Note that when K branch nodes are fixed, although both hyperplane decision trees and axis-parallel decision trees are bounded by $O(K!)$, the number of feasible trees in the hyperplane case is much smaller than in the axis-parallel case. This is because, in the axis-parallel decision tree case, ***every*** *pair of hyperplanes is mutually ancestral*, a situation that rarely occurs in the hyperplane split case.

Complexity of the decision tree problem over binary feature data Lastly, the case for binary feature data differs. In the previous discussions, if hyperplane $h_1 \swarrow h_2$ or $h_1 \searrow h_2$, it can never be the case that $h_1 \searrow h_2$ or $h_1 \swarrow h_2$, i.e. if h_2 is in the left branch of h_1, it cannot lie in the right branch of it.

However, in decision trees with binary feature data, hyperplanes are abstracted as the selection or non-selection of features from the data. This allows any pair of branch nodes to be mutually ancestral and *symmetric*. By symmetric, we mean that any branch node can serve as either the left or right branch of the root node. In other words, selecting features x_i and x_j, both $x_i \swarrow x_j$ and $x_i \searrow x_j$ are possible when x_i is the root, and $x_j \swarrow x_i$ and $x_j \searrow x_i$ are possible with x_j as root. This holds because, in all previous cases, hyperplanes are characterized by data points, and similarly, in the axis-parallel tree case, only **one** data item is needed to define each hyperplane, and this item always lies on one side of another hyperplane. However, for binary feature data, this method of characterization is no longer applicable. In the case of binary feature data, it is only necessary to consider whether a particular feature is selected or not and there are K features in total used to define decision trees. Hence, for binary feature data, the number of possible decision trees depends not only on the shape of the tree (which can be counted using the Catalan number) but also on all possible permutations of the labels.

Therefore, assume D is much larger than K, the worst-case combinatorial complexity for the decision tree problem with binary feature data is

$$O\left(\begin{pmatrix} D \\ K \end{pmatrix} + K! \times \text{Catalan}(K) \times \begin{pmatrix} D \\ K \end{pmatrix}\right) = O\left(D^K\right), \tag{150}$$

where $\text{Catalan}(n)$ is the Catalan number of n.

Therefore, the complexity analysis presented by Hu et al. [2019] is also incorrect. Both permutation of labels and the symmetrical representation of the tree can generate different decision trees. This is a unique characteristic of the decision tree problem with binary feature data.

III.3.11 Further discussion

III.3.11.1 Acceleration techniques

Combinatorial constraints In the study of tree-based models, it is common to incorporate constraints such as requiring the number of data points in each leaf to be greater than $N_{\min}$ to avoid overfitting. Despite its simplicity and effectiveness, classical decision tree algorithms optimized through heuristic

methods, such as CART or C4.5, struggle to incorporate such constraints. By contrast, adding more splitting hyperplanes (branch nodes) typically decreases the number of data points in each leaf. This satisfies the segment-closed property and thus can be effectively integrated into the `kpermsAlg` algebra. By incorporating these constraints, the filtering process reduces the number of configurations generated, thereby making the algorithms developed above, more efficient.

Hyperplanes lying on the convex hull Hyperplanes that lie on the convex hull of the dataset can be safely discarded. This is because, for such hyperplanes, at least one of the resulting leaves is empty, thereby contributing nothing to the prediction. Consequently, the partition of the dataset remains unchanged without this hyperplane.

This principle is not limited to hyperplanes lying on the convex hull of the entire dataset. When splitting each leaf of a partial tree, hyperplanes that lie on the convex hull of the data points within a leaf node can also be ignored, as they do not alter the partition of the data within that region.

Pessimistic upper bound and optimistic lower bound

Definition 38. *Fixed leaves.* Fixed leaves are defined as leaves for which no new branch nodes are added to their ancestors; rather, only new branch nodes are added to their subtrees, thereby splitting the decision regions determined by these fixed leaves into smaller regions.

For instance, when adding a hyperplane h_1 **before** another hyperplane h_2, but after hyperplane h_3, this new tree is determined by $[h_3, h_1, h_2]$. The decision region of h_2 will be modified based on the decision region of h_1, whereas the decision region determined by h_3 will stay unchanged but is split into smaller regions by h_2 and h_1. Here, h_3 is referred to as a fixed leaf and h_2 is considered an *unfixed* leaf.

The determination of the pessimistic upper bound and optimistic lower bound for a partial tree is based on the following facts.

Theorem 21. Adding more hyperplanes to the decision regions defined by the fixed leaves can only decrease the 0-1 loss of this region.

Proof. When adding a new decision hyperplane (branch node) to one of the leaves of a partial tree, there are two possible scenarios. Denote the dataset in a leaf as M, with M^+ correctly classified and M^- misclassified data points, where $|M^+| \geq |M^-|$ by definition. Adding a new hyperplane results in two smaller leaves, M_1 and M_2. There are two cases to consider. If the prediction class in both new leaves remains unchanged after adding the hyperplane, then the misclassified data points are distributed between M_1 and M_2. Then $M_1^- \cup M_2^- = M^-$. Therefore, the 0-1 loss for these two new leaves is $\left|M_1^-\right| + \left|M_2^-\right| = |M^-|$.On the other hand, if the prediction in either of the new leaves changes, denote these leaves as M_1' and M_2'. Assume the prediction in M_2' has changed. According to the definition, $\left|M_2'^-\right| \leq \left|M_2^-\right|$ because the label assigned is the majority class in this region. Thus the $\left|M_1^-\right| + \left|M_2'^-\right| \leq |M^-|$. A similar result holds if the prediction of M_1 changes or if predictions in both M_1 and M_2. □

Given a partial tree with $K - i$ fixed leaves, the pessimistic upper bound can be derived by assuming that the 0-1 loss remains the same after adding new hyperplanes to the current fixed leaves. Conversely, the optimistic lower bound can be obtained by assuming that the decision regions of the i leaves can be

perfectly classified (i.e., zero 0-1 loss). Therefore, if the objective value of a tree configuration is worse than the global upper bound or the optimistic lower bound of a partial tree, then this partial tree can be safely discarded without further extension.

III.4 The K-clustering problem

III.4.1 Related studies

Clustering in machine learning, is the grouping of similar objects; clustering of a set of data points is a partition of the set elements chosen to minimize some measure of dissimilarity. There are various kinds of measures of dissimilarity, called "distance." There are some frequently used distance metrics, such as L_p norm, the special case for $p = 1$ is known as the *taxicab* distance or *Manhattan* distance and $p = 2$ the *Euclidean* distance. These two distances relate to the well-known *K-medians problem* and *K-means problem*. A tractable and exact algorithm for the K-clustering problem will have a huge impact on many fields. Unfortunately, the K-clustering problem is well known to be NP-hard for all dimensions $D \geq 2$ [Aloise et al., 2009, Mahajan et al., 2012].

Numerous studies have been conducted to obtain exact solutions for the K-means problem. For instance, Du Merle et al. [1999] developed an algorithm that combines an interior point method with branch-and-bound. Interestingly, the computation time of their algorithm tends to decrease rather than increase with the number of clusters. Diehr [1985] proposed a branch-and-bound algorithm as well, but its performance degrades significantly as the number of clusters increases and the separation between clusters diminishes. Cutting-plane algorithms have also been employed to solve the K-means problem, as seen in the works of Grötschel and Wakabayashi [1989] and Peng and Xia [2005]. Peng and Xia [2005]'s cutting-plane algorithm is a refined version of Tuy [1964]'s cutting-plane algorithm.

A relaxed version of the K-clustering problem involves constraining the cluster centers to be chosen only from the input data itself; this variant is known as the *K-medoids problem*. Unlike the K-means problem, which allows centroids to be at any point in the feature space, the K-medoids problem restricts centroids (medoids) to be selected exclusively from the actual data points. This approach allows for the precomputation of pairwise distances between data items, thereby eliminating the need to compute distances during the algorithm's execution. As a result, the K-medoids problem becomes a dimension-independent problem. Additionally, the K-medoids problem can accommodate arbitrary dissimilarity measures. Despite being a relaxed form of the K-clustering problem, the K-medoids problem remains NP-hard to optimize directly [Megiddo and Supowit, 1984].

The use of the BnB method dominates research on this problem [Ren et al., 2022, Elloumi, 2010, Christofides and Beasley, 1982, Ceselli and Righini, 2005]. An alternative approach is to use off-the-shelf *mixed-integer programming solvers* (MIP) such as Gurobi [Gurobi Optimization, 2021] or GLPK (GNU Linear Programming Kit) [Makhorin, 2008]. These solvers have made significant achievements, for instance, Elloumi [2010], Ceselli and Righini [2005]'s BnB algorithm is capable of processing medium-scale datasets with a very large number of medoids. More recently, Ren et al. [2022] designed another BnB algorithm capable of delivering tight **approximate** solutions—with an optimal gap of less than 0.1%—on very large-scale datasets, comprising over one million data points with three medoids, although this required a massively parallel computation over 6,000 CPU cores.

III.4.2 Problem specification

Equation (105), provides definitions of the *K-means problem* for both continuous variable $\overrightarrow{\boldsymbol{\mu}}$ and combinatorial variable s. The definition of the K-clustering problem is almost the same, with the only difference being that the distance function used in the K-clustering problem is not restricted to the Euclidean distance, but can be any distance function.

By fixing a data set $\mathcal{D}$ and fixing the cluster number to K, the objective function of the K-clustering problem over continuous variables can be defined as

$$E_{\text{kcluster}}\left(\overrightarrow{\boldsymbol{\mu}}\right) = \sum_{\boldsymbol{\mu}_k \in \mathcal{U}} \sum_{\boldsymbol{x}_n \in C_k} d\left(\boldsymbol{x}_n, \boldsymbol{\mu}_k\right) \quad . \tag{151}$$

Then the *K-clustering problem* requires finding an optimal centroid vector $\overrightarrow{\boldsymbol{\mu}} \in \mathbb{R}^{DK}$ which minimizes the K-clustering objective function,

$$\overrightarrow{\boldsymbol{\mu}}^* = \underset{\overrightarrow{\boldsymbol{\mu}} \in \mathbb{R}^{DK}}{\operatorname{argmin}} \; E_{\text{kcluster}}\left(\overrightarrow{\boldsymbol{\mu}}\right). \tag{152}$$

Similarly, the K-clustering problem defined over the combinatorial variable $s = (\alpha_1, \alpha_2, \ldots, \alpha_N) \in \mathcal{S}_{\text{kasgns}}$ is,

$$s^* = \underset{s \in \mathcal{S}_{\text{kasgns}}}{\operatorname{argmin}} \; E_{\text{kcluster}}\left(s\right) = \sum_{k \in \mathcal{K}} \sum_{n \in \mathcal{N}} \mathbf{1}\left[s_n = k\right] d\left(\boldsymbol{x}_n, \boldsymbol{\mu}_k\right)^2 \quad , \tag{153}$$

where function $\mathbf{1}\left[\,\right]$ returns 1 if the Boolean argument $s_n = k$ is true, and 0 if false.

As mentioned earlier, both the K-clustering and the K-medoids problems attempt to minimize the sum of the within-cluster distances for arbitrary distance functions. By contrast to the K-clustering problem, K-medoids chooses some data points in the dataset $\mathcal{D}$ as the centroids (medoids). In other words, $\mathcal{U} \subseteq \mathcal{D}$ for the K-medoids problem. Thus, the K-medoids problem is given as,

$$\begin{aligned} \mathcal{U}^* = \underset{\mathcal{U}}{\operatorname{argmin}} \; & E_{\text{kmedoids}}\left(\mathcal{U}\right) \\ \text{s.t. } & \mathcal{U} \subseteq \mathcal{D}, |\mathcal{U}| = K, \end{aligned} \tag{154}$$

where $E_{\text{kmedoids}}\left(\mathcal{U}\right) = \sum_{k \in \mathcal{K}} \sum_{x_n \in C_k} d\left(\boldsymbol{x}_n, \boldsymbol{\mu}_k\right)$ is the objective function for the K-medoids problem, and $\mathcal{U}^*$ is a set of centroids that optimize the objective function $E\left(\mathcal{U}\right)$.

The constraints of the K-medoids problem enforce the continuous variable $\overrightarrow{\boldsymbol{\mu}} \in \mathbb{R}^{DK}$ to become a combinatorial variable $\mathcal{U} \subseteq \mathcal{D}$, making the K-medoids problem *dimension-independent*. In this case, the distance between each data item and the medoids can be precomputed by calculating the pairwise distances between data items, which requires only $O\left(D \times N^2\right)$ time. By contrast, in the K-clustering problem, the centroids can lie anywhere in $\mathbb{R}^D$, and each distinct set of centroids results in a unique objective value. From the perspective of the combinatorial variable s, each distinct s in $\mathcal{S}_{\text{kasgns}}$ produces a different set of centroids. Since the size of $\mathcal{S}_{\text{kasgns}}$ is $O\left(K^N\right)$, it is impractical to precompute all possible centroids and their distances to each data item for large-scale data.

III.4.3 The essential combinatorial properties of K-clustering problems

K-medoids problem The choice of centroids in the K-clustering problem is isomorphic to $\mathbb{R}^{D \times K}$, which is infinitely large. However, by constraining the K-clustering problem, the choice of centroids is finite.

The search space in the K-medoids problem is much more restricted than in the general K-clustering problem, by the MIP specification of the K-medoids problem, $C(\boldsymbol{z}_c)$ constraint the choice of centroids to be distinct and chosen from the data. Following the exhaustive search paradigm that mentioned before, the obvious strategy for solving the K-medoids problem is to enumerate all possible centroids $\boldsymbol{z}_c = \{\boldsymbol{\mu}_k\}, \forall k \in \mathcal{K}$, wherein lies at least one set of such centroids determining an assignment which is optimal. It follows that there are only $\frac{N\times(N-1)\times\cdots\times(N-K+1)}{K!} = \begin{pmatrix} N \\ K \end{pmatrix}$ ways of selecting centroids whose corresponding assignments are potentially distinct. In other words, the K-medoids problem can be solved exactly by selecting the best K-combinations of data points as the centroids.

K-means problem Following our discussion in Section II.3.4, Lemma 15 demonstrate that the optimal solution to the K-means clustering problem must be a Voronoi partition. At the same time, Thm. 14 shows that all possible Voronoi partitions for the K-means problem is essentially the Cartesian product of k-combinations subset and length k binary assignments, for all $1 \leq k \leq K + (K-1)D - 1$. Thus solving the K-means problem exactly requires exhaustively enumerating all possible Cartesian products of k-combinations and length k binary assignments, and hence the size of the search space of the K-means problem $\mathcal{S}_{\text{kmean}}$ has a complexity of

$$|\mathcal{S}_{\text{kmeans}}| = \sum_{d=1}^{K+(K-1)D-1} 2^d \begin{pmatrix} (K-1)N \\ d \end{pmatrix} = O\left(N^{K+(K-1)D-1}\right). \tag{155}$$

2-means problem In the special case of $K = 2$, it has been proved earlier that the 2-means clustering problem is equivalent to the linear classification problem in Section II.3.4. Hence the 2-means clustering problem can be solved exhaustively by enumerating all possible cells of the dual arrangement $\phi(\mathcal{D})$. Thus the combinatorial search space of the 2-means clustering problem has a size of $\sum_{d=0}^{D} \begin{pmatrix} N \\ d \end{pmatrix} = O\left(N^D\right)$.

III.4.4 Further discussion

We present a relatively short chapter on the K-clustering problem because our contributions to these problems lie primarily in the geometric aspects, which have been thoroughly discussed in Section (III.3.4). It is disappointing that we did not make any algorithmic contributions to these problems. This is because, after being simplified using geometry, the combinatorics for these clustering problems consist of the simplest combinatorial objects (all combinations), leaving little room for applying algorithmic techniques to speed up the process. Additionally, due to time constraints, we were unable to explore acceleration techniques for these algorithms. Nevertheless, even for the simplest brute-force solution, the resulting algorithms for solving these problems remain the fastest in terms of worst-case complexity, as BnB algorithms for solving these problems exhibit exponential complexity. We will further explore the K-medoids algorithm in Section IV.2 of the final part, where we will show that even the most naive brute-force algorithm, implemented using the `kcombs` generator, is more efficient than the state-of-the-art BnB algorithm [Ren et al., 2022] on every dataset we tested. Moreover, we will also demonstrate that Ren et al. [2022]'s algorithm exhibits exponential complexity in the worst case.

Time-space complexity trade-off in designing exact algorithms

In all the previous discussions, the focus has been primarily on the *time efficiency* of the algorithms, as the central concern is designing combinatorial optimization and generator algorithms. However, as noted many times before, achieving superior time efficiency often comes at the cost of increased memory usage. This trade-off is a critical consideration in algorithm design, particularly in large-scale problems where both time and memory resources are limited. To address this trade-off effectively, it is necessary to navigate the balance between time and space complexities, using specific techniques that mitigate excessive memory usage while maintaining reasonable execution times.

The purpose of the discussion here is to provide a brief guide on the operational choices one can make when navigating the trade-offs, rather than offering a comprehensive analysis of how each operational choice affects the performance of the algorithms. We explore three key aspects that can help guide the management of these trade-offs.

Selection of generators The algorithm design framework of this thesis is based on deriving an efficient algorithm from an initially inefficient exhaustive search specification. It is a characteristic of many successful theories, in mathematics as well as in natural science, that they can be presented in several apparently independent ways, which are in a useful sense provably equivalent [Hoare, 1997]. Different definitions can be safely and consistently used at different times and for different purposes.

In this context, different generators for generating the same combinatorial structures will serve as definitions for different purposes. Chapter II.1 of Part II introduces four classes of combinatorial generators. Below, the advantages and limitations of each class of generator in the context of combinatorial optimization, is summarized.

First, the **lexicographical generation** method is inefficient for exhaustive generation and is non-recursive. As a result, it does not benefit from acceleration techniques introduced earlier, such as fusion or dominance relations. Since configurations are generated one-by-one, this generator has the advantage of being embarrassingly parallelizable and consumes only $O(1)$ space during run-time.

Second, **sequential decision processes**, which are the central focus of this thesis, are particularly well-suited for most combinatorial optimization tasks. These generators are both efficient and flexible. Their *efficiency* is demonstrated by their low *amortized* time complexity, embarrassingly parallelizable nature, and ability to incorporate various acceleration techniques, such as fusion and thinning. Their *flexibility* lies in two key properties. Firstly, backtracking can be integrated, allowing the use of different search strategies for different tasks. Second, the principles for designing more complex SDP generators are directly applicable, as discussed in Subsection II.2.3.4.

Combinatorial Gray code generation, as noted, can be considered a subclass of SDP generation. However, when analyzed independently, it possesses nearly all the advantages of SDP generators but lacks embarrassingly parallelizability and the ability to incorporate backtracking. This is because the ordering of configurations already been fixed, and introducing either technique would alter the intrinsic ordering between configurations.

Lastly, the **integer sequential decision process** lies (qualitatively speaking) between SDP generation and Gray code generation. It may be more efficient than classical SDP generators for generating

Methods	Efficiency	Memory Usage	Parallelizability
Catamorphism with backtracking	Better best-case complexity	Much Less memory usage	Requires communication
Catamorphism without backtracking	Better worst-case complexity	More memory usage	No communication

Table 3: Qualitative efficiency comparison between the catamorphism with backtracking technique and ordinary catamorphism over join-list datatype.

all configurations of the same combinatorial structure, as manipulating integers is typically faster than handling combinatorial configurations, which are usually stored in lists. However, when applied to combinatorial optimization, this generator may require running an unranking function for each subconfiguration to evaluate their objective values. This additional step can result in a slower algorithm compared to SDP generators.

Evaluation in partial fusable generators This trade-off is primarily observed in generators that are only *partially fusable* with the evaluator. Partially fusable generators, are those generators where a *non-prefix-closed* predicate is relaxed to become *prefix-closed* in order to enable fusion within the generator. This often results in situations, encountered frequently in this thesis, where an incomplete configuration (i.e. partial configurations that satisfy the relaxed predicate but not the final predicate) cannot be evaluated.

In such cases, there is often a choice between evaluation strategies. The first option is to *evaluate the complete configuration* directly during generation. By evaluating the objective of a complete configuration while generating, it is only necessary to store the best configuration encountered at each recursive step. This allows for the fusion of the selector into the generator, although this fusion is only partially applicable. Since the predicate has been relaxed, there is limited information to justify the optimality of these incomplete configurations. Moreover, this strategy is well-suited to vectorized implementations and thus also appropriate for parallel implementation

Alternatively, configurations can be evaluated *incrementally* during the recursion. This approach has been demonstrated earlier to lead to significant speed-up, particularly in solving the 0-1 loss linear classification problem [He and Little, 2023a]. However, this method requires more memory than the previous approaches and is more difficult to implement in parallel.

Search strategies Search strategies are discussed in detail in Subsection II.2.3.4, and are summarized in Table 3. In brief, the primary advantage of using backtracking techniques is the potential for significantly reduced memory usage.

However, with backtracking, communication between processors becomes necessary, as some processors may need to wait for others to finish due to dependencies introduced by the backtracking process. By contrast, the classical catamorphism approach, which does not use backtracking and is commonly referred to as a breadth-first search strategy in BnB studies, requires no inter-processor communication and is much easier to implement on GPUs.

Part IV

End-to-end implementation in Haskell

This final part of the thesis presents two Haskell implementations of the algorithms for solving the *0-1 loss linear classification problem* and the *K-medoids problem*. The definitions of these two problems have been discussed in detail in Chapter III.1 and Chapter III.4 in Part III.

From the discussion in Part III, both exact algorithms are the *first* polynomial-time algorithms for solving their respective problems and are *embarrassingly parallelizable*. In this section, it is demonstrated empirically that these polynomial-time complexity predictions hold true. Moreover, it is shown here that the state-of-the-art MIP solver (GLPK) and branch-and-bound (BnB) algorithms exhibit exponential asymptotic complexity in the worst case. For the classification problem, the empirical analysis on the widely-used UCI benchmark datasets [Dua and Graff, 2019] shows that the novel exact algorithm of this thesis consistently achieves the best 0-1 loss among other algorithms. Similarly, the novel EKM algorithm always obtains the best objective value compared with existing approximate algorithms on both UCI and synthetic datasets.

IV.1 Exact 0-1 loss linear classification algorithm

In previous research, [He and Little, 2023b], the author and collaborates presented an *end-to-end implementation*—the complete process of designing, developing, testing, and deploying the algorithm in a manner that covers all the steps from the initial input to the final output—of an exact 0-1 loss linear classification algorithm, E01-ICE, short for "exact 0-1 loss incremental cell enumeration algorithm," which was based on the 0-1 loss linear classification theorem 15. This algorithm was constructed using a catamorphism over the snoc-list datatype through the `foldl` operator. Although this algorithm is referred to as a "cell enumeration algorithm," it more accurately enumerates only the "vertices" of the dual arrangement, $\mathcal{H}_{\mathcal{D}}$. It is referred to as a cell enumeration algorithm to emphasize the fact that it implicitly enumerates all possible "cells", $\mathcal{H}_{\mathcal{D}}$. This chapter revisits this problem and demonstrates an end-to-end implementation of the same algorithm, but now based on the join-list datatype.

IV.1.1 An efficient combination-sequence generator

According to Thm. 15, the 0-1 loss linear classification problem can be solved exhaustively by enumerating all possible size-D sublists (combinations) of data items. To evaluate the objective value of each hyperplane, each size-D sublist must be paired with the data sequence. This implies that the configuration for solving the problem is the Cartesian product of sequences and combinations. A *D-sublist-sequence generator* will enumerate all such pairs of combinatorial configurations, and by evaluating the 0-1 loss over these configurations, it is guaranteed to test every possible assignment, ensuring that an optimal solution is eventually found.

Chapter II.2 Section II.2.3 defined the following algebras based on the join-list datatype for enumerating size-D sublists and sequences

```
dsubsAlg :: Int -> ListFj a [[a]] -> [[a]]
dsubsAlg d Nil = [[]]
dsubsAlg d (Single a) = [[],[a]]
dsubsAlg d (Join x y) = filter (maxlen k)(crj x y)
 where maxlen d x = (length x) <= d

seqnAlg :: ListFj a [[a]] -> [[a]]
seqnAlg Nil = [[]]
seqnAlg (Single a) = [[a]]
```

where the size-D *sublists generator* is obtained by incorporating the segment-closed predicate `maxlen` into the ordinary sublists generator.

According to the Thm. 3, the *Cartesian product* of D-sublists and sequence can be constructed easily by applying the *Cartesian product fusion algebra* `cpalg`. Thus the Cartesian product of the D-sublists and the sequence generator can be defined as

```
dsubsseqnAlg d = cpalg (ksubsAlg d) seqnAlg
dsubsseqn d = cata (dsubsseqnAlg d)
```

Evaluating `dcombsseqn 2 [1,2,3]` gives the Cartesian product of sublists with sizes smaller than or equal to two and input sequence

```
[([],[1,2,3]),([3],[1,2,3]),([2],[1,2,3]),([2,3],[1,2,3]),([1],[1,2,3]),([1,3],[1,2,3]),([1,2],[1,
```

Alternatively, the generator `dcombsseqn` can be defined explicitly as the following join-list algebra

```
dsubsseqnAlg :: Int -> ListFj a [([a],[a])] -> [([a],[a])]
dsubsseqnAlg d Nil = [([],[])]
dsubsseqnAlg d (Single a) = [([],[a]),([a],[a])]
dsubsseqnAlg d (Join x y) = filter (maxlenfst d) (crp merge x y)
 where maxlenfst k x = (length (fst x)) <= k

merge :: ([a],[a]) -> ([a],[a]) -> ([a],[a])
merge x1 x2 = ((fst x1) ++ (fst x2), (snd x1) ++ (snd x2))

dsubsseqn' d = cata (dsubsseqnAlg d)
```

Evaluating `dcombsseqn' = cata dcombsseqnAlg` will return the same result given by `dcombsseqn`.

IV.1.2 Exhaustive, incremental cell enumeration based on join-lists

Now, all the ingredients to construct the algorithm, which will enumerate all these linear classification decision hyperplanes and thus solve (123), are in place. Some basic linear algebra will be needed, such as real-valued `Vector` and `Matrix` types, solving linear systems `linearsolve :: Matrix -> Vector -> Vector` and matrix-vector multiplication `matvecmult:: Matrix -> Vector -> Vector` which are defined in the imported `Linearsolve` module and listed in the Appendix A.

Dataset First, the input `Dataset` is defined

```
type Label = Int
type Item = (Vector, Label)
type Dataset = [Item]
```

which is a set of data `Items` which comprise a tuple of a real-valued `Vector` data point and its associated integer training `Label`, for clarity extracted from the tuple using

```
label (x,l) = l
point (x,l) = x
```

Linear classification A linear model is the unique hyperplane parameter of type `Vector` which goes through a given set of data points, where the number of data points is equal to the dimension of the space

```
ones:: Int -> Vector
ones n = take n [1.0,1.0..]

fitw :: Double -> [Vector] -> Vector
```

```
fitw sense dx = [-sense] ++ (map (*sense) (linearsolve dx (ones (length
   (head dx)))))
```

Here, `dx` is a list of vectors of length D, in other words a $D \times D$ matrix, and the function `fitw` solves a linear system of equations to obtain the normal vector of the hyperplane in the homogeneous coordinates for all data in `dx`. The `sense` parameter, taking on the values $\{-1.0, +1.0\}$, is used to select the orientation of the normal vector. The function `head :: [a] -> a` extracts the first element of a list (which must be non-empty); here it is used to find the dimension D of the dataset.

With an (oriented) linear model obtained this way, it can be applied to a set of data points in order to make a decision function prediction

```
evalw :: [Vector] -> Vector -> [Double]
evalw dx w = matvecmult (map ([1.0]++) dx) w
```

which is the oriented distance of all data items in `dx` to the linear model with normal vector `w`. Given that prediction function value, the corresponding predicted assignment in $\{0, 1, -1\}$ can be obtained (which is zero for points which lie on the decision boundary and which actually define the boundary):

```
plabel :: [Double] -> [Label]
plabel = map (round.signum.underflow)
 where
  smalleps = 1e-8
  underflow v = if (abs v) < smalleps then 0 else v
```

The Haskell `where` keyword is a notational convenience which allows local function and variable definitions that can access the enclosing, less indented, scope. The reason for the `underflow` correction, is that numerical imprecision leads to predictions for some points which are not exactly on the boundary, where they should be. The function `round` type casts the label prediction to match the label type (integer). Lastly, combining these two functions above obtains

```
pclass :: [Vector] -> Vector -> [Label]
pclass dx w = plabel (evalw dx w)
```

which, given a set of data points and a hyperplane, obtains the associated labels with respect to that hyperplane.

Loss Next, given a pair of labels, it is necessary to compute the corresponding term in the 0-1 loss. This makes use of Haskell guard equations

```
loss01 :: Label -> Label -> Int
loss01 l1 l2
  | l1 == 0 = 0
  | l2 == 0 = 0
  | l1 /= l2 = 1
  | otherwise = 0
```

This function handles the situation where either label is 0, which occurs for data points which lie on the defining hyperplane and whose predicted class is always assumed to be the same as training label, and also the default case (`otherwise`) to ensure that `loss01` is total. Using this, the 0-1 loss, $E_{0\text{-}1}$ can be computed for a given pair of label lists

```
e01 :: [Label] -> [Label] -> Integer
e01 x y = sum (map (\(lx,ly) -> loss01 lx ly) (zip x y))
```

making use of the Haskell function `zip :: [a] -> [b] -> [(a,b)]` which pairs every element of the first given list with the corresponding element of the second given list.

Configuration The recursive combination-sequence SDP given earlier, requires a partial configuration datatype which is updated by application of the decisions. For computational efficiency, the linear classification hyperplane defined by the size-D combination is packaged up with the 0-1 loss for its corresponding sequence, which together define the type (classification) `Model`

```
type Model = (Vector, Int)

modelw :: Model -> Vector
modelw (w,l) = w

modell :: Model -> Int
modell (w,l) = l
```

and, combining this with the combination-sequence datatype gives us the SDP configuration `Config`:

The first element in this tuple is the combination of data items (with maximum size D) that is used to construct a linear model, the second element is a sequence of data items which have been encountered so far in the recursion. Note that here, the value for `Model` in the configuration is *optional*. This is indicated by the use of Haskell's `Maybe` datatype, the initial configuration has empty combination-sequence pairs, and a `Nothing`-valued model

```
empty :: Config
empty = Cnfg { comb = [], seqn = [], model = Nothing }
```

The reason for the model pair being optional should be obvious: for combinations of insufficient size, it is not possible to compute a model or corresponding 0-1 loss.

Algorithms The exposition is now in a position to give the main recursion `e01gen`, which is defined by a join-list algebra `iceAlg`

```
mergecnfg :: Double -> Int -> Config -> Config -> Config
mergecnfg sense dim c1 c2 = (updcomb ,updseqn , updloss)
  where
   updcomb = (comb c1) ++ (comb c2)
   updseqn = (seqn c1) ++ (seqn c2)
```

```
    updloss = if (length updcomb == dim) then Just (w, e01 (map label
       updseqn) (pclass (map point updseqn) w)) else Nothing
     where w = fitw sense (map point updcomb)

iceAlg :: Double -> Int -> Int -> ListAlg Item [Config]
iceAlg sense ub dim = alg
  where
   alg Nil = [empty]
   alg (Single a) = [([],[a], Nothing),([a],[a], Nothing)]
   alg (Join x y) = filter (retain) (crp (mergecnfg sense dim) x y)
    where
     feasible dim = (<= dim) . length . comb
     viable ub c = case (model c) of
       Nothing -> True
       Just (w,l) -> (l <= ub)
     retain c = (feasible dim c) && (viable ub c)

e01gen :: Int -> Int -> Dataset -> [Config]
e01gen ub dim xs = (cata (iceAlg 1 ub dim) xs) ++ (cata (iceAlg (-1) ub
   dim) xs)
```

Given an orientation parameter `sense :: Double`, an approximate upper bound on the 0-1 loss `ub :: Integer`, and a (non-empty) dataset `xs :: Dataset`, `e01gen` outputs a list of candidate solutions to the of type `[Config]` which are potential globally optimal solutions to (123), with 0-1 loss no worse than `ub`. This efficient vertices generator is derived using all the same principles as the D-sublist-sequence generator introduced above, but additionally includes updates to the configurations of type `Config`, and the evaluation of the objective value for each configuration. The name `iceAlg` stands for "incremental vertex enumeration algebra", since it enumerates the vertices of the dual arrangement, $\mathcal{H}_\mathcal{D}$.

In the pattern defined by the `Join` constructor, all partial configurations `c1` in `x` and partial configurations `c2` in `y`, are merged. In this merge operation, `updcomb` and `updseqn` are obtained by joining the combination and sequence in `c1` with `c2` respectively. Furthermore, a new linear model for the configuration (using `fitw`) is computed, when its combination reaches size D for the first time. Thus, the `Maybe` value of the model in the configuration, undergoes a one-way *state transition* from undefined (`Nothing`) to computed (`Just m`); when computed, the linear boundary hyperplane remains unchanged and the configuration's 0-1 loss is updated on each subsequent recursion step.

Looking at the filtering, the predicate `retain` is the conjunction of two separate predicates `feasible` and `viable`. The first predicate, `feasible`, checks whether the size of the combination is smaller than `dim`, which is the same as the `maxlen` predicate introduced before. The predicate `viable :: Config -> Bool` checks whether a linear hyperplane model is defined for a configuration, and if so (case `Just m`), returns `True` when the 0-1 loss of this configuration is at most equal to the approximate upper bound. Both predicates are *segment-closed*, the `viable` predicate is segment-closed because the 0-1 loss is non-decreasing as more data is scanned by the recursion; this is a very useful computational efficiency improvement

since it can eliminate many non-optimal partial solutions. Since the conjunction of two segment-closed predicates is also segment-closed, the integration `retain :: Config -> Bool` of these two predicates is also segment-closed.

Having generated partial solutions, the next stage is to select an optimal one. This involves a straightforward recursive iteration through a non-empty list of partial configurations, comparing adjacent configurations remaining in the list using the fold operator for non-empty lists `foldl1 :: (a -> a -> a) -> [a] -> a`. The best of the pair, that is, one with 0-1 loss at most as large as the other, is selected, using function `best :: Config -> Config -> Config`. At the same time, configurations with `Nothing`-valued (undefined) models are simultaneously removed in the same iteration. This leads to the following `sel01opt` function, which selects the configuration with the minimal 0-1 loss:

```
sel01opt :: [Config] -> Config
sel01opt = foldl1 best
 where
  best c1 c2 = case (model c1) of
   Nothing -> c2
   Just (w1,l1) -> case (model c2) of
    Nothing -> c1
    Just (w2,l2) -> if (l1 <= l2) then c1 else c2
```

Finally, the program for solving problem (123) can be given. It generates all positive and negatively-oriented decision boundaries (which are viable with respect to the approximate upper bound `ub`) and selects an optimal one:

```
ice_join ub dim  = sel01opt . (e01gen ub dim)
```

An approximate upper bound may be computed by any reasonably good approximate method, for instance, the support vector machine (SVM). The tighter this bound, the more partial solutions are removed during iteration of `e01gen` which is desirable in order to achieve practical computational performance.

Symmetry fusion In the previous discussion of the linear classification problem, the symmetry fusion Thm. 16 states that the 0-1 loss for the negative orientation of a hyperplane can be calculated from the positive orientation of the same hyperplane. Therefore, the 0-1 loss linear classification problem can be solved by enumerating only the positive or negative-oriented hyperplanes, rather than both.

As a result, half of the computations can be saved by modifying the `iceAlg` as

```
iceAlg' :: Int -> Int -> Int -> ListAlg Item [Config]
iceAlg' n ub dim = alg
  where
   alg Nil = [empty]
   alg (Single a) = [([],[a], Nothing),([a],[a], Nothing)]
   alg (Join x y) = filter (retain) (crp (mergecnfg' dim) x y)
    where
     feasible dim = (<= dim) . length . comb
     viable ub c = case (model c) of
```

```
        Nothing -> True
        Just (w,l) -> (l <= ub) || (l >= n - dim - ub)
      retain c = (feasible dim c) && (viable ub c)

mergecnfg' ::  Int -> Config -> Config -> Config
mergecnfg' dim c1 c2 = (updcomb ,updseqn , updloss)
  where
   updcomb = (comb c1) ++ (comb c2)
   updseqn = (seqn c1) ++ (seqn c2)
   updloss = if (length updcomb == dim) then Just (w, e01 (map label
      updseqn) (pclass (map point updseqn) w)) else Nothing
    where w = fitw (1) (map point updcomb)
```

The above program discards the use of the `sense` parameter to generate hyperplanes of negative orientation. Since both positive and negative orientations correspond to the same hyperplane, and the 0-1 loss of the negative oriented hyperplane can be obtained by calculating `negl = n - dim -l`, where `n` is the data size and `l` is the 0-1 loss of the positive hyperplane. Similarly, if `negl` is greater than the upper bound `ub`, it can be discarded. Thus, the predicate for testing the negative hyperplane can be defined as `negl <= ub` which is equivalent to `n - dim - ub <= l`.

Therefore, the new generator for the 0-1 loss linear classification problem can be defined as

```
e01gen' ub dim xs = cata (iceAlg' (length xs) ub dim) xs
```

Similarly, the selector should also be modified as

```
sel01opt' :: Int -> [Config] -> Config
sel01opt' dim = foldl1 best
 where
  best c1 c2 = case (model c1) of
   Nothing -> c2
   Just (w1,l1) -> case (model c2) of
    Nothing -> c1
    Just (w2,l2) -> if (l1 <= l2) && (l1 <= n - dim - l2) then c1 else c2
```

Finally, the 0-1 loss linear classification problem can be solved efficiently by running

```
ice_join' ub dim = sel01opt' dim . (e01gen' ub dim)
```

IV.1.3 Empirical analysis

In this section, the computational performance of the novel ICE algorithm given above, is estimated on both synthetic and real-world data sets. The evaluation aims to test the following predictions: (a) the ICE algorithm always obtains the best 0-1 loss (classification error) among other algorithms (hence obtains optimal prediction accuracy); (b) wall-clock run-time matches the worst-case time complexity analysis,

UCI dataset	N	D	Incremental cell enumeration (ICE) (ours)	Support vector machine (SVM)	Logistic regression (LR)	Linear discriminant analysis (LDA)
Habermans	306	3	**21.6% (66)**	24.8% (76)	23.9% (73)	25.2% (77)
Caesarian	80	5	**22.5% (18)**	27.5% (22)	27.5% (22)	27.5% (22)
Cryotherapy	90	6	**4.4% (4)**	8.9% (8)	**4.4% (4)**	10.0% (9)
Voicepath	704	2	**2.7% (19)**	3.3% (23)	3.4% (24)	3.4% (24)
Inflammations	120	6	**0.0% (0)**	**0.0% (0)**	**0.0% (0)**	**0.0% (0)**

Table 4: Empirical comparison of the classification error performance (smaller is better), $\hat{E}_{0\text{-}1}$, of the novel incremental cell enumeration (ICE) algorithm presented in this thesis, against approximate methods (linear kernel support vector machines, logistic regression, Fisher's linear discriminant) on real-world datasets from the UCI machine learning repository [Dua and Graff, 2019]. Misclassification rates are given as classification error percentage, $\hat{E}_{0\text{-}1}/N\%$, and number of classification errors, $\hat{E}_{0\text{-}1}$ (in brackets). Best performing algorithm is marked bold. As predicted, ICE, being exact, outperforms all other non-exact algorithms.

and (c) viability filtering using the approximate upper bound leads to polynomial decrease in wall-clock run-time.[24]

IV.1.3.1 Real-world data set classification performance

Various linear classification algorithms were applied to classification data sets from the UCI machine learning repository [Dua and Graff, 2019]. The exact algorithm ICE is compared against approximate algorithms: linear kernel support vector machines (SVM), logistic regression (LR) and linear discriminant analysis (LDA). As predicted (Table 4), the ICE algorithm always finds solutions with smaller 0-1 loss than approximate algorithms (except for the Inflammations data set which is linearly separable).

IV.1.3.2 Out-of-sample generalization tests

After having computed the in-sample accuracy, the out-of-sample prediction performance is tested using cross-validation (90% of data is used for training, 10% is used for testing) (see Table 5), where the out-of-sample predictions use the *maximum margin representative* of the equivalence class of the exact hyperplane [Vapnik, 1999]. The reason we choose this representative, as discussed thoroughly in Chapter 19 of Part III, is that the larger the margin, the higher the probability of obtaining a more accurate solution on test datasets.

IV.1.3.3 Run-time complexity analysis

In the worst case situation, $\texttt{ub} \geq N/2$, viability filtering with the approximate global upper bound will do nothing because every combinatorial configuration will be feasible (if a model's objective function value $E_{0-1} \geq N/2$, its negative part can be obtained by reversing the direction of the normal vector; the resulting model will have the 0-1 loss smaller than $N/2$, both models represented by the same hyperplane).

[24] For practical purposes, all our results are obtained using a direct, efficient C++ translation of the Haskell code given in this paper.

UCI dataset	ICE train (%)	ICE test (%)	SVM train (%)	SVM test (%)	LR train (%)	LR test (%)	LDA train (%)	LDA test (%)
Habermans	**21.5** (0.6)	**23.8** (6.8)	24.8 (0.6)	25.8 (5.7)	24.9 (1.2)	26.4 (6.8)	25.0 (1.0)	27.1 (6.0)
Caesarian	**16.4** (1.4)	**37.5** (16.8)	30.3 (4.9)	42.5 (13.9)	27.9 (3.7)	43.8 (15.1)	29.6 (2.8)	43.8 (15.1)
Cryotherapy	**4.4** (0.6)	**9.0** (10.5)	7.8 (1.7)	17.8 (10.1)	**4.4** (0.6)	**9.0** (9.2)	10 (1.7)	16.7 (9.0)
Voicepath	**2.7** (0.2)	**3.7** (1.4)	3.3 (0.2)	4.0 (1.7)	3.4 (0.2)	3.7 (1.3)	3.3 (0.3)	3.9 (1.1)
Inflammations	**0.0** (0.0)	**0.0** (0.0)	**0.0** (0.0)	**0.0** (0.0)	**0.0** (0.0)	**0.0** (0.0)	**0.0** (0.0)	**0.0** (0.0)

Table 5: Ten-fold cross-validation out-of-sample tests on UCI data set of the novel incremental cell enumeration (ICE) algorithm presented in this thesis, against approximate methods (linear kernel support vector machine, SVM; logistic regression, LR; linear discriminant analysis, LDA). Mean classification error (smaller is better) percentage $\hat{E}_{0\text{-}1}/N\%$ is given (standard deviation in brackets), for the training and test sets. Best performing algorithm is marked bold. As predicted, ICE always obtains better solutions on average on out-of-sample datasets than other, non-exact algorithms.

Therefore, all $O\left(N^D\right)$ configurations will be enumerated for all N dataset items. In each iteration, a configuration takes constant time to update its 0-1 loss, followed by $O\left(N\right)$ time required to calculate the complete 0-1 loss of a configuration. Hence, the ICE algorithm will have $O\left(N^{D+1}\right)$ in the worst case.

The wall clock run-time of the novel ICE algorithm is tested on four different synthetic data sets with dimension ranging from $1D$ to $4D$. The $1D$-dimensional data set has data size ranging from $N = 1000$ to 60000, the $2D$-dimensional ranges from 150 to 2400, $3D$-dimensional from 50 to 500, and $4D$-dimensional data ranging from 30 to 200. The worst-case predictions are well-matched empirically (see Fig. 29).

Viability filtering using the approximate global upper bound `ub` can substantially speed up the ICE algorithm. Next, the effectiveness of the upper bound (see Fig. 30) is evaluated. Five synthetic datasets with dimension ranging from $D = 1$ to $D = 4$ are generated, and `ub` is varied from $\hat{E}_{0\text{-}1}$ to N. The synthetic datasets are chosen such that they all have optimal 0-1 loss $E^*_{0\text{-}1}$ approximately equal to $0.1N$ and $0.2N$. Fig. 30 shows polynomial degree decrease in run time as `ub` is decreased from $N/2$ to $E^*_{0\text{-}1}$, and it remains stable when $\texttt{ub} \geq N/2$ because then all configurations are viable.

All decision boundaries computed by exact algorithms entail the same, globally optimal 0-1 loss. Therefore, the only meaningful comparison between ICE and any other exact algorithms is in terms of run-time complexity. Here, the wall-clock run time of the ICE algorithm is tested against the exact branch-and-bound algorithm of Nguyen and Sanner [2013]. As a branch-and-bound algorithm, in the worst case it must test all possible assignments of data points to labels which requires an exponential number of computations, by comparison to ICE's worst case polynomial time complexity arising from the enumeration of dichotomies instead. Empirical computations confirm this reasoning (see Fig. 31), predicting for instance that for the $N = 150$ data size with $D = 3$, ICE would take 1.2 seconds worst-

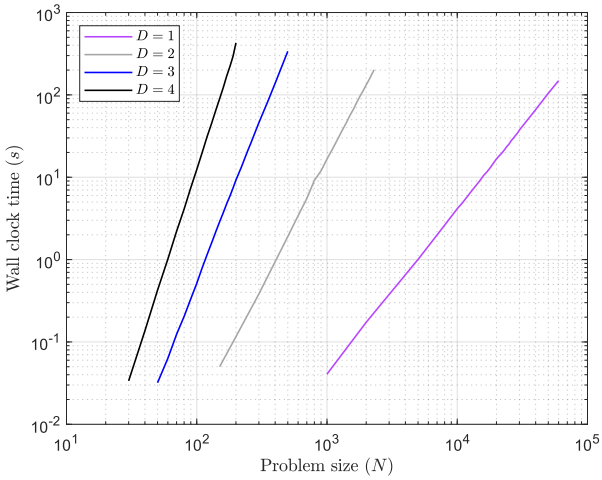


Figure 29: Log-log wall-clock run time (seconds) for the ICE algorithm in $1D$ to $4D$ synthetic datasets, against dataset size N, where the approximate upper bound is disabled (by setting it to N). The run-time curves from left to right (corresponding to $D = 1, 2, 3, 4$ respectively), have slopes 2.0, 3.1, 4.1, and 4.9, a very good match to the predicted worst-case run-time complexity of $O\left(N^2\right)$, $O\left(N^3\right)$, $O\left(N^4\right)$, and $O\left(N^5\right)$ respectively.

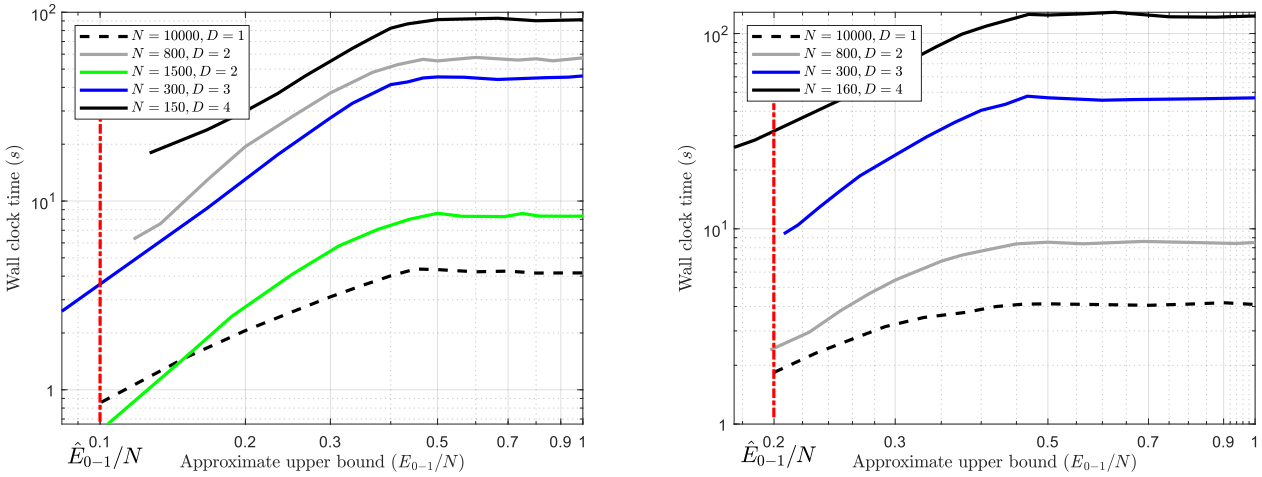


Figure 30: Log-log wall-clock run time (seconds) of the ICE algorithm on synthetic data, as the approximate upper bound viability is varied, $E^*_{0\text{-}1} \leq \texttt{ub} \leq N$, for $E^*_{0\text{-}1}$ approximately $0.1N$ (left), and approximately $0.2N$ (right). It can be seen that the empirical run-time decreases polynomially as `ub` tends towards the exact $E^*_{0\text{-}1}$ of the dataset.

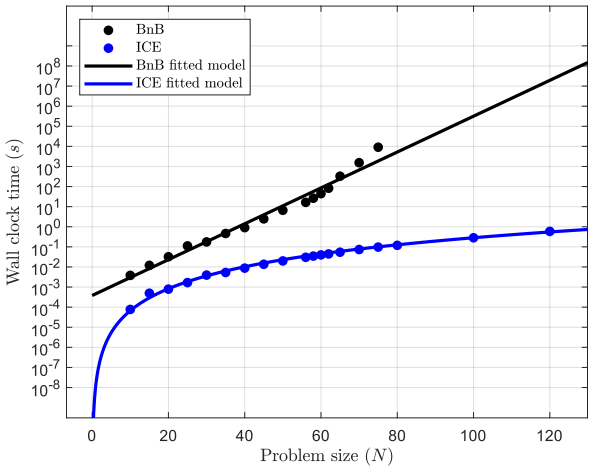


Figure 31: Log-linear wall-clock run time (seconds) plot comparing the ICE algorithm against the branch-and-bound (BnB) algorithm of Nguyen and Sanner [2013] (MATLAB implementation provided by the authors) on three dimensional synthetic data. On this log-linear scale exponential run time appears as a linear function of problem size N, whereas, polynomial run time is a logarithmic function of N. Fitting appropriate models (lines) to the computational experiment data (dots) provides clear evidence of this prediction.

case whereas BnB would take approximately 10^{10} seconds (nearly 317 years), demonstrating the clear superiority of the ICE algorithm. Similar findings would be expected to hold for other implementations such as the use of generic MIP solvers such as GLPK.

IV.2 Exact K-medoids algorithm

The K-medoids problem has a similar combinatorics to the 0-1 loss linear classification problem.

IV.2.1 Exhaustive, K-medoids enumeration based on join-list

Dataset For the unsupervised learning problem, the input `Dataset` is defined as

```
type Item = Vector
type Dataset = [Item]
```

which is a set of `Items`.

Squared distance evaluation As discussed already, the K-medoids problem is defined over an arbitrary objective function, the most common choice is the squared Euclidean distance function $d_2(\boldsymbol{x}, \boldsymbol{\mu})^2 = \|\boldsymbol{x} - \boldsymbol{\mu}\|_2^2$. In Haskell, the squared distance between a pair of data points can be defined as

```
sqrdist :: Item -> Item -> Double
sqrdist a b = sum $ map (^2) $ zipWith (-) a b
```

The `zipWith (-)` function uses every element of `a` minus the corresponding element of the second list `b`, then each element in the resulting list are squared by `map (^2)` function, and finally the results are summed together by `sum` function. The `sqrdist` function takes $O(D)$ time to evaluate the distance of two data items in $\mathbb{R}^D$. In practice, the distance for each pair of data points can be pre-calculated and stored in a distance matrix. This requires evaluating N^2 pairs of distance and evaluating the distance for a pair of points requires $O(D)$ complexity. Thus, the overall complexity for calculating the distance matrix is $O(N^2)$. Once the distance matrix is computed in advance, the sum-of-squared error (SSE) for each set of centroids can be obtained by indexing, which requires only $O(N)$ time. Thus evaluating the SSE for each K-combination is independent of dimension.

Configuration and assignment Analogous to the 0-1 loss linear classification problem, the SSE of a configuration is just a floating point `type SSE = Double` and `Maybe SSE` is defined to represent the existence of the SSE in a particular configuration. Combining this with the combination-sequence datatype, the configuration datatype is defined as follows

```
data Config = Cnfg {comb::[Item], seqn::[Item], model::Maybe SSE}
```

The prediction labels of a given set of centroids are referred to as an *assignment*, which is just a sequence of `Int`s (labels), defined as

```
type Assignment = [Int]
```

Updating objective function value Given a set of medoids `ms :: [Item]` and a sequence of data item `xs :: [Item]`, the SSE of `ms` with respect to data sequence `xs` can be calculated by function

```
updsse :: [Item] -> [Item] -> SSE
```

```
updsse xs ms = sum $ map sum[[sqrdist x m | (x,i) <- zip xs asgn, i == j]
   | (j, m) <- zip [0..] ms]]
  where asgn = getasgn xs ms
```

The `updsse` function first generates the assignment `asgn` of medoids `ms` with respect to `xs`, then each data item `x` in sequence `xs` is paired with its corresponding assignment in `asgn`. The squared distance of `x` to its corresponding centroids `ms!!j` is calculated by `sqrdist`. Finally, the SSE of each cluster is obtained by the `map sum` function, and the total SSE is obtained by summing the SSE of each cluster.

The assignment of medoids `ms` with respect to `xs` is generated by the `getasgn` function, which is defined as

```
getasgn :: [Item] -> [Item] -> Assignment
getasgn xs ms = map argmin listdists
  where listdists = [map (sqrdist x) ms | x <- xs]

argmin :: [Double] -> Int
argmin dists = fst $ minimumBy (comparing snd) (zip [0,1..] dists)
```

the `listdist` calculate the distances of each data point `x` to all medoids `ms` and then the `map argmin` function outputs the indexes of the medoids that have the minimal distances to each data point `x` in `xs`.

Algorithm The join-list algebra for solving the K-medoids problem can now be defined:

```
kmedAlg :: (Config -> Bool) -> Int -> ListAlg Item [Config]
kmedAlg p k = alg where
    alg Nil = [([],[], Nothing)]
    alg (Single a) = [([],[a], Nothing),([a],[a], Nothing)]
    alg (Join x y) = filter p (cpp (mergecnfg k) x y)

mergecnfg :: Int -> Config -> Config -> Config
mergecnfg k x1 x2 = (updcomb ,updseqn , upd)
  where
   updcomb = (comb x1) ++ (comb x2)
   updseqn = (seqn x1) ++ (seqn x2)
   upd = if (length updcomb == k) then (Just (updsse (updseqn) (updcomb)))
                                  else Nothing
```

In this case, instead of defining the fused algebra directly, a more generic form of filter-fused algebra is provided. Any segment-closed predicate `p` can be fused into `kmedAlg`. For instance, the following filter-fused algebra with a segment-closed predicate can be defined, consisting of the conjunction of *three* predicates

```
kmedFiltAlg nmin ub k = kmedAlg (retain nmin ub k) k
 where
  retain nmin ub k c = (clustSize k c) && (viable ub c) && (minData nmin c)
```

```
viable ub c = case (sse c) of
  Nothing -> True
  Just e -> (e <= ub)
clustSize k = (<= k) . length . comb
```

In addition to the `viable` and `feasible` test that were used in the 0-1 loss linear classification problem, an additional segment-closed predicate, `minData`, is introduced. This predicate checks whether the number of data items in each cluster is greater than `nmin`. The predicate `minData` is as follows,

```
minData :: Int -> Config -> Bool
minData nmin c
 | (length (comb c) == 0) = True
 | otherwise = lstElem x <= nmin
  where x = countData (length (comb c)) (getasgn (seqn c) (comb c))

lstElem :: [Int] -> Int
lstElem x = minlist (<=) x

countData :: Int -> Assignment -> [Int]
countData k asgn = [count i asgn | i <- [0..(k-1)]]
 where count i = length . filter (== i)
```

where `countData` counts the number of data items in each cluster based on the assignment with respect to the medoids `comb c`, then function `lstElem` finds the smallest cluster size. If this smallest cluster size is greater than the constrained size `nmin` then all other clusters will also have a size greater than `nmin`. This predicate is segment-closed because, as new medoids are introduced, the number of data items in each cluster can only decrease.

The selector for the K-medoids problem can be defined as

```
selsse ::  [Config] -> Config
selsse = foldl1 best
 where
  best c1 c2 = case (sse c1) of
   Nothing -> c2
   Just e1 -> case (sse c2) of
    Nothing -> c1
    Just e2 -> if (e1 <= e2)then c1 else c2
```

Finally, the K-medoids problem with the additional cluster size constraint can be solved efficiently by running:

```
kmed_filt nmin ub k = selsse . cata (kmedFiltAlg nmin ub k)
```

IV.2.2 Empirical analysis

In this section, we analyze the computational performance of our algorithm EKM on both synthetic and real-world data sets. Our evaluation aims to test the following predictions: (a) EKM always obtains the best objective value[25]; (b) wall-clock run-time matches the worst-case polynomial time complexity analysis; (c) the state-of-art BnB algorithm [Ren et al., 2022] for solving the K-medoids problem will have exponential time complexity even for fixed K in the worst-case. In our implementation, the matrix operations required at every recursive step are batch processed on a single GPU. We executed all the experiments on an Intel Core i9 CPU, with 24 cores, 2.4-6 GHz, 32 GB RAM and GeForce RTX 4060 Ti GPU. Remarks, all comparison about the time complexity is executed in the sequential version of our algorithm over CPU only.

IV.2.3 Performance on real-world datasets

We test the performance of our EKM algorithm against the approximate algorithms partition around medoids (PAM), Faster-PAM and Clustering Large Applications based on RANdomized Search (CLARANS)[26] on 18 datasets from the UCI Machine Learning Repository, two datasets from Ren et al. [2022] (UK, HCV), and two open-source datasets (PR2392, HEMI) from [Wang et al., 2022, Padberg and Rinaldi, 1991, Ren et al., 2022]. We show that, as expected, no other algorithms can achieve better objective function values (see Table 6), except in cases where Ren et al. [2022]'s BnB algorithm returned incorrect solutions, which were clearly invalid as they were several orders of magnitude lower than our exact solutions.

Our experiments included real-world datasets with a maximum size of $N = 5,000$. To the best of our knowledge, the largest dataset for which an exact solution has been previously obtained is $N = 150$, as documented by Ceselli and Righini [2005] with $K = 3$. Existing literature on the K-medoids problem has only reported exact solutions on very small datasets, primarily due to the use of BnB algorithms. Given their unpredictable run-time and worst-case exponential time complexity, most reported usage of BnB algorithms impose a hard computational time limit to avoid memory overflow or intractable run times.

In summary, Ren et al. [2022]'s algorithm returned only **approximate** solutions (with an optimality gap greater than zero), whereas our algorithm consistently produced **provably exact** solutions in **significantly less time**. Furthermore, for challenging datasets, such as WDG, Ren et al. [2022]'s algorithm produced a solution with an optimality gap of **800%** even after running for three hours! Moreover, in *nearly all* datasets tested in our experiments but not included in Ren et al. [2022] (e.g. IC, Yearst, WDG, wine, LD, VC, UKM and LM), their algorithm produces obvious errorness solutions where upper bounds that were **lower than** our **exact** solutions, which is fundamentally incorrect as an upper bound cannot be lower than the exact solution.

The only reason that Ren et al. [2022] claim their algorithm can handle datasets with over a million instances is that they test on datasets that are inherently easy to classify—so that even approximate algorithms can obtain exact solutions. As we have demonstrated, almost all the datasets they use can be solved exactly using PAM or Faster-PAM, which achieve exact solutions with significantly fewer resources. In contrast, Ren et al. [2022]'s algorithm requires an excessive amount of computational power (6,000 CPU

[25] The squares Euclidean distance function was chosen for the experiments, any other proper metrics could also be used.

[26] We set the maximum number of neighbors examined as 4, and the number of iteration as 5.

UCI dataset	N	D	EKM (ours)	Ren's BnB	PAM	Faster-PAM	CLARANS
LM	338	3	**3.96×10^{1}** (8.20×10^{-1})	1.21×10^{1} (2.31×10^{1}) 0	3.99×10^{1} (4.02×10^{-3})	4.07×10^{1} (3.01×10^{-3})	5.33×10^{1} (6.14)
UKM	403	5	**8.36×10^{1}** (1.37)	5.51×10^{1} (1.62×10^{3}) $\leq 0.1\%$	8.44×10^{1} (8.57×10^{-3})	8.40×10^{1} (3.21×10^{-3})	1.16×10^{2} (4.98×10^{1})
LD	345	5	**3.31×10^{5}** (8.3×10^{-1})	1.21×10^{1} (2.47×10^{1}) $\leq 0.1\%$	3.56×10^{5} (4.11×10^{-3})	**3.31×10^{5}** (3.87×10^{-3})	4.68×10^{5} (3.40)
Energy	768	8	**2.20×10^{6}** (1.37×10^{1})	2.20×10^{6} (1.68×10^{1}) $\leq 0.1\%$	2.28×10^{6} (6.95×10^{-3})	2.28×10^{6} (3.94×10^{-3})	2.97×10^{6} (2.71)
VC	310	6	**3.13×10^{5}** (6.82×10^{-1})	1.50×10^{5} (3.83×10^{2}) $\leq 0.1\%$	**3.13×10^{5}** (3.15×10^{-3})	3.58×10^{5} (5.36×10^{-3})	5.27×10^{5} (2.58)
Wine	178	13	**2.39×10^{6}** (2.22×10^{-1})	1.16×10^{4} (5.17×10^{1}) $\leq 0.1\%$	**2.39×10^{6}** (1.06×10^{-3})	2.63×10^{6} (2.34×10^{-3})	6.86×10^{6} (5.56×10^{-1})
Yeast	1484	8	**8.37×10^{1}** (1.74×10^{2})	6.57×10^{1} (1.08×10^{4}) $\leq 39.19\%$	8.42×10^{1} (9.54×10^{-2})	8.42×10^{1} (6.08×10^{-2})	1.05×10^{2} (1.73×10^{2})
IC	3150	13	**6.9063×10^{9}** (4.53×10^{3})	6.18×10^{9} (6.37×10^{3}) 0	6.9105×10^{9} (8.68×10^{-1})	**6.9063×10^{9}** (1.91×10^{-1})	1.44×10^{10} (2.70×10^{1})
WDG	5000	21	**1.67×10^{5}** (5.23×10^{4})	1.60×10^{5} (1.08×10^{4}) $\leq 785.05\%$	**1.67×10^{5}** (1.34)	**1.67×10^{5}** (1.97×10^{-1})	2.77×10^{5} (5.32×10^{3})
IRIS	150	4	**8.40×10^{1}** (1.57×10^{-1})	8.46×10^{1} (2.51×10^{1}) $\leq 27.1\%$	8.45×10^{1} (2.51×10^{-3})	8.45×10^{1} (1.03×10^{-3})	1.57×10^{2} (2.32×10^{-1})
SEEDS	210	7	**5.98×10^{2}** (2.85×10^{-1})	5.98×10^{2} (2.42×10^{1}) $\leq 0.1\%$	**5.98×10^{2}** (1.14×10^{-3})	**5.98×10^{2}** (3.59×10^{-3})	1.12×10^{3} (7.82×10^{-1})
GLASS	214	9	**6.29×10^{2}** (2.90×10^{-1})	6.29×10^{2} (3.13×10^{1}) $\leq 0.1\%$	**6.29×10^{2}** (1.01×10^{-3})	**6.29×10^{2}** (1.62×10^{-3})	1.04×10^{3} (2.27)
BM	249	6	**8.63×10^{5}** (3.96×10^{-1})	8.63×10^{5} (1.19×10^{2}) $\leq 0.1\%$	8.76×10^{5} (4.12×10^{-3})	**8.63×10^{5}** (1.61×10^{-3})	1.33×10^{6} (1.02×10^{1})
HF	299	12	**7.83×10^{11}** (6.26×10^{-1})	**7.83×10^{11}** (5.20×10^{1}) 0	**7.83×10^{11}** (1.00×10^{-3})	**7.83×10^{11}** (4.87×10^{-3})	1.88×10^{12} (6.55×10^{-1})
WHO	440	7	**8.33×10^{10}** (1.98)	8.33×10^{10} (3.71×10^{2}) $\leq 0.1\%$	**8.33×10^{10}** (5.62×10^{-3})	**8.33×10^{10}** (2.81×10^{-3})	1.21×10^{11} (8.12)
UK	258	5	**5.08×10^{1}** (3.78×10^{-1})	5.08×10^{1} (1.43×10^{3}) $\leq 0.1\%$	**5.08×10^{1}** (2.47×10^{-3})	**5.08×10^{1}** (2.47×10^{-3})	6.89×10^{1} (2.67×10^{1})
HCV	572	12	**2.75×10^{6}** (5.48)	2.75×10^{6} (8.59×10^{1}) $\leq 0.1\%$	**2.75×10^{6}** (5.79×10^{-3})	**2.75×10^{6}** (2.21×10^{-3})	4.75×10^{6} (6.89×10^{1})
ABS	740	19	**2.32×10^{6}** (1.17×10^{1})	2.32×10^{6} (6.23×10^{2}) $\leq 0.1\%$	**2.32×10^{6}** (2.11×10^{-2})	2.38×10^{6} (5.00×10^{-3})	2.96×10^{6} (7.80×10^{1})
TR	980	10	**1.13×10^{3}** (2.53×10^{1})	1.14×10^{3} (1.08×10^{4}) $\leq 89\%$	**1.13×10^{3}** (5.14×10^{-2})	**1.13×10^{3}** (1.14×10^{-2})	1.38×10^{3} (2.59×10^{2})
SGC	1000	21	**1.28×10^{9}** (3.87×10^{1})	1.28×10^{9} (1.75×10^{2}) $\leq 0.1\%$	**1.28×10^{9}** (1.71×10^{-1})	**1.28×10^{9}** (4.22×10^{-2})	2.52×10^{9} (2.24)
HEMI	1995	7	**9.91×10^{6}** (9.00×10^{2})	9.91×10^{6} (3.92×10^{2}) $\leq 0.1\%$	**9.91×10^{6}** (3.64×10^{-1})	**9.91×10^{6}** (6.99×10^{-2})	1.66×10^{7} (9.53)
PR2392	2392	2	**2.13×10^{10}** (1.29×10^{3})	2.13×10^{10} (1.54×10^{3}) $\leq$ 0.1%	**2.13×10^{10}** (3.66×10^{-1})	**2.13×10^{10}** (8.38×10^{-2})	3.47×10^{10} (1.15×10^{2})

Table 6: Empirical comparison of our novel exact K-medoids algorithm, EKM, against widely-used approximate algorithms (PAM, Fast-PAM, and CLARANS) and the state-of-art exact BnB algorithm developed by Ren et al. [2022], for $K = 3$, in terms of sum-of-squared errors (E), smaller is better. For Ren et al. [2022]'s aglorithm, we include both the *upper bound* and the *optimal gap*. The best-performing algorithm is highlighted in bold, while incorrect solutions are marked in red. Incorrect solutions are defined as those with objective values lower than the optimal solution. Bold font is applied only for datasets where the algorithm has run to completion. Although some of the upper bounds returned by Ren et al. [2022] are exact, these values are not marked in bold, as the optimal gap has not yet converged to zero. Wall clock execution time in brackets (seconds), the maximal running time of Ren et al. [2022]'s algorithm is 1.08×10^{4} second because of the three hours time limit.

cores) compared to approximate algorithms.

Moreover, we observed that Ren et al. [2022]'s algorithm exhibits **exponential** complexity even when K is fixed. This is evident from the **non-polynomial growth** in running time for experiments on datasets such as UK, BM, and Seeds. Although these datasets have nearly identical sizes, their running time differs significantly in Ren et al. [2022]'s algorithm. In the following section, we provide an empirical analysis of their algorithm to further investigate this behavior.

IV.2.4 Time complexity analysis without parallelization

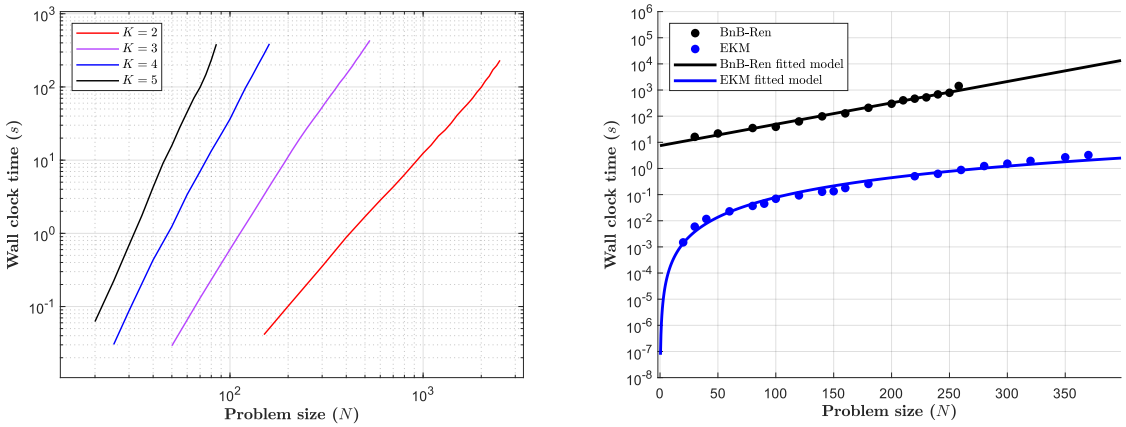


Figure 32: Log-log wall-clock run time (seconds) for our algorithm (EKM) tested on synthetic datasets (left panel). The run-time curves from left to right (corresponding to $K = 2, 3, 4, 5$ respectively), have slopes 3.005, 4.006, 5.018, and 5.995, an excellent match to the predicted worst-case run-time complexity of $O\left(N^3\right)$, $O\left(N^4\right)$, $O\left(N^5\right)$, and $O\left(N^6\right)$ respectively. Log-linear wall-clock run-time (seconds) comparing EKM algorithm against Ren et al. [2022]'s algorithm by sampling UK dataset with $K = 3$ (right panel). On this log-linear scale, exponential run-time appears as a linear function of problem size N, whereas polynomial run-time is a logarithmic function of N.

We test the wall-clock time of our novel EKM algorithm on a synthetic dataset with cluster sizes ranging from $K = 2$ to 5. When $K = 2$, the data size N ranges from 150 to 2,500, $K = 3$ ranges from 50 to 530, $K = 4$ ranges from 25 to 160, and $K = 5$ ranges from 30 to 200. The worst-case predictions are well-matched empirically (Figure 32, left panel).

As predicted, Ren et al. [2022]'s algorithm exhibits worst-case exponential time complexity even for a fixed $K = 3$ (Figure 32, right panel), whereas our algorithm runs in polynomial time in the worst case.

Part V

Conclusion

This thesis presents a general framework that integrates three key themes—combinatorial geometry, combinatorial generation, and constructive algorithmics—to solve combinatorial optimization problems involving finite hyperplanes and data points. Our foundational theory builds upon the work of Bird and De Moor [1996], Bird and Gibbons [2020] in constructive algorithmics, Edelsbrunner [1987] in combinatorial geometry, and Kreher and Stinson [1999] in combinatorial generation.

Our novel contributions to these themes include, for example, in constructive theme, the cross-product fusion law and the reformulation of BnB using catamorphisms. In particular, we believe the cross-product fusion law, along with other fusion laws proposed in Subsection II.2.3.4 holds tromendous potential, which will be discussed shortly in the discussion of the open problems below. In particular, we believe the cross-product fusion law, along with other fusion laws proposed in Subsection II.2.3.4, holds tremendous potential, which will be further discussed in the open problems section next. In the combinatorial generation theme, we contribute by introducing two generic generators—the `kcombs` and `kperms` generators discussed in II.2.3.3—and the new class of generator that we introduced—the integer SDP generator. Finally, for combinatorial geometry, we analyze the relationships (counting and incidence relations) between dichotomies and cells in an arrangement using inhomogeneous coordinates, whereas classical analyses typically use homogeneous coordinates. Additionally, our geometric analysis to the K-means problem and 2-means problem are novel.

The primary contribution of this thesis is the application of knowledge from these three themes to solving combinatorial optimization problems in machine learning, particularly the empirical risk minimization (ERM) problem for ReLU networks and the optimal hyperplane decision tree problem. In these cases, insights from all three themes are integrated to develop effective solutions.

However, for the classification and clustering problems, while combinatorial geometry has been used to simplify the combinatorial structures, and the efficient generators proposed in II.2.3.3 have been applied to solve these problems, program calculation has not been utilized. This is because these problems involve the simplest combinatorial structures, such as combinations and binary assignments, leaving no room for applying algorithmic techniques.

Nevertheless, as demonstrated in Part IV, even the simplest brute-force algorithm for solving the linear classification and K-medoids problems significantly outperforms state-of-the-art BnB algorithms. Moreover, these methods remain amenable to further speed-up in future studies.

Open problems

Open questions in constructive algorithmics and combinatorial generation

- *Constructive proofs for unproven Gray code functions.* e provide Haskell implementations of the ranking and unranking functions for several common combinatorial Gray code generators. Some of these differ from the definitions given in classical combinatorial generation textbooks, such as Kreher and Stinson [1999], Ruskey [2003]. While our resulting implementation is more succinct and

has undergone manual verification to ensure correctness, a constructive proof is still required. Given that this thesis does not employ CGC generators, ranking, or unranking functions elsewhere, we leave this proof as an interesting topic for future research.

- *Incompartibility of* `kcombs` *generator.* We found that the Cartesian product fusion law is incompatible with the `kcombs` generator, partly because the use of convolution (which involves `zip` function in the definition) prevents the generator from being strictly an SDP. How to generalize the Cartesian product fusion law to a more general setting would be an interesting topic to explore further.

- *Reverse design process for constructing combinatorial generator.* We believe the potential implications of several fusion laws discussed in II.2.3.4—for constructing generators for complex combinatorial structures based on simple ones—are far more significant than the applications demonstrated in this thesis. In the study of analytic combinatorics, Flajolet [2009] proposed a *symbolic method* for deriving *generating functions* (GFs), which are used to obtain counting formulas for complex combinatorial structures. This is achieved by combining the GFs of basic structures through primitive set construction operations—such as *disjoint union*, *Cartesian* product, *set restriction*, and *multiset*—to derive the GF of a complex combinatorial structure. It is intriguing to explore how to develop a "*reverse design process*" for constructing combinatorial generators using constructive algorithmics. Specifically, given the counting formula of a combinatorial structure and its composition from simple combinatorial structures, we aim to develop an efficient generator for this structure using a similarly symbolic approach. We have taken some initial steps in Subsection II.2.3.4, but more sophisticated principles need to be explored to build generators for more complex combinatorial structures. Such principles could have a significant impact on many fields of science, including computational/combinatorial geometry, combinatorial optimization, and combinatorics, since knowing how to generate combinatorial structures is far more important than merely enumerating (counting) them.

- *Graphical algorithms.* Graphical algorithms are used ubiquitously in combinatorial optimization. However, although graphical algorithms are frequently employed in statistical machine learning—such as in constructing probabilistic models like MCMC (Markov chain Monte Carlo)—the graph structure is rarely utilized in classical machine learning research, which mainly concerns deterministic methods for building machine learning models. Because we focus on designing exact algorithms, we did not explore any graph algorithms in this thesis. In future studies, it would be interesting to investigate how constructive algorithmics can be used to rigorously reformulate graph algorithms in machine learning.

Open questions in combinatorial geometry

- *Specialized cell enumeration algorithm and dominance relations for K-means problem.* Although the algorithm we propose to solve the K-means problem is polynomial in the worst case, its best-case complexity matches its worst-case complexity, leaving significant room for future speed-ups. There are two potential speed-ups: First, as mentioned in II.3.4.5, designing a specialized cell enumeration algorithm tailored to enumerate the cells in the arrangement represented by the K-means problem. Second, designing tailored dominance relations, introduced in Subsection II.2.6.3, for solving the K-means problem.

- *Numerical issues in hyperplane-based method.* The hyperplane-based method for cell enumeration requires finding D data points that lie precisely on the hyperplane to characterize it uniquely. However, this often introduces severe numerical issues with real-world datasets, as it is frequently the case that many data points are very close to each other due to their representation with finite-bit floating-point numbers rather than real values. As a result, the matrix inversion involved—where the matrix consists of D data points in $\mathbb{R}^D$—often fails because the data points are not in general position. A similar numerical issue arises in Algorithm 3, which requires determining a value t_d that must be sufficiently large to ensure all vertices of the hyperplane arrangement $\mathcal{H}$ lie below E_{t_d}.

What I would do differently if I could begin anew

When I started on my PhD, I entered as a naive student with no background in computer science and only rudimentary knowledge of linear algebra and probability theory. I am deeply grateful to my supervisor, Max, for accepting me as his student and for his patient guidance throughout this journey. Nevertheless, the oversights and gaps in my preparation during the early stages of my doctoral research led me down numerous detours. Reflecting on this experience, I have identified several aspects I would approach differently if granted the opportunity to start anew.

Avoid spending too much time reading the literature on exact machine learning algorithms

One big change I would make is to avoid over-investing time in reviewing state-of-the-art ML literature on exact algorithms, which I now regard as largely inefficient for my objectives. For those aspiring to devise original and ingenious exact algorithms to tackle fundamental problems in ML, I warn against investigating too deeply into the state-of-the-art literature in ML. In my view, much of the contemporary ML literature is written to please reviewers—often junior researchers or even undergraduate students—who tend to value complexity over elegance and simplicity. Consequently, I found that many papers on exact ML algorithms are obscure and ambiguous, with pseudocode either oversimplified to the point of diverging from actual implementations or overloaded with excessive detail that obscures understanding.

As one of my own experiences, I devoted considerable time and effort to studying state-of-the-art algorithms for the optimal decision tree problem. After months of fruitless investigation, I had to abandon the effort. The literature was full of undefined terms, misleading descriptions, and, most critically, lack of justifications for the correctness, which compounded my frustration. It was only after turning away from this body of work and engaging with the remarkable work done by Bird and De Moor [1996], Bird and Gibbons [2020] that I began to rethink the problem independently. By setting aside the convoluted prior research, I developed a solution that proved both more elegant and more effective than existing approaches.

In retrospect, I would adopt an approach that balances foundational learning with independent exploration, rather than relying heavily on a literature that, while extensive, often obscures rather than illuminates. This shift in perspective not only streamlined my research process but also underscored the value of clarity and simplicity in algorithmic design—principles I now hold as central to my scholarly philosophy.

Haskell VS C++ implementation

Optimizing Haskell implementations. Due to time constraints, although we developed various algorithms and generators using Haskell, we did not conduct experiments in Haskell. This decision was made for several reasons. First, we aim to convince the machine learning (ML) community to adopt a more rigorous and scientific approach, as we have identified numerous errors in the literature. However, to effectively persuade them, or at least to publish a paper in an ML journal, we need to invest significant effort in experiments, as ML researchers tend to focus exclusively on empirical evidence rather than rigorous reasoning. As a result, ML research has become, to some extent, a "numbers game." Indeed, almost half of my Ph.D. journey has been spent implementing various algorithms in languages commonly used within the ML community, such as Python and C++, because very few ML researchers are familiar with functional programming languages like Haskell. Unfortunately, it is extremely challenging to run large-scale datasets for exact algorithms. Our ultimate goal is to implement our algorithm on CUDA, as NVIDIA is the leading producer of high-performance GPUs. Since CUDA is based on C++, we chose to first implement our algorithm in classical imperative languages such as C++ and Python, with plans to develop an efficient CUDA implementation as the next step. However, as my understanding of both parallelization and functional programming deepens, I increasingly feel that implementing the parallelized algorithm in a side-affect free functional programming language initially would be a more compelling approach. We leave this intriguing topic of parallelizing our algorithm in a functional programming language as an interesting avenue for future research.

Quickcheck

Haskell QuickCheck is a helpful tool in the Haskell programming language that automatically creates random test cases to check if your code works correctly. It uses a method called property-based testing, where you write general rules your code should follow instead of making specific tests by hand. Looking back, I am upset that the combination generators from Chapter II.1 were only tested manually. It would have been much better to use QuickCheck for those functions.

Sadly, I only learned about this tool late in my PhD. This happened because I did not talk much with other functional programmers. If I had, I might have found out about QuickCheck sooner and used it in my work. I missed a chance to use a great tool that fits my research on combinations and algorithms. It also teaches me how important it is to connect with others and look for useful tools earlier.

References

Sina Aghaei, Mohammad Javad Azizi, and Phebe Vayanos. Learning optimal and fair decision trees for non-discriminative decision-making. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1418–1426, 2019.

Sina Aghaei, Andrés Gómez, and Phebe Vayanos. Strong optimal classification trees. *ArXiv preprint ArXiv:2103.15965*, 2021.

Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning optimal decision trees using caching branch-and-bound search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3146–3153, 2020.

Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Pydl8. 5: A library for learning optimal decision trees. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 5222–5224, 2021.

Srinivas M Aji and Robert J McEliece. The generalized distributive law. *IEEE transactions on Information Theory*, 46(2):325–343, 2000.

Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. NP-hardness of Euclidean sum-of-squares clustering. *Machine learning*, 75:245–248, 2009.

Elaine Angelino, Nicholas Larus-Stone, Daniel Alabi, Margo Seltzer, and Cynthia Rudin. Learning certifiably optimal rule lists for categorical data. *Journal of Machine Learning Research*, 18(234):1–78, 2018.

Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding deep neural networks with rectified linear units. *ArXiv preprint ArXiv:1611.01491*, 2016.

Florent Avellaneda. Efficient inference of optimal decision trees. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3195–3202, 2020.

David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete applied mathematics*, 65(1-3): 21–46, 1996.

Egon Balas and Paolo Toth. Branch and bound methods for the traveling salesman problem. *Carnegie-Mellon University, Design Research Center*, 1983.

R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, 1985. doi: 10.1109/TSE.1985.231877.

Arindam Banerjee, Srujana Merugu, Inderjit S Dhillon, Joydeep Ghosh, and John Lafferty. Clustering with Bregman divergences. *Journal of Machine Learning Research*, 6(10), 2005.

Rodrigo C Barros, Ricardo Cerri, Pablo A Jaskowiak, and André CPLF De Carvalho. A bottom-up oblique decision tree induction algorithm. In *11th International Conference on Intelligent Systems Design and Applications*, pages 450–456. IEEE, 2011.

Peter L Bartlett, Nick Harvey, Christopher Liaw, and Abbas Mehrabian. Nearly-tight vc-dimension and pseudodimension bounds for piecewise linear neural networks. *Journal of Machine Learning Research*, 20(63):1–17, 2019.

F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gantz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Semelson, M. Wirsing, and H. Wössner. *The Munich Project CIP*, volume 1: The Wide Spectrum Language CIP-L. Springer Berlin, Heidelberg, Berlin, 1985. LNCS 183.

Mikhail Belkin, Daniel J Hsu, and Partha Mitra. Overfitting or perfect fitting? risk bounds for classification and regression rules that interpolate. *Advances in Neural Information Processing Systems*, 31, 2018.

Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116 (32):15849–15854, 2019a.

Mikhail Belkin, Alexander Rakhlin, and Alexandre B Tsybakov. Does data interpolation contradict statistical optimality? In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1611–1619. PMLR, 2019b.

Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

Richard Bellman. *Eye of the Hurricane*. World Scientific, 1984.

Kristin P Bennett. Decision tree construction via linear programming. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1992.

Kristin P Bennett and Jennifer A Blue. Optimal decision trees. *Rensselaer Polytechnic Institute Math Report*, 214(24):128, 1996.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

Daniel Bertschinger, Christoph Hertrich, Paul Jungeblut, Tillmann Miltzow, and Simon Weber. Training fully connected neural networks is r-complete. *ArXiv preprint ArXiv.2204.01368*, 10, 2022.

Dimitris Bertsimas and Jack Dunn. Optimal classification trees. *Machine Learning*, 106:1039–1082, 2017.

Dimitris Bertsimas and Jack Dunn. *Machine learning under a modern optimization lens*. Dynamic Ideas LLC Charlestown, MA, 2019.

Dimitris Bertsimas, Jean Pauphilet, and Bart Van Parys. Sparse regression. *Statistical Science*, 35(4): 555–578, 2020.

Richard Bird. *Pearls of functional algorithm design*. Cambridge University Press, 2010. ISBN 9780521513388. URL http://www.cambridge.org/gb/knowledge/isbn/item5600469.

Richard Bird and Oege De Moor. The algebra of programming. *NATO ASI DPD*, 152:167–203, 1996.

Richard Bird and Jeremy Gibbons. *Algorithm design with Haskell*. Cambridge University Press, 2020.

Richard S Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design: International Summer School directed by FL Bauer, M. Broy, EW Dijkstra, CAR Hoare*, pages 5–42. Springer, 1987.

Richard S Bird. Lectures on constructive functional programming. In *Constructive Methods in Computing Science: International Summer School directed by FL Bauer, M. Broy, EW Dijkstra, CAR Hoare*, pages 151–217. Springer, 1989.

Richard S. Bird. Zippy tabulations of recursive functions. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction*, pages 92–109, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-70594-9.

R.S. Bird and John Hughes. The alpha-beta algorithm: An exercise in program transformation. *Information Processing Letters*, 24(1):53–57, 1987. ISSN 0020-0190. doi: https://doi.org/10.1016/0020-0190(87)90198-0. URL https://www.sciencedirect.com/science/article/pii/0020019087901980.

Christopher M Bishop. Pattern recognition and machine learning. *Springer Google Schola*, 2:1122–1128, 2006.

Anders Björner. *Oriented matroids*. Number 46. Cambridge University Press, 1999.

Anders Björner and Günter M Ziegler. Introduction to greedoids. *Matroid Applications*, 40:284–357, 1992.

Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. Learnability and the vapnik-chervonenkis dimension. *Journal of the ACM*, 36(4):929–965, 1989.

Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge Un0iversity Press, 2004.

L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen. *Classification and Regression Trees*. Taylor & Francis, 1984. ISBN 9780412048418.

Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001a.

Leo Breiman. Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical science*, 16(3):199–231, 2001b.

J Paul Brooks. Support vector machines with the ramp loss and the hard margin loss. *Operations Research*, 59(2):467–479, 2011.

Alexander Bunkenburg. The boom hierarchy. In *Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop on Functional Programming, Ayr, Scotland, 5–7 July 1993*, pages 1–8. Springer, 1994.

R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977. ISSN 0004-5411. doi: 10.1145/321992.321996. URL https://doi.org/10.1145/321992.321996.

Yuliang Cai, Huaguang Zhang, Qiang He, and Jie Duan. A novel framework of fuzzy oblique decision tree construction for pattern classification. *Applied Intelligence*, 50:2959–2975, 2020.

Venanzio Capretta, Tarmo Uustalu, and Varmo Vene. Recursive coalgebras from comonads. *Information and Computation*, 204(4):437–468, 2006.

Emilio Carrizosa, Amaya Nogales-Gómez, and Dolores Romero Morales. Strongly agree or strongly disagree?: Rating features in support vector machines. *Information Sciences*, 329:256–273, 2016.

Bob F Caviness and Jeremy R Johnson. *Quantifier elimination and cylindrical algebraic decomposition*. Springer Science & Business Media, 2012.

Alberto Ceselli and Giovanni Righini. A branch-and-price algorithm for the capacitated p-median problem. *Networks: An International Journal*, 45(3):125–142, 2005.

Yann Chevaleyre, Frédéerick Koriche, and Jean-Daniel Zucker. Rounding methods for discrete linear classification. In *International Conference on Machine Learning*, pages 651–659. Proceedings of Machine Learning Research, 2013.

Nicos Christofides and John E Beasley. A tree search algorithm for the p-median problem. *European Journal of Operational Research*, 10(2):196–204, 1982.

Nicos Christofides, Aristide Mingozzi, and Paolo Toth. Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations. *Mathematical Programming*, 20:255–282, 1981.

Murray I Cole. *Algorithmic skeletons: Structured management of parallel computation*. Pitman London, 1989.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

Thomas M Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers*, (3):326–334, 1965.

David Cox, John Little, Donal O'shea, and Moss Sweedler. *Ideals, varieties, and algorithms*, volume 3. Springer, 1997.

David R Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):215–232, 1958.

David Roxbee Cox. Some procedures connected with the logistic qualitative response curve. *Research Papers in Statistics*, pages 55–71, 1966.

George B Dantzig. Linear programming and extensions. In *Linear programming and extensions*. Princeton university press, 2016.

Constantinos Daskalakis, Richard M Karp, Elchanan Mossel, Samantha J Riesenfeld, and Elad Verbin. Sorting and selection in posets. *SIAM Journal on Computing*, 40(3):597–622, 2011.

Oege De Moor. Categories, relations and dynamic programming. *Mathematical Structures in Computer Science*, 4(1):33–69, 1994.

Oege De Moor. A generic program for sequential decision processes. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 1–23. Springer, 1995.

Oege de Moor and Jeremy Gibbons. Bridging the algorithm gap: A linear-time functional program for paragraph formatting. *Science of Computer Programming*, 35(1), 1999. URL http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/bridging.ps.gz.

Emir Demirović and Peter J Stuckey. Optimal decision trees for nonlinear metrics. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3733–3741, 2021.

Emir Demirović, Anna Lukina, Emmanuel Hebrard, Jeffrey Chan, James Bailey, Christopher Leckie, Kotagiri Ramamohanarao, and Peter J Stuckey. Murtree: Optimal decision trees via dynamic programming and search. *Journal of Machine Learning Research*, 23(26):1–47, 2022.

George Diehr. Evaluation of a branch and bound algorithm for clustering. *SIAM Journal on Scientific and Statistical Computing*, 6(2):268–284, 1985.

Olivier Du Merle, Pierre Hansen, Brigitte Jaumard, and Nenad Mladenovic. An interior point algorithm for minimum sum-of-squares clustering. *SIAM Journal on Scientific Computing*, 21(4):1485–1505, 1999.

D. Dua and C. Graff. UCI Machine learning repository, 2019. URL http://archive.ics.uci.edu/.

Jack William Dunn. *Optimal trees for prediction and prescription*. PhD thesis, MIT, 2018.

Herbert Edelsbrunner. *Algorithms in combinatorial geometry*, volume 10. Springer Science & Business Media, 1987.

Samuel Eilenberg and Jesse B Wright. Automata in general algebras. *Information and Control*, 11(4): 452–470, 1967.

Sourour Elloumi. A tighter formulation of the p-median problem. *Journal of Combinatorial Optimization*, 19(1):69–83, 2010.

Kento Emoto, Sebastian Fischer, and Zhenjiang Hu. Filter-embedding semiring fusion for programming with mapreduce. *Formal Aspects of Computing*, 24:623–645, 2012.

David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F Italiano. Sparse dynamic programming i: Linear cost functions. *Journal of the ACM*, 39(3):519–545, 1992.

Hatem A Fayed and Amir F Atiya. A mixed breadth-depth first strategy for the branch and bound tree of euclidean k-center problems. *Computational Optimization and Applications*, 54:675–703, 2013.

J-A Ferrez, Komei Fukuda, and Th M Liebling. Solving the fixed rank convex quadratic maximization in binary variables by a parallel zonotope construction algorithm. *European Journal of Operational Research*, 166(1):35–50, 2005.

P Flajolet. *Analytic combinatorics*. Cambridge University Press, 2009.

Maarten M Fokkinga. An exercise in transformational programming: Backtracking and branch-and-bound. *Science of Computer Programming*, 16(1):19–48, 1991.

Maarten M Fokkinga. *Law and order in algorithmics.* PhD thesis, University of Twente, 1992.

Komei Fukuda. Lecture: Polyhedral computation, spring 2016. *Institute for Operations Research and Institute of Theoretical Computer Science, ETH Zurich. https://inf. ethz. ch/personal/fukudak/lect/pclect/notes2015/PolyComp2015. pdf*, 2016.

Zvi Galil and Raffaele Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64(1):107–118, 1989.

Susan L. Gerhart. Correctness-preserving program transformations. In *Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '75, pages 54–66, New York, NY, USA, 1975. Association for Computing Machinery. ISBN 9781450373517. doi: 10.1145/512976.512983. URL https://doi.org/10.1145/512976.512983.

Thomas Gerstner and Markus Holtz. Algorithms for the cell enumeration and orthant decomposition of hyperplane arrangements. *University of Bonn*, 2006.

Jeremy Gibbons. *Algebras for tree algorithms.* PhD thesis, University of Oxford, 1991.

Jeremy Gibbons. Computing downwards accumulations on trees quickly. *Theoretical Computer Science*, 169(1):67–80, 1996.

Surbhi Goel, Adam Klivans, Pasin Manurangsi, and Daniel Reichman. Tight hardness results for training depth-2 relu networks. *ArXiv preprint ArXiv:2011.13550*, 2020.

Joseph A Goguen, James W Thatcher, Eric G Wagner, Jesse B Wright, et al. Abstract data types as initial algebras and the correctness of data representations. *Computer Graphics, Pattern Recognition and Data Structure*, pages 89–93, 1975.

John C Gower and Pierre Legendre. Metric and Euclidean properties of dissimilarity coefficients. *Journal of classification*, 3:5–48, 1986.

Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. Why do tree-based models still outperform deep learning on typical tabular data? *Advances in Neural Information Processing Systems*, 35:507–520, 2022.

Martin Grötschel and Yoshiko Wakabayashi. A cutting plane algorithm for a clustering problem. *Mathematical Programming*, 45(1):59–96, 1989.

Oktay Günlük, Jayant Kalagnanam, Minhan Li, Matt Menickelly, and Katya Scheinberg. Optimal decision trees for categorical data via integer programming. *Journal of Global Optimization*, 81:233–260, 2021.

LLC Gurobi Optimization. Gurobi optimizer reference manual. 2021. URL https://docs.gurobi.com/projects/optimizer/en/current/index.html.

Robin Hartshorne. *Algebraic geometry*, volume 52. Springer Science & Business Media, 1977.

Susumu Hasegawa, H Imai, M Inaba, N Katoh, and J Nakano. Efficient algorithms for variance-based k-clustering. In *Proceedings of the First Pacific Conference on Computer Graphics and Applications, World Scientific*, pages 75–89. Citeseer, 1993.

Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: Data mining, inference, and prediction*, volume 2. Springer, 2009.

Xi He and Max A. Little. Exact 0-1 loss linear classification algorithms, April 2023a. URL https://github.com/XiHegrt/E01Loss.

Xi He and Max A Little. An efficient, provably exact algorithm for the 0-1 loss linear classification problem. *ArXiv preprint ArXiv:2306.12344*, 2023b.

Xi He and Max A Little. EKM: An exact, polynomial-time algorithm for the K-medoids problem. *ArXiv preprint ArXiv:2405.12237*, 2024.

Christoph Hertrich. *Facets of neural network complexity*. Technische Universitaet Berlin (Germany), 2022.

Ralf Hinze. Adjoint folds and unfolds—an extended study. *Science of Computer Programming*, 78(11): 2108–2159, 2013.

Ralf Hinze and Nicolas Wu. Histo- and dynamorphisms revisited. In *Proceedings of the 9th ACM Special Interest Group on Programming Languages Workshop on Generic Programming*, pages 1–12, 2013.

Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Unifying structured recursion schemes. *ACM SIGPLAN International Conference on Function Programming*, 48(9):209–220, 2013.

Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Conjugate hylomorphisms–or: The mother of all structured recursion schemes. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 50 (1):527–538, 2015.

Charles Anthony Richard Hoare. Unified theories of programming. In *Mathematical methods in program development*, pages 313–367. Springer, 1997.

Charles Antony Richard Hoare. Chapter ii: Notes on data structuring. In *Structured Programming*, pages 83–174. 1972.

Charles Antony Richard Hoare and Jifeng He. The weakest prespecification. *Information Processing Letters*, 24(2):127–132, 1987.

Robert C Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63–90, 1993.

Hao Hu, Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. Learning optimal decision trees with maxsat and its integration in adaboost. In *29th International Joint Conference on Artificial Intelligence and the 17th Pacific Rim International Conference on Artificial Intelligence*, 2020.

Xiyang Hu, Cynthia Rudin, and Margo Seltzer. Optimal sparse decision trees. *Advances in Neural Information Processing Systems*, 32, 2019.

Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. *ACM SIGPLAN International Conference on Functional Programming*, 31(6):73–82, 1996.

Liang Huang. Advanced dynamic programming in semiring and hypergraph frameworks. *Coling 2008: Advanced Dynamic Programming in Computational Linguistics: Theory, Algorithms and Applications-Tutorial Notes*, pages 1–18, 2008.

Toshihide Ibaraki. The power of dominance relations in branch-and-bound algorithms. *Journal of the ACM*, 24(2):264–279, 1977.

Mary Inaba, Naoki Katoh, and Hiroshi Imai. Applications of weighted voronoi diagrams and randomization to variance-based k-clustering. In *Proceedings of the tenth annual symposium on Computational geometry*, pages 332–339, 1994.

Johan Theodoor Jeuring. *Theories for algorithm calculation*. PhD thesis, Utrecht University, 1993.

Su Jia, Fatemeh Navidi, R Ravi, et al. Optimal decision tree with noisy outcomes. *Advances in Neural Information Processing Systems*, 32, 2019.

Richard M Karp and Michael Held. Finite-state processes and dynamic programming. *SIAM Journal on Applied Mathematics*, 15(3):693–718, 1967.

Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

Donald E Knuth. Structured programming with go to statements. *ACM Computing Surveys*, 6(4):261–301, 1974.

Walter H Kohler and Kenneth Steiglitz. Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. *Journal of the ACM*, 21(1):140–156, 1974.

Warren L. G. Koontz, Patrenahalli M. Narendra, and Keinosuke Fukunaga. A branch and bound clustering algorithm. *IEEE Transactions on Computers*, 100(9):908–915, 1975.

Donald L Kreher and Douglas R Stinson. Combinatorial algorithms: Generation, enumeration, and search. *ACM Special Interest Group on Algorithms and Computation Theory News*, 30(1):33–35, 1999.

Frank R Kschischang, Brendan J Frey, and H-A Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.

Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, October 1980. ISSN 0004-5411. doi: 10.1145/322217.322232. URL https://doi.org/10.1145/322217.322232.

Joachim Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161, 1968.

Hyafil Laurent and Ronald L Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.

Jimmy Lin, Chudi Zhong, Diane Hu, Cynthia Rudin, and Margo Seltzer. Generalized and scalable optimal sparse decision trees. pages 6150–6160. Proceedings of Machine Learning Research, 2020.

Max A Little. *Machine learning for signal processing: Data science, algorithms, and computational statistics*. Oxford University Press, USA, 2019.

Max A Little, Xi He, and Ugur Kayas. Polymorphic dynamic programming by algebraic shortcut fusion. *Formal Aspects of Computing*, May 2024. ISSN 0934-5043. doi: 10.1145/3664828. (in press).

Yufeng Liu and Yichao Wu. Variable selection via a combination of the l0 and l1 penalties. *Journal of Computational and Graphical Statistics*, 16(4):782–798, 2007.

Philip M Long and Rocco A Servedio. Random classification noise defeats all convex potential boosters. In *Proceedings of the 25th International Conference on Machine learning*, pages 608–615, 2008.

Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. The planar k-means problem is NP-hard. *Theoretical Computer Science*, 442:13–21, 2012.

Andrew Makhorin. GLPK (gnu linear programming kit). *http://www.gnu.org/s/glpk/glpk.html*, 2008.

Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14 (2-3):255–279, 1990.

Petros Maragos, Vasileios Charisopoulos, and Emmanouil Theodosis. Tropical geometry and machine learning. *Proceedings of the IEEE*, 109(5):728–755, 2021.

Rahul Mazumder, Xiang Meng, and Haoyue Wang. Quant-bnb: A scalable branch-and-bound method for optimal decision trees with continuous features. In *International Conference on Machine Learning*, pages 15255–15277. PMLR, 2022.

Lambert Meertens. Algorithmics : towards programming as a mathematical activity. In *Towards programming as a mathematical activity. Mathematics and computer science*, pages 289–334, 1986.

Lambert Meertens. First steps towards the theory of rose trees. *Centrum Wiskunde & Informatica, Amsterdam*, 1988.

Nimrod Megiddo and Kenneth J Supowit. On the complexity of some common geometric location problems. *SIAM Journal on Computing*, 13(1):182–196, 1984.

Bartosz Milewski. *Category theory for programmers*. Blurb, 2018.

Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

Andrey Mokhov. Algebraic graphs with class (functional pearl). *ACM SIGPLAN International Conference on Function Programming*, 52(10):2–13, 2017.

Rajeev Motwani and Prabhakar Raghavan. Randomized algorithms. *ACM Computing Surveys (CSUR)*, 28(1):33–37, 1996.

Sreerama K Murthy, Simon Kasif, and Steven Salzberg. A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research*, 2:1–32, 1994.

Nina Narodytska, Alexey Ignatiev, Filipe Pereira, and Joao Marques-Silva. Learning optimal decision trees with sat. In *27th International Joint Conference on Artificial Intelligence*, pages 1362–1368. International Joint Conferences on Artificial Intelligence Organization, 7 2018. doi: 10.24963/ijcai.2018/189. URL https://doi.org/10.24963/ijcai.2018/189.

John Ashworth Nelder and Robert WM Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society: Series A (General)*, 135(3):370–384, 1972.

Tan Nguyen and Scott Sanner. Algorithms for direct 0–1 loss optimization in binary classification. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1085–1093, Atlanta, Georgia, USA, 17–19 Jun 2013. Proceedings of Machine Learning Research. URL https://proceedings.mlr.press/v28/nguyen13a.html.

Siegfried Nijssen and Elisa Fromont. Optimal constraint-based decision tree induction from itemset lattices. *Data Mining and Knowledge Discovery*, 21:9–51, 2010.

Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *Society for Industrial and Applied Mathematics Review*, 33(1): 60–100, 1991.

Judea Pearl, Madelyn Glymour, and Nicholas P Jewell. *Causal inference in statistics: A primer*. John Wiley & Sons, 2016.

Jiming Peng and Yu Xia. A cutting algorithm for the minimum sum-of-squared error clustering. In *Proceedings of the 2005 SIAM International Conference on Data Mining*, pages 150–160. SIAM, 2005.

Adolphe Quetelet et al. *Correspondance mathématique et physique*, volume 2. Impr. d' H. Vanderkeriehove, 1826.

J Ross Quinlan. *C4. 5: Programs for machine learning*. Elsevier, 2014.

Miroslav Rada and Michal Cerny. A new algorithm for enumeration of cells of hyperplane arrangements and a comparison with Avis and Fukuda's reverse search. *SIAM Journal on Discrete Mathematics*, 32 (1):455–473, 2018.

Jiayang Ren, Kaixun Hua, and Yankai Cao. Global optimal K-medoids clustering of one million samples. *Advances in Neural Information Processing Systems*, 35:982–994, 2022.

Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5):206–215, 2019.

Cynthia Rudin and Joanna Radin. Why are we using black box models in ai when we don't need to. *Harvard Data Science Review*, 1(2):1–9, 2019.

Frank Ruskey. Combinatorial generation. *Preliminary Working Draft. University of Victoria, Victoria, BC, Canada*, 11:20, 2003.

Norbert Sauer. On the density of families of sets. *Journal of Combinatorial Theory, Series A*, 13(1): 145–147, 1972.

Gunther Schmidt and Thomas Ströhlein. *Relations and graphs: Discrete mathematics for computer scientists*. Springer Science & Business Media, 2012.

Alexander Schrijver et al. *Combinatorial optimization: Polyhedra and efficiency*, volume 24. Springer, 2003.

Robert Sedgewick. Permutation generation methods. *ACM Computing Surveys*, 9(2):137–164, 1977.

Michael Ian Shamos and Dan Hoey. Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science*, pages 151–162. IEEE, 1975.

Micha Sharir. Almost tight upper bounds for lower envelopes in higher dimensions. *Discrete & Computational Geometry*, 12(3):327–345, 1994.

Ravid Shwartz-Ziv and Amitai Armon. Tabular data: Deep learning is not all you need. *Information Fusion*, 81:84–90, 2022.

Richard P Stanley et al. An introduction to hyperplane arrangements. *Geometric Combinatorics*, 13 (389-496):24, 2004.

Yufang Tang, Xueming Li, Yan Xu, Shuchang Liu, and Shuxin Ouyang. A mixed integer programming approach to maximum margin 0–1 loss classification. In *2014 International Radar Conference*, pages 1–6. IEEE, 2014.

Cristina Tîrnăucă, Domingo Gómez-Pérez, José L Balcázar, and José L Montaña. Global optimality in k-means clustering. *Information Sciences*, 439:79–94, 2018.

Csaba D Toth, Joseph O'Rourke, and Jacob E Goodman. *Handbook of discrete and computational geometry*. Chemical Rubber Company press, 2017.

Hoang Tuy. Concave programming under linear constraints. *Soviet Math.*, 5:1437–1440, 1964.

The Univalent Foundations Program. *Homotopy type theory: Univalent foundations of mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

Berk Ustun and Cynthia Rudin. Supersparse linear integer models for optimized medical scoring systems. *Machine Learning*, 102:349–391, 2016.

Berk Ustun and Cynthia Rudin. Learning optimized risk scores. *Journal of Machine Learning Research*, 20(150):1–75, 2019.

Berk Tevfik Berk Ustun. *Simple linear classifiers via discrete optimization: Learning certifiably optimal scoring systems for decision-making and risk assessment*. PhD thesis, Massachusetts Institute of Technology, 2017.

Tarmo Uustalu and Varmo Vene. Primitive (co) recursion and course-of-value (co) iteration, categorically. *Informatica*, 10(1):5–26, 1999.

Tarmo Uustalu, Varmo Vene, and Alberto Pardo. Recursion schemes from comonads. *Nordic Journal of Computing*, 8(3):366–390, 2001.

Pravin M Vaidya. Speeding-up linear programming using fast matrix multiplication. In *30th Annual Symposium on Foundations of Computer Science*, pages 332–337. IEEE Computer Society, 1989.

Vladimir Vapnik. *The nature of statistical learning theory*. Springer Science & Business Media, 1999.

Sicco Verwer and Yingqian Zhang. Learning optimal classification trees using a binary linear program formulation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1625–1632, 2019.

Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359, 1989.

Edward Wang, Riley Ballachay, Genpei Cai, Yankai Cao, and Heather L Trajano. Predicting xylose yield from prehydrolysis of hardwoods: A machine learning approach. *Frontiers in Chemical Engineering*, 4: 994428, 2022.

Wolfgang Wechler. *Universal algebra for computer scientists*, volume 25. Springer Science & Business Media, 2012.

Dominic JA Welsh. *Matroid theory*. Courier Corporation, 2010.

Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57: 509–533, 1935.

Darshana Chitraka Wickramarachchi, Blair Lennon Robertson, Marco Reale, Christopher John Price, and Jennifer Brown. Hhcart: An oblique decision tree. *Computational Statistics & Data Analysis*, 96:12–23, 2016.

Edwin B Wilson and Jane Worcester. The determination of ld 50 and its sampling error in bio-assay. *Proceedings of the National Academy of Sciences*, 29(2):79–85, 1943.

Zhixuan Yang and Nicolas Wu. Fantastic morphisms and where to find them: A guide to recursion schemes. In *International Conference on Mathematics of Program Construction*, pages 222–267. Springer, 2022.

Rui Zhang, Rui Xin, Margo Seltzer, and Cynthia Rudin. Optimal sparse regression trees. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 11270–11279, 2023.

Weixiong Zhang. *Branch-and-bound search algorithms and their computational complexity*. University of Southern California, Information Sciences Institute, 1996.

Index

A Haskell linear algebra functions

Here we list the `Linearsolve` module functions used to perform the linear algebra for the ICE algorithm.

```
module Linearsolve
(vecmult, matvecmult, matmult, linearsolve, triangular, rotatePivot,
   resubstitute, resubstitute', Matrix, Vector)
where
import Data.List
type Vector = [Double]
type Row    = [Double]
type Matrix = [Row]

vecmult::Vector -> Vector->Double
vecmult a b = sum (zipWith (*) a b)

matvecmult :: Matrix -> Vector -> Vector
matvecmult rows v = [sum (zipWith (*) row v) | row <- rows]

matmult :: Matrix -> Matrix -> Matrix
matmult a b
  | null a || null b = []
  | length (head a) /= length b = error "Incompatible␣matrices␣for␣
     multiplication"
  | otherwise = [[sum (zipWith (*) row col) | col<-(transpose b)] |
     row<-a]

linearsolve :: Matrix -> Vector -> Vector
linearsolve a b = x
 where
  b' = map (\y -> [y]) b
  a' = zipWith (++) a b'
  x  = resubstitute $ triangular a'


triangular :: Matrix -> Matrix
triangular [] = []
triangular m  = row:(triangular rows')
 where
  (row:rows) = rotatePivot m
  rows' = map f rows
  f bs
```

```
    | (head bs) == 0 = drop 1 bs
    | otherwise      = drop 1 $ zipWith (-) (map (*c) bs) row
    where
     c = (head row)/(head bs)


rotatePivot :: Matrix -> Matrix
rotatePivot (row:rows)
 | (head row) /= 0 = (row:rows)
 | otherwise       = rotatePivot (rows ++ [row])


resubstitute :: Matrix -> Vector
resubstitute = reverse . resubstitute' . reverse . map reverse


resubstitute' :: Matrix -> Vector
resubstitute' [] = []
resubstitute' (row:rows) = x:(resubstitute' rows')
 where
  x     = (head row)/(last row)
  rows' = map substituteUnknown rows
  substituteUnknown (a1:(a2:as')) = ((a1-x*a2):as')
```

B Proofs

B.1 Proof of correspondence between functional and relational algebra

Corollary 6. Given a functional algebra `alg :: func [a] -> [a]` and a relational algebra `algR :: func a -> a`. Assume the base functor is the cons-list `ListFr a`. We have following equality establish the connection between the functional algebra and the power transpose of the relational algebra `ΛalgR :: func a -> [a]`

$$\texttt{concat . map (ΛalgR . (Cons a)) = alg . (Cons a)} \tag{156}$$

Proof. we have following equational reasoning

```
  concat . map (ΛalgR . (Cons a))
```
≡ Λ-fusion law and `Cons a` is a function
```
  concat . map Λ(algR . Cons a)
```
≡ **E**`R = concat . map (ΛR)`, where `R` is a relation, **E** is the existential image functor
```
  E(algR . cons a)
```
≡ definition of **E**
```
 Λ( algR . Cons a . ∈)
```
≡ definition of the functor
```
 Λ( algR . fmap ∈ . Cons a)
```
≡ Eilenberg-Wright Lemma: `alg = Λ( algR . fmap ∈)`
```
 alg . (Cons a)
```

□

The formal definition of the existential image functor **E** and inclusion relation $\in$ can be found in Bird and De Moor [1996].

More generally, given a base functor **F**, we have

$$\texttt{concat . P (ΛalgR . F) = alg . F}. \tag{157}$$

The proof when the base functor **F** is defined by the join-list datatype, we have

$$\texttt{concat . crp (ΛalgR . F) = alg . F}$$

Proof. It can be proved by following

```
  concat . P (ΛalgR . F)
```
≡ Λ-fusion law and **F** is a function
```
  concat . P (Λ(algR . F))
```
≡ $\mathbf{P}f = \Lambda\,(f\,.\,\in)$, where f is a function
```
  concat . Λ(Λ(algR . F) . ∈)
```
≡ Λ-fusion law
```
  concat . Λ(ΛalgR . F . ∈)
```
≡ definition of the functor
```
 concat . Λ(Λ(algR . fmap ∈ . F))
```
≡ `fmap` $\in$. **F** is a function, and Λ-fusion law
```
 concat . Λ(ΛalgR . fmap ∈ . F))
```
≡ definition of the cross product `crp (f . F)` = Λ(`f . fmap` $\in$. **F**)
```
 concat . crp (ΛalgR . F)
```

□

B.2 Proof of nested combination generator

Given `dcombsKcombsAlg d k` defined as

```
<se d.kcsA d.func fst, kcsA k.(uncurry Join).<kcs k.!!d.kcsA d.func fst,kcsA k.func snd>>.
```
(158)

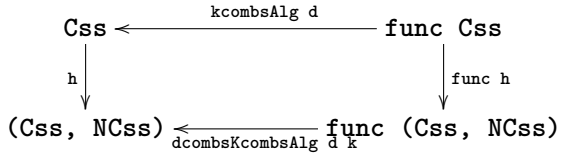
where `kcs`, `kcsA` are short for `kcombs`, `kcombsAlg`.

We need to verify the following fusion condition

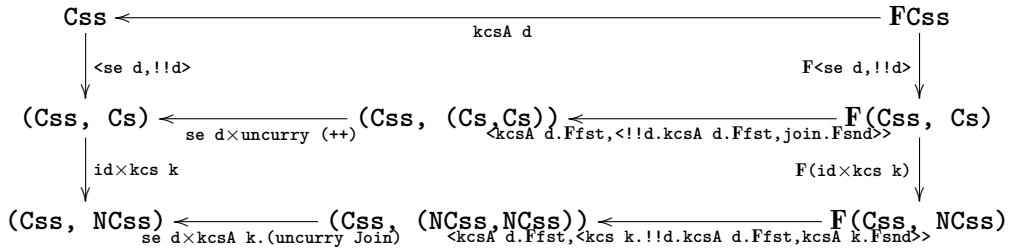
```
h . kcombsAlg d = dcombsKcombsAlg d k. func h,
```
(159)

where `h = <se d, kcombs k.(!!d)>`.

n other words, we need to prove that the following diagram commutes



However, proving that the above diagram commutes is challenging. Instead, we expand the diagram by presenting all intermediate stage explicitly. For brevity, we use symbol again **F** to replace with `func`, and `kcs`, `kcsA` are short for `kcombs`, `kcombsAlg`.



To prove the fusion condition, we first need to verify the two paths between **FCss** and `(Css, Cs)`. In other words, we need to prove

```
<se d,!!d>.kcsA d=se d×uncurry (++).<kcsA d.Ffst,<!!d.kcsA d.Ffst,join.Fsnd>>.F<se d,!!d>
```

This can be proved by following equational reasoning

```
se d×uncurry (++).<kcsA d.Ffst,<!!d.kcsA d.Ffst,join.Fsnd>>.F<se d,!!d>
```

≡ Product fusion and × absorption law

```
<se d.kcsA d.Fse d,uncurry (++).<!!d.kcsA d.Fse d,join.F!!d>>
```

≡ Definition of `se d`

```
<se d.kcsA d,uncurry (++).<!!d.kcsA d.Fse d),join.F!!d>>
```

≡ Definition of Combination

```
<se d,!!d>.kcsA d
```

where `join (Join a b) = a + b`. Note that, the equality between the third equation and the last equation is a assertion of fact, rather than a results can be proved (verified). This equivalence comes from the fact that size K-combinations can be constructed by joining all possible combinations of size i and size $K - i$ combinations, where $0 \leq i \leq K$.

Next, we prove the two paths between **F**`(Css, Cs)` and `(NCss, Cs)` are equivalent.

```
  <se d.kcsA d.Ffst, kcsA k.(uncurry Join).<kcs k.!!d.kcsA d.Ffst,kcsA k.Fsnd>>.Fid×kcs k
```

≡ Definition of ×

```
  <se d.kcsA d.Ffst, kcsA k.(uncurry Join).<kcs k.!!d.kcsA d.Ffst,kcsA k.Fsnd>>.F<fst,kcs k.snd>
```

≡ Product fusion

```
  <se d.kcsA d.Ffst, kcsA k.(uncurry Join).<kcs k.!!d.kcsA d.Ffst,kcsA k.F(kcs k.snd)>>
```

≡ Definition of catamorphism

```
  <se d.kcsA d.Ffst, kcsA k.(uncurry Join).<kcs k.!!d.kcsA d.Ffst,kcs k.In.snd)>>
```

≡ Definition of product

```
  <se d.kcsA d.Ffst, kcsA k.(uncurry Join).kcs k<!!d.kcsA d.Ffst,In.snd)>>
```

≡ Definition of `In` and functor

```
  <se d.kcsA d.Ffst, kcs k.(uncurry (++)).<!!d.kcsA d.Ffst,join.snd)>>
```