Recursive optimization: Exact and efficient combinatorial optimization algorithm design principles (DRAFT)

November 13, 2024

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* methods.

Our framework is grounded in Bird's theory of the algebra of programming, which is a relational formalism for deriving efficient algorithms from provably correct specifications. We present this theory in a novel way through Haskell code, focusing specifically on the context of combinatorial optimization. Additionally, we extend Bird's results by incorporating the *backtracking technique*, which integrates the branch-and-bound method into our framework, enabling a unified approach to 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. 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 essence of combinatorial problems, which allows us to significantly simplify the combinatorial complexity of the problem.

In addition to providing algorithm design principles. To demonstrate the effectiveness of our framework, in the *specialized theory* part, we address four fundamental problems in machine learning: *classification*, *clustering*, *decision tree*, and *empirical risk minimization for ReLU neural network*. We provide a detailed analysis of the combinatorial essence 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 available in terms of worst-case complexity, and their performance can be further sped up through the acceleration techniques we introduce. Finally, two example problems are selected to show *end-to-end implementations* in Haskell. We believe these implementations provide a clear demonstration of the effectiveness and practical applicability of our proposed framework.

Contents

Part I
Background

Our journey begins with a brief overview of machine learning and the 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 relies solely on the branch-and-bound (BnB) algorithms and mixed-integer programming 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 *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.

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 discussion 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 an intelligent machine which can think and act like the human species. Machine learning (ML) has been widely recognized as a subfield of AI, and 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 a revolutionized impact on almost any field where computation plays a role.

Scientists often assume that a stable underlying mechanism exists, which is organized to form nature [Pearl et al., 2016]. The main task of machine learning is to recover the underlying mechanism through an inductive learning process on a *finite* number of electronic data. The problems in ML involve human-labelled datasets are called *supervised learning problems*. Typical examples of supervised learning problems include classification and regression. The algorithms for the classification problem have output restricted to a finite set of values (usually a finite set of integers, called *labels*). The algorithms for regression problems have output restricted to numerical values like real values. On the other hand, problems involving unlabelled data are called *unsupervised learning problems*, the K-clustering problems and dimension reduction problems 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 the 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 prediction error [Mohri et al., 2018].

Objectives of machine learning problems and their optimization methods Nevertheless, due to the finite nature of 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. In optimization, solving a problem with a discrete objective is typically much more difficult than solving one with continuous objective. Therefore, for most of the machine learning problems, finding the ERM solution with a combinatorial objective is computationally intractable [Little, 2019].

Alternatively, the combinatorial objectives are replaced with the convex surrogate objective functions, which are differentiable and smooth. Then, various continuous optimization methods, such as gradient descent, can be applied. On the other hand, solving ML problems using *combinatorial optimization methods* is particularly scarce. This disparity arises from several factors:

1. **Problems are defined over continuous variables.** Although many ML problems possess a combinatorial nature, the variables used to optimize machine learning problems are defined over continuous variables [Bishop, 2006].

2. **Intractable combinatorics**. The combinatorics of many ML problems are exponentially large, 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**. Studies on 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) methods, the expectation-maximization (EM) algorithm, and the gradient descent algorithm.

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

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 partition given the specification of the clustering problem. 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 framework that offers a generic solution for designing practical algorithms for solving combinatorial machine learning problems exactly.

Machine learning problems concerned in this thesis The main subjects examined in 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 related to hyperplanes will help us gain a better understanding about the combinatorial essence of these PWL models.

Finally, due to the 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.

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 from George Dantzig [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 algorithms* will always select the *best solution* in the hypothesis set. In other words, a *provably exact algorithms cannot be improved in terms of accuracy.* In contrast, while approximate algorithms offer advantages in terms of efficiency and scalability, the potential risks associated with inaccuracies and lack of reliability must be carefully considered.

In most practical engineering applications, it might not important whether the global optimal solution is found, 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 have a polynomial-time worst-case guarantee, this trade-off can be managed with a high degree of precision. In contrast, this trade-off is generally not possible for most machine learning algorithms. 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 accuracy and 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 a trade-off between accuracy and interpretability—namely, that an interpretable model cannot surpass an uninterpretable one in 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 somewhat led to a more blind belief that there exists a trade-off between accuracy and interpretability. However, this is not necessarily true for problems that have structured data 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 linear model.

Does 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.

However, 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 (e.g., CART) demonstrate poor approximation to the ground truth. However, as training data increases, these approximate algorithms become as accurate out-of-sample as other methods [Bertsimas and Dunn, 2017].

In data-poor environments, 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, 2023] 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, the *general-purpose algorithms*[2] 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. This 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 so that no known algorithm can solve all instances of the problem in polynomial time.

However, for many ML problems, the problems specified for proving NP-hardness are not the same as their original definitions used in practical ML applications. Hence the polynomial-time algorithm do exist for many "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, 2023]. Similarly, the polynomial time algorithms for solving the K-means problem have also been developed [Inaba et al., 1994, Tîrnăucă et al., 2018]. If the 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

[2] general-purpose algorithms

complexity in the worst case and offer limited insight into the problem itself—can hinge 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 that must be addressed urgently:

1. **Lack of formal proof**. 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 CO algorithms should derive from an exhaustive search specification,

2. **Lack of systematic characterization**. Most existing general-purpose algorithms are designed in an ad-hoc manner through intuitions. The insights obtained from one particular problem are very hard to be used for another.

3. **No worst-case guarantee**. The worst-case time and space complexity analysis is rarely reported in studies of BnB algorithms. Indeed, many of these general-purpose algorithms exhibit exponential 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. **Parallel implementation**. 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 exists. An embarrassingly parallel program might be the only feasible approach for solving large-scale problems. However, such powerful techniques are rarely discussed or utilized in the study of these general-purpose algorithms. Indeed, the embarrassingly parallel implementation is very difficult to achieve for algorithms based on *generative recursions* (see Subsection I.2.3.2 for definition), such as *cutting-plane algorithms*.

5. **Acceleration techniques**. Although many acceleration techniques, such as upper bound/lower bound, 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. **Flexibility to incorporate constraints**. 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 CO algorithms in a broader optimization context. We refer to this framework as the *Recursive Optimization Framework* (ROF).

We believe that, by showing many seemingly intractable or even NP-hard problems can be solved exactly in polynomial time (with some fixed parameters), will illuminate the path to designing reliable and tractable CO algorithms that are 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 $\mathcal{D}$ 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 associated with a unique *true label* $t_n : \mathbb{T}$. For regression task 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 in a *combinatorial search space* $\mathcal{S}$. The task of a *learner* (algorithm) is to use dataset $\mathcal{D}$ to 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 $l : \mathbb{R}^D \times \mathbb{T} \times \mathbb{R}^M \times \mathcal{S} \times \Theta \to \mathbb{R}$, and $\theta : \Theta$ is the model *hyperparameter*. 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 *surjective 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)

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

where $p : \mathcal{S} \to \mathbb{B}$ is a *predicate function*, it 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 different styles. 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

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

where the generator function $gen : \mathcal{D} \to [\mathcal{S}]$, 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 $gen\left(n\right)$, $\forall n \in \mathcal{N}$. The *evaluator* $eval_E : \left[\left[\mathbb{R}^D\right]\right] \to \left[\left(\left[\mathbb{R}^D\right], \mathbb{R}\right)\right]$ computes the objective values $r = E\left(s\right)$ for all configurations s generated by $gen\left(n\right)$ 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 returns true by 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 E.

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 of the use of *currying* in point-free programming, the function can be *partially applied*, and the type of the input can be inferred from the type declaration. 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 *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 efficient CO algorithms from provably correct specification (4), thus the correctness of the algorithm is assured yet ensuring efficiency. Furthermore, classical combinatorial optimization algorithms, such as greedy algorithm, dynamic programming (DP), divide-and-conquer (D&C), branch-and-bound (BnB) are unified in 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 CO algorithms from provably correct specifications, avoiding tedious induction proofs or informal explanations.

I.2.1.2 Combinatorial generation and combinatorial optimization

Following the above discussion, (4) presents a universal approach for solving any COP, 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 found that once we have *efficient brute-force algorithm*, the efficient CO algorithm for this problem will follows 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 horribly inefficient, especially for large-scale problems, as they fail to use any structure or heuristics to reduce the search space. The key insight is that both the efficient generator and the CO algorithm for a problem often require exploring the same principles but within different algebraic structures. We hope this concern will be addressed through the discussions in this thesis.

One notable principle is *distributivity*, in its simplest form, states that $ab + ac = a \times (b + c)$. The solution of 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 property $\min x + \min y = \min \{a + b \mid a \in x \wedge b \in y\}$. Assume x, and y are length N lists, the right side involves N^2 arithmetic operations whereas the left side involves only $2N + 1$ computation. 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 the generalization of distributivity—monotonicity—plays a crucial role in designing efficient exhaustive search algorithms, and consequently, efficient combinatorial optimization (CO) 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. In contrast, D&C method 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 CO 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 CO generator, which is often simpler than directly designing an efficient CO algorithm

2. **Reusability**: Since many COPs share similar combinatorial structures, a generic combinatorial generator can be reused for various problems, whereas CO 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 properties. This flexibility allows for the design of *tailored* CO 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), meaning that the amount of computation is proportional to the number of objects listed. In other words, each configuration should take constant time to generate. The smaller this constant, the more efficient the generator is.

The key to constructing a CAT generator lies in identifying common substructures shared among configurations, which involves a *efficient factorization* to 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. In Section II.1.2, we will first introduce a range of efficient SDP-based generators, including those for such as permutations, subsets, K-permutations, K-combinations (K-subsets), sequences, and partitions (segmentation). These generators will serve as a comprehensive library that can be directly applied to future 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 its basic concept to provide the audience with some intuition before illustrating a more detailed 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 vaguely characterizes the SDP as the general problem of sequential choice among several actions. De Moor [1995] gives a more precise definition of what is SDP, the sequential nature is captured by expressing it as an instance of the operator `fold`. Readers who are not familiar with functional programming may not familiar with this operator as well. In Haskell, `foldr` is defined as

```
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` ``[3], the function `foldr` traverses the list `(a:xs) = [a,a1,..,an]`(`:` is the operator that append a element to the list) recursively from right to left starting with the seed value `e`. In other words, `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 extensions of the SDP.

We have noted that greedy algorithms, DP, and the BnB method can all be characterized as SDP. However, there are several characteristics 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 time and again 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`. In contrast, many DP problems involve *memorization* techniques that retain results from several previous steps. For instance, the Fibonacci sequence DP recursion $f(n) = f(n-1) + f(n-2)$, which depends on previous step $f(n-1)$ and previous two step $f(n-2)$.

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 *catamorphism*, along with its extensions. Catamorphisms allow programmers to write statically-checkable generic shape-dependent 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 natural number $\mathbb{N}$ 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 appending new values to an empty list. In fact, both can be defined inductively through polynomial functors.

Furthermore, in Section II.2.7, we will show that different search strategies used in the BnB method can be derived from the specification of catamorphisms. This provides formal proof of why different search strategies that used in BnB algorithm 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 CO methods are commonly understood. In the second Subsection, we summarize the key components of designing efficient CO algorithms, with efficiency being our sole concern. In the third Subsection, we summarize how these different CO 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 are often written in a style like 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 different 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.

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

Greedy method Greedy method are widely used in many different COPs. 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 method do not always yield optimal solutions, but for problems that satisfy the *greedy condition*, 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]. 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 COP 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 its perceived impressiveness. 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].

The applications of DP have been found in numerous fields. Classical DP algorithms like 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 essence of DP. Many believe they understand dynamic programming, yet their definitions 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 of the principle of optimality.

> **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, several Terminology, such as "policy", "decisions", and "initial states" are not rigorously defined. Unitil 1967, 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.

Until today, it is widely accepted that an efficient recursive program incorporated with the principle of optimality is insufficient to become a DP algorithm The 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 regarding 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 (SDP) is characterized as a *step-by-step* recursive formula, it is a recursion that has access only to the previous step. In other words, the use of memoization is not amenable in 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*[4] are widely accepted as the DP algorithms, but there is no memorization technique 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 is well-known 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 essence of DP was first rigorously outlined by Karp and Held [1967], in their work, Dijkstra's algorithm is considered the prototype DP algorithm for illustration. 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

[4]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.

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. Despite these arguments, the community has not reached a consistent agreement on this matter, and the debate has persisted for almost a century.

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 subdivide recursively until the sub-problems are small enough (base cases), that they are solved directly. Then the solutions to the sub-problems are 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 is serving to reduce the running time to a lower polynomial.

In the later Subsection II.2.4.3, we will provide an alternative definition for the D&C method, instead of splitting the problem into two equal halves, we consider all possible subdivisions of a problem. This approach allows us to relate to perhaps the most 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 COPs. This method relies on the principle of decomposing a complex problem into smaller subproblems through branching rules, analogous to 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. 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 time at 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.2.7, 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—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 misapplication leads to the misconception that the BnB method is exceedingly general. Moreover, some literature also classified 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., catamorphism. This approach is essential for the study of BnB algorithms, especially 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.2.7.

I.2.2.2 Summary of key components in designing efficient combinatorial algorithms

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

1. **Identify combinatorics through geometry.** For most machine learning 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 $\mathcal{D}$.

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 utilize 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 the computations*. *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 CO algorithms can hence be derived by verifying the conditions of these theorems.

4. **Dominance relation.** The dominance relation, often employed through 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**. Thinning is a (relational or functional) program used to delete provably suboptimal configurations and is parameterized by the dominance relations. It is roughly equivalent to what many other 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 the thinning approach 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 implemented 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*: the extending ordering is also known as search strategy in BnB studies, for a list of configurations, extending ordering determines the sequence in which configurations are extended. Well-known examples are depth-fisrt search and breadth-first search strategies. 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.

Our discussions here are solely on improving time efficiency. However, in solving intractable COPs, 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.6 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 method, are often treated as distinct approaches, they are closely related to each other. In our framework, these methods are related based on their abstraction levels. We summarize their inclusion relationships as follows:

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

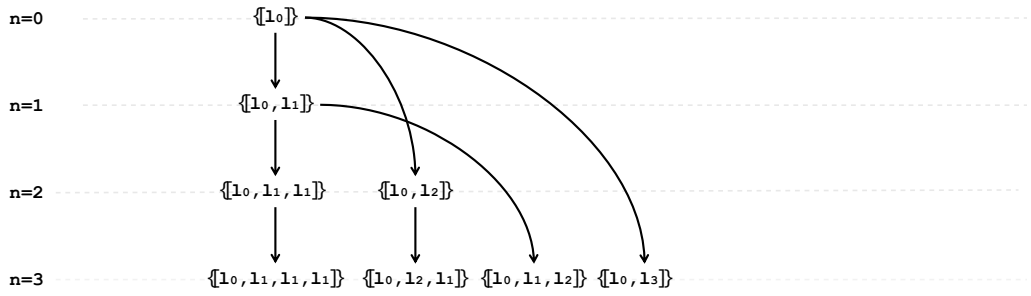


Figure I.2.1: Rod-cutting problem generation tree. IIn this generation tree, the segments at each level depend on the segments from all preceding levels. For example, the length-three segments (configurations at level $n = 3$) are constructed using length-two, length-one, and empty segments. Configurations 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 the thinning process and alternative search strategies. The inclusion relations 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 efficient dynamic programming algorithm from scratch

After introducing the key components in designing efficient algorithms, to give audiences some intuition about how to design algorithms, 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 to provide 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.* The rod-cutting problem is a classical combinatorial optimization problem that can be solved efficiently using a dynamic programming algorithm. Assume we have a rod of length N that we want to cut into pieces. Pieces with different lengths are worth different amount 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 amounts 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 $\mathcal{S}_n$ be all possible rod pieces of a length n rod. The all possible rod pieces for length three rod consist of

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

we then establish the following equivalence relations

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

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. I.2.1.

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. I.2.1 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*, 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 2 $\{[l_0,l_1,l_1],[l_0,l_2]\}$ 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.

Analogue 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 explores 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*. Indeed, Little et al. [2024] have shown that we can freely swap any semirings if we have a recursion has type information similar to 8, this is a consequence of Wadler [1989]'s *free theorem*. Little et al. [2024]'s result is based on the distributivity of semiring, 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 the 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. Recursions are well-studied in the functional programming community, particularly in the study of *structured recursion schemes*. 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 development 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*" is usually referred to as a function or a process that is defined by 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 or the input data. The Fibonacci sequence recursion (11) above is an example of the structured recursion. The structured recursions includes almost all recursion with tree structure, binary tree search or 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 $r = a \bmod b$ 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 Development history of constructive algorithmics

In the 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 Following the pioneering work of Meertens [1986] and Bird [1987, 1989], they form a new programming formalism for structuring data and the method for processing these data. This formalism is the calculus built around recursive/corecursive datatypes and homomorphisms on those datatypes. These datatypes are typically various forms of trees or datatypes similar to trees (for example, nature 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 the *initiality* to prove the equality of the program rather than tedious induction. It is later known as *constructive algorithmics*, *the Bird-Meertens formalism*, or *Squiggle formalism*, due to its lavish use of squiggly notation.

Initial algebras, recursive datatypes and catamorphisms Datatypes such as *Nat*, *List*, and *Btree* look very similar, and their homomorphism has the same recursive structures. It is reasonable to consider using abstract language to generalize all these datatypes. In 1990, with the help of category theory, Malcolm [1990] found that these datatypes can be modeled by the *initial algebra* in the category of **F**-algebras. 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 *corecursive (coinductive) datatypes* (conatural numbers, colists, streams, etc.) [Fokkinga, 1992].

The homomorphisms between the *initial* **F**-algebra and **F**-algebras are called *catamorphisms*, it is named by Meertens [1988]. The initial algebras are the abstraction of recursive datatypes, hence the catamorphisms are the abstraction of the homomorphisms of different datatypes, in other words, catamorphisms are the abstraction of fold operator. Indeed, 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] generalize the Bird-Meertens formalism to relations, specifications are viewed as input–output *relations* (non-deterministic functions) instead of *total functions*. This is later known as *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 scheme - 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, or SDPs, is characterized by *catamorphisms*.

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, *paramorphism*, on the other hand, 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. *Zygomorphism* is a variation of catamorphism 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 essence of *divide-and-conquer*. A more formal characterization of this approach will be provided after we introduce 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 scheme The various generalizations of catamorphism can be perplexing to the uninitiated, as many of these morphisms appear quite similar. Numerous research efforts have been made 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] made another attempt at 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 like C++, Python, or MATLAB, it offers several notable advantages. Its strongly-typed system helps eliminate side effects and ensures code correctness, while its support for functional composition and currying results in more concise and readable code. 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 [Program, 2013]. Category theory is an abstraction about compositions 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 the 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. The formal definition of category is defined 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 identity morphism.
- For each pair of morphisms f, g, such that $t(f) = s(g)$ there exists a specified morphism $f \circ g$, called the composite morphism, such that $s(g \circ f) = s(f)$ and $t(g \circ f) = t(f)$. Or graphically

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

The compositions satisfy the following properties

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

Objects in category theory are abstract nebulous entities. All you can ever know about it 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 known as *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. It is self-evident that programming is closely related to category theory. An example of a category is the *fiction category* **Hask** by considering the functional programming language Haskell as a category. Haskell itself 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. Despite this limitation, the analogy remains useful for understanding the basic concepts of category theory.

Categories itself can be considered as objects in bigger categories, this is a 2-*category*, known as *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:

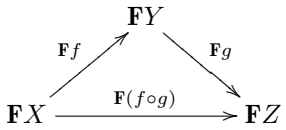
- For each object X of $\mathcal{C}$, an object $\mathbf{F}X$ of $\mathcal{D}$
- For each morphism $X \to Y$ of $\mathcal{C}$, a morphism $\mathbf{F}f : \mathtt{F}X \to \mathbf{F}Y$ of $\mathcal{D}$

and the following functoriality axioms hold:

- Unitality : for every object X of $\mathcal{C}$, $\mathbf{F}\left(id_X\right) = id_{\mathbf{F}X}$, where id_X is the identity function for object X in category $\mathcal{C}$ and $id_{\mathbf{F}X}$ is the identity function for object $\mathbf{F}X$ in category $\mathcal{D}$
- Compositionality: for every pair of composable morphisms f and g

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

in $\mathcal{C}$ we have $\mathbf{F}\left(f \circ g\right) = \mathbf{F}f \circ \mathbf{F}g$. That is the following diagram commute



A functor between categories $\mathcal{C}$ and $\mathcal{D}$ must give an object of $\mathcal{D}$ for every object of $\mathcal{C}$. This thinking leads us to consider how to construct new objects from given ones. In the realm of programming, this is referred to as "*type constructors*." A type constructor is a model for a *datatype* or *data structure*. Functors are a special kind of type constructor that operate on both types and functions: they 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 explore how category theory assists in constructing algebraic datatypes by using *polynomial functors*. Further, the fundamental relationships between functors are known as *natural transformation*s. 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 relationship (*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 -> ()`. 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 coming to it from any object in the category. We normally use "0" to represent an initial object, and "1" to represent a terminal object.

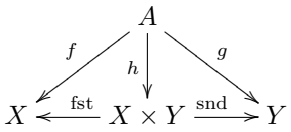
The initial object is defined by its *mapping out* property. For instance, in a *partially ordered set* (poset) category, an initial object 0 is the smallest element, because there is only one morphism from $0 \to A$ for any A in the *poset*

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 the 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, we need to think in terms of relationships. Analogue to Cartesian product of sets, if we want to define a function $f : A \to X \times Y$, we can define a pair of function $f_X : A \to X$ 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.

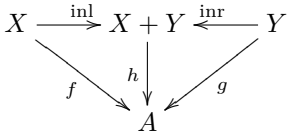
Definition 3. *Product.* Let X and X be objects in a category $\mathcal{C}$. A product of x and y consists of three things: an object, denoted $X \times Y$ and two morphisms $\text{fst} : X \times Y \to X$ and $\text{snd} : X \times Y \to Y$, with the following *universal property*: For any other such 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 coproduct with the mapping-out property.

Definition 4. *Coproduct.* Let X and Y be objects in a category $\mathcal{C}$. A product 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 category of things that expression fits. The expression `True` is a Boolean type `Bool`, `1` is an integer type `Int`, 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 normal math-style. However, it is worth noting that both type variables and value variables in function definitions are represented in 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
```

For the uninitiated reader, there is no need to be panic. Although both the type variables and value variables of this function are denoted using lower-case letters, we can easily distinguish them. To clarify, everything following `::` are types or type variables, while variables directly applied to a function after white space represent value variables.

List and tuple List is a *homogenous data structure*. 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 `:` 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. 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)`. However, it is important to note that 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, 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 capitalized.

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, called *the field* of `Maybe` type constructor. when I say fields, I actually mean parameters. In this case, the variable of this datatype can be any type, for instance, field `x` can be can be `Int`, or `String`, corresponding to type `Maybe Int`, or `Maybe String`, 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, we will explain it in more detail when we use it.

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

```
data Either Int 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 just *constant variables*. If this constant field is unknown, we denoted it as `a` to distinguish the free field `x`.

Type synonyms do not really do anything, they are just about giving some types of different names so that they make more sense to someone reading our code and documentation. In Haskell preclude, the `String` is a synonym for a list of characters `[Char]`

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

Haskell has type a inference system, which means that if we write a number, we do not have to tell Haskell it is a number. Haskell can infer any missing type information where possible, and then we do not have to explicitly write out the types of our functions or expressions to get things right. Therefore, if the codomain of a function `f` matches another function `g`'s domain, we can compose it without claiming their type.

Haskell pattern matching and map, 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 you to define separate function bodies for different patterns. This leads to really neat code that is more readable. Here is an example to define a useful function called `map` by using pattern matching

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

`_` means 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 you call function `map`, the definition patterns will be checked from top to bottom, and when input conforms to a 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 in the way commonly done in imperative languages, such as `f(x,y)`. Instead, all functions in Haskell that appear to take several parameters are actually *curried functions*.

To understand 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 *partial 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 imperative programming and 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, you can use `a `+` b`, `a `*` b`, `a `max` b` to write binary functions infix. Any function in prefix form can also be partially applied by using a *section*. To section a function, you surround it with parentheses and supply 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 describing its application and arguments.

```
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` using a point-free style.

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

the function composition is `.` 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 already included in Haskell, 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 example

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

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

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

List comprehension In set theory, a set can be constructed from another set, this is known as *set comprehension*, 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. In Haskell, list comprehension is usually rendered as

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

it produces a list of values of the form `f x` , the operator `f` is drawn from a list of functions `fs` and the element `x` is drawn from a list `xs`. For instance, expression `[f x | x <- [1,2,3], f <- [(+1),(+2)]` produce a 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*; "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]`. As previously mentioned, every natural transformation is a polymorphic function.

Functors in Haskell In Haskell, functors are defined as endofunctors within the category **Hask**. Categorically, a functor is a mapping between two categories 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 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 connects $\mathbf{F}A$ and $\mathbf{F}B$. In Haskell, we define the morphisms part of the functor as

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

where `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`.

Defining `fmap` for each new datatype can be tedious. Fortunately, we can use the `deriving` keywords 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 defining

```
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 algorithms, and geometry in machine learning.

Constructing efficient exhaustive search algorithms – Combinatorial generation

As discussed, designing an efficient CO algorithm hinges on developing effective brute-force algorithms, and the 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, the 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 elegant 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.2.2 and Section I.2.1 will be explained more formally. There are several focuses in this Chapter, the first part focuses 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 programs*, will be explored with a particular focus on its application to the context of this thesis—combinatorial optimization. This formalism offers a systematic approach to deriving efficient programs from provably correct specifications, providing us with simple, elegant derivation steps. We will see in our applications that the algorithms designed using this formalism are *concise* and *readily comprehensible*.

Combinatorail geometry – Geometric algorithms and combinatorial essences of ML models

In the final theme, Geometry, we highlight two key aspects: how geometric insights can simplify the combinatorial complexity of intractable problems, and how the algorithmic design principles we introduced 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*. In 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 closely tied to many machine learning models, as nearly all successful ML models can be represented as piecewise linear functions. Understanding these properties is crucial for grasping the combinatorial essence 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 combinatorial essence 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 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 CO algorithms for these problems can be derived directly by combining the basic combinatorial generators introduced in the general theory. Furthermore, we have also discussed 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 problem

In this chapter, 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 $O\left(N^D\right)$. 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 $G = \left(\begin{array}{c} D+W \\ D \end{array} \right) - 1$, and W is the degree of the polynomial used to define the hypersurface.

Empirical risk minimization for feedforward neural networks with rectified linear unite

In this chapter, 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 complexity of $O\left(N^{DK}\right)$, where K denotes the number of hidden nodes. 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.

Decision tree problems

We propose a unified algorithmic process for solving decision tree problems that can handle *axis-parallel splits*, *hyperplane splits*, and *polynomial hypersurface splits*. The worst-case complexity of the proposed algorithms is as follows: 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 this Chapter, 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 two example problems: the 0-1 loss linear classification problem and the K-medoids problem. We provide comprehensive implementations in Haskell, along with detailed experiments. These examples encompass most of the techniques introduced in general theory.

I.4 Contributions

Contributions in general theory In the first theme, combinatorial generation, we provide a comprehensive summary of efficient combinatorial generators for a wide range of fundamental combinatorial structures and make progress in designing a new class of generator, called the integer sequential decision process generator, by integrating the strengths of generators in different classes. These generators will form a comprehensive library, applicable directly to solving optimization problems.

In the second theme, We extend Bird's results by incorporating the backtracking technique, thereby integrating the well-known branch-and-bound method into our framework, which allows us to present a unified framework for designing greedy, dynamic programming, divide-and-conquer, and branch-and-bound algorithms. In addition to reformulating the combinatorial generators introduced in the first theme using a datatype-generic approach, we introduces two new generators, `kcombs` and `kperms` based on join-list datatype, for enumerating K-combinations and K-permutations, both of which are embarrassingly parallelizable. Alongside this, we establish systematic principles for designing catamorphism generators, demonstrating how these SDP generators can serve as foundational "atoms" for constructing more complex "compounds".

In the last theme, combinatorial geometry, we demonstrate two key aspects: how geometric insights can simplify the combinatorial complexity of many intractable COPs, and how the algorithmic design principles we introduced assist in solving fundamental geometric problems. We have made several significant contributions in this section, one of the most important being the introduction of novel algorithms for enumerating the cells of a hyperplane arrangement—a fundamental problem in combinatorial geometry. Our algorithms are both optimally efficient and embarrassingly parallelizable. Furthermore, we explore the geometric and combinatorial properties of two important geometric objects: Voronoi diagrams and hyperplanes. These properties are essential for understanding the combinatorial essence of many fundamental machine learning problems, which we will examine in detail in the third part of this thesis.

Contributions in specialized theory The specialized theory section of this thesis addresses four fundamental problems in machine learning: classification, clustering, decision tree, and empirical risk minimization for ReLU neural network. Exact solutions for all these problems are known to be NP-hard. We provide a detailed analysis of the combinatorial essence of these NP-hard problems, demonstrating that these problems can be solved in polynomial time when certain parameters, such as dimensionality or number of branches, are fixed. To the best of our knowledge, all algorithms we propose are the fastest available in terms of worst-case complexity, and their performance can be further sped up through the acceleration techniques we introduce, thus improving their efficiency beyond the worst-case complexity bounds.

Part II
General theory: Principles for designing efficient combinatorial optimization algorithms

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

In the first theme, **Combinatorial generation**, we focus 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, we will explore 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, *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, underscoring the elegance and power of systematic algorithmic derivation.

Finally, the third theme, **Combinatorial geometry**, explores the combinatorial essence of many machine learning models by identifying the 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 robust framework for understanding and advancing the design of algorithms at the intersection of combinatorial optimization and machine learning, offering both theoretical insights and 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 types of combinatorial generators: sequential decision process, *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 of Chapter II.2. Moreover, We will see how these basic SDP generators can serve as the "atoms" for constructing more complex "compounds" in Subsection II.2.3.4.

II.1.1 Containers/datatypes

Combinatorial structures are consists of by a sequence/set of elements. The containers to store these combinatorial structures are called *datatypes*. For readers who are computer science background may very familiar with *data structures*. Datatypes are *abstraction* of data structures, each datatypes 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.

Among the numerous datatypes, we are particularly interested in a class of datatypes, known as the *Boom-hierarchy family* [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 especially significant for combinatorial generation.

In this section, we briefly introduce several datatypes that are prominently used 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 family is the free algebra of its binary operator $\cup$, called *"join"*, and unit $\emptyset$ [5], called "*empty*." Different datatypes are distinguished by the laws satisfied by their join operator $\cup$. The four laws considered in the Boom-hierarchy family are

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

We can classify different datatypes by identifying the laws of its binary operator $\cup$ satisfied, there are four laws in total, hence there are 2^4 number of datatypes in the Boom hierarchy family. Adding a law to an algebra can be thought of as partitioning the carrier of the algebra into equivalence classes induced by that law and regarding each class as one element [Bunkenburg, 1994]. For instance, set concatenation has commutativity, hence $\{1, 2\}$ and $\{2, 1\}$ belong to the same equivalence classes.

Lists/sequences A (finite) list or sequence is an ordered collection of values of the same type which are called the *elements/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 $x \cup y \neq y \cup x$, for all lists x, y.

In many imperative programming languages, like 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 `[]`.

[5] 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.

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 list. 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 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 process for basic combinatorial structures

The combinatorial generator based on the Sequential Decision Process (SDP) is 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 decision nature, and nearly all SDP-based generators are *optimally efficient* in terms of asymptotic complexity. Furthermore, SDPs can be generalized to offer a data-type 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 sequential decision process 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

```
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`[6] 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 filter out all configuration that do not satisfy the predicate.

When a filtering process is needed, directly use `sdp_gen` program is inefficient because the configurations returned by `foldl` are usually exponential large. In some applications, we can sometimes *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, the filter fusion is possible if and only if the predicate `p` is *prefix-closed.* We call a predicate `p` prefix-closed if `p (f x a) = q (f x a) && p x` [De Moor, 1995], where the original predicate `p` with respect to updated configuration `f x a` is true if and only if the new 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 literature [Bird, 1987]. In practice, calculating `q (f a x)` is often more efficient than directly calculating `p (f a x)`. For instance, the well-known eight queens problem has predicate `p` check no queen is attacked by any other queens, whereas the predicate `q` check whether the newly added queen does not attack the others.

II.1.2.2 Sublists, sequence and K-sublists

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

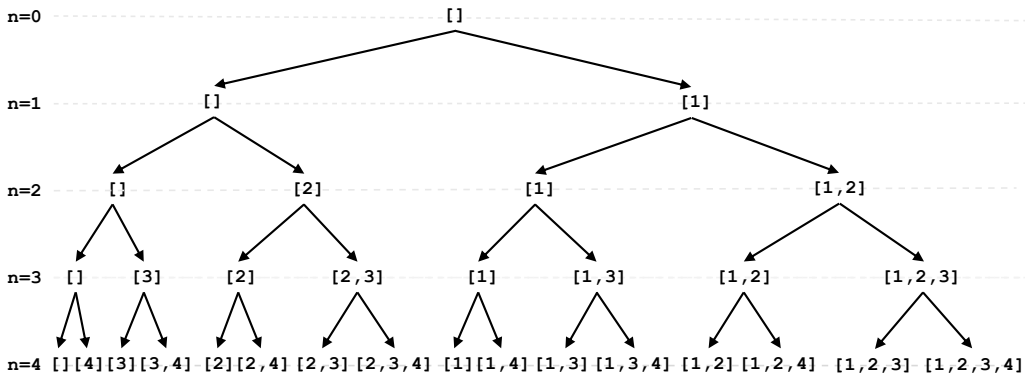


Figure II.1.1: The generation tree for a sequential decision process (SDP) sublist generator with the input $[1, 2, 3, 4]$. At each recursive step, we either append a new element to the end of each partial configuration (sublist) or leave the configuration unchanged.

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. We denote $\mathcal{S}_{\text{subs}}$ as the set of all sublists. 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 functions

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

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

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

The SDP generator for sublists is then rendered as

```
sdp_subs :: Num a => [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. II.1.1 draws the generation tree for `sdp_subs [1,2,3]`.

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

```
sdp_seq :: Num a => [a] -> [[a]]
sdp_seq = sdp_gen seq_p seq_fs seq_e
 where
  seq_fs = [append]
```

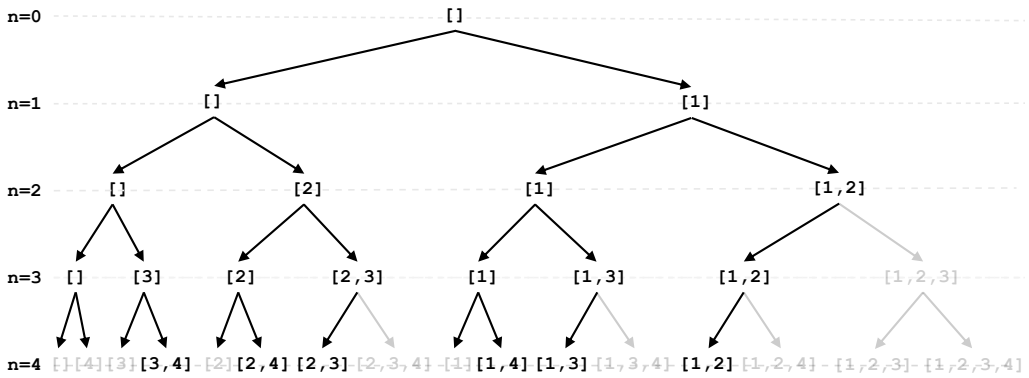


Figure II.1.2: The sequential decision process generation tree for the 2-combination of list $[1, 2, 3, 4]$. The shaded sublists are filtered sublists, violate the predicate `ksub_q = (k >=) . length`.

```
seq_p = const True
seq_e = [[]]
```

evaluate `sdp_seq [1,2,3]` will returns `[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. We can achieve this by combining the sequence generator with the configuration generator of the problem using the "tupling" method.

K-sublists The terms "K-sublists" and "K-combinations" are synonymous, both representing sublists of size K. We have developed two types of generators for generating 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 $\left(\begin{array}{c} N \\ K \end{array} \right)$, which is defined as

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

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** prefixed-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 we can construct 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_subs' 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. II.1.2.

II.1.2.3 Assignments

Binary assignments There exist 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 equal 1 and 0 otherwise. The binary assignments are also called the *characteristic vector* of a sublist. 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 Multiary assignments are a direct generalization of binary assignments. Instead of having 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 list, there are M possible choices, resulting in a total of M^N possible assignments. We denote the set of all multiary assignments as $\mathcal{S}_{\text{masgns}}$.

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

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

Generating permutations can be done by inserting a new element into the existing partial permutations. For instance, we can inserting 3 to partial permutation $[1,2]$ and $[2,1]$, we obtain all possible permutations $[[3,1,2], [1,3,2], [1,2,3], [3,2,1], [2,3,1], [2,1,3]]$ for 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)
```

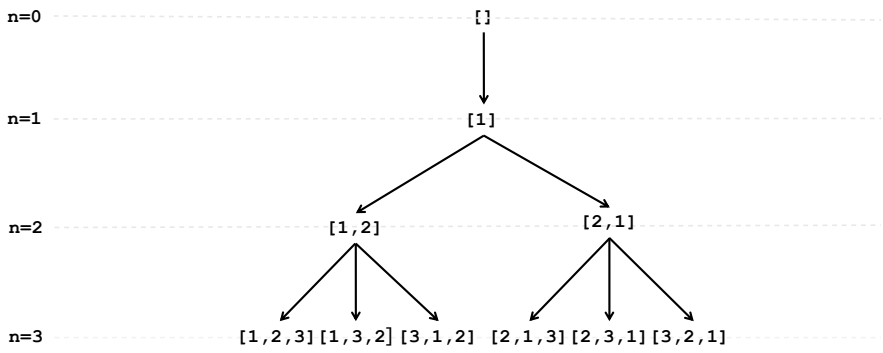


Figure II.1.3: The sequential decision process generation tree for the permutations of list $[1, 2, 3]$. In each recursive step, a new element is inserted into every possible position between elements. For a list of length n list, there are $n + 1$possible choice of insertion.

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 recursively. We can define the decision functions for the permutations generator as being parameterized by an integer `k`

```
perms_fs :: Int -> [a -> [a] -> [a]]
perms_fs k = [insertKth i | i <- [0..k]]
```

In contrast, our previous `choice` function could only define fixed decision functions `fs`. To accommodate varying decision functions recursively, 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 $ length (xs!!0)]
```

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
```

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. II.1.3 draws the generation tree for `sdp_perms [1,2,3]`.

II.1.2.5 K-permutations

K-permutations is nothing more than a combination of permutation and K-combinations. In other words, the K-permutations of a list are the rearrangement of the size K sublists. It has many applications in ML research, such as the decision tree problem [Hu et al., 2019] and rule list [Angelino et al., 2018].

The most common 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 $\mathcal{S}_{\text{kperms}}$.

This selection process leads to a new definition for the choice function. In this case, we do not merely apply the decision functions to each configuration. Instead, at each step, we need to select elements from the input list that have not been previously selected. This requires checking if the newly selected element already exists in the current configuration. We can define a relation as follows

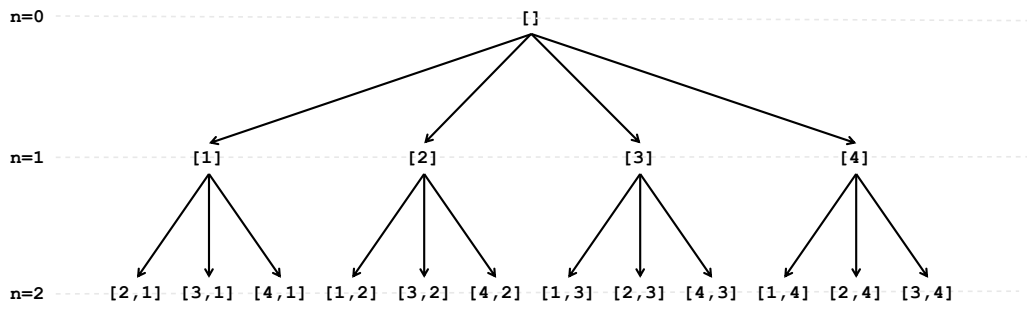


Figure II.1.4: The sequential decision process generation tree for the 2-permutations of list $[1,2,3,4]$. This process recursively selects new elements from the input list and appends them to the end of the current configuration. If the current configuration has n elements, there are $N-n$ possible choice of selection.

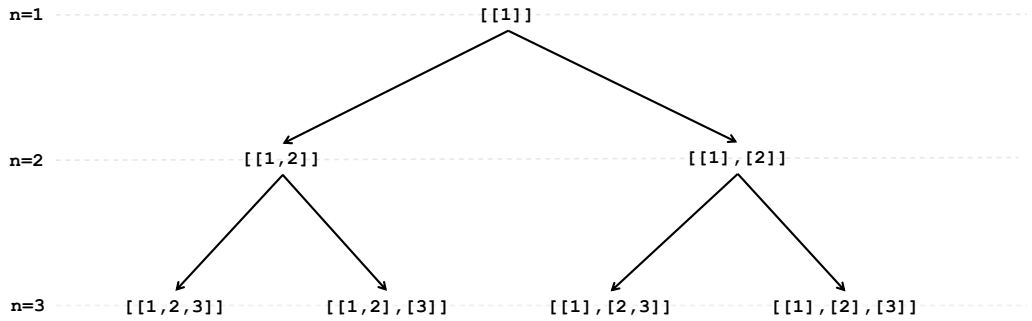


Figure II.1.5: The sequential decision process generation tree for the list partitions of list $[1,2,3]$. In each recursive step, a new element is either appended to the last element of a configuration or used to create a singleton list containing this new element.

```
r :: Eq a => a -> [a] -> Bool
r i y = all (\b -> b /= i) y
```

This function checks if element `i` exist in configuration `y`. We can now define the new choice function `choice3` and the new SDP generator `sdp_gen3` as

```
sdp_gen3 k p e x = foldl (choice3 x) e (take k x)
 where choice3 x ys a = filter p [y ++ [a] | a <- x, y <- ys, r a y]
```

where the operation `(++ [a])` can be understood as our decision function `f`, and we only apply it when `r a y` equals `True`.

Now we can define the K-permutations SDP generator as

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

Evaluating `sdp_kperms 2 [1,2,3,4]` returns `[[2,1],[3,1],[1,2],[3,2],[1,3],[2,3]]`. Fig. II.1.4 draws the generation tree for `sdp_kperms 2 [1,2,3,4]`.

II.1.2.6 List partitions

A partition of a list is to divide this list into non-empty consecutive 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 continuous 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 | 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]]

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

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, extent]
  parts_p = const True
  parts_e x = [[take 1 x]]
```

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

Fig. II.1.5 draws the generation tree for `sdp_parts [1,2,3]`.

II.1.3 Lexicographic generation

Generating combinatorial structures based on SDPs is not very common in the study of combinatorial generation. A significant proportion of research in this field focuses on generating combinatorial configurations by assuming an intrinsic *ranking* or *ordering* among all configurations. The most common ordering is to sort the configurations based on some ordering. The most common one is lexicographic ordering, which is based on the familiar concept of the ordering of words in dictionaries. We use the symbol $\prec_l$ to represent the lexicographic ordering between values, and $<_l$ to denote the lexicographic ordering between configurations. We now define what is lexicographic ordering formally.

Definition 5. *Lexicographic ordering.* In lexicographic order we have $[a_1, a_2, \ldots, a_N] <_l [b_1, b_2, \ldots, b_M]$ if either

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

Once we have an ordering among configurations, we immediately can define a *ranking function* and its inverse *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 $\mathcal{R} = \{1, 2, \ldots, 2^N\}$.

The primary use of a lexicographical generator is to generate random configurations with equal probability based on their rank. Thus lexicographical generators are designed for random generation rather than for generating 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 extensive 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.

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 (bit 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*.

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*. Research on combinatorics generation with minimal change ordering is referred to as *combinatorial Gray codes* (CGCs). Minimal change ordering aligns with the principle of optimality but within the context of combinatorial generation. Combinatorial Gray codes 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

In combinatorial optimization, randomly generating a few configurations using a lexicographic generator cannot guarantee exactness. 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 $[\,]$, $[1]$, $[1,2]$, $[1,2,3]$, $[1,3]$, $[2]$, $[2,3]$, $[3]$. 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 explores the concept of *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 [7] between them.

Minimal change ordering organizes configurations so that, at each recursive step, only one new element is added to 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 crucial.

There is a vast amount of research on CGC generators, and we do not have space to cover all of it. Instead, we will present a few simple and classical examples. More importantly, the underlying principles of these generators can be further explored within our framework. To streamline our discussion, we will focus solely on the *recursion*, *ranking*, and *unranking algorithms* for each Gray code algorithm Proofs and construction methods are beyond the scope of our study, interested readers can refer to Kreher and Stinson [1999].

II.1.4.1 Sublists

Definition 6. *Sublists distance.* Given a list of sublists $\mathcal{S}_{\text{subs}}$, and two sublists $s_1, s_2 \in \mathcal{S}_{\text{subs}}$, we define the *symmetric difference* of two sets s_1 and s_2 as

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

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

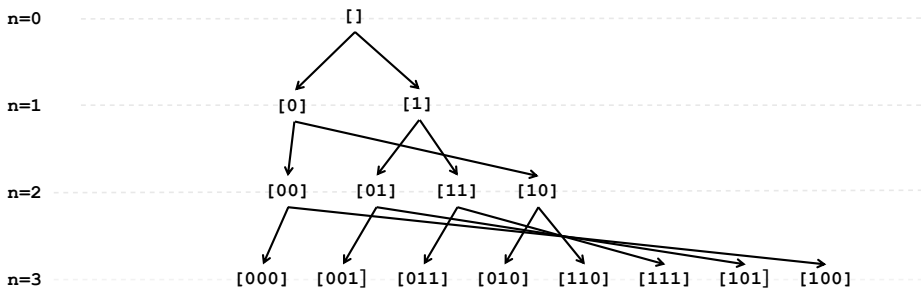


Figure II.1.6: The combinatorial Gray code generation tree for the binary assignment of a length 3 list.

where $s_1 - s_2$ is the list difference defined as $[x \mid x \in s_1 \wedge x \notin s_2]$, for instance $[1,2,3] - [2] = [1,3]$. Next, we can define the distance of two sublists as 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 we described in Subsection II.1.2.3, $d_{\text{sub}}(s_1, s_2)$ equal to the number of entries that $s_1', s_2' \in \mathcal{S}_{\text{basgns}}$ are different. The number of different entries for two configurations $s_1', s_2' \in \mathcal{S}_{\text{basgns}}$ is called *Hamming distance* $d_{\text{Ham}}(s_1', s_2')$. In other words, $d_{\text{sub}}(s_1, s_2) = d_{\text{Ham}}(s_1', s_2')$, it represents the number of elements that need to be added to and/or deleted from one sublist in order to obtain the other.

The minimal change ordering for two sublists $s_1, s_2 \in \mathcal{S}_{\text{subs}}$ is precisely when $d_{\text{sub}}(s_1, s_2) = 1$. An example of minimal change ordering for 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 a Hamming 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 of Gray codes. For sublist generation, we will examine a particularly simple class of Gray codes known as *the binary reflected Gray codes*.

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 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 reverse function is the reason that why it is called reflected Gray code. The Gray code G^n is constructed from G^{n-1} by two steps, we 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 two 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 generate ordering, the generation tree for the binary reflected Gray code G^3 is drawn in Fig. II.1.6.

Next, we will discuss the ranking and unranking functions with respect to the ordering used in CGC. 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 function 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 $r = \sum_{i=0}^{N} b_i 2^i \wedge b_N = 0$, and $G_r^N = [a_{N-1}, \ldots, a_0] \in G^N$ is a 0-1 binary assignment in all possible assignment generated by G^N. Then we have

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

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 give rise to the definition for unranking and ranking function. We briefly introduce the definition here, more detailed explanation can be found in Kreher and Stinson [1999].

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

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

where each value a_n is defined as

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

Similarly, we can define ranking function as below.

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

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

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

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, we need to redefine a minimal change ordering on $\mathcal{S}_{\text{ksubs}}$ to ensure that any two consecutive combinations have a distance of one. This special ordering is known as the *resolving door ordering.*

The revolving door algorithm The *revolving door algorithm* for generating K-sublists 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, the recursive program for generating the K-sublists of the list $[1, 2, \ldots, N]$ in resolving 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 is defined as follows

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

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 below.

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

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

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

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. It is defined recursively as

$$G^N = \left[rev\left(ins\left(N, \left[G_0^{N-1} \times N\right]\right)\right), [N] \circ_{ins} \left[G_1^{N-1} \times N\right], rev\left([N] \circ_{ins} \left[G_2^{N-1} \times N\right]\right), \ldots\right], \tag{35}$$

where G^N is a list of all permutations for list $[1, \ldots N]$, and $\left[G_r^{N-1} \times N\right]$ means duplicates N times for the r_{th} configuration in G^{N-1}. The $ins\left(N, \left[G_0^{N-1} \times N\right]\right)$ function inserts the new element N into the space between two adjacent elements of each G_0^{N-1} in $\left[G_0^{N-1} \times N\right]$, when r is even, we insert new element from the end to the beginning of G_r^{N-1}. If r is odd, we inset N from the beginning to the end of G_r^{N-1}. For instance, $G^2 = [[1, 2], [2, 1]]$, we can construct G^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. II.1.7.

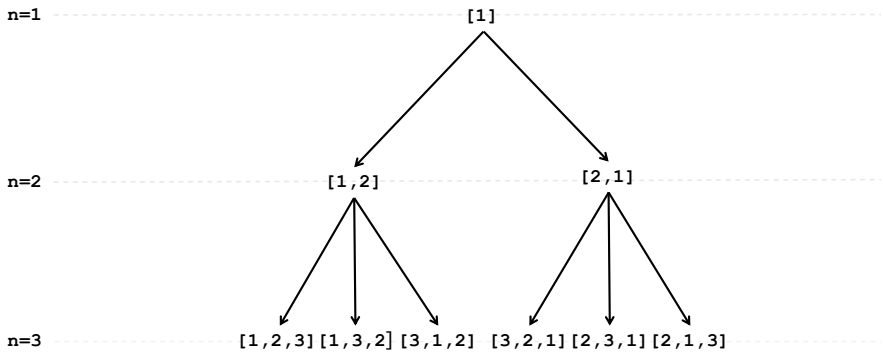


Figure II.1.7: The combinatorial Gray code generation tree for the permutations of list $[1, 2, 3]$.

Ranking and unranking functions Both ranking and unranking functions for the Trotter-Johnson ordering can be calculated recursively, this will help us to construct an integer SDP generator. Observing that descendants for a configuration G_r^{N-1} will have rank lies between $n \times r$ and $nr + n - 1$, this is 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 go after the descendants of G_r^{N-1}. Now we can define the ranking and unranking function 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

$$rank([\pi(1), \pi(2), \ldots, \pi(N)], N) = N \times rank([\pi(1), \ldots, \pi(k-1), \pi(k+1), \ldots, \pi(N)], N-1) + c, \tag{36}$$

where $\pi(k) = N$ and

$$c = \begin{cases} n - k & \text{if } rank([\pi(1), \ldots, \pi(k-1), \pi(k+1), \ldots, \pi(N)], N-1) \text{ is even} \\ k - 1 & \text{if } rank([\pi(1), \ldots, \pi(k-1), \pi(k+1), \ldots, \pi(N)], N-1) \text{ is odd.} \end{cases} \tag{37}$$

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

$$unrank(r, N) = [\pi(1), \pi(2), \ldots, \pi(N)], \tag{38}$$

such that $\pi(k) = N$ where

$$k = \begin{cases} N - (r - N \times unrank(\lfloor r/N \rfloor, N-1)) & \text{if } \lfloor r/N \rfloor \text{ is even} \\ r - N \times unrank(\lfloor r/N \rfloor, N-1) + 1 & \text{if } \lfloor r/N \rfloor \text{ is odd.} \end{cases} \tag{39}$$

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, 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* $\log(|\mathcal{S}_{\text{perms}}|)$ bits.

We shall refer to the SDP combinatorial generation process, which includes an embedded ordering (such as lexicographic or any other type of ordering) for each configuration, as the *ordered SDP combinatorial generator*. By replacing each configuration with the integer representing its rank, we obtain the *integer SDP c*ombinatorial *generator*.

In this section, we provide 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, we demonstrate how their corresponding integer SDP generators can be easily derived.

II.1.5.1 The binary reflected Gray code SDP generator

We have explained the binary reflected Gray code generator in mathematical form (23). In this Subsection, we further investigate the binary reflected Gray code by implementing the recursion (23) as an ordered SDP generator, and its integer SDP combinatorial generator can be implemented subsequently.

Ordered SDP combinatorial generator According to the binary reflected Gray code recursion (23), we can define the following two update functions

```
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` receives an input configuration list `xs` and keeps it unchanged. The `right_upd` function append the new element `a` to the front of every configuration in `xs`, then reverse 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 $ concat [f xs a | f <- 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]` gives us `[[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 After constructing the ordered SDP generator, the integer SDP generator can be constructed very easily by observing how integers are updated. Observing the updating patterns for `left_upd` and `right_upd` functions, we can design the corresponding integer update functions as

```
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` represents the recursion stage. Calculating `right_upd [0,1,2,3] 3` returns `[4,5,6,7]`.

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]
  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]`, if we run the unranking function on these integers, we can recover the original configurations.

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

We have already introduced a K-sublist generator `ksubs` in previous discussion. Given a list of sequence `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, we introduced a new K-sublist generator, which will generate all possible sublists, but the sublists

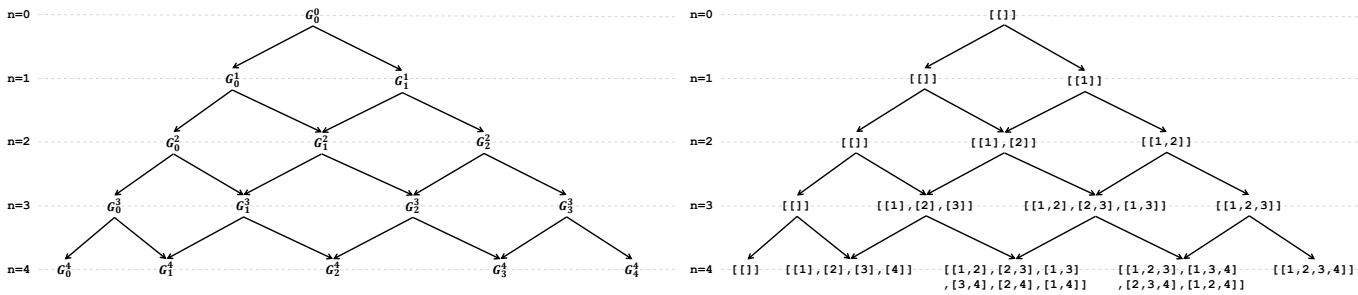


Figure II.1.8: The sequential decision process generation tree for all k-combinations, $0 \leq k \leq N$, of the input list $[1, 2, 3, 4]$ in revolving door ordering. The left panel depicts the generation tree representing the combinations of the same size in a configuration G_k^n. The combinations contained in G_k^n are illustrated in the right panel of the figure.

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 distinguish with `ksubs` and `subs`, we name this generator as `kcombs`, connotes "K-combinations generator."

Ordered SDP combinatorial generator As we introduced 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 collection 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 the 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.

To address the above issue, we need to generalize the ordinary SDP generators defined in Section II.1.2, we can still consider 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 becomes a single.

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-combinations 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` apply `append` function to every k-combinations in configuration `(reverse xs)`, the use of `reverse` operation is because we want to maintain the revolving door ordering, so we analogue the update rule of recursion (32). To illustrate,
`upd_left [[1],[2]] 3 = [[2,3], [1,3]]`.

Next, we need to merge some configurations generated by applying `upd_left` and `upd_right` functions. For instance, applying `upd_left` and `upd_right` to a list of configurations $\left[G_0^1, G_0^1\right] = [[[0]], [[1]]]$ gives us $[[[0]], [[2]], [[1]], [[1, 2]]]$. Clearly, $[[2]]$, $[[1]]$ should belong to a single configuration in G_1^2 because both of them are *size-one* combinations. We can infer 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 the pascal's formula (31). More generally, in the left panel of Fig. II.1.8, we draw the generation tree of the K-sublist SDP generator with revolving door ordering.

When we merge the right update of G_{k-1}^{n-1} and the left update of G_k^{n-1}, we must exchange their ordering 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]]`.

Observing that, two special configurations G_0^n and G_n^n 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 merge with any other updates. Following this observation, we can now define 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 satisfies 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
```

Now, we have following generator for generating 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 us all possible k-combinations, for all $0 \leq k \leq K$ in revolving door ordering $\left[G_0^4, G_1^4, G_2^4, G_3^4, G_4^4\right]$. The right panel of Fig. II.1.8 draws 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]` gives us `[[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, we tuple each rank list with an integer k. Then we can 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, it represents the `n`th level in the generation tree (see Fig. II.1.8), `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, we 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 rendered as

```
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 us
`[([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

In this chapter, we introduce 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.

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, they are still difficult to solve large-scale problems. In machine learning, distributed algorithms that require minimal or no communication between workers are often necessary to handle large-scale problems effectively, as seen 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 (COPs) 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 utilized 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 or polynomially. Storing all these configurations is both inefficient and memory-intensive.

Fortunately, both issues can be effectively addressed through the introduction of a datatype-generic generalization of the SDP—catamorphism. This generalization allows us to define combinatorial generators in a more succinct and elegant manner. Moreover, the generality offered by catamorphism 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 SDP. This flexibility facilitates 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, we must first discuss some motivations to study constructive algorithmics.

In programming, a recursive function $f : A \to B$ is a function that call themselves from within their own definition. The most common definition for recursion can take the abstract form

$$f = \varphi\,(f) \tag{40}$$

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

The recursion function abstraction (40) has three immediate problems:

1. **No unique solution guarantee.** Although function φ can be arbitrary functions with the required type, the (40) is not guaranteed to be a meaningful definition for all possible φ, it may have a unique solution up to isomorphisms or we can prove there is *no canonical choice exists.*

2. **Lack of structure** .Because φ can be arbitrary forms, 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 problem 2. Due to the difficulty in observing the structure of the recursion, no systematic and formal approach to verify the correctness of the program. Proof of the programs can only rely on tedious induction, making it arduous to conduct and prone to errors.

Most BnB algorithms are recursive functions and are typically presented in the form of (40). 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 (40), we dedicated to studying a more structured abstraction for recursion, *hylomorphisms*. The recursion specified by a hylomorphism takes the form

$$f = \phi \cdot \mathbf{F} f \cdot \psi \tag{41}$$

where the $\mathbf{F} : \mathcal{C} \to \mathcal{C}$ is a functor called *base functor* which determined the structure of the recursion, and the function $\psi : A \to \mathbf{F}A$ and $\phi : \mathbf{F}B \to B$ called $\mathbf{F}$-*coalgebra* and $\mathbf{F}$-*algebra*, we will see later these two name is originated from the study of universal algebra. The recursion (41) is a more structured instance of (40), in other words, any 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 to study the theory of constructing *structured recursion* in the form of (41). There are several benefits to examining recursions of this form: first, the sophisticated definition provided by (41) 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* studies, originally explored by Meertens [1986], Bird [1987]. The theory of constructive algorithmics involves deriving programs from specifications with an initial focus on producing the clearest and most understandable program possible, disregarding efficiency concerns. Subsequently, an efficient program is derived without altering the results or meaning of the original specification. In other words, it 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 consequences. Constructive algorithmics was specifically developed to design reliable programs for these tasks. It provides a foundation for formal methods in algorithm designs, 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 catamorphism

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 elegant language to unify these datatypes, and then we can know how to construct similar recursive datatypes for free.

In this section, we abstract datatypes categorically using **F**-*algebra.* An **F**-algebras is an arrow with type $alg : \mathbf{F}A \to A$, the object A is called the *carrier* of the initial algebra, and functor **F** is referred to as the *base functor*, which is defined through *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 **F**, the **F**-algebras, along with the homomorphisms between different **F**-algebras form a category $Alg(\mathbf{F})$. This category contains an *initial algebra*, which serves as the initial object in the category of **F**-algebras, with their carriers representing the *least fixed point* for the given base functor **F**. Some researchers will directly call the initial **F**-algebras as the models for recursive datatypes, rather than the carriers of the initial **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 $Alg(\mathbf{F})$, the homomorphisms between different datatypes (**F**-algebras) are called **F**-homomorphisms. These **F**-homomorphisms are *structure-preserving maps* that generalize the *homomorphisms* in universal algebra. Furthermore, when the domain of the **F**-homomorphisms is an initial algebra in the category of **F**-algebras, these **F**-homomorphisms are called *catamorphisms*, which can be implemented as a recursive program and the recursive structures of catamorphisms depending on 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 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 **F**
- The **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 nature numbers can be constructed from the empty seed $[\,]$ and then append 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

```
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. On the left-hand side of the equation, the term is referred to as a *type constructor* because it is utilized to construct a new datatype, with two parameters `a` and `x`. On the right-side of the equation 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 function means in the next section. The word "*polynomial*" refers to how the construction of the polynomial functor mimics the construction of 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*.

The astute reader may ask, can the type variable `x` in datatype `ListFl` be arbitrary types? The answer is affirmative. Indeed, 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
```

this definition exemplifies a *recursive datatype*. `Listl a` is called the *least fixed point*[8] of the functor `ListFl a`. This substitution can be made more systematically for any polynomial functors, because we can prove that `Listl a` is isomorphic to `ListFl (Listl a) a`, that is `Listl a` $\cong$ `ListFl (Listl a) a`. This is a consequence of *Lambek's lemma*. 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, can be implemented as follows

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

the `length` function serves as a homomorphism from snoc-list to an integer, it recursively calculate the length of a list from the left to right.

Equivalently, we can define the length function more compactly by using the `foldl` operator a

```
length :: (Num a) => [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 input, 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 use `ListFl a` as the input, we can feed 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 catamorphisms—allows us to produce module programs, we do not need to write a new program when we encounter a new 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`. The functions constructed by generic program `foldl` is more simple and compact.

II.2.2.2 Polynomial functors

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. 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.

The four algebraic rules used to construct polynomial functors are outlined below.

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 **A** maps every object in this category to the same object **A**, and morphisms to the identity function with respect to object A, i.e., $\mathbf{A}X = A$ and $\mathbf{A}f = id_A$.

[8] The term "least" connotes that it possesses a unique algebraic mapping to any other fixed point of the functor.

Product functor Given two arrows $f : C \to A$ and $g : C \to B$, categorically, we use the symbol $\langle\ \rangle$ to denote the *pairing arrow* $\langle f, g\rangle : C \to A \times B$ that apply two arrows f, g to the same object, where $A \times B$ called the *product* of two objects A and B. Some researchers 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
```

If all objects in a category $\mathcal{C}$ has a product, 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 product category, this bifunctor maps two objects two objects A and B to the Cartesian product of A and B. 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*[9], it has type $cross\,(f, g) : A \times B \to C \times D$ 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 = paring (f . fst) (g . snd)
cross f g = \(x, y) -> (f x, g y)
```

.

Coproduct functor Coproduct is dual to product, the operation on it can be defined dually. In Haskell, the coproduct functor on objects is just different value constructor that is separated 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 researchers 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 coparing/case arrow $[f, g] : A + B \to C$ is implemented as the pattern matching in Haskell. In python, we use `match case` statement to implement the case arrow.

Similarly, the morphisms between coproduct objects is rendered as

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

In the category of 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 $m \times n$.

[9]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.

Example 2. Snoc-list functor. 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 of 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. We will explore their connections and differences in this discussion, which may provide a better understanding of the theory of **F**-algebras in the later sections.

The algebra theory 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 $h(a * b) = h(a) + h(b)$, for $\forall a, b \in \mathbb{R}$. In the case of ordinary $+$, $*$ operation on the set of real numbers, $h = \log$. Let's explore how to generalize the idea of monoids and monoid homomorphism in universal algebra theory 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 $\mathtt{alg} : \mathbf{F}A \to A$ 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) is defined as `alg :: Functor func =>func a -> a`.

A morphism between algebras `alg1 :: func a -> a` and `alg2 :: func b -> b` is called **F**-homomorphism, denoted as `h`, which are defined by the mapping between their carrier `h :: a -> b` such that
`h . alg1 = alg2 . (fmap h)`.

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 $\mathbf{F} = \mathbf{id} \times \mathbf{id}$ as the type synonyms of Haskell built-in tuples

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

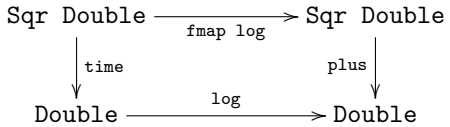
Then the monoid operation $* : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$ can hence be defined as an `Sqr`-algebra
`alg :: Sqr Double -> Double`, which can be implemented in Haskell as follows

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

Similarly, the monoid operation $+$ is defined as

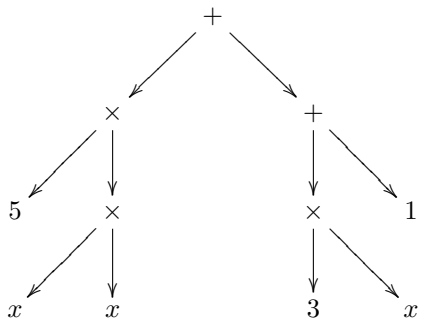
```
plus :: Sqr Double -> Double
plus (a, b) = a + b
```

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



The generality of **F**-algebra to the universal algebra lies in three-fold: Firstly, while all operators in universal algebra theory are binary, we are allowed to define non-binary operators 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, it first receives an argument of type `ListF a` and then an argument `a`. Third, the operations it generalize to any SCPO category—the category of complete partial orders with continuous functions—whereas universal algebra theory is restricted to the category of sets.

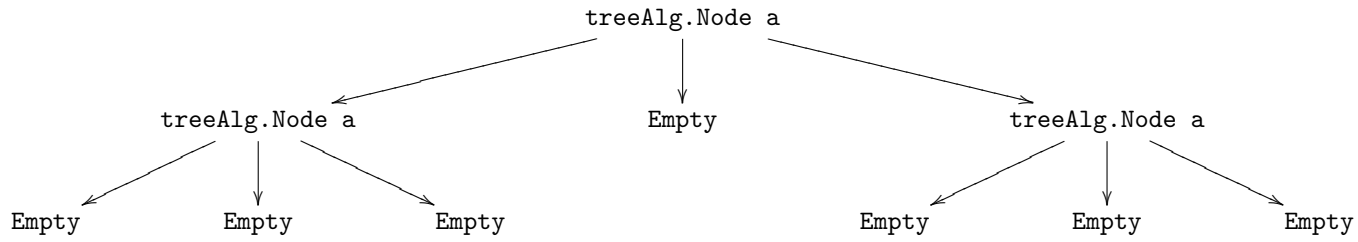
In computer science, arithmetic expressions like $5x^2 + 3x + 1$can be represented as the following expression tree



the 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, we can easily define a ternary tree by a polynomial functor $\mathbf{F_A} = \mathbf{1} + \mathbf{A} \times \mathbf{id} \times \mathbf{id} \times \mathbf{id}$. In Haskell, the object parts of this functor is defined by

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

When we define a `Ttree`-algebra `treeAlg`, the catamorphisms `cata treeAlg` will recursively evaluate the tree by using algebra from the bottom to up. The process diagram 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 not only 2 elements, and the initial `Ttree`-algebra `Node` contain elements of two different types.

II.2.2.4 Catamorphism characterization theorem

Fixed point operator **F**-homomorphisms can be composed and have an identity, it follows immediately that **F**-algebras and **F**-homomorphisms form a category called $Alg(\mathbf{F})$. The initial **F**-algebra $in : \mathbf{F}\mu\mathbf{F} \to \mu\mathbf{F}$ is the *initial object* in the category $Alg(\mathbf{F})$[Malcolm, 1990, Bird and De Moor, 1996], where type $\mu\mathbf{F}$ is the carrier. The carrier of the initial **F**-algebra is called 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. Many literature commonly refer to initial **F**-algebras as models for recursive datatypes, and we adopt this convention. According to Lambek's lemma [Lambek, 1968], the initial property induced an isomorphism $\mu\mathbf{F} \cong \mathbf{F}\mu\mathbf{F}$. Therefore, we can apply functor **F** infinite time to the datatype $\mu\mathbf{F}$, it is still isomorphic to itself. Hence $\mu\mathbf{F}$ is a recursive datatype. At the same time, the isomorphism between $\mu\mathbf{F}$ and $\mathbf{F}\mu\mathbf{F}$ tells us that the existence of initial algebra $in : \mathbf{F}\mu\mathbf{F} \to \mu\mathbf{F}$ must has a corresponding reversed arrow $\mathtt{out} : \mu\mathbf{F} \to \mathbf{F}\mu\mathbf{F}$ called *terminal algebra*, such that $in \cdot out = id$ and $out \cdot in = id$.

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 **F** or the fixed point of functor **F**. We can implement the fixed point operator μ as following

```
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 in the right-hand side of `=` 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 $Alg(\mathbf{F})$.

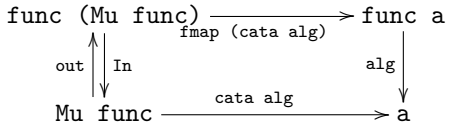
As an example, in the above section, we have explicitly defined the recursive definition for the snoc-list functor `ListFl a` as `Listl a`, now we can equivalently define `Listl a` by just calling `Mu (ListFl a)`, the least fixed point of functor `ListFl a` will automatically be generated by `Mu (ListFl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

Initiality and catamorphism As we have 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* **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 **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

$$\texttt{cata alg} = \texttt{alg . fmap (cata alg) . out}, \tag{42}$$

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



from the definition of catamorphism, we immediately have the *reflection law* `cata In = id`. This is because a catamorphism always maps an initial algebra `In` to another algebra `alg`, when `alg = In`, the catamorphism maps the initial algebra `In` to itself, thus the catamorphism becomes the identity function.

Given an **F**-algebra `alg :: Functor func =>func a -> a`, where base functor `func` is constrained in *functor class* `Functor`, the catamorphism characterization theorem (42) can be defined as

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

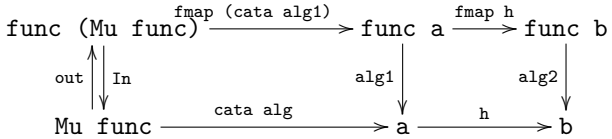
where the catamorphism `cata` takes an algebra `alg :: Functor func =>func a -> a` and a fixed point type `Mu func` as input. Although the fixed point input is not explicitly given in the definition of the function, it is reflected in the type declaration. This is again because of the use of *currying*. 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 corollary 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. The catamorphism fusion law is defined as follows.

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 state the following implication

$$\texttt{h . (cata alg1)} = \texttt{cata alg2} \Longleftarrow \texttt{h . alg1 = alg2 . (fmap alg1)}, \tag{43}$$

and the following diagram commute



The proof for the fusion law beyond the scope of this thesis, interested readers can refer to [Bird and De Moor, 1996, Jeuring, 1993]. The fusion law (43) is also sometimes referred to as the *promotion law*. The *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 demonstrates again why the theory of **F**-algebra generalize the universal algebra.

Fusion is a special case of *reorder of computation*. A function `h` satisfies the right-hand side equality can replace the `alg1` with a new algebra `alg2`. The program `cata alg2` is usually much more efficient than `h . (cata alg1)`. In the context of COPs, `cata alg1` represents a combinatorial generator `gen`, and then the function `h` performs

some computations on the configurations generated by `gen`. It is tempting to think if we can fuse computations, such as filtering or evaluation on combinatorial configurations, into our generator `gen`. Indeed, both the evaluator and the filter can be fused under certain conditions, and we will elaborate on this in a later section.

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. We will see various useful datatypes in the next section, and how these datatypes determine the structure of the recursion will be elaborated.

F-algebras represent the *semantics* of programs, for which determine the content of the recursion. In our context, the content we aim to describe includes recursive optimization algorithms and recursive combinatorial generators. We will present various useful algebras for combinatorial generation in Section II.2.3.

II.2.2.5 Various useful recursive datatypes

To highlight the effectiveness of polynomial functors, we construct several datatypes that are frequently used in ML research and demonstrate 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 is 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 we mentioned 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 (Listj 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)
```

In the join-list datatype, we can freely join the list from both sides. This gives us the freedom to split the list in an arbitrary way and then join them together using a `Join` operator. Therefore, the catamorphism with a join-list as input follows a recursive pattern: it splits the problem in an arbitrary way, solves each sub-problem independently, and then the solutions are "joined" together using a `ListFj`-algebra . In particular, if this algebra is

associative, we can easily construct an *embarrassingly parallelizable recursive program* based on join-list datatype. We will investigate this fact further and 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` defined by us. For example, the transformation from `[a]` to `Mu (ListFj a)` can be defined through the following function

```
conv :: [a] -> Mu (ListFj a)
conv [] = In (Nil)
conv [a] = In (Single a)
conv (a:x) = In (Join (conv [a]) (conv x))
```

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

```
cata :: [a] -> a
cata alg = alg . fmap (cata alg) . out . conv
```

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

Rather than define a converting function `conv`, it is much more convenient to define an ad-hoc 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:y) = Join [x] y
```

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

Similarly, for the cons-list, we can define an ad-hoc terminal algebra as

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

Note that the `out` function defined here is an ad-hoc definition specific to the join-list functor `ListFj a` and the cons-list functor `ListFr a`. In 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)`.

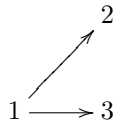
Algebraic directed graph Graph is a ubiquitous datatype in ML research. For instance, the directed graph is widely used in modeling *probabilistic* or *causal relations*. The most common definition for the graph is to define a pair $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ 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, if we have a graph defined as $G = (\{1\}, \{(1, 2)\})$, the vertex set $\{1\}$ contains only one vertex, but the edge $(1, 2)$ refer to the non-exist vertex 2.

This naturally raises the question of whether we can formulate a definition for a graph 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, we denote it 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 graph from the empty graph and single-vertex graph. These two primitive operations are called *overlay* and *connect*, are defined as

$$\begin{aligned} \text{Overlay}\left((V_1, E_1), (V_2, E_2)\right) &= (V_1 \cup V_2, E_1 \cup E_2) \\ \text{Connect}\left((V_1, E_1), (V_2, E_2)\right) &= (V_1 \cup V_2, E_1 \cup E_2 \cup (V_1 \times V_2)) . \end{aligned} \tag{44}$$

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* (ADPs). 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 can be found in Mokhov [2017].

The four primitive operations in ADG can be defined by a polynomial functor $\mathbf{F_A} = \mathbf{1} + \mathbf{A} + \mathbf{id} \times \mathbf{id} + \mathbf{id} \times \mathbf{id}$. In Haskell, we can implement it by defining datatype

```
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)
```

The directed graphs constructed in this way are safe and flexible. We can add additional axioms to extend the definition of directed graphs to represent undirected, reflexive, and hypergraphs, etc.

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 `fmap` based on functor `DiGraphF a` functor can be implemented explicitly as

```
fmap :: (y -> x) -> DiGraph a y -> DiGraph b x
fmap _ Empty = Empty
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) (fu y)
```

Binary tree In the definition of a binary tree, every node in the tree has an associated node element and two subtrees. Hence, we can define the binary tree functor as $\mathbf{F_A} = \mathbf{1} + \mathbf{id} \times \mathbf{A} \times \mathbf{id}$. The datatype implement this functor is defined as

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

The explicit recursive definition is rendered as

```
data Btree a = Empty | Node (Btree a) a (Btree a)
```

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

```
fmap :: (y -> x) -> BtreeF a y -> BtreeF b x
fmap _ Empty = Empty
fmap f (Node x a y) = Node (f x) a (f y)
```

II.2.3 Catamorphism combinatorial generation

In this section, we will reformulate 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 conducted in an imperative language such as Python or C++. Our goal 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, we explain how to create efficient generators for more complex combinatorial structures 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 from our construction of 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 elegant 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, it applies a binary function `f` to each element `a` in `x` and each element `b` in `y`, and store them in the list. For instance, that Cartesian product operator can be defined as `cp x y = crp (,) x y`, evaluate `cp [1,2] [3,4]` will return `[(1,3),(1,4),(2,3),(2,4)]`.

We can define 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
```

evaluate `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, we can define the "cross join, right" operator as `crjr =crpr (++)` and "cross join, left" operator as `crjl = crpl (++)`.

The cross product operator is frequently employed in various combinatorial generation tasks, particularly for catamorphisms based on 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 explore 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

The 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 with 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, to fuse a filtering process into a catamorphism defined over cons-list algebra, the predicate must still be prefix-closed

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

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 `q`.

In this Subsection, we redefine most of the generators illustrated in Section II.1.2 using catamorphisms based on cons-lists. Additionally, we introduce two new combinatorial generators that produce all possible initial or tail segments of a list. These new generators will enable us to construct 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 = cata (ksubsAlg k)
```

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

Also, borrows the definition of `upd_left`, `upd_right` and `sort_revol` in 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, we can define the sequence generator and its algebra as follows

```
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)]]`. We can therefore define these two generators and their algebras 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 We have defined the permutation generator in Section II.1.2 based on inserting a new element to the existing partial permutation, where we have defined an insertion function `insertKth` that inserts a new element `a` to the existing partial permutation.

Notice that in the definition of `insertKth`, we always take the first `k` elements of list `x` and join it with new element `a` and remaining list, for all `k` in `[0..len(x)]`. This is essentially the same as split `x` to a length `k` initial segment and a length `len(x)-k` tail segment. Therefore, we can define 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

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

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

and the all possible insertion of `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 we can define a more efficient permutation generator based on `adds` function, which can be defined as

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

perms = cata permsAlg
```

Also, by defining a `crjrwd` operator (short for, "cross join, right, without duplicates"), the K-permutations generator can be defined more compactly as

```
r :: Eq a => a -> [a] -> Bool
r i y = all (\b -> b /= i) y

kpermsAlg :: Eq a => [a] -> ListFr a [[a]] -> [[a]]
kpermsAlg x Nil = [[]]
kpermsAlg x (Cons a ys) = crjrwd x ys
 where crjrwd x ys = [y ++ [a] | a <- x, y <- ys, r a y]

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

In this section, we provide 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 [Emoto et al., 2012]. Associativity enables us to perform a sequence of operations without regard to the order of these operations. This is inherent in list join operation, that is

$$\texttt{x ++ y ++ z = (x ++ y) ++ z = x ++ (y ++ z)} \ . \tag{45}$$

Therefore, once we have a join-list algebra that is associative, we immediately obtain an embarrassingly parallelizable program. 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 is insufficient. Instead, a stronger condition is required, the predicate `p` must be *segment-closed*

$$\texttt{p (x ++ y) = p x \&\& p y}, \tag{46}$$

for all `x` and `y`.

Sublists, K-sublists Intuitively, the join-list sublists generator is constructed from 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
```

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) = [[],[a]]
ksubsAlg k (Join x y) = filter (maxlen k)(crj x y)
 where maxlen k = (<= k) . length

ksubs k = cata (ksubsAlg k)
```

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, we need to merge a list of permutations `xs` 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 the associativity, which is defined as follows

```
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, in order to merge two partial permutations `x` and `y`, we need to insert the element of `y` into all possible space of `x` while preserving the original ordering of elements in `y`. For instance, evaluating `merge [1,2] [3,4]` gives us `[[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 `2` is ahead of `1`. Indeed, this merge function is the join-list version of the *interleave function* [Bird and Gibbons, 2020]. We will discuss this function in greater detail in Chapter III.3 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. This process is indeed a special kind of *convolution product*, which is defined as

```
convol_filt :: ([a] -> [a] -> [a]) -> Int -> [[a]] -> [[a]] -> [[a]]
convol_filt f k x y = map process [0..k]
  where
    process n = concat [result i j | i <- [0..n], let j = n - i, i < length x, j
       < length y, (i + j <= k) ]
    result i j = f (x !! i) (y !! j)
```

For instance, combinations of size `3` can only be constructed from two combinations with size, `3` and `0`, `1` and `2`, `2` and `1`, and `0` and `3`.

Assume we want to construct combinations of size K from two lists of combinations `xs` and `ys`, with size sum up to K, we need to combine the combinations in `xs` and combinations in `ys` in all possible ways, i.e., the cross join of `xs` and `ys`. The K-combinations generator `kcombs` can thus be defined as

```
kcombsAlg :: Int -> ListFj a [[[a]]] -> [[[a]]]
kcombsAlg k Nil = [[[]]]
kcombsAlg k (Single a) = [[[]],[[a]]]
kcombsAlg k (Join xss yss) = convol_filt crj k xss yss

kcombs k = cata (kcombsAlg k)
```

Analogue to the design of the K-combination generator, the K-permutation generator is nothing more than the *integration of K-combinations and permutation*. Therefore, in order to design a K-permutations generator, we just need to 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 = [[[]]]
kpermsAlg k (Single a) = [[[]],[[a]]]
kpermsAlg k (Join xss yss) = convol_filt crm k xss yss
  where crm xs ys = concat $ crp (merge) xs ys

kperms k = cata (kpermsAlg k)
```

because both the `merge` and `crp` are associative, then `crm` is also associative.

II.2.3.4 Built complex combinatorial structures from the simpler basic structures

In our exploration, we have identified 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 us to evaluate the objective values of configurations during the recursive generation process, this is known as *evaluation fusion*. In many combinatorial optimization literature, evaluation fusion is often employed by default without undergoing verification for correctness. The Cartesian product fusion law now formalizes it.

Filter fusion We have seen the use of filtering in constructing the K-sublists algebra, where configurations that do not satisfy the predicate `p = (<= k) . length` are filtered out. In principle, we can define any predicates we want to select configurations from the set of all candidate configurations in the search space $\mathcal{S}$.

However, if our goal is to create an efficient combinatorial generator, it's crucial to identify partial configurations that will not satisfy the predicate before extending them to complete configurations. By eliminating these infeasible configurations early on, we can avoid wasting computational resources. This technique is known as *filter fusion*.

Nevertheless, in the previous two Subsections, we have explained that the filter fusion condition for cons-list algebra is prefix-closed, which is defined as

$$\texttt{p (x ++ y) = p x.} \tag{47}$$

For join-list algebra, the predicate is required to be segment-close

$$\texttt{p (x ++ y) = p x \&\& p y.} \tag{48}$$

Product fusion (banana-split law) Consider we have a list of integers, and we want to calculate its mean. 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 defined by `sum` operator. Equivalently, it can be implemented by using catamorphism as

```
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, we can define the function `average` by

```
average = uncurry div . (pair sum' length')
```

where the function `uncurry div` calculate the division of a pair of values.

This naive implementation `average` traverse the input list twice, once for catamorphism `sum'`, once for catamorphism `length'`. The astute reader may immediately realize that we can 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`, we have the following equality

$$\texttt{pair (cata alg1) (cata alg2) = cata (prodalg alg1 alg2)}, \tag{49}$$

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 notations for catamorphisms. In literature, we often use the symbol (|alg|) to represent catamorphism `cata alg`, the brackets (| |) are looks like bananas. We choose to call it *product fusion law*, due to its intrinsic product fusion nature.

Applying the product fusion to the above example, we have 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 evaluate `uncurry div . (cata sumlenalg)` is equivalent to evaluate
`uncurry div . (cata (prodalg sumalg lenalg))`.

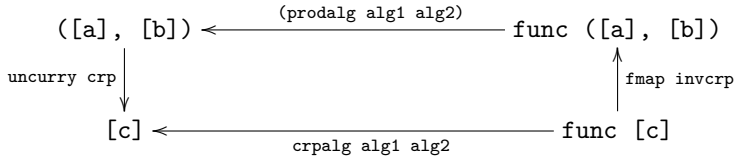
In the context of combinatorial generation, if we want to generate combinatorial configurations for two (or more) different combinatorial structures, the product fusion allows us to generate them by calling a single catamorphism. For instance, if we want to generate all sublists and permutations of a given list `xs = [1,2]`, evaluating `cata (prodalg subsalg permsalg) xs` gives us `([[],[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 Our earlier two primitive constructions—filter fusion and product fusion—may seem limited in certain situations. In many cases, building more complex combinatorial structures demands fuse operations that go beyond simple filtering or pairing. It is worth exploring whether we can fuse the cross product of any two catamorphism generators. The answer is given below.

Lemma 3. *Cross product fusion law.* Given any two algebras `alg1 :: func a -> a` and `alg2 :: func b -> b`. Assume function `f :: a -> b -> c4` is a bijective function. Then we have following equality

```
uncurry (crp f) . (pair (cata alg1) (cata alg2)) = cata (crpalg alg1 alg2),        (50)
```

where
`crpalg alg1 alg2 = uncurry (crp f) . pair (alg1.(fmap fst)) (alg2.(fmap snd))) . (fmap (invcrp f))`,
called *cross product fusion algebra.* In other words, the following diagram commute



Intuitively, the above diagram means that applying `crpalg alg1 alg2` to `[c]` is equivalent to: first recovering a pair of lists `([a],[b])` such that their cross product is equal to `[c]`, then calculate the result of `alg1 (func [a])`and `alg2 (func [b])` by applying `prodalg alg1 alg2` to `func ([a], [b])`. Finally, we apply the cross product operator again to `([a],[b])` to obtain the updated `[c]`.

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 (43)
  uncurry (crp f) . (prodalg alg1 alg2) = (crpalg alg1 alg2) . fmap (crp f)
≡ define invcrp f as the inverse of crp f
  (crpalg alg1 alg2) = uncurry (crp f) . (prodalg alg1 alg2). fmap (invcrp f)
```

The function `invcrp :: (a -> b -> c) -> [c] -> ([a], [b])` is the inverse of of the cross product function `crp`, 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])`. □

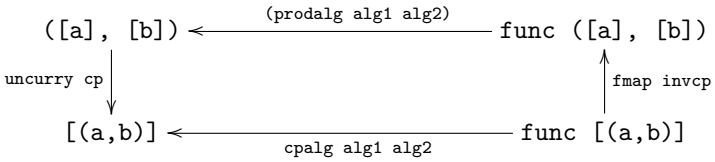
Cartesian product fusion Let's analyze cross product fusion in the simplest case: *Cartesian product fusion*. In the previous section, the Cartesian product function `cp` was described as a special case of the cross product `crp`. It can also be explicitly defined using a list comprehension `cp x y = [ (a, b) | a <- x, b <- y]` where `','` has the type `',' :: a -> b -> (a,b)`. By examining the construction order in the Haskell list comprehension, we can define 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, based on the construction ordering in Haskell list comprehensions. In other languages, the function `invcp` might require a different definition. According to the result of Lemma 3, we have the Cartesian product fusion algebra defined as

`cpalg alg1 alg2 = (uncurry cp) . (pair (alg1.(fmap fst)) (alg2 . (fmap snd))) . (fmap invcp)`, it makes the following diagram commute



In COPs, we often 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, we 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, we can apply the Cartesian product fusion law to fuse these two catamorphism generators. This allows us to apply 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]`, which gives us `[([],[1,2]),([2],[1,2]),([1],[1,2]),([1,2],[1,2])]`.

Similarly, evaluating `cata (cpalg subsalg permsalg) [1,2]` gives us `[([],[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, `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 here is 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 (54) and the foundations of relational algebra theory in Section II.2.5.

Time-space trade-off in fused cross product generator From the above examples, we can see that using cross product fusion allows us to construct efficient complex combinatorial generators easily. The 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, when constructing the cross product of two combinatorial generators, we 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.

In 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, we cannot introduce all of them in this thesis. Instead, we will focus on two of them that are most relevant to our research, namely *anamorphisms* and *hylomorphisms*.

II.2.4.1 Anamorphism

Let $\mathbf{F}$ be an endofunctor from $\mathbf{F} : \mathcal{C} \to \mathcal{C}$. Dual to $\mathbf{F}$-algebras `alg :: func a -> a`, a $\mathbf{F}$-coalgebra has type `coalg :: a -> func a`, in other words, the $\mathbf{F}$-coalgebra wraps an object A in the context of $\mathbf{F}$. We have seen one example of $\mathbf{F}$-coalgebra from the above discussion — the terminal coalgebra `out :: Mu func -> func (Mu func)`.

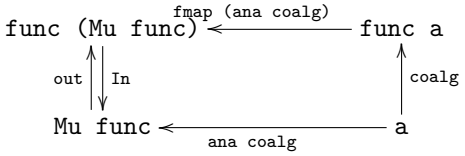
Analogue homomorphisms in $\mathbf{F}$-algebras, homomorphisms between two $\mathbf{F}$-coalgebras `coalg1 :: a -> func a` and `coalg2 :: b -> func b`, is defined by a morphism between their carrier `h :: a -> b` such that `coalg1 . h = (fmap h) . coalg2`.

The catamorphisms is a homomorphism from the initial $\mathbf{F}$-algebra to another $\mathbf{F}$-algebra, an $\mathbf{F}$-algebras `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 $\mathbf{F}$-coalgebra `coalg :: a -> func a` can be thought of as generating a data structure from a seed. The idea of a coalgebra is that you are given a seed and you use it to create a *single level* of a recursive data structure. This single-level data structure is precisely described by a functor $\mathbf{F}$, and the corecursive data structures related to $\mathbf{F}$-coalgebras are modeled by *the greatest fixed point*. Categorically, the greatest fixed point is denoted as $\nu\mathbf{F}$. In Haskell (or any SCPO categories more generally), the least fixed point $\mu\mathbf{F}$ and the greatest fixed point $\nu\mathbf{F}$ coincide [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 catamorphisms case, $\mathbf{F}$-coalgebras and homomorphisms between them form a category $coAlg(\mathbf{F})$. A $\mathbf{F}$-coalgebra is said to be a terminal $\mathbf{F}$-coalgebra if it is a *terminal object* in category $coAlg(\mathbf{F})$, denote as $out : \mathbf{F}\mu\mathbf{F} \to \mu\mathbf{F}$, 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 $\mathbf{F}$-coalgebra will correspond to the co-recursive/co-inductive datatypes. because the terminal $\mathbf{F}$-coalgebra is the terminal object in category $coAlg(\mathbf{F})$, initiality implies that there exists a unique morphism from other coalgebras in $coAlg(\mathbf{F})$ to terminal $\mathbf{F}$-coalgebra `out`. This unique homomorphism, called *anamorphism*, which 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

$$\texttt{ana coalg = In . fmap (ana coalg) . coalg}, \tag{51}$$

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, we can implement anamorphisms as

```
ana :: Functor f =>  (a -> f a) -> a -> Mu f
ana coalg = In . fmap (ana coalg) . coalg
```

Perhaps the simplest corecursive 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 dropped. The `Nil` constructor, which enables recursion termination, is absent. Hence, streams will never terminate. Categorically, the stream functor is defined as $\mathbf{F_A} = \mathbf{A} \times \mathbf{id}$. 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 `StreamF a` functor, then the `Stream` datatype can be defined by the following types of 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 of anamorphism is the stream generator. In Haskell, we can generate an infinite list starting with `n` by using syntax `[n..]`. We can implement a similar function `intsFrom` to generate 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, we convert a `Stream` to an (infinite) list by using a helper function that is defined as

```
toList :: Stream a -> [a]
toList (In (Cons x xs)) = x : toList xs
```

when evaluate `toList (intsFrom 3)` (lazily) generate an infinite list `[3,4,5,6,7..]`.

II.2.4.2 Hylomorphism

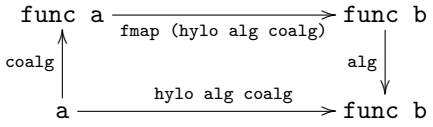
Hylomorphism is a very powerful and generic recursive morphism, it subsumes almost all practical recursions, including both *structured* and *generative* recursions, and almost all structured recursive schemes are special cases of hylomorphism.

The definition of hylomorphism becomes straightforward once we understand what anamorphism and catamorphism 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`, we can implement a hylomorphism in Haskell as

```
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 from `hylo alg coalg :: a -> b` that maps an coalgebra `coalg :: a -> func a` to an algebra `alg :: func a -> a` such that



commutes, where functor `func` is the base functor **F** of hylomorphism.

The commutative diagram of hylomorphism, as depicted in the above diagram, closely resembles that of catamorphism. Indeed, hylomorphism can be understood as a *generalized form of catamorphism* in the sense that the terminal algebra `out` in catamorphism is substituted with a **F**-coalgebra. This substitution affords an extensive degree of flexibility in program construction compared to catamorphism, 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. In contrast, `cata alg` and `ana coalg` always total if `alg` and `coalg` are. Indeed the hylomorphism `hylo alg coalg` has a unique solution if and if the `coalg` is a *recursive coalgebra*. In section II.2.4.4, we will clarify the conditions under which coalgebras become recursive coalgebras.

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; we will elaborate on the differences shortly in the next subsection. Thus, we refer to hylomorphisms as *generalized divide-and-conquer* (D&C).

One issue with using hylomorphisms is that, although we can define coalgebras and algebras as total functions, the resulting hylomorphism may not be total. In contrast, catamorphisms and anamorphisms are always total if their respective algebras and coalgebras are total. To address this issue, 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.

II.2.4.3 Hylomorphisms and divide-and-conquer algorithms

Our definition of the generalized D&C method given above differs from the classical definition 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. We make it explicit here, as we consider it a defining feature distinguishing D&C from dynamic programming. Classical examples include the mergesort and quicksort algorithms. By employing this binary subdivision strategy, D&C typically achieves logarithmic speed-up.

Therefore, our definition of the generalized D&C method given subsumes the classical one, as it allows for arbitrary subdivisions and shared subsolutions. More precisely, our definition of the D&C is *equal* to the traditional definition of the D&C when the intermediate datatype in hylomorphism has a recursive structure similar to the *binary tree*, such as join-list datatype or binary tree datatype that we mentioned in Subsection II.2.2.5.

In this section, we will further explore the connection between hylomorphisms and the divide-and-conquer method by analyzing two classical divide-and-conquer algorithms, the *mergesort* and the *quicksort* algorithms, and characterizing them in terms of hylomorphisms.

Mergesort algorithm The *mergesort algorithm* is often considered the archetypal D&C algorithm, utilized for sorting a list of elements into a specified order. The mergesort comprises two fundamental steps:

1. **Recursively splitting**: Divide the unsorted list recursively into two equal halves until each sublist consists of singleton lists.

2. **Recursively merging**: Merge the sublists recursively to generate new sorted sublists until only one element 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` split 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 consists of three pattern matchings, the empty list pattern, the singleton pattern 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. Indeed, the join-list functor `ListFj a` can serve as the base functor for the mergesort algorithm using hylomorphism.

The `ListFj a`-coalgebra of the hylomorphism corresponds to a one-step decomposition process. Here we subdivide the unsorted list into two equal halves and store them in the intermediate datatype

```
split :: [a] -> (ListF a [a])
split []  = Nil
split [x] = Single x
split xs  = Join l r where
  (l, r) = splitAt (length xs `div` 2) xs
```

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 used in hylomorphism corresponds to the recombine stage in the recursion which is implemented by

```
mcombine :: Ord a => ListF 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 hylomorphism. The problem is initially decomposed by `ListFj a`-coalgebra `split`, then solved by `hylo`, and finally, the solutions are recombined by `ListFj a`-algebra `mcombine`.

Quicksort algorithm The *quicksort algorithm* shares a nearly identical decomposition structure with mergesort and has also gained widespread acceptance as a D&C algorithm. The quicksort algorithm consists of three essential steps:

1. **Pivot selection and recursive partitioning**: Select a random 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 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

```
qucikSort' :: Ord a => [a] -> [a]
qucikSort' [] = []
qucikSort' (a:x) = (qucikSort' l) ++ [a] ++ (qucikSort' 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 gives us a subdivision that is similar to the binary tree datatype that we 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
```

By combining the `BtreeF a`-algebra and the `BtreeF a`-coalgebra together, we obtain the hylomorphism implementation for the quicksort algorithm is rendered as

```
qucikSort :: Ord a => [a] -> [a]
qucikSort = hylo qcombine partition
```

Implication of the hylomorphism Astute readers may notice that the definition of `quickSort'` algorithm consists of *two patterns*, whereas the definition of `mergeSort'` algorithm consists of *three patterns*. This discrepancy is not coincidental, it arises from the fact that the hylomorphism for constructing the `mergeSort` algorithm is based on the join-list functor `ListFj a`, which is defined by *three value constructors*. In contrast, the hylomorphism for constructing the `quickSort` algorithm is based on the *binary tree functor* `BtreeF a`, which is defined by two value constructors. This observation strongly indicates that the hylomorphism reflects the structure of the decomposition.

Indeed, the classical definition of a divide-and-conquer algorithm is precisely when the intermediate data type in a hylomorphism exhibits a *binary split* structure. For example, in the cases discussed, the intermediate datatypes are join-list and binary tree, both of which have functors with constructors that include two free fields `x`. However, the utility of hylomorphisms extends beyond binary-tree-like datatypes. Hylomorphisms can be applied to intermediate datatypes such as ternary trees, quaternary trees etc.

II.2.4.4 Recursive coalgebras

Hylomorphism can be understood as a three phrases program: First step a problem is divided into sub-problems by the **F**-coalgebra, where **F** is the base functor; Second, sub-problems are solved recursively and independently to form sub-solutions; Finally, these sub-solutions are combined together to form the complete solutions of the original problem.

The recursive structure of hylomorphism is determined by the base functor **F**. Depending on the shape of base functor **F**, hylomorphisms can capture various constructs such as while-loops, and divide-and-conquer schemes. In fact, any practical program can be cast into a hylomorphism form [Hu et al., 1996].

As mentioned earlier, the generality of hylomorphisms can lead to solutions that are either non-unique or not well-defined. Although we have mentioned that 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 *adjunction* and *conjugate*, which are very hard to explain or give a concrete Haskell example in limited space. This thesis aims to present a semi-tutorial explanation regarding constructive algorithmics to a wide range of audience. We think it is better to establish a few simple conditions that can help readers quickly understand and reuse them in the future.

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 *algebras*, or alternatively, explore *corecursive algebras* to ensure uniqueness for any *coalgebras*. This thesis specifically emphasizes the development of recursive coalgebras, leaving the construction of corecursive algebras to be addressed dually.

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 coalgebra provides another kind of initiality, and the category of recursive coalgebras *rec* (**F**) forms a subcategory of the category of coalgebras *coalg* (**F**). The notion of recursive coalgebras generalizes the terminal algebras insofar as structured recursion is concerned. In many applications, we need more than initial algebra to solve a problem, as we can see in the above mergesort and quicksort algorithms, both `split` and `partition` are not terminal coalgebras. We believe 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.

We have the following two simple facts for constructing recursive coalgebras.

Fact 1. If an endofunctor **F** has an initial algebra `in`, the inverse of the initial algebra is a terminal recursive coalgebra, for instance, in catamorphism the terminal algebra `out` is also a recursive coalgebra.

This observation is easily comprehensible given our previous understanding that catamorphisms possess unique solutions. This aspect provides a clearer explanation as to why initiality guarantees the uniqueness of the catamorphism.

Fact 2. For a base functor **F**, if the category *coalg* (**F**) has a terminal **F**-coalgebra `out`, if there exists an anamorphism from **F**-coalgebra `coalg` to `out`.

The fact 2 comes from a more general result of Capretta et al. [2006], proposition 8. It states that 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. Consequently, Fact 2 naturally emerges as a specific instance, since every anamorphism has a left-inverse, which is a catamorphism.

This explains why the coalgebras `split` and `partition` in mergesort and quick sort algorithm has a unique solution, because they are `BtreeF`-coalgebras, and the category of *coalg* (`BtreeF`) has a terminal `F`-coalgebra `out`.

II.2.5 Foundations for the algebra of programming

As we mentioned, the theory of the algebra of programming is a generalization of the Bird-Meertens formalism from *total functions* to *relations*. Before we go into details about the theory, we will illustrate a few foundational concepts for understanding the algebra of programming theory.

II.2.5.1 Motivations for using relational algebra

In the classical Bird-Meertens formalism, it 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. **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.

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*, and we denote the converse of the relation by using a superscript $^\circ$. For instance, the converse of a relation `r a b` can be defined as `r° a b = r b a`.

The 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 shortcoming in implementation: a Boolean-valued function cannot capture non-deterministic behavior. Specifically, we want to give an input `a` to relation `r`, and `r a` will return all `b`s such that `r a b == True`.

We cannot define a non-deterministic function in Haskell. Instead, when necessary, we will represent non-deterministic functions 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`.

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,

$$\mathtt{r} \subseteq \mathtt{s} \Longleftrightarrow (\forall \mathtt{a}, \mathtt{b} : \mathtt{r\ a\ b} \Longrightarrow \mathtt{s\ a\ b}) . \tag{52}$$

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 *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 we have explained in Section II.2.3, any COP can be specified through the exhaustive search paradigm, which can be re-formulated in Haskell as

```
sel r . filter p . (map eval) . (cp dataseqn) . gen
```

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 allows evaluation fusion feasible, we can 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, and we have seen various combinatorial generators based on catamorphism in Section II.2.3. If a predicate `p` is prefixed-close, the filtering and evaluation processes are integrated into the generator, a COP 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, when we need to decompose a problem using a coalgebra `coalg`, then the generator `gen` can be specified as a hylomorphism. Consequently, the COP can be expressed in a sophisticated form through hylomorphism.

```
sel r . hylo alg coalg
```

As we 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 a total order of the form `cost°. leq . cost`, where the `cost` function calculates the objective values of a configuration. Therefore, in Haskell, we can define the selector over the list 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
```

which selects the minimal element in a list with respect to the preorder `r`. We will always denote the preorder used in selector `sel` as the lowercase letter `r`, referred to as the *selector relation*.

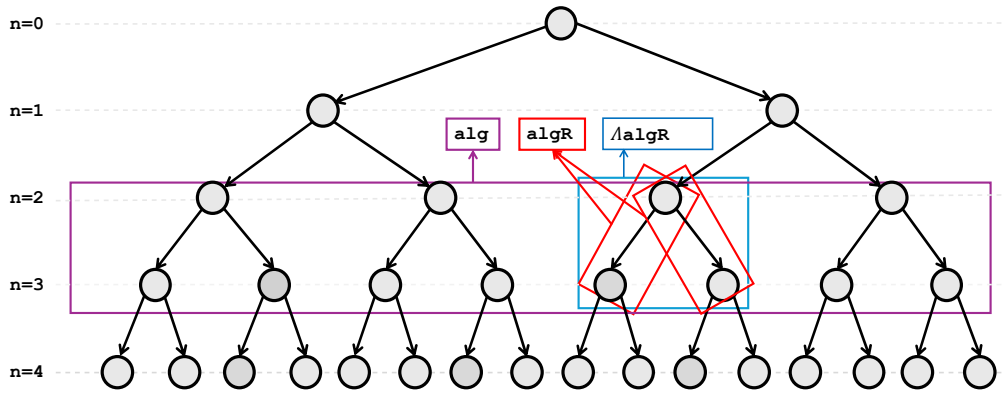


Figure II.2.1: The difference between functional algebra `alg` (purple), relational algebra `algR` (red), and the power transpose of the relational algebra `ΛalgR` (blue) in the context of the sequential decision process (catamorphism over cons-list datatype). The arrows in red boxes represent possible 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 `ΛalgR:: func a -> [a]` maps a configuration to all possible outcomes 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.

II.2.5.4 Relational F-algebras

Relational F-algebras/coalgebras Before discussing more general results, we need to distinguish the difference between *relational* **F**-*algebras* (`algR`) and *functional* **F**-*algebras* (`alg`). All algebras that we have seen before are functional **F**-algebras. Introducing relational **F**-algebras will allow us to reason programs more easily. 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 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 our discussion here, for the ease of understanding, we leave the identity between functional algebra and relational algebra implicitly, and assume 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 $\tau^{-1}\,(\texttt{alg}) = \texttt{algR}$. The definitions of functional **F**-algebras and relational **F**-algebras are defined as follows.

Definition 14. *Functional* **F**-*algebra and relational* **F**-*algebra.* Denote base functor **F** as `func`. Given a functional algebra `alg :: func [a] -> [a]`, its corresponding relational algebra has a type `algR :: func a -> a`, such that $\tau\,(\texttt{algR}) = \texttt{alg}$ and $\tau^{-1}\,(\texttt{alg}) = \texttt{algR}$.

Similarly, we can define the functional **F**-coalgebra and relational **F**-coalgebra 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 SDP, the distinctions between relational algebra and functional algebra are depicted in Fig. II.2.1. 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

$$\texttt{Λr a = [b | b <- (r a b == True)]}, \tag{53}$$

it means that `Λr a` returns all `b` such that `r a b == True` and store all `b`s in a list. The list comprehension here is defined as a pseudo-Haskell code style, as the relation `r` here is a non-deterministic function. When we implement the power transpose of a relation `Λr` in Haskell, we denote it as `r_pt`.

There is a subtle difference we need to care about. Given a relational algebra `algR`, the functional algebra `alg :: func [a] -> [a]` is different with `ΛalgR :: func a -> [a]`, despite both being functions. As depicted in Fig. II.2.1, `alg :: func [a] -> [a]` means apply *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 symbol Λ can not be implemented in Haskell, we use `algR_pt` (short for "relational algebra, power transpose") 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

$$\texttt{concat . map (ΛalgR . (Cons a)) = alg . (Cons a)}, \tag{54}$$

over the cons-list catamorphism. 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 :: ListFr a`. In contrast, the power transpose of a relational algebra `ΛalgR :: func a -> [a]` can only update a single configuration. To address this, we parameterize it with `Cons a :: ListFr 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, we apply the `concat` function, which removes the inner brackets and transforms the list of lists into a single list. The correctness of equality (54) can be generalized to other base functors as well. However, we focus on the cons-list functor for ease of understanding.

More generally, given a base functor **F**, we have

$$\texttt{concat . } \mathbf{P}\ \texttt{(ΛalgR . } \mathbf{F}\texttt{) = alg . } \mathbf{F}, \tag{55}$$

where the `map` function is replaced by the power functor **P** and `Cons a` is replaced with the datatype constructor (functor on objects) **F**. In the case where the base functor corresponds to the join-list datatype, it can be shown that the power functor **P** is equivalent to the cross product operator. This equivalence explains why the Cartesian product fusion algebra `cpalg'` is equivalent to `cpalg`. The `pairlist` operator used to define `cpalg'` serves as the power transpose of the relational version of `cpalg`. In other words, `cpalg = ΛcpalgR`, where `cpalgR` represents the relational version of `cpalg`.

The proof for equality (54) involves several unnecessary definitions in relational algebras that will not be used elsewhere, so we have placed the proof in Corollary 5 in the appendix.

Defining a combinatorial optimization problem through relation We have seen how to specify a COP through hylomorphism in the previous section where the algebras and coalgebras are functional. Given a relational algebra `algR :: func a -> a`, and a relational coalgebra `algR :: func a -> a`, a COP can be specified through a relational hylomorphism as

$$\texttt{sel r . Λ(hylo algR coalgR)}, \tag{56}$$

where `Λ(hylo algR coalgR) = hylo alg coalg`.

Similarly, a COP can also be specified through a relational catamorphism as

$$\texttt{sel r . Λ(cata algR)}, \tag{57}$$

where `Λ(cata algR) = cata alg`.

II.2.5.5 Monotonic algebras

The 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

$$\texttt{algR . (fmap rel)} \subseteq \texttt{rel . algR}. \tag{58}$$

Example 3. *Sequential decision process.* For combinatorial optimization algorithms based on SDPs, `algR :: func Config -> Config` can be considered as a single decision function. The condition of (58) states that if `a` is a predecessor of configuration `a'`, i.e., `a' = alg (func a)`, then `rel a b` implies that `rel a' b'`, where `b' = alg (func b)`. Indeed, in the context of SDP, the monotonicity condition in (58) can be simplified to the following

$$\texttt{rel a b} \subseteq \texttt{rel a' b'}, \tag{59}$$

which aligns precisely with the classical definition of monotonicity in existing literature [Ibaraki, 1977, Karp and Held, 1967], and the abstraction (58) takes a much more abstract and generic form.

Example 4. *Universal algebras.* Consider the base functor `Sqr`, which represents the squared functor. We previously defined an algebra `plus :: Double -> Double -> Double` (or equivalently `plus :: Sqr Double -> Double`). This algebra is monotonic with respect to relation `leq`, which can be expressed as `plus . (fmap 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'` $\Longrightarrow$ `c` $\leq$ `a' + b'`, where `+` and $\leq$ is the infix notation of `plus` and `leq`.

When `algR` is a function, we denoted it as `algRf`, the monotonicity has two useful facts.

Fact 3. When `algR` is a function, we can prove that `alg` is monotonic on `R` if and only if it is monotonic on $\texttt{R}^\circ$.

Fact 4. When `algR` is a function, the monotonicity is equivalent to the *distributivity*. We say `algR` distributes over `r` if

$$\texttt{algR . min r} \subseteq \texttt{sel r . alg}, \tag{60}$$

For instance, `plus` distributes over `leq` can be interpreted point-wise as `min x + min y = min [ a + b | a <- x, b <- y ]`.

II.2.6 Thinning

The *thinning algorithm* has often been called dominance relation in many algorithm design literature [Ibaraki, 1977, Galil and Giancarlo, 1989, Eppstein et al., 1992]. The use of thinning or dominance relation is concerned with improving the time complexity of naive dynamic programming algorithms. In the discussion here, we characterize the thinning algorithm by parameterizing it with a dominance relation.

The thinning technique explores 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 condition is known as the *greedy condition*. This condition characterizes situations where a problem can be solved using a *greedy algorithm*.

On 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 insightful observations about the problem. To address this issue, we introduce two generic dominance relations: the *global upper bound* and the *finite dominance relation*. These two dominance relations are widely applicable to many machine learning or combinatorial optimization problems where easily accessible approximate algorithms are available and possess only finite data.

II.2.6.1 What is thinning

In this Subsection, we will characterize the theorems and corollaries of the thinning technique, which are formalized by Bird and De Moor [1996]. Following this, we will expand upon the discussion regarding the connection between the greedy condition in the *thinning theorem* and the greedy condition in *matroid theory*. The latter is the 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]` select 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` $\subseteq$ `r`, otherwise applying thinning operation is the same as applying `sel r` directly.

We can introduce the thinning relation into an optimization problem with the following *thin-introduction rule*

$$\texttt{sel r = sel r . thin domR}, \tag{61}$$

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 requires to be a preorder, t, which allows us to utilize transitivity. If `a` dominate `b`, and `b` dominate `c` we have a dominate `c`. In practice, this enables us to only consider configurations that are "most likely" dominate others, if `a` is not dominate `c` and `a` dominate `b`, then `b` is impossible to dominate `c`, comparing `b` and `c` is a waste of time.

Thinning theorem

Theorem 1. *Thinning theorem.* Given a base functor `func`. If `algR :: func a -> a` is monotonic on `domR`$^\circ$, then we have following implication

$$\texttt{sel r . (cata ((thin domR). alg))} \subseteq \texttt{sel r . } \Lambda\texttt{(cata algR)}. \tag{62}$$

The thinning theorem 1 states that solutions returned by the thinning algorithm (left-hand side of the inclusion (62)) are also solutions of the brute-force algorithm (right-hand side of the inclusion (62)). It immediately follows from the theorem that the thinning algorithm becomes a brute-force algorithm—the inclusion (62) becomes an identity—if `domR == id`.

The greedy algorithm is then be characterized by the following corollary.

Theorem 2. *Greedy theorem.* A COP specified in the form of (57), if and only if `algR` is monotonic on $\boldsymbol{r}^\circ$. Then we have

$$\texttt{cata (sel r . } \Lambda\texttt{algR)} \subseteq \texttt{sel r . } \Lambda\texttt{(cata algR)}. \tag{63}$$

where `cata (sel r . ΛalgR))` is the specification of the greedy algorithm

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 (61). 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 COP specified in the form of (57), if `f` is monotonic on `r`$^\circ$, such that `f` $\subseteq$ `sel r . ΛalgR`. Then we have

$$\texttt{cata f} \subseteq \texttt{sel r . } \Lambda\texttt{(cata algR)} \tag{64}$$

In the context of SDP, the function `f` in Corollary 2 can be understood as the *best decision function* with respect to the objective in choice function list `ΛalgR`. Diagrammatically, in Fig. II.2.1, 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`, we can just replace the monotonic condition on `r`$^\circ$ with `r`. In practice, this greedy condition is very easy to verify when the COP is specified as a SDP, all we need to do is to check if the *decision functions are monotonic on* $\boldsymbol{r}^\circ$, which is usually the preorder with respect to the objective function.

Compared with the classical greedy condition where the problem requires finding a *matroid* [Schrijver et al., 2003]. This is almost as hard as finding a greedy algorithm directly. In contrast, our characterization is significantly simpler, more concise, and much easier to verify in practice. We will next explore how these two characterizations of the greedy condition are related to each other.

Greedy algorithm and matroid theory The *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 this collection of independent sets. Matroid theory has exactly the same relationship to linear algebra as does point set topology to the theory of real variables [Welsh, 2010].

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. if $I \in \mathcal{I}$ and $J \subseteq I$, then $J \in \mathcal{I}$
2. if $I, J \in \mathcal{I}$ and $|I| < |J|$, then $I + z \in \mathcal{I}$ for some $z \in J \backslash I$

and 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 Schrijver et al. [2003].

Theorem 3. *Greedy theorem in matroid.* Let $\mathcal{I}$ be a nonempty collection of subsets of a finite set S closed under taking subsets. For any weight function$w : S \to \mathbb{R}$ we want 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 $w(I \cup y)$ as small as possible. We stop 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.

Corollary 3. An exhaustive specification through catamorphism (57) introduces a matroid. If the algebra `algR` of it satisfies the greedy condition given in Thm. 2, then the resulting greedy algorithm of Thm. 2 is a valid greedy algorithm in matroid.

Proof. We first show that any exhaustive specification through catamorphism (57) introduces a matroid, and then prove that the 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 finite times of algebra `algR` to a partial configuration `a'` $\in \mathcal{S}'$, then `a` $\in \mathcal{S}$ such that `a = algR func (.. algR (func a'))`. Given $\mathcal{I}_{\text{SDP}}$ consists of all possible configurations $\mathcal{S}$ and set all partial configurations $\mathcal{S}'$, it thus the first condition of the matroid is satisfied. If `a'` $\in \mathcal{S}'$ is a partial configuration, assume `b' = algR (func a')`, then `b'` is either a partial configuration or a complete configuration. Defining `|a'|`as the number of times we need to apply `algR` to obtain `a'` from empty, thus `|b'|` $\geq$ `|a'|`. Therefore, the second condition of the matroid is satisfied. Hence $\mathcal{I}_{\text{SDP}}$ is a independent set of the matroid $M_{\text{SDP}} = (S, \mathcal{I}_{\text{SDP}})$, where S is the input list.

Our 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 repeat selection process in matroid greedy algorithm is rendered as

$$y = \underset{y \in S \backslash I}{\operatorname{argmin}}\, w(I \cup y). \tag{65}$$

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 our definition of the decision function `algR func I = algR (Cons y I)` over the cons-list datatype. However, instead of selecting the smallest $I \cup y$ such that $y \in S \backslash I$, we select 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 $M_{\text{SDP}} = (S, \mathcal{I}_{\text{SDP}})$, then `cata (sel r . ΛalgR))` is a valid greedy algorithm □

The key distinctions between the classical greedy theorem in matroid theory and our characterization lie in both the formulation of the greedy algorithm and the definition of "independence." In our approach, 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 we choose a $w(y)$ as small as possible. Additionally, in the matroid greedy theorem, "independence" is characterized by the matroid, while in our characterization, independence is an inherent property of the SDP formulation itself.

Given these distinctions, we believe 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 essence of the greedy algorithm, making its derivation easier and more systematic.

II.2.6.2 Different implementations of thinning

In this Subsection, we will discuss how to implement the thinning algorithm in practice. The naive *perfect* thinning algorithm requires comparing all pairs of configurations, thus $O\left(M^2\right)$ computations in the number of evaluations of dominance relation `rel` are required, where M is the size of partial configurations which is usually exponentially/polynomially large. By perfect, we mean all *dominated partial configurations* are purged.

Certainly, the naive approach is impractical in practice. The ideal implementation of the thinning algorithm should be a linear-time program—linear with respect to the number of partial configurations—that produces the shortest possible result by eliminating as many partial configurations as possible. However, a linear-time program does not always guarantee the shortest result. Achieving both goals at the same time is a paradox in itself because deleting more partial configurations means that we need to evaluate the dominance relation more frequently.

In practice, we 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 should 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 = [(2, 5), (1, 2), (2, 3), (3, 2), (1, 3), (1,5)]
rel a b = (a, b) `elem` relList
```

if element `(a, b)` exists in list `relList` means `rel a b == True`. In this case, we assume the preorder `rel` is a partial order, i.e., it has 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]`.

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: we 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` we delete `b` from `x`, and whenever `rel b a == True`, we stop comparing and delete `a` from `l`. Then we recursively apply this process to `x'`, where `x'` is the list 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]
```

where `@` create 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, we retain `a` in the front of the list `l` , otherwise we delete it, and list `x'` is strictly smaller than `x` since we delete some elements from `x` to create `x'`, this guarantees that `thin_exh` will terminate.

For instance, evaluating `thin_exh r l` gives us list `[1,4]`. Another example is when the relation `r` is the total order `<`, run `thin_exh (<) [2,3,4,1,5]` we obtain `[1]`.

In the worst case, we need to compare all pairs of elements in `l`, thus the `thin_exh` function will evaluate `rel` $O\left(M^2\right)$ times, 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, we can sort a list with respect to preorder `rel` first, then scan the list from start to end. We can find the first element `b` dominated by the first element `a`, and all elements after `b` should be purged. 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*[10] $O(M \log M)$, 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, we assume we have a sorting function `sort` that sorts our list in order. For instance, given preorder `r` the list `l = [1,2,3,4,5]` is sorted to `[1,4,2,3,5]`.

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, evaluating `thin rel [1,4,2,3,4,5]` will return the perfect thinning result `[1,4]`.

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, we need only compare the "best" configuration `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`, (i.e., configurations for which `rel b c = True`) will automatically satisfy `rel a c = False` without further examination.

Thinning by bumping After introducing the perfect thinning algorithms, we now introduce *linear-time* thinning algorithms, these implementations are only perfect in certain situations. The first linear-time method is called *bumping*, the Haskell implementation of bumping 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 (acc!!0) b = bump acc bs
      | rel b (acc!!0) = bump ((tail acc) ++ [b]) bs
      | otherwise = bump (acc ++ [b]) bs
```

the `bump` function has a accumulator `acc` which initialized as the first element `a` of list `l`, we always use the first element `acc!!0` of accumulator to recursively compared with the element `b` in `x`, if `r (acc!!0) b == True`, we ignore `b` and compare the next elements in `bs`, if `r b (acc!!0) == True`, i.e., the first element in `acc` is dominated by some `b`, then we delete `acc!!0` 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 $3 * M$, 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) [1,4,2,3,5] = [1,4]`, exchange only two elements in the list will obtain completely different results.

Thinning by squeezing In the previous study, De Moor [1995] proposed another method for implementing a linear-time thinning algorithm, known as "squeezing." This method removes an element from a list if its neighbor is "smaller" in the preorder `rel`. In Haskell, we can define it as

[10] Query complexity is the number of comparisons performed

```
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))
```

Same as `thin_bump` function, we have no guarantee that `thin_squeeze` is perfect in general. For instance, evaluating `thin_squeeze rel [1,4,5,3,2]` will give us `[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 = (thin_squeeze rel1) . mmerge rel2
```

In practice, the relation in `thin_squeeze` may different with `mmerge`.

II.2.6.3 Dominance relations

Dominance relations are used extensively across various fields and have led to the construction of many successful algorithms. Discovering dominance relations often involves insightful observations about the problem, and such observations are typically not reusable across different problems. To address this issue, we propose two ingenious dominance relations: the *global upper bound* (GUB) and the *finite dominance relation* (FDR). In the discussion of Part III, we will explore how these two dominance relations are ubiquitous in machine learning and combinatorial optimization research. In these fields, approximate algorithms are easy to obtain, and data is always finite. These characteristics simplify the design and implementation of the GUB and FDR relations.

Previously, we have derived the classical definition of monotonicity in the context of SDP in Subsection II.2.5.5. We have discussed this in Thm. 62 that a relation `domR` satisfying monotonicity can be fused into the thinning algorithm However, we have not yet formally characterized what we mean by a dominance relation. We now present a formal definition of the dominance relation in the context of SDP as follows.

Definition 16. *Dominance relation.* Given a relational algebra `algR :: func Config -> Config`, and two configuration `a :: Config` and `b :: Config`. A relation `domR :: Config -> Config` is called dominance relation if

1. `domR` is a preorder
2. `domR a b` $\implies$ `domR a' b'`, where `a' = algR (func a)` and `b' = algR (func b)`
3. `domR` $\subseteq$ `r`

where relation `r :: Config -> Config` is the preorder with respect to the objective function value. The second condition is essentially the point-wise expression of (59). The preorder requirement is necessary because we need transitivity to make the thinning process more efficient.

Global upper bound dominance relation 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, by using approximate algorithms, we can obtain an approximate solution very cheaply, since these algorithms typically have a low-order polynomial time complexity.

The global upper bound (GUB) dominance relation exploits the simple fact that the global optimal solution will always be at least as good as the local optimal solutions. Hence, we can use any approximate/heuristic algorithms 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 run from several hours to just a few seconds after applying this technique.

To simplify our notation, we use a triple in the form of `cnfg = (c,s,e) :: Config` to represent a configuration. This configuration triple consists of a combinatorial configuration, a data sequence, and the objective value for this configuration. Suppose we have another *fictitious configuration* `fict = (_,_,ub)`[11]. We define the GUB relation as `gubdomR fict cnfg = True` if `leq ub e = True`, i.e., `ub` $\leq$ `e`.

Next, we need to show that relation `gubdomR` is indeed a valid dominance relation that satisfies the monotonicity. According to Def. 16, a relation `gubdomR` is defined based on `leq`, thus, it is a preorder. Also `gubdomR` satisfies the following monotonicity condition

$$\texttt{gubdomR fict cnfg} \Longrightarrow \texttt{gubdomR fict cnfg'}, \tag{66}$$

where `cnfg' = algR (func cnfg)` is the updated configuration for partial configuration `cnfg`. For most machine learning problems, the objective function is defined as the summation of non-negative loss terms, one for each data item. This immediately implies that `eval cnfg'` $\geq$ `eval cnfg` $\geq$ `ub`. This holds true for any problems where the objective value does **not** decrease after the configuration is updated.

Finite dominance relation The finite dominance relation is another generic and useful dominance relation which explores the finiteness of the dataset. Since we have only a limited amount of data, in many applications, we can estimate how the objective function value of a partial configuration will change if we extend it to complete. There are two types of estimations: we 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` 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. We now formalize this idea and prove it in the context of SDP.

Assuming we have 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 finite dominance relation is defined as `fdomR a b == True` if `pes_ub a` $\leq$ `opt_lb b`. We need to verify

$$\texttt{fdomR a b} \Longrightarrow \texttt{fdomR a' b'}, \tag{67}$$

where `a' = algR (func a)` and `b' = algR (func b)` is the updated configuration for partial configuration `a` and `b`. This implication holds because of the properties of `pes_ub` and `opt_lb` functions. The `pes_ub` has a property that `pes_ub a'` $\leq$ `pes_ub a`, because function `pes_ub a` evaluate the "worst-case" objective value of `a`, any update `a'` of `a` will have smaller or equal worst-case objective value, since `a'` may not be the *worst update* of `a`. Similarly, we have `opt_lb b'` $\geq$ `opt_lb b`, because any update `b'` may not be the *best update* of `b`. Therefore, the above implication trivially holds.

Another way to use finite dominance relation is to combine it with the global upper bound technique. We know that 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 we can freely discard it. To verify this intuition, we define a

[11] In Haskell, symbol "_" denotes that the value is irrelevant.

dominance relation as `fubdomR fict a == True` if `ub` $\leq$ `opt_lb a`. We need to verify `fubdomR` satisfies

$$\texttt{fubdomR fict a} \Longrightarrow \texttt{fubdomR fict a}. \tag{68}$$

This implication is true because `opt_lb a'` $\geq$ `opt_lb a` $\geq$ `ub`.

The below two examples illustrate the applications of finite dominance relation in machine learning problems.

Example 5. *Classification problem.* In our previous work on the 0-1 loss classification problem [Xi and Little, 2023], we applied the finite dominance relation 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 step n. Hence if a configuration `loss a + n <= loss b`, we can know configuration `b` will be non-optimal, then `a` dominate `b` . This result can be easily generalized to the weighted 0-1 loss classification problem and other loss functions with non-decreasing objectives concerning increasing 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 us to eliminate any partial configurations that have an optimistic lower bound greater than the global upper bound.

II.2.7 Backtracking and branch-and-bound

Branch-and-bound (BnB) algorithms are methods for global optimization in combinatorial optimization problems. It has been broadly 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, and 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 research.

The underlying idea 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 that we introduced above. If the computed lower bound exceeds the objective value of the best upper bound currently available, it 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. Starting from a seed configuration, it is recursively updated according to some procedure, once the partial configuration becomes infeasible with respect to a predicate, then we *backtrack* to the previous feasible partial configuration.

On the other hand, in the study of BnB algorithms. The problem requires finding one optimal solution instead of generating all possible configurations. The 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.

Branch-and-bound 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, we will explore how to formally characterize

the BnB method within our framework. As we noted in Section I.2.2, BnB algorithms consist of four main factors: *branching rules*, *pruning*, *bounding rules*, and *search strategies*. Through our exposition, we can immediately connect these terms to decision functions, the thinning process, and the dominance relations that we have introduced above. However, search strategies and backtracking techniques have not been discussed in previous sections. In this section, we will analyze these aspects within our framework. As we will show, 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.

In Section II.1.2, we illustrated the combinatorial generation tree for nearly all the SDP generators we introduced. Drawing a generation tree diagram for a catamorphism generator is the same as representing a *recursion* as an *iteration*. Indeed, transforming a recursion as iteration is always possible for any recursion that is defined by a catamorphism. Indeed, 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 stage after the introduction of constructive algorithmics by Meertens [1986], his proof relies on an inductive style and spans nearly five pages. We now provide a simpler characterization based on the definition of catamorphisms.

Theorem 4. *Branch-and-bound characterization theorem.* Given a catamorphism `cata alg x` on cons-list `ListFr a`, where `x = [an...a1]` is a finite list. We have following equality

```
cata alg x = concat $ map (alg_pt_iter x 0 algR_pt) e,                    (69)
```

where `e = alg_pt Nil`, and `alg_pt_iter` is defined as

```
algR_pt_iter x i algR_pt y
    | i == length x = [y]
    | i < length x = concat $ map (algR_pt_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 (69) is referred to as an iterative catamorphism or branch-and-bound algorithm (catamorphism with backtracking).

Proof. A catamorphism over cons-list can be expended as

```
cata alg [an..a1] = alg (Cons an (alg ... alg (Cons a1 (alg []))))),          (70)
```

we have the following equational reasoning steps

```
  alg (Cons an (alg ... alg (Cons a1 (alg []))))
≡ let e = alg []
  alg (Cons an (alg ... alg (Cons a1 e)))
≡ equivalent equation
  alg . (Cons an) (alg ... alg . (Cons a1) e)
≡ equation (54) 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
```

□

Indeed, a seed configuration `e` is repeatedly extended by `algR_pt` in a *depth-first way* because the *laziness* of Haskell[12]. If a partial configuration fails to pass the filter or is proven to be non-optimal, the process "backtracks" to the last valid point where further alternatives are still available.

[12]Haskell is a lazy language. It 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.

The iterative catamorphism in (69) is performed in a depth-first way, because the `map` function always starts from the *first* elements in the list. We can generalize the iterative catamorphism (69) (BnB algorithms) to perform *arbitrary search strategies* by changing the definition of the map function

```
map' :: (Ord a, Ord b) => [a] -> [a] -> (a -> b) -> [a] -> [b]
map' (sort r) f xs = map f ((sort r) xs)

iter_cata :: [a] -> (ListFr a [a] -> [[a]]) -> [[a]]
iter_cata x algR_pt = concat $ map' (alg_pt_iter x 0 algR_pt) e
  where e = [[]]
```

where `sort r :: Rel a a -> [a] -> [a]` sort the original list with respect to a *total ordering* `r` and the `map` function in `alg_pt_iter` should change accordingly. If `r` is the total order with respect to the objective value, then the program `iter_cata` is performed in a *best-first way*.

The iterative catamorphism, `iter_cata`, provides proof of exhaustiveness for combinatorial optimization and generation algorithms that use backtracking techniques. In other words, it demonstrates that *the BnB algorithm can be derived from an exhaustive search specification* defined by a cons-list catamorphism. Furthermore, `iter_cata` provides a formal and generic definition for the BnB method, where different search methods can be implemented by simply modifying the `map'` function.

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 crucial 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. In contrast, the ordinary catamorphism generator based on the join-list is embarrassingly parallelizable, requiring zero communication between processors.

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.2.8 Recursive optimization framework

II.2.8.1 Hylomorphism recursive optimization framework

Given a COP that is specified in the form

$$\texttt{sel r . } \Lambda\texttt{(hylo algRf coalgR)}, \tag{71}$$

consider only the case where the relational algebra is a function `algRf :: func a -> a` (function is a special case of relation), we call it *relational function-algebra*. Note that both relational function-algebra `algRf` and its functional correspondence `alg :: func [a] -> [a]` are functions, we distinguish them through their type information).

Given a problem specified in the form of (71), Bird and De Moor [1996] present the following theorem.

Theorem 5. *Dynamic programming theorem.* If `algRf` is monotonic on `R`, then the solution to specification (71) can be obtained by

$$\texttt{hyROF = sel r . (map (algRf . (fmap hyROF))) . } \Lambda\texttt{coalgR}), \tag{72}$$

where the relational algebra has type `algRf :: func a -> a`, and the power transpose of the relational coalgebra has type Λ`coalgR :: a -> [func a]`.

When the DP algorithm exists, it is precisely when the condition of Theorem 5 holds for the *cons-list datatype*. Considering the greater expressivity of hylomorphism, which extends beyond the cons-list datatype, thus Theorem 5 provides a more general and powerful framework for constructing recursive optimization programs.

We can implement (71) in Haskell as

```
hyROF :: (Functor func, Eq a)=> 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 we can not implement power transpose in Haskell, so we use function `coalgR_pt` to represent the function after applying power transpose to relational coalgebra `coalgR`. It is very easy to verify that when relational function-coalgebra `coalgR` is the terminal algebra `out`, the program (72) becomes a catamorphism[13].

In many cases, for some special relations, the selector relation `r` is difficult to be implemented in a functional language. To address this issue, we can instead replace selector `minlist r` with a function `polymin` of type `polymin :: [b] -> b`, and `hyROF` can be reformulate as

```
hyROF_poly :: (Functor func,Eq a)=> ([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, and thinning algorithm is applicable, we have the following corollary.

Theorem 6. *Hylomorphism recursive optimization theorem.* If `algRf` is monotonic on `R`, given a dominance relation `domR :: Rel (func a) (func a)`, if `domR` is a preorder satisfying

$$\texttt{algRf . (fmap (hylo algRf coalgR)) . domR}^\circ \subseteq \texttt{r}^\circ\texttt{. algRf . (fmap (hylo algRf coalgR))}, \tag{73}$$

then solution obtained by

$$\texttt{hyROF\_thin = sel r . (map (algRf . (fmap hyROF))) . thin domR . }\Lambda\texttt{coalgR)}, \tag{74}$$

is a solution of (71).

We can implement `hyROF_thin` 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
```

We refer to problems that satisfy the conditions given by Thm. 6 are problems that can be solved by the *hylomorphism recursive optimization framework* (Hy-ROF).

II.2.8.2 Catamorphism recursive optimization framework

In many cases, specifying a COP specified through hylomorphism (56) 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 (57). Compared with hylomorphisms, the sequential decomposition process in catamorphism provides more 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 COP problem specified as (57), 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

$$\texttt{sel r . cata (thin domR . alg)}, \tag{75}$$

is also a solution of (57), 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`, (75) becomes a greedy algorithm

We call any problems that satisfied the condition given by Corollary 4, the problems that can be solved by the *catamorphism recursive optimization framework* (Cata-ROF).

[13] By applying the rule `min r . map X` $\subseteq$ `X .` $\in$ and $\in$ `.` Λ `= id`. The inclusion becomes equality when `X` is a function

II.2.9 Reconcile combinatorial optimization methods

Inclusion relationships between different combinatorial optimization methods Now, after introducing the hylomorphism and catamorphism recursive optimization frameworks, it is time to rediscuss the question, what is the recursive optimization framework, and how are classical CO methods related to it?

The recursive optimization framework is a way to construct efficient recursive programs for solving COPs. It subsumes the classical algorithm design strategies, such as greedy method, dynamic programmings and divide-and-conquer methods, and they have the following inclusion relations

$$\text{SDP} \subseteq \text{Greedy algorihtm} \subseteq \text{BnB} \subseteq \text{General SDP} \subseteq \text{ DP} \subseteq \text{General D\&C}, \tag{76}$$

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*. The inclusion relations above are implied by the following inclusion relations of their abstractions

$$\text{Catamorphism} \subseteq \text{Cata-ROF} \subseteq \text{Hy-ROF}. \tag{77}$$

The most basic strategy is the ordinary sequential decision process, which corresponds to catamorphism. According to Greedy Theorem 2 and Branch-and-Bound Characterization Theorem 4, both the greedy and BnB method are subsumed within Cata-ROF, which can be understood as incorporating a ordinary catamorphism with a thinning algorithm and alternative search strategies. The greedy algorithm is a special case of the thinning algorithm, and the backtracking technique can be incorporated by reformulating the catamorphism.

Similarly, DP algorithms are characterized by the Dynamic Programming Theorem 5, which can be further refined based on whether the DP problem's subproblems have a *layered* or *symmetric* structure [Bird, 2008]. However, the Thm. 5 itself does not allow one to use the memoization technique which is often considered as the main characteristic of the DP algorithm. We argue 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 external optimization. The property of overlapping subproblems is what distinguishes the DP method from classical D&C method. The classical D&C method decomposes a problem into subproblems arbitrarily, **without** overlap. In contrast, dynamic programming always **involves** overlapping subproblems in its decomposition. Clearly, both DP and classical D&C methods can be represented by hylomorphism, i.e., general D&C.

When implementing memoization, an alternative perspective can be helpful by viewing DP algorithms as a form of the *course-of-values recursion*. This approach integrates catamorphism with context-sensitive computations modeled by *comonads*. In this way, DP can be characterized as a histomorphism [Hinze and Wu, 2013].

We classify BnB method as a subclass of DP method because the recursive structure of DP algorithms is more complex than a ordinary catamorphism. A detailed analysis of their difference will be provided in the next subsection.

We can now conclude the inclusion relation as described in (5) is indeed valid.

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 the DP method are subclass of BnB method. However, we disagree with this view for two reasons.

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 the branching rules as the decomposition of the complete problem into *disjoint* subproblems.

Second, using alternative search strategies in problems 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 power of our framework, we use it 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 another is well-known to have a DP solution. In this section, we show how to design these two algorithm from scratch.

In particular, for each COP, we sometimes need to solve either the *optimal configuration problem* or the *optimal value problem*. The optimal configuration problem involves finding the optimal configuration of a COP, whereas the optimal value problem focuses on determining the optimal objective value of a COP. In our framework, these tasks can be achieved by modifying the corresponding algebra or coalgebra. In nearly every case, the optimal value of the problem can be obtained from the optimal configuration. However, executing a program that produces the optimal value is often more efficient in terms of actual wall-clock run-time, as the optimal value problem primarily involves numerical operations.

In the analysis of this section, we will provide 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 is well-known and has a greedy solution. In this section, we explain how to derive an efficient algorithm for this problem within our framework. Based on our 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`. 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
```

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, we can integrate the selector `maxlist` 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, we need to show the relational algebra of the generator satisfies the greedy condition given in Thm. 2, i.e., we need to prove `subsalgR` is monotonic with respect to `r° = sum° . leq . sum`. 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 metaphorically to represent logical "or" since we cannot define a relation in Haskell.

In the context of SDP, the meaning of monotonicity is given in (59). Let's verify this is true for algebra `subsalgR` and relation `r° = sum° . leq . sum`. If two configuration (sublists) `x` and `y` satisfies `r° x y = True`, i.e., `x` have a smaller sum value compared with `y`, then it is easy to verify `r° x y` $\subseteq$ `r° x' y'`, where `x' = subsAlgR (Cons a x)` and `y' = subsAlgR (Cons a y)`. Following the result of the Greedy Theorem 2, we have

$$\texttt{cata (max . } \Lambda\texttt{subsalgR)} \subseteq \texttt{max . } \Lambda\texttt{(cata subsalgR)}$$

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, we may be interested only in obtaining 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° = sum° . leq . sum`. 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`.

II.2.10.2 Sequence alignment problem

The sequential alignment problem is a well-known problem in bioinformatics that can be solved by using a dynamic programming algorithm. We have two DNA or protein sequences, and we want to infer if they are homologous or not. To do this, we need to measure the similarity of the two sequences `x`and `y`, and their similarity is measured by finding the *shortest* editing sequence. In the basic case, we assume there are just three basic editing operations: *match*, *delete* and *insert*. The match operation matches a character `a` which is an element of both in `x`and `y`, the delete operation delete a character `a` from `x`, and the insert operation insert a character `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 If fact, the sequence of editing operations contains enough information to recover 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]`, we can easily define a cons-list algebra such that the *catamorphism* of it receives an editing sequence and recursively constructs the original two strings from scratch.

```
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
```

For instance, evaluating `cata editalg [Del 2,Ins 1,Mat 3]` 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 important reason to use this relational framework. 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, its converse `edit°` should be a relation that takes a pair of sequences and returns an editing sequence. In other words, `edit°` should possess a type information `edit° :: ([a],[a]) -> [Op a]`. As a result, from the definition, `Λedit°` should have a type `Λedit° :: ([a],[a]) -> [[Op a]]`, i.e., a function that takes a pair of sequence and returns *all* editing sequence. Hence the specification of the sequence alignment problem can be formally characterized as

$$\texttt{min r . } \Lambda(\texttt{edit}^\circ) \tag{78}$$

where `edit° = ana editalg°` because the `edit` is defined as a catamorphism, its converse `edit°` should be an anamorphism. Although `editalg` is a function, its inverse `editalg°` should be a non-deterministic relation, since any pair of sequences has three possible choices of operations. Let `editalg° = 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)) = Cons (Mat a) (x,y)
                           or Cons (Del a) (x,b:y)
                           or Cons (Ins b) (a:x,y)
```

the derivation of `uneditCoalgR` based on the fact 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, we construct a new `Mat` operation, or we can either delete the first element from `x` or insert a new element to `y`. Observe that, when `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 (78), we need to verify the algebra for (78) is monotonic to the selector relation `r`. However, at first glance, there exists no algebra for specification (78), since this is an anamorphism instead of a hylomorphism.

Nonetheless, based on our previous discussion, we know that 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
```

By defining the selector relation as `r = length° . leq . length` , it is trivial to verify that initial algebra `in` is monotonic with respect to `r`. Therefore, according to Thm. 5, the problem (78) 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 problem and generator In order to solve the optimal value problem, we simply need to simply modify the algebra `mesAlg` as following

```
mesValAlg :: ListFr (Op a) Int -> Int
mesValAlg Nil = 0
mesValAlg (Cons a acc) = 1 + acc
```

which 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 editing 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
```

On the other hand, for the generation problem, it is not easy to define the selector relation `r` that is used to define the selector, it will involves to define many terms that will not be used elsewhere in the thesis. To simplify our discussion we can instead of defining the selector for the generator problem as `sel r = concat`, the `concat :: [[a]] -> [a]` function flatten a list of lists to a list by removing inner brackets, it thus satisfies the type requirement in order to use program `hyROF_poly`, thus the generator for the sequence alignment problem

```
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])` generates all possible editing sequence
`[[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

In this chapter, we present 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 simplicity and elegance, Bird and De Moor [1996]'s relational calculus may be hard to grasp for uninitiate, as most of us are more familiar wit functions. It is natural to ask whether this formalism can be expressed purely in functional. 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, we have 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. In the discussion of Part III, we demonstrate how these basic generators, along 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] did for sublist, permutation, and partition generators over cons-list catamorphisms.

We believe the algorithm design framework proposed here is particularly important in the design of exact 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. In other words, it not only enables the creation of correct programs but also provides simple proofs for their correctness. In contrast, formal proofs for classical BnB algorithms often involve tedious inductive steps, and proofs for MIP solvers require managing generative recursions, making termination proofs necessary and parallelization difficult.

II.3 Combinatorial geometry

In this Chapter, we focus on geometry objects consist of *finite hyperplanes* and *data points*. Two key reasons motivate our study of these objects: First, the combinatorics of these two objects are well-studied and the combinatorial problems related to these two objects are central topics in combinatorial geometry. Second, 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 combinatorial essence of these problems.

A dissection of $\mathbb{R}^D$ by a finite number of hyperplanes is called a hyperplane arrangement. At first glance, 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 configuration and hyperplane arrangement, we will see later 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 structures.

Our goal in this chapter is to demonstrate that applying geometric principles can provide new insights and interpretations into the combinatorics of various combinatorial 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 our discussion.

In this section, we provide a brief introduction to the foundational geometric definitions that will be frequently used in our discussion. Throughout the discussion, we limit our focus to Euclidean space over the real closed field $\mathbb{R}$.

II.3.1.1 Affine varieties and polynomials

We denote both the affine D-space and the vector D-space over $\mathbb{R}$ as $\mathbb{R}^D$, both of them 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 17. *Monomial.* A *monomial* with respect to a D-tuple $\boldsymbol{x} = (x_1, x_2 \ldots, x_D)$ is a product of form

$$M = \boldsymbol{x}^{\boldsymbol{\alpha}} = x_1^{\alpha_1} \cdot x_2^{\alpha_2} \ldots \cdot x_D^{\alpha_D}, \tag{79}$$

where $\boldsymbol{\alpha} = (\alpha_1, \alpha_2 \ldots, \alpha_D)$, $\alpha_1, \alpha_2 \ldots, \alpha_D$ 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 18. *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 $\mathbb{R}$ (field)) of monomials. We will write a polynomial $P(\boldsymbol{x})$, or P in short, in the form

$$P(\boldsymbol{x}) = \sum_i w_i \boldsymbol{x}^{\boldsymbol{\alpha}_i}, w_i \in \mathbb{R}, \tag{80}$$

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 as $\mathbb{R}[x_1, x_2 \ldots, x_D]$ or $\mathbb{R}[\boldsymbol{x}]$ if variables $\boldsymbol{x} = (x_1, x_2 \ldots, x_D)$.

Let $P = \sum_i w_i \boldsymbol{x}^{\boldsymbol{\alpha}_i}$ be a polynomial in $\mathbb{R}[\boldsymbol{x}]$, we call $\boldsymbol{\alpha}_i$the *coefficient* of the monomial $\boldsymbol{x}^{\boldsymbol{\alpha}_i}$. If $w_i \neq 0$, then we call $w_i \boldsymbol{x}^{\boldsymbol{\alpha}_i}$ a *term* of P. The *maximal degree* of P, denoted $deg(P)$, is the maximum $|\boldsymbol{\alpha}_i|$ such that the coefficient a_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.

The number of possible monomial terms of a degree K polynomial is equivalent to select K variables from the multisets of $D+1$ variables[14]. This is again equivalent to the *size K combinations of $D+1$ elements with replacement*. In other words, we need to select K variables from variables set $(x_0, x_1, \ldots, x_D)$ in homogeneous coordinate with repetition, and we have the following fact.

Fact 5. 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 term at most.

As an example, a polynomial in $\mathbb{R}[x_1, x_2]$ with maximal degree 2 (a 2-dimensional conic section) has 6 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 *polynomial ring*.

When we have a polynomial $P \in \mathbb{R}[\boldsymbol{x}]$, we can talk about the set of *zeros* of P, namely $V(P) = \left\{(x_1, x_2 \ldots, x_D) \in \mathbb{R}^D : P(x_1, x_2 \ldots, x_D) = 0\right\}$. Geometrically, the zeros set of a polynomial P determines a subset of $\mathbb{R}^D$, this subset is called *surface* and it has dimensionality smaller than D, and it is called *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, we want to know the intersection of the zeros sets for a set of polynomials, which is equivalent to the intersection set of their corresponding surfaces in $\mathbb{R}^D$. This intersection set is called *algebraic variety*. We can define it formally as follows.

Definition 19. *Algebraic variety.* Given a set of polynomial $\mathcal{P} = \{P_i : i \in \mathcal{I}\}$ 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

$$Var(\mathcal{P}) = \left\{(x_1, x_2 \ldots, x_D) \in \mathbb{R}^D : P_i(x_1, x_2 \ldots, x_D) = 0, \forall i \in \mathcal{I}\right\}, \tag{81}$$

is called the affine variety defined by $\mathcal{P}$.

In particular, the algebraic variety for one polynomial P_i is a surface denoted as $S_i = \left\{\boldsymbol{x} \in \mathbb{R}^D : P_i(\boldsymbol{x}) = 0\right\}$, 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

In this Subsection, we review basic terminology and combinatorics of the arrangement of the *hypersurface* and *hyperplane*.

Definition 20. *Surface arrangement.* Given a system of polynomials $\mathcal{P} = \{P_i : i \in \mathcal{I}\}$. A finite *surface arrangement* $\mathcal{S}_{\mathcal{P}} = \{S_i : i \in \mathcal{I}\}$, where $S_i = \left\{\boldsymbol{x} \in \mathbb{R}^D : P_i(\boldsymbol{x}) = 0\right\}$, is a finite set of surfaces defined by polynomial system $\mathcal{P}$. We call $\mathcal{P}$ a *central* surface arrangement, if the coefficient w for the zero degree term $w\boldsymbol{x}^{\mathbf{0}}$ equal to zero.

In particular, if these surfaces are hyperplanes (with dimension $D-1$ and maximal degree-one), we obtain the definition of hyperplane arrangement.

Definition 21. *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. We call $\mathcal{H}$ 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 *affine hyperplane* if $c \neq 0$, and *linear hyperplane* if $c = 0$.

A vitally important assumption in the study of arrangements is called *general position*. It means the *general case situation*, as opposed to some more special or coincidental cases that are possible, which is 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 defined as below.

[14]There are $D+1$ variables if we consider polynomial in homogeneous coordinate, i.e., projective space $\mathbb{P}^D$

Definition 22. *General position for hyperplanes.* A hyperplane arrangement $\mathcal{H} = \{H_i : i \in \mathcal{I}\}$ is in *general linear position* (or general position in short) if

$$\{H_1, ..., H_k\} \in \mathcal{H},\ 1 \leq k \leq D \Longrightarrow \dim(H_1 \cap \cdots \cap H_k) = D - k, \tag{82}$$

where $\dim(H_1 \cap \cdots \cap H_k)$ is the *dimension* of the set $H_1 \cap \cdots \cap H_k$ and

$$\{H_1 \cap \cdots \cap H_k\} \in \mathcal{H},\ k > D \Longrightarrow H_1 \cap \cdots \cap H_k = \{\emptyset\}. \tag{83}$$

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 only the normal vectors of every D hyperplanes in $\mathcal{H}$ are *linearly independent*.

Definition 23. *Sign vector and hyperplane arrangement.* Given a *hyperplane* arrangement $\mathcal{H} = \{H_i : i \in \mathcal{I}\}$. We denote with $H_i^+ = \left\{\boldsymbol{x} \in \mathbb{R}^D : \boldsymbol{w}_i^T \boldsymbol{x} > b_i\right\}$, $H_i^- = \left\{\boldsymbol{x} \in \mathbb{R}^D : \boldsymbol{w}_i^T \boldsymbol{x} < b_i\right\}$ the open sets "above" or "below" H_i. We associate to each point $\boldsymbol{x} \in \mathbb{R}^D$, *the sign vector* of data point $\boldsymbol{x}$ with respect to arrangement $\mathcal{H}$ is 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 defined as

$$\delta_i(\boldsymbol{x}) = \begin{cases} +1 & \boldsymbol{x} \in H_i^+ \\ 0 & \boldsymbol{x} \in H_i \\ -1 & \boldsymbol{x} \in H_i^- \end{cases}, 1 \leq i \leq I. \tag{84}$$

Similarly, for more general case, we 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. Also, 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 the discussion on hyperplane arrangements. In particular, we are interested in the connected components (regions) intersected by hyperplanes. These connected regions are the *faces* of an arrangement, which is defined as follows.

Definition 24. *Faces of a hyperplane arrangement.* We denote by $\mathcal{F}_{\mathcal{H}}$ the set of all sign vectors $\text{sign}_{\mathcal{H}}(\boldsymbol{x})$ in $\mathbb{R}^D$ for arrangement $\mathcal{H}$, which is defined as

$$\mathcal{F}_{\mathcal{H}} = \left\{\text{sign}_{\mathcal{H}}(\boldsymbol{x}) : \boldsymbol{x} \in \mathbb{R}^D\right\}, \tag{85}$$

A *face* f (connected component) of an arrangement $f \subseteq \mathbb{R}^D$is a subset of $\mathbb{R}^D$, such that all $\boldsymbol{x} \in f$ has 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, we denote $\text{sign}_{\mathcal{H}}(f)$ as the sign vector for any point in f.

Different vectors in $\mathcal{F}_{\mathcal{H}}$ define different faces of the arrangement $\mathcal{H}$. We call a face *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 closure 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.

We assume 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 the arrangements

Before we study the theory of arrangements, we first investigate the combinatorial complexity of hyperplane and hypersurface arrangements, and these results will become useful when we analyze the combinatorial complexity of the problem.

Theorem 7. *Faces 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

$$F_D(\mathcal{H}) = \sum_{i=0}^{k} \binom{D-i}{k-i} \binom{N}{D-i}. \tag{86}$$

The maximum is attained exactly when $\mathcal{H}$ is simple [Toth et al., 2017].

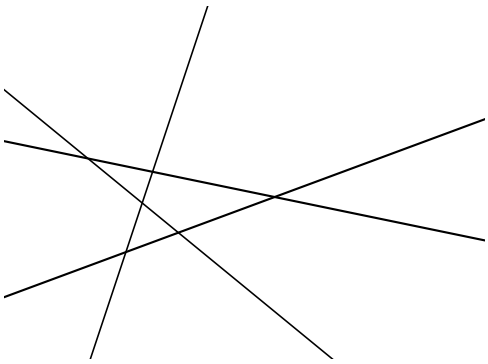


Figure II.3.1: An arrangement of four lines in $\mathbb{R}^2$. Four lines intersect such that no three lines meet at a single point. This arrangement divides the plane into three closed regions (bounded cells) and eight open regions (unbounded cells).

A special case of (86) is when $k = D$, we obtain the following lemma for counting the number of cells of an arrangement.

Lemma 4. *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

$$C_D(\mathcal{H}) \leq \sum_{d=0}^{D} \binom{N}{d}. \tag{87}$$

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

$$C_D(\mathcal{H}) \leq 2 \sum_{d=0}^{D-1} \binom{N-1}{d}. \tag{88}$$

The maximum is attained exactly when $\mathcal{H}$ is simple [Fukuda, 2016].

For instance, the non-central arrangement of four lines shown in Fig. II.3.1 has 11 cells.

The cell counting formula for central arrangements (88) is, in fact, equivalent to the well-known *Cover's dichotomies 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** hyperplane) is given by:

$$\text{Cover}(N, D+1) = 2 \sum_{d=0}^{D} \binom{N-1}{d} \tag{89}$$

Astute readers may notice that in this summation, the upper limit is D rather than $D-1$, as seen in (88). This difference arises because a dataset in the general position in $\mathbb{R}^D$ is isomorphic to a central arrangement in $\mathbb{R}^{D+1}$. The relationship between Cover's dichotomies 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 called 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, we can calculate the number of bounded cells by the following Lemma [Stanley et al., 2004].

Lemma 5. 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

$$B_D(\mathcal{H}) \leq \binom{N-1}{D}. \tag{90}$$

The maximum is attained exactly when $\mathcal{H}$ is simple.

As depicted in Fig. II.3.1. In a simple arrangement of four lines, the number of bounded cells is three.

Theorem 8. *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 $\mathcal{P}$[15]. The asymptotically highest complexity is obtained when the surfaces are in general positions [Sharir, 1994].

II.3.1.4 Points and hyperplanes duality

The concept of duality in geometry establishes a profound relationship between points and hyperplanes, revealing their inherent one-to-one correspondence. By constructing a dual transformation, points can be systematically mapped to hyperplanes and vice versa, while preserving incidence relations between them.

In an affine space, a *point configuration* is a set of data points $\boldsymbol{p} = (p_1, p_2, \ldots, p_n) \in \mathbb{R}^D$, $n \in \mathcal{N} = \{1, \ldots, N\}$. When we fix an origin, a point configuration becomes a set of vectors, which is equivalent to the data set that we defined previously, thus we denote a point configuration as $\mathcal{D} = \{\boldsymbol{p}_n : n \in \mathcal{N}\}$.

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 $k = 2, 3, ..., D+1$. We will see 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 25. *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

$$p_1 x_1 + p_2 x_2 + ... + p_{D-1} x_{D-1} - x_D = p_D, \tag{91}$$

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 $\phi^{-1}(H) = (w_1, w_2, \ldots, w_D)^T$.

We use the terms *primal space*, and *dual space* to refer to the spaces before and after transformation by ϕ and ϕ^{-1}. The dual transformation is naturally extended 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. We sometimes denote the dual hyperplane arrangement of points set $\mathcal{D}$ as $\mathcal{H}_{\mathcal{D}}$ and the dual point configuration of $\mathcal{H}$ as $\mathcal{D}_{\mathcal{H}}$.

Theorem 9. *The Incidence relations of dual transformation.* Given $\boldsymbol{p}$ 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 $\phi(\boldsymbol{p})$.

The dual transformation preserves the incidence relations 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 diagram

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. The applications of the Voronoi diagram 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 diagram.

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.

[15] However, if the maximal degree of the $\mathcal{P}$ is very high, there will be a very large constant term in the exponent hided by the big O notation

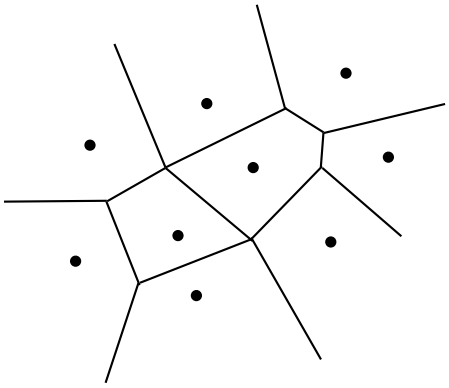


Figure II.3.2: An Euclidean Voronoi diagram in $\mathbb{R}^2$. The black points represent the centroids of each Voronoi cell (connected regions in $\mathbb{R}^2$), and the black lines denote the boundaries of the connected regions.

Definition 26. *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

$$d_p(\boldsymbol{x}, \boldsymbol{y}) = \left(\sum_{i=1}^{D} |y_i - x_i| \, |^p \right)^{1/p}. \tag{92}$$

A special case of this metric is the *Euclidean distance* or L_2 *distance* is defined as

$$d_2(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|_2^2 = \|\boldsymbol{y} - \boldsymbol{x}\|_2^2 = \sqrt{\sum_{i=1}^{D} (y_i - x_i)^2}, \tag{93}$$

similarly we have L_1 *distance*

$$d_1(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{y} - \boldsymbol{x}\|_1 = |\boldsymbol{y} - \boldsymbol{x}| = |\boldsymbol{y} - \boldsymbol{x}| = \sum_{i=1}^{D} |y_i - x_i|. \tag{94}$$

Definition 27. *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 $\mathcal{V}(\mathcal{D})$ of 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 $\mathcal{D}$ partition the ambient space $\mathbb{R}^D$ into *Voronoi regions/polygons* $\mathcal{V}(\mathcal{D}) = \{V_1, V_2, \ldots, V_N\}$ defined by

$$V_n = \left\{ \boldsymbol{x} \in \mathbb{R}^D \mid d(\boldsymbol{x} - \boldsymbol{x}_n)^2 \leq d(\boldsymbol{x} - \boldsymbol{x}_j)^2, \forall n, j \in \mathcal{N} \wedge k \neq j \right\}. \tag{95}$$

We can safely 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 *Euclidean Voronoi diagram*. An example of Euclidean Voronoi diagram is depicted in Fig. II.3.2. We may define various 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*, it is referred to as a *Bregman Voronoi diagram*. In this thesis, however, we will focus on investigating the Euclidean Voronoi diagram in more detail.

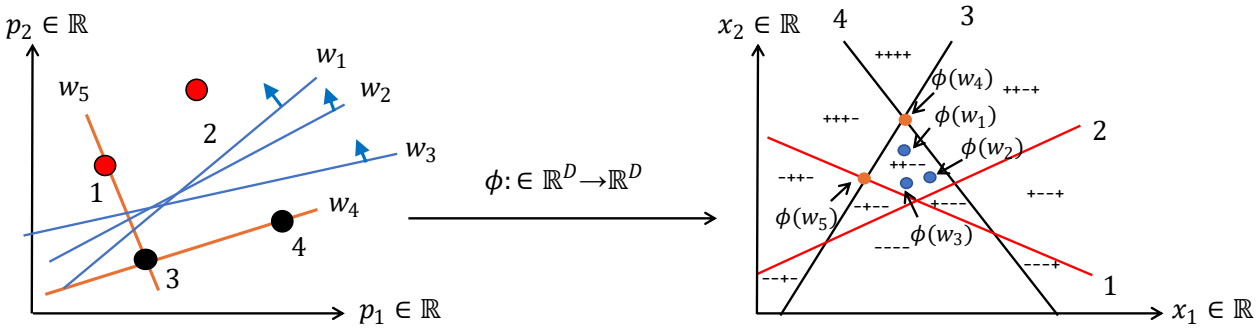


Figure II.3.3: A point configuration $\mathcal{D}$ (left-panel) and its dual arrangement $\mathcal{H}_{\mathcal{D}}$ (right-panel). The yellow hyperplanes w_4, w_5 with two points lies on it in $\mathbb{R}^D$ corresponds to the yellow points in the dual space, which is the intersection of corresponding 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 $\phi(w_1)$, $\phi(w_2)$, $\phi(w_2)$ lies in the same cell of dual arrangement $\phi(\mathcal{D})$.

II.3.2 Classification problems and duality

The classification problems, especially the linear classification problems, are the central theme of machine learning studies. The effectiveness of a linear classifier depends on 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.

In this section, we 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, we gain a new perspective on the linear classification problem. This perspective can facilitate the development of an efficient and general algorithm capable of solving linear classification problems with arbitrary objectives. Then we will generalize our discussion to non-linear (polynomial hypersurface) classification problems.

II.3.2.1 Linear classification and duality

Previously, we have introduced that the combinatorial complexity of linear dichotomies with respect to data set $\mathcal{D}$ is given by (89). However, Cover's theorem only provides insights into the combinatorial complexity of possible dichotomies. How to enumerate these dichotomies remains an open problem.

Indeed, in our previous studies [He and Little, 2023], we showed 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 the Thm. 9. 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. II.3.3).

In this Subsection, we begin 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 algorithms that we design.

We have 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, we explain the correspondence between Cover's dichotomies and the cells of a dual arrangement $\mathcal{H}_{\mathcal{D}}$, which offers an alternative approach to proving Cover's counting theorem.

Lemma 6. For a set points $\mathcal{D} = \{x_n \in \mathbb{R}^D : n \in \mathcal{N}\}$ 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 $\mathcal{H}_{\mathcal{D}}$.

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 the number of linear separable dichotomies given by affine hyperplanes is

$$\text{Cover}\left(N, D+1\right) = 2\sum_{d=0}^{D}\left(\begin{array}{c} N-1 \\ d \end{array}\right). \tag{96}$$

The original Cover's function counting theorem counts the number of linear 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 $\mathbb{R}^{D+1}$($\bar{\mathcal{D}}$ is the *homogeneous dataset*, which is obtained by embedding $\mathcal{D}$ in homogeneous space)

In Subsection II.3.1.3, we have shown 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 information together, we obtain

$$\begin{aligned} B_D\left(\mathcal{H}_{\mathcal{D}}\right) + C_D\left(\mathcal{H}_{\mathcal{D}}\right) =& \left(\begin{array}{c} N-1 \\ D \end{array}\right) + \sum_{d=0}^{D}\left(\begin{array}{c} N \\ d \end{array}\right) \\ =& \left(\begin{array}{c} N-1 \\ D \end{array}\right) + \sum_{d=0}^{D}\left[\left(\begin{array}{c} N-1 \\ d \end{array}\right) + \left(\begin{array}{c} N-1 \\ d-1 \end{array}\right)\right] \\ =& \sum_{d=0}^{D}\left(\begin{array}{c} N-1 \\ d \end{array}\right) + \sum_{d=0}^{D}\left(\begin{array}{c} N-1 \\ d \end{array}\right) \\ =& 2\sum_{d=0}^{D}\left(\begin{array}{c} N-1 \\ d \end{array}\right) \\ =& \text{Cover}\left(N, D+1\right). \end{aligned} \tag{97}$$

□

The equivalence between the number of dichotomies and the sum of bounded and unbounded 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 7. 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 $\phi^{-1}\left(H\right)$ lies above (below) hyperplane $\phi\left(\boldsymbol{x}\right)$. 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 $\mathcal{D}_H^+ = \left\{\boldsymbol{x}_n : \boldsymbol{w}^T\boldsymbol{x} > 0\right\}$ and $\mathcal{D}_H^- = \left\{\boldsymbol{x}_n : \boldsymbol{w}^T\boldsymbol{x} < 0\right\}$, and according to the Thm. 9, $\mathcal{D}$ has a unique associated dual arrangement $\phi\left(\mathcal{D}\right)$. Thus, the sign vector of the point $\phi^{-1}\left(H\right)$ with respect to arrangement $\phi\left(\mathcal{D}\right)$ partitions the arrangement into two subsets $\phi_H\left(\mathcal{D}\right)^+ = \left\{\phi\left(\boldsymbol{x}_n\right) : \boldsymbol{\nu}_{\phi(\boldsymbol{x}_n)}^T\phi^{-1}\left(H\right) > 0\right\}$ and $\phi_H\left(\mathcal{D}\right)^- = \left\{\boldsymbol{\nu}_{\phi(\boldsymbol{x}_n)}^T\phi^{-1}\left(H\right) < 0\right\}$, where $\boldsymbol{\nu}_{\phi(\boldsymbol{x}_n)}^T$ is the normal vector to the dual hyperplane $\phi\left(\boldsymbol{x}_n\right)$, in other words, point $\phi^{-1}\left(H\right)$ lies in a cell of arrangement $\phi\left(\mathcal{D}\right)$.

Next, we need to prove that bounded cells have no complement cell. The reverse assignment of the bounded cells of the dual arrangements $\phi\left(\mathcal{D}\right)$ 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 $\phi\left(\boldsymbol{x}_n\right)$ 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, we only need to take the cell f incident with $\boldsymbol{r}_1$, and in the positive direction with respect to $\phi(\boldsymbol{x}_n)$, take $\mathtt{g}$ to be the cell incident with $\boldsymbol{r}_2$, and in the negative direction with respect to $\phi(\boldsymbol{x}_n)$. In this way, we obtain two unbounded cells f and g with opposite sign vectors. It means that, for point $\phi^{-1}(H)$ in these unbounded cells, this hyperplane H partition the data set to $\mathcal{D}_H^+$ and $\mathcal{D}_H^-$, we can move the position of hyperplane H in the primal space, there exists a new hyperplane H' by moving H, it partitions the data set to $\mathcal{D}_{H'}^+ = \mathcal{D}_H^-$ and $\mathcal{D}_{H'}^- = \mathcal{D}_H^+$. In other words, H' has the 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, we can move a hyperplane from the 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 7 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. Thus we immediately have the following theorem.

Theorem 10. *Linear classification theorem.* Given a data set $\mathcal{D}$ 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 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 8. For a data set $\mathcal{D}$ in general position, a hyperplane with k data items lies 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, corresponding 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 28. *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

$$\operatorname{sep}(f, g) = \{n \in \mathcal{N} : \delta_n(f) = -\delta_n(g) \neq 0\}, \tag{98}$$

using which, we say that the two faces f, g are *conformal* if $\operatorname{sep}(f, g) = \emptyset$.

Two faces that are conformal is essentially the same as saying that two faces have consistent assignments.

Lemma 9. 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 proper cell 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 our analysis 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 10. 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 $\operatorname{sign}_{\mathcal{H}}(f)$. For an arbitrary $(D-d)$-dimension face g, $0 < d \leq D$, the number of different signs of $\operatorname{sign}_{\mathcal{H}}(g)$ with respect to $\operatorname{sign}_{\mathcal{H}}(f)$ is larger than or equal to d, where equality holds only when g is conformal to f (g is a subface of f).

Proof. Define the number of different signs of $\operatorname{sign}_{\mathcal{H}}(g)$ with respect to $\operatorname{sign}_{\mathcal{H}}(f)$ as $E_{0\text{-}1}(g)$. In a simple arrangement, the sign vector $\operatorname{sign}_{\mathcal{H}}(f)$ 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 $\operatorname{sign}_{\mathcal{H}}(g)$ with respect to $\operatorname{sign}_{\mathcal{H}}(f)$ must larger than or equal to d, i.e., $E_{0\text{-}1}(f) \geq d$. If the $\operatorname{sep}(f, g) = \emptyset$, then $E_{0\text{-}1}(g) = d$ according to definition of $\operatorname{sep}(f, g) = \emptyset$. In this case, f, g are conformal. In contrast, if f, g are not conformal, i.e., $\operatorname{sep}(f, g) \neq \emptyset$, and assuming $|\operatorname{sep}(f, g)| = C$, according to the definition of the objective function and conformal faces, $E_{0\text{-}1}(\operatorname{sign}_{\mathcal{H}}(f)) = d + C$. Hence, $E_{0\text{-}1}(\operatorname{sign}_{\mathcal{H}}(g)) \geq d$, and equality holds only when g is conformal to f. □

In the linear classification problem with the 0-1 loss objective, Lemma 10 will later be used to prove Thm. 13. 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 is consistent with the predictions of this optimal boundary with D points lies on it. Since there are only $\binom{N}{D}$ such hyperplanes, we can solve the 0-1 loss linear classification problem in polynomial time.

II.3.2.2 Growth function and the complexity classification problem

In the study of *foundational machine learning theory*, known as *statistical learning theory*. A vitally important concepts is called 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 space.

Definition 29. *Growth function.* The growth function $\Pi_{\mathbb{H}} : \mathbb{N} \to \mathbb{N}$ ($\mathbb{N}$ stands for nature numbers) for a hypothesis set $\mathbb{H}$ defined by

$$\forall N \in \mathbb{N}, \Pi_{\mathbb{H}}(N) = \max_{\{\boldsymbol{x}_n : n \in \mathcal{N}\}} \left| \left\{ \operatorname{sign}\left(\boldsymbol{w}^T \bar{\boldsymbol{x}}_n\right) : \boldsymbol{w} \in \mathbb{R}^{D+1} \right\} \right|, \tag{99}$$

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* can be realized by hypothesis.

The next result, known as *Sauer's lemma*, clarifies the connection between the notions of growth function and *VC-dimension* [Mohri et al., 2018]. The VC-dimension is a key measure of the complexity of a classification model. The simple D-dimensional *linear hyperplane* classification model, which we discuss in detail below, has VC-dimension $D+1$. This is lower than that of other widely used models, such as the decision tree model l (axis-parallel hyper-rectangles), which has a VC-dimension of $2D$; the K-degree polynomial, which has a VC-dimension of $O\left(D^K\right)$; and the L-layer, W-weight piecewise linear deep neural networks, which have a VC-dimension of $O(WL\log(W))$ [Blumer et al., 1989, Bartlett et al., 2019, Vapnik, 1999].

Lemma 11. *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

$$\Pi_{\mathbb{H}}(N) \leq \sum_{d=0}^{D} \binom{N}{d}. \tag{100}$$

It is clear that when $\operatorname{VCdim}(\mathbb{H}) = D+1$, for N data points in D-dimensional space, the Cover's functional counting theorem satisfies the above inequality

$$\begin{aligned} \operatorname{Cover}(N, D+1) =& 2\sum_{d=0}^{D} \binom{N-1}{d} \\ =& \sum_{d=0}^{D} \binom{N-1}{d} + \sum_{d=0}^{D} \binom{N-1}{d} \\ \leq& \sum_{d=0}^{D} \binom{N-1}{d+1} + \sum_{d=0}^{D} \binom{N-1}{d} \\ =& \sum_{d=0}^{D+1} \binom{N}{d+1}. \end{aligned} \tag{101}$$

Since the right-hand side of the (100) is always *polynomially* large when the VC-dimension of the hypothesis set is finite, it tells us that we can always construct a polynomial-time exact classification algorithm for any hypothesis set with finite VC-dimension.

In the next section, we will show 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 us to extend Thm. 10 from linear classification problems to polynomial hypersurface classification problems. This extension allows us to construct an algorithm that can find an optimal non-linear (polynomial hypersurface) classifier in polynomial time.

II.3.2.3 Non-linear (polynomial) classification and Veronese embedding

Based on the point-hyperplane duality, we successfully establish equivalence relations for linear classifiers on finite sets of data. However, a linear classifier is often too restrictive in practice, as many problems require more complex decision boundaries. It is natural to ask whether we can extend our theory to non-linear classification? In this section, we explore a well-known concept in algebraic geometry, the *W-tuple Veronese embedding*, which allows us to generalize our previous strategy for solving classification problem with *hyperplane classifier* to problems involves

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 for defining the hypersurface. If both W and D are fixed constant, this again gives us a polynomial 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 [2013], the following embedding in projective space is demonstrated.

Definition 30. *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 $\mathbb{R}^D$ if we forget 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 $G = \begin{pmatrix} D+W \\ D \end{pmatrix} - 1$. We define a mapping $\rho_W : \mathbb{P}^D \to \mathbb{P}^G$ by sending 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 exists 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 12. The W-tuple of $\mathbb{P}^D$ is an injective isomorphism onto its image.

Proof. see exercise 3.4, chapter I [Hartshorne, 2013]. □

A consequence of this lemma states 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 containing 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. Given 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 isomorphic to a hyperplane H in $\mathbb{P}^5$ (y_0, y_1, y_2, y_3, y_4, y_5), namely the hyperplane defined by equation $w_0 y_0 + w_1 y_1 + \ldots + w_4 y_4 + w_5 y_5 = 0$.

This explains why 5 points can determine a conic section in $\mathbb{R}^2$, because a conic section (conic section corresponds to polynomial of degree 2) in $\mathbb{R}^2$ is isomorphic to a hyperplane in $\mathbb{R}^5$.

Therefore, by showing 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 the hyperplane have duality, the hypersurface corresponds to it also have the duality. Therefore, we can generalize the Linear Classification Theorem 10 to the following hypersurface case.

Theorem 11. *Hypersurface classification theorem.* Given a data set $\mathcal{D}$ in general position in $\mathbb{R}^D$. Assume t_{eval} is the time required to evaluate the objective value. The exact solutions for hypersurface classification problem with an 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 $\rho_W(\bar{\boldsymbol{x}}) = (M_0(\bar{\boldsymbol{x}}), M_1(\bar{\boldsymbol{x}}), \ldots M_G(\bar{\boldsymbol{x}})) \in \mathbb{P}^G$ (which is equivalent to a point $(1, M_1(\bar{\boldsymbol{x}}), \ldots M_G(\bar{\boldsymbol{x}})) \in \mathbb{R}^{G+1}$).

II.3.3 Methods for cell enumeration

We have seen that the key to solving linear classification problems 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, 2023], 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, we have not yet explained how to enumerate all possible cells of an arrangement. We denote the set of all cells of $\mathcal{H}$ as $\mathcal{S}_{\text{cell}}(\mathcal{H})$. When the arrangement $\mathcal{H}$ is clear from the context, we denote it as $\mathcal{S}_{\text{cell}}$ directly. From Lemma 4, we know that the number of possible cells for an arrangement $\mathcal{H}$ is given by $|\mathcal{S}_{\text{cell}}(\mathcal{H})| = C_D(\mathcal{H}) = \sum_{d=0}^{D} \binom{N}{d}$, which is polynomial large with fixed D. Therefore, if we develop an efficient algorithm for enumerating the cells of an arrangement, we would also have a polynomial-time algorithm for solving all the aforementioned problems related to cell enumeration.

In this section, we provide a comprehensive review of methods for enumerating the cells of a hyperplane arrangement. Based on the combinatorial properties of different cell enumeration algorithms, we classify them into two major categories: *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. We will elaborate on these differences in the following discussion.

II.3.3.1 Linear programming-based method 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 an 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 a N-tuple $\{+,-\}^N$, where each $+$ or $-$ sign indicate whether a point lies on the positive or negative side of the hyperplane h_n, $\forall n \in \mathcal{N}$, one way to enumerating cells of an arrangement $\mathcal{H}$ is by generating the possible sign vectors that $\mathcal{H}$ can represents. Note, the sign vectors is equivalent to binary assignments, and there are 2^Npossible 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 $\mathcal{H}$, one can solve the following linear programming problem

$$\begin{aligned} &\min_{\boldsymbol{x}} \ \mathbf{0}^T \boldsymbol{x} \\ &\text{s.t. } \operatorname{diag}(c)^T \overline{\boldsymbol{W}} \boldsymbol{x} \geq \mathbf{1}, \end{aligned} \tag{102}$$

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. $\boldsymbol{W}$ is the normal vector matrix, where each row corresponds to a normal vector $\boldsymbol{w}_n \in \mathbb{R}^{D+1}$, $n \in \mathcal{N}$. We 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* (102), which will be the focus of this Subsection. we will refer as linear programming-based cell generation (LP-CG) algorithms, and we denote the cell feasibility predicate as p_{cell} 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, one can solve the linear program (102). 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 *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 crucial 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. We shall call c the *unique child* of c'. The definition of q is not unique, the detailed definition of q can refer to Ferrez et al. [2005].

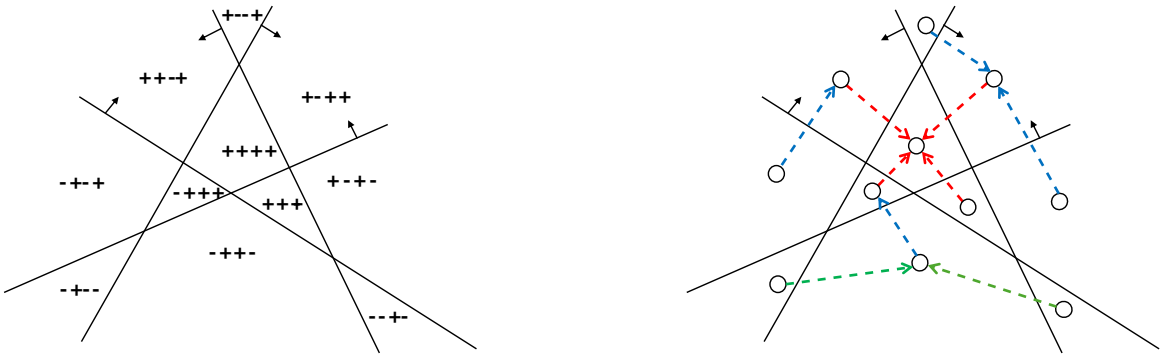


Figure II.3.4: A hyperplane arrangement (left panel) consists of four hyperplanes, with black arrows indicating their directions. The positive and negative labels represent the underlying sign vector of each cell. The right panel illustrates the generation process of the most well-known cell enumeration 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.

Indeed, this generation process can be described as the following sequential decision process

$$\begin{aligned} gen_{\text{rev}}(0) &= \text{adjcells}(c^*) \\ gen_{\text{rev}}(N) &= gen(N-1) \cup filter_q\left(map_{\text{adjcells}}\left(gen(N-1)\right)\right), \end{aligned} \tag{103}$$

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, which is defined as

$$\text{adjcells}(c) = \left[\text{flip}_n(c) \,|\forall n \in c^+, \text{if } p_{\text{cell}}(\text{flip}_n(c)) = \text{True}\right], \tag{104}$$

where p_{cell} is the cell feasibility test defined in (102), and $\text{flip}_n(c)$ function flip the n_{th} index of sign vector c. The $filter_q$ function here is symbolic, when we generate a list of parent cells $\text{adjcells}(c)$ from its child c, we need 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 as the algorithmic process in Algorithm 1, the 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. The time complexity of this algorithm is $O(N \times \text{LP}(N, D) \times C_D(\mathcal{H}))$ because in the worst case, we need to run $O(N)$ times linear program 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. II.3.4.

Incremental sign construction algorithm In our previous analysis for solving the 0-1 loss linear classification problem [Xi and Little, 2023], we developed a generic cell enumeration generator based on the *binary assignment generator* introduced in Section II.2.3. We refer to this as the *incremental sign construction algorithm*, due to its inherent nature of constructing sign vectors incrementally. Rada and Cerny [2018] independently discovered the depth-first search version of this algorithm.

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 $p(x \cup y) = p(x) \wedge p(y)$. 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

$$\begin{aligned} gen_{\text{cell}}(0) &= [[\,]] \\ gen_{\text{cell}}(N) &= filter_{p_{\text{cell}}}\left([1] \circ gen_{\text{cell}}(N-1) \cup [-1] \circ gen_{\text{cell}}(N-1)\right), \end{aligned} \tag{105}$$

Algorithm 1 Abstraction of the reverse search algorithm

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**: $\mathcal{S}_{\text{cell}}$

where $\circ$ is the cross-join operator. In the n_{th} recursive step of program (105), there are at most $\text{Cover}\,(n, D+1) = 2\sum_{d=0}^{D} \begin{pmatrix} n-1 \\ d \end{pmatrix}$ cells, which differs from the number of cells in the reverse search algorithm, as this generator also accounts for the dual cells. Thus program (105) 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 (105), 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 subdivided 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 $c \in \{1,-1\}^{n-1}$ belong to H_n^+ or H_n^- then there must exists 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 we determine the sign vector of c', 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 (102). Thus, only half of the sign vectors need to run the linear program (102), 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 (105) is

$$\text{Cover}\,(n, D+1) - \text{Cover}\,(n-1, D+1) = 2\sum_{d=0}^{D} \begin{pmatrix} n-1 \\ d \end{pmatrix} - 2\sum_{d=0}^{D} \begin{pmatrix} n-2 \\ d \end{pmatrix} = \text{Cover}\,(n-1, D)\,. \tag{106}$$

Thus the actual complexity of the algorithm will be

$$O\left(\sum_{n=0}^{N} (\text{LP}\,(n, D) \times \text{Cover}\,(n-1, D))\right) = O\left(\sum_{n=0}^{N} \left(\text{LP}\,(n, D) \times n^{D-1}\right)\right). \tag{107}$$

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 2 Obvious cell enumeration algorithm

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** $v \in \mathcal{S}_{\text{vert}}$ **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**: $\mathcal{S}_{\text{cell}}$

algorithm with worst-case complexity of $O\left(N^3\right)$ [Vaidya, 1989], leads to the cell enumerator's complexity (107), being upper-bounded by $O\left(N^{D+3}\right)$.

Compared with the reverse search algorithm (103), the incremental sign construction algorithm (105) solves more but smaller linear programming problems. In the previous analysis by Rada and Cerny [2018], the incremental sign construction algorithm (105) 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, hyperplane-based methods represent a hyperplane by identifying the data points that lie on it, which provides us with a new perspective to look at hyperplanes and eventually results a new class of algorithms that have a different combinatorial structure.

Vertex enumeration and 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]. It states that, given a set of data points $\mathcal{D}$ of size N in $\mathbb{R}^D$, the possible way to partition $\mathcal{D}$ by a linear hyperplane is $2^D \binom{N}{D}$. This is obtained from the intuition that every D points can determine a hyperplane, and for each hyperplane, there are 2^D distinguish ways assign different labels to D data points that lies on the hyperplane. As depicted in the left-panel of Fig. II.3.3, in $\mathbb{R}^D$, each yellow hyperplane can be determined by arbitrary two data item, and we can shift them infinitesimally to obtain 2^D hyperplanes without affecting the prediction to other data points.

In the space of hyperplane arrangement, this corresponds to the arbitrary D hyperplane has a intersection points (vertex). Each vertex in the dual space will adjacent to exactly 2^D *cells*, because we assume data points are in general position. By enumerating all possible vertices first and then enumerating the adjacent cells of each vertices, we can enumerate all possible cells in time $2^D \binom{N}{D}$. This strategy can be described as the algorithmic process 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 $c \in \{1, -1\}$. We can obtain the adjacent cells of v by simply replacing the 0 signs with arbitrary 1 or -1 signs.

However, as we can read from the Fig. II.3.3, there are many *overlapped cells* by using the obvious strategy described above. For instance, according to Lemma 4, When $N = 100$ and $D = 3$, the number of cells is

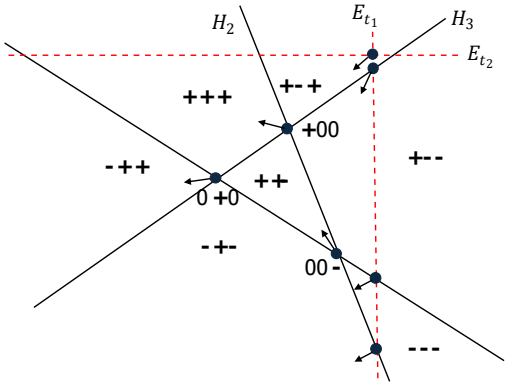


Figure II.3.5: Intersection of axis-parallel hyperplanes E_{t_d}, $1 \leq d \leq 2$ with an arrangement consists of three lines in $\mathbb{R}^2$

$\sum_{d=0}^{3} \binom{100}{d} = 166751$, whereas the combinatorial complexity of the obvious strategy is $2^3 \binom{100}{3} = 1293600$, the combinatorial complexity of this obvious strategy wasted almost $\left(2^D - 1\right) \binom{N}{D}$ computations for enumerating duplicates cells!

In the following, we will introduce a generic cell enumeration algorithm for enumerating the cells of a non-central arrangement $\mathcal{H}$, and each cell will be visited only once, thus the resulting algorithms will have a complexity of $\sum_{d=0}^{D} \binom{N}{d}$.

Efficient cell enumeration based on K-combination generator Consider a *non-central* and *simple* arrangement $\mathcal{H}$, consists of N hyperplane in $\mathbb{R}^D$. In this Section, we present a ingenious cell enumeration algorithm based on enumerating the vertices of an arrangements. The algorithms constructed in this section follows an important lemma 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 $\boldsymbol{e}_d$ the vector with all components equal to zero except for the d_{th} component which is equal to one. 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. II.3.5, two axis-parallel hyperplane E_{t_1} and E_{t_2} in $\mathbb{R}^2$ indicated by dashed lines, such that all vertices (black dots in Fig.) of $\mathcal{H}$ are *below* E_{t_d}, $\forall 1 \leq d \leq D$. We have following lemma, 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.

Lemma 13. Given a simple arrangement $\mathcal{H}$. Let the vertices 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 vertices set $\mathcal{V}_{\mathcal{H}}$ is one-to-one corresponds to 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, we did not prove it here, as it 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 be 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.

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 $O\left(D^3\right)$ time in the worst-case, is much more efficient than calculating a linear

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 lies 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**: $\mathcal{S}_{\text{cell}}$

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: Comparison between LP-based methods and H-based methods for cell enumeration in terms of worst/best case time/space complexity.

program with N inequalities in D variables. We summarize the comparison between these two classes of methods in Table 1 considered solely on their performance in 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. In 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 COPs, 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. In 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 often insufficient to construct a hyperplane. Moreover, these d-combinations must be stored in order to construct complete D-combinations. Therefore, CO algorithms based on H-based methods typically have better worst-case time complexity but worse best-case time complexity in CO tasks.

II.3.4 Euclidean Voronoi diagram and K-means problem

II.3.4.1 K-means problem and Euclidean Voronoi partition

Clustering is the grouping of similar objects and clustering of a set is a partition of elements that is chosen to minimize some measure of similarity, there are various kinds of measures of dissimilarity. The K-clustering problem fix a set of centroids $\mathcal{U} = \{\boldsymbol{\mu}_k : k \in \mathcal{K}\}$, where $\mathcal{K} = \{1, \ldots, K\}$ in $\mathbb{R}^D$. We can take all centroids in $\mathbb{R}^D$ together in a DK-tuple, denote as $\overrightarrow{\boldsymbol{\mu}} = (\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_K) \in \mathbb{R}^{DK}$. Each centroid $\boldsymbol{\mu}_k$ associated with a unique subset of data points, these data points are the closest data points to this centroid than other centroids (in terms of distance), this subset is called a *cluster*, denoted as C_k. Together, they forms a K-clusters set $\mathcal{C} = \{C_1, C_2, \ldots, C_K\}$. Thus $\mathcal{K}$ is 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 continuous variable $\overrightarrow{\boldsymbol{\mu}}$ as

$$\overrightarrow{\boldsymbol{\mu}}^* = \underset{\overrightarrow{\boldsymbol{\mu}} \in \mathbb{R}^{DK}}{\operatorname{argmin}} E_{\text{K-means}}\left(\overrightarrow{\boldsymbol{\mu}}\right) = \sum_{\boldsymbol{\mu}_k \in \mathcal{U}} \sum_{\boldsymbol{x}_n \in C_k} d_2\left(\boldsymbol{x}_n, \boldsymbol{\mu}_k\right)^2 \quad . \tag{108}$$

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 we fixed assignment s the optimal centroids $\mathcal{U}$ which solve the (108) are uniquely determined, which means that we can find the map between a set of class labels and a set of centroids analytically by

$$\boldsymbol{\mu}_k = \frac{1}{\mid C_k \mid} \sum_{\boldsymbol{x}_n \in C_k} \boldsymbol{x}_n, \quad k \in \mathcal{K}. \tag{109}$$

Given the correspondence between class labels and centroids, we can define a combinatorial parameter $s = (\alpha_1, \alpha_2, \ldots, \alpha_N) \in \mathcal{S}_{\text{kasgns}} = \mathcal{K}^N$ 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

$$s^* = \underset{s \in \mathcal{S}_{\text{kasgns}}}{\operatorname{argmin}} E_{\text{K-means}}(s) = \sum_{k \in \mathcal{K}} \sum_{n \in \mathcal{N}} \mathbf{1}\left[s_n = k\right] d_2\left(\boldsymbol{x}_n, \boldsymbol{\mu}_k\right)^2 \quad , \tag{110}$$

where function $\mathbf{1}[\,]$ returns 1 if the Boolean argument $s_n = k$ is true, and 0 if false.

Similarly, if we have 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 partition the space into K Voronoi regions. Given a data set $\mathcal{D}$ and a set of centroids set of centroids $\mathcal{U} = \{\mu_1, \mu_2, \ldots, \mu_K\}$. Every set of centroids $\mathcal{U} = \{\mu_1, \mu_2, \ldots, \mu_K\}$ have an associated Voronoi diagram $\mathcal{V} = \{V_1, V_2, \ldots, V_K\}$, each centroid $\boldsymbol{\mu}_k$ partition the ambient space $\mathbb{R}^D$ into regions defined by

$$V_k = \left\{\boldsymbol{x} \in \mathbb{R}^D \mid d_2\left(\boldsymbol{x} - \boldsymbol{\mu}_k\right)^2 \leq d_2\left(\boldsymbol{x} - \boldsymbol{\mu}_j\right)^2, \forall k, j \in \mathcal{K} \wedge k \neq j\right\}, \tag{111}$$

which naturally partition the data set $\mathcal{D}$ into K clusters $\mathcal{C} = \{C_1, C_2, \ldots, C_K\}$ defined by

$$C_k = \left\{\boldsymbol{x} \in \mathcal{D} \mid d_2\left(\boldsymbol{x} - \boldsymbol{\mu}_k\right)^2 \leq d_2\left(\boldsymbol{x} - \boldsymbol{\mu}_j\right)^2, \forall k, j \in \mathcal{K} \wedge k \neq j\right\}, \tag{112}$$

the data items lies in region V_k assigned to the cluster C_k. We will call the clusters set $\mathcal{C}$ the *Voronoi partition* to distinguish with the Voronoi diagram $\mathcal{V} = \{V_1, V_2, \ldots, V_K\}$, the K-class assignment s 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 for solving this issue 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 s 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 the *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 14. The optimum partition for the K-clustering problem must be a Voronoi partition.

Proof. This theorem is easy to prove by showing that any non-Voronoi partitions is non-optimal. Assume we have an assignment s that assign a data item $\boldsymbol{x}_n$ to cluster C_k, and $\boldsymbol{x}_n$ has distance relation $d_2\left(\boldsymbol{x}_n - \boldsymbol{\mu}_k\right)^2 > d_2\left(\boldsymbol{x}_n - \boldsymbol{\mu}_j\right)^2$. If a new assignment s' assign x_n to cluster C_j and keep other data items fixed, the squared error will decrease because $d_2\left(\boldsymbol{x}_n - \boldsymbol{\mu}_k\right)^2 > d_2\left(\boldsymbol{x}_n - \boldsymbol{\mu}_j\right)^2$. Thus assignment s' is better than assignment s, and s is non-optimal. □

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 fact comes from the fact that the Voronoi diagram can be represented by a hyperplane arrangement consisting of $NK\left(K-1\right)/2$ 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 (112), instead of the actual value of $d_2\left(\boldsymbol{x} - \boldsymbol{\mu}_k\right)^2$. 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

We can reformulate the expression $d_2\left(\boldsymbol{x}_n - \boldsymbol{\mu}_k\right)^2 \leq d_2\left(\boldsymbol{x}_n - \boldsymbol{\mu}_j\right)^2$ to the equivalent form

$$\boldsymbol{\mu}_k^2 - \boldsymbol{\mu}_j^2 - 2\boldsymbol{x}_n^T\left(\boldsymbol{\mu}_k - \boldsymbol{\mu}_j\right) \leq 0, \tag{113}$$

if the data items $x_n$fixed, this formula becomes a polynomial with variable $\boldsymbol{\mu}_k$, $\boldsymbol{\mu}_j$.

WE can define $d_2\left(\boldsymbol{x}_n - \boldsymbol{\mu}_k\right)^2 - d_2\left(\boldsymbol{x}_n - \boldsymbol{\mu}_j\right)^2$ as a polynomial parameterized by $\boldsymbol{x}_n$and with variable $\boldsymbol{\mu}_k$, $\boldsymbol{\mu}_j$

$$P_{\boldsymbol{x}_n,k,j} = \sum_{d=1}^{D}\mu_{kd}^2 - \sum_{d=1}^{D}\mu_{jd}^2 - 2\sum_{d=1}^{D}x_{nd}\left(\mu_{kd} - \mu_{jd}\right), \tag{114}$$

where $P_{\boldsymbol{x}_n,k,j}$ is a polynomial in $\mathbb{R}\left[\overrightarrow{\boldsymbol{\mu}}\right]$, and $\overrightarrow{\boldsymbol{\mu}} = \left(\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_K\right) \in \mathbb{R}^{DK}$ is the K-tuples of K centroids in $\mathbb{R}^D$.

For each pair of centroids and a data items we can define a polynomial (114). Therefore, the total number of polynomials equations for a size N data set with K centroids in $\mathbb{R}^D$ is $M = N\begin{pmatrix} K \\ 2 \end{pmatrix} = NK\left(K-1\right)/2$. These polynomials form a 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 $S_{\boldsymbol{x}_n,k,j} = \left\{\boldsymbol{x}_n \in \mathbb{R}^D : \sum_{d=1}^{D}\mu_{kd}^2 - \sum_{d=1}^{D}\mu_{jd}^2 - 2\sum_{d=1}^{D}x_{nd}\left(\mu_{kd} - \mu_{jd}\right)\right\}$. We call $\mathcal{S}_{\mathcal{P}}$ as the *Voronoi arrangement* defined by data set $\mathcal{D}$.

Lemma 15. Assume there are no data lies 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

$$C_k = \left\{\boldsymbol{x}_n \in \mathcal{D} : P_{\boldsymbol{x}_n,k,j} < 0 \wedge P_{\boldsymbol{x}_n,j,k} > 0\right\}, \forall k, j \in \mathcal{K} \wedge j \neq k. \tag{115}$$

In fact, the actual value of $P_{\boldsymbol{x}_n,k,j}$ is not interested to us, since we only want to know if $\boldsymbol{x}_n$ is closer to $\boldsymbol{\mu}_k$ or $\boldsymbol{\mu}_j$.In fact, this information can be obtained by knowing the sign vector of $\overrightarrow{\boldsymbol{\mu}} \in \mathbb{R}^{DK}$ to the polynomial system $\mathcal{P} = \{P_{\boldsymbol{x}_n,k,j} \mid \forall n \in \mathcal{N}, \forall k, j \in \mathcal{K} \wedge j \neq k\}$.

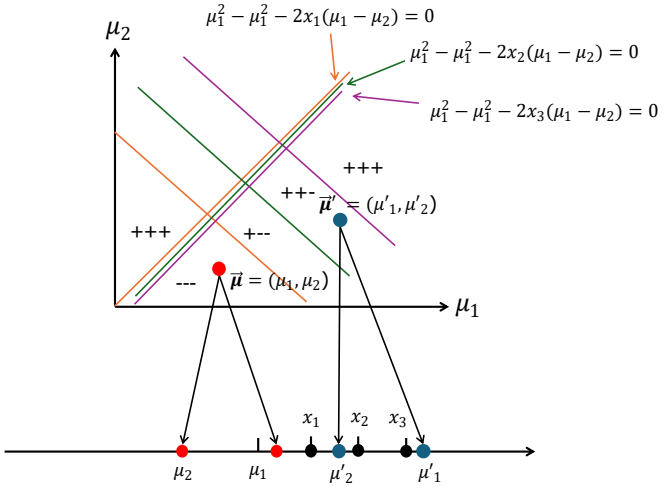


Figure II.3.6: The hypersurface arrangement introduced by the Voronoi arrangement of three data points x_1, x_2 and x_3 in $\mathbb{R}$. The centroids vector $\overrightarrow{\boldsymbol{\mu}}$ hat lies within a given cell produces the same partition of data points. For example, two centroid vectors (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.

Thus we can reformulate (115) using the sign vector of the polynomial system $\mathcal{P}$ as stated in the following lemma.

Lemma 16. 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 there are no data lies on the boundary of each partition, a cluster set $\{C_1, C_2, \ldots, C_K\}$ is a Voronoi partition if and only if

$$C_k = \left\{\boldsymbol{x}_n \in \mathcal{D} : \delta_{\boldsymbol{x}_n,k,j}\left(\overrightarrow{\boldsymbol{\mu}}\right) = -1 \wedge \delta_{\boldsymbol{x}_n,j,k}\left(\overrightarrow{\boldsymbol{\mu}}\right) = 1\right\}, \forall k \in \mathcal{K}. \tag{116}$$

Geometrically, $\delta_{\boldsymbol{x}_n,k,j}\left(\overrightarrow{\boldsymbol{\mu}}_{k,j}\right) = -1$ means $d_2\left(\boldsymbol{x}_n - \boldsymbol{\mu}_k\right)^2 \leq d_2\left(\boldsymbol{x}_n - \boldsymbol{\mu}_j\right)^2$, i.e., $\boldsymbol{x}_n$ is closer to $\boldsymbol{\mu}_k$ than $\boldsymbol{\mu}_j$.

Conversely given a centroids vector $\overrightarrow{\boldsymbol{\mu}}$, the Voronoi partition $\{C_1, C_2, \ldots, C_K\}$ associated to $\overrightarrow{\boldsymbol{\mu}}$ is defined as

$$C_k = \left\{\boldsymbol{x}_n \in \mathcal{D} : \delta_{\boldsymbol{x}_n,k,j}\left(\overrightarrow{\boldsymbol{\mu}}\right) = -1 \wedge \delta_{\boldsymbol{x}_n,j,k}\left(\overrightarrow{\boldsymbol{\mu}}\right) = 1\right\}, \forall k \in \mathcal{K}. \tag{117}$$

The K-clustering problems are 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 8, 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 we have three points $x_1 = 1$, $x_2 = 2$, and $x_3 = 3$ in $\mathbb{R}$, and we have $K = 2$ clusters, the arrangement $\mathcal{S}_\mathcal{P}$ for polynomial system defined by (114) are three 2D conic sections that shown in Fig. II.3.6. 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, thus the centroids vector $\overrightarrow{\boldsymbol{\mu}}$ lies in $\mathbb{R}^2$. In Fig. II.3.6, we can see that the space of $\mathbb{R}^2$ is partitioned into eight disjoint regions. For arbitrary points in region with $(-,-,-)$ sign (the red point in the Fig.), they correspond to centroids with 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 is 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 [Basu et al., 1995, Caviness and Johnson, 2012]. However, enumerating the cells of a hypersurface arrangement is much

more harder than enumerating the cells of a hyperplane arrangement. Fortunately, because of the special geometry of the K-clustering problem, we can enumerate the cells of the arrangement $\mathcal{S}_{\mathcal{P}}$ in a much simpler way.

In the next section, we will show that the hypersurface arrangement $\mathcal{S}_{\mathcal{P}}$ defined by quadratic polynomials (114) can be transformed into a hyperplane arrangement.

II.3.4.4 Variable replacement and optimal K-means clustering

Our discussion here is similar to the discussion of [Tîrnăucă et al., 2018], it shows that the system of polynomials in (114) can be solved by a system of linear equations by applying a change of variables.

Consider polynomials defined in (114) with KD variables, an important observation is that $P_{\boldsymbol{x}_n,k,j} = P_{\boldsymbol{x}_n,1,j} - P_{\boldsymbol{x}_n,1,k}$, $\forall 2 \leq k < j \leq K$. If we replace the variable in (114) through 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

$$\begin{aligned} P'_{\boldsymbol{x}_n,1,k} &= Y_1 - Y_k - 2\sum_{d=1}^{D} x_{nd} Z_{(k-1)d},\ \forall k \in \{2,\ldots,K\} \\ P'_{\boldsymbol{x}_n,k,j} &= P'_{\boldsymbol{x}_n,1,j} - P'_{\boldsymbol{x}_n,1,k},\ \forall 2 \leq k < j \leq K, \end{aligned} \tag{118}$$

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 from $\overrightarrow{\boldsymbol{\mu}}$ through above variable replacement. We denote the hyperplane arrangement defined by 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 $K + D(K-1)$ 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 we enumerate the cells of the arrangement $\mathcal{H}_{\mathcal{P}'}$. Therefore, although there are $NK(K-1)/2$ polynomial equations in system $\mathcal{P}'$, the number of linearly independent equations are $(K-1)N$. In contrast, the old system has $NK(K-1)/2$ independent equations.

Example 8. Given a dataset $\mathcal{D}$ of size N and $K = 3$. The coefficients for linear system (118) can be represented by following coefficient.

$$\begin{bmatrix} 1 & -1 & & -2x_{11} & -2x_{12} & & & & \\ \vdots & \vdots & & \vdots & \vdots & & & & \\ 1 & -1 & & -2x_{N1} & -2x_{N2} & & & & \\ 1 & & -1 & & & -2x_{11} & -2x_{12} & & \\ \vdots & & \vdots & & & \vdots & \vdots & & \\ 1 & & -1 & & & -2x_{N1} & -2x_{N2} & & \\ 1 & & & -1 & & & & -2x_{12} & -2x_{12} \\ \vdots & & & \vdots & & & & \vdots & \vdots \\ 1 & & & -1 & & & & -2x_{N2} & -2x_{N2} \end{bmatrix}, \forall n \in \mathcal{N}. \tag{119}$$

By applying the variable replacement above, it is easy to show the following lemma is true.

Lemma 17. Let $\mathcal{P}_{\mathcal{I}} = \{P_i \mid i \in \mathcal{I}\}$ be a polynomial sub-system of $\mathcal{P} = \{P_{\boldsymbol{x}_n,k,j} \mid \forall n \in \mathcal{N}, \forall k,j \in \mathcal{K} \wedge j \neq k\}$, where $P_{\boldsymbol{x}_n,k,j}$ is defined by (114) 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*

$$\operatorname{sign}_{\mathcal{S}_{\mathcal{P}_{\mathcal{I}}}}\left(\overrightarrow{\boldsymbol{\mu}}\right) = \alpha_i, \forall i \in \mathcal{I}, \tag{120}$$

has a solution, then

$$\operatorname{sign}_{\mathcal{H}_{\mathcal{P}_{\mathcal{I}}'}}\left(\overrightarrow{\boldsymbol{\mu}}'\right) = \alpha_i, \forall i \in \mathcal{I}, \tag{121}$$

also has.

A consequence of Lemma 17 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 $\mathcal{P}' = \left\{ P'_{\boldsymbol{x}_n,j,k} : \forall n \in \mathcal{N}, k \neq j \right\}$.

Follows from the result of [Tîrnăucă et al., 2018], we have following theorem for enumerating cells in arrangement $\mathcal{H}_{\mathcal{P}'}$.

Theorem 12. In order to enumerate all possible sign vectors (cells) of arrangement $\mathcal{P}'$, can be enumerated 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 $|\mathcal{J}| \leq K + (K-1)\, D$.

II.3.4.5 Duality and 2-means problem

Indeed, Thm. 12 is similar to the obvious hyperplane-based method (114) 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 in all Bregman divergences [Banerjee et al., 2005]. In this section, we 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

$$\begin{bmatrix} 1 & -1 & -2x_{11} & -2x_{12} & & -2x_{1D} \\ 1 & -1 & -2x_{21} & -2x_{22} & & -2x_{2D} \\ & & & \vdots & \cdots & \vdots \\ 1 & -1 & -2x_{n1} & -2x_{n2} & & -2x_{nD} \\ & & & \vdots & & \vdots \\ 1 & -1 & -2x_{N1} & -2x_{N2} & & -2x_{ND} \end{bmatrix}. \tag{122}$$

At first glance, the affine flats represented by the matrix (122) form an arrangement in $\mathbb{R}^{D+2}$. However, the linear system (hyperplane arrangement) in (122) has a very special form, it is 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 (122), we obtain a set of data points $\left\{\phi^{-1}\left(\boldsymbol{w}_n\right) = (1, -1, -2\boldsymbol{x}_1, \ldots, -2\boldsymbol{x}_D) \in \mathbb{R}^{D+2} \mid \forall n \in \mathcal{N}\right\}$. We define the data points of this form as following.

Definition 31. *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

$$\bar{\bar{\boldsymbol{x}}}_n = (c_1, c_2, x_{n1}, x_{n2}, \ldots, x_{nD}) \in \mathbb{R}^{D+2}, \tag{123}$$

where c_1, c_2 are some constant.

In the case of data set $\left\{\phi^{-1}\left(\boldsymbol{w}_n\right) = (1, -1, -2\boldsymbol{x}_n) \in \mathbb{R}^{D+2} \mid \forall n \in \mathcal{N}\right\}$. We can imagine a set of dataset $\{-2\boldsymbol{x}_n : \forall n \in \mathcal{N}\}$ been moved to the homogeneous coordinate twice. Geometrically, this means that we have two hyperplanes $Y_1 = 1$ and $Y_2 = -1$ (subspace with dimension $D + 2 - 1$, remember that the dimension for the linear system (122) is $D + 2$), their intersection forms a $D + 2 - 2 = D$ dimensional subspace which is equivalent to the dimension of our 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 (122). 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

In this chapter, we present a comprehensive analysis of the geometric and combinatorial properties of Voronoi diagrams and hyperplanes. One of the main contributions includes the development of novel algorithms for enumerating the cells of hyperplane arrangements, along with a reformulation of the reverse search algorithm, which is typically expressed in a depth-first way in literature.

These algorithms hold potential interest for both combinatorial geometers and optimization researchers. For geometers, the construction of efficient cell enumeration algorithms is one of the most fundamental problems in combinatorial geometry studies, and existing algorithms are often difficult to parallelize.

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.2.7, often requires extensive communication between processors due to backtracking. In contrast, our cell enumeration algorithms, including the reformulated reverse search algorithm, are much easier to parallelize and can be efficiently implemented on both CPUs and GPUs.

This is also of interest to optimization researchers because many intractable COPs [Ferrez et al., 2005, Xi and Little, 2023, Bertsimas et al., 2020] can be solved in polynomial time by applying these cell enumeration algorithms.

Additionally, we explore the geometric foundations of many essential machine learning models, such as K-means clustering and linear (or polynomial) classification models. These insights almost already give us a polynomial time algorithm for solving these intractable machine learning problems, given the generator that we have introduced in Section II.1.2 and Section II.2.3. In Part III, we 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 explains the **Specialized Theory**, which examines the combinatorial essence 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, and hypersurface decision boundaries.

Although all the problems examined in this part have been proven to be NP-hard, we will show 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 essence, we demonstrate 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: a *review of related studies*, a *problem definition*, the *combinatorial essence 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 a *further discussion*. The further discussion addresses acceleration techniques tailored to each problem, which can lead to algorithms with performance that is provably better than their worst-case complexity, potentially achieving near-linear time efficiency in the best case. Additionally, the advantages, drawbacks, and possible extensions to related problems are examined, providing a comprehensive understanding of each problem's challenges and opportunities.

By analyzing the combinatorial essence of these commonly used machine learning problems, we will demonstrate the great potential of using our framework to design efficient and exact algorithms tailored to the most pressing challenges in machine learning.

III.1 Terminology

In this chapter, we provide a brief summary of the Terminology that will be used in the discussion. Some of these terms may have been explained in previous sections, and we re-summarize them here to assist the audience in recapping.

Given a data set $\mathcal{D}$ consists of N *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.* We can use a matrix to store our data set, by putting the transpose of each column vector $\boldsymbol{x}_n$ in the row of the matrix, we obtain a $N \times D$ matrix $\boldsymbol{X}$. In the K-clustering problem, we do not know the label for each data point $\boldsymbol{x}_n$, since the K-clustering problem is an unsupervised learning problem. In the classification problem, each data point has a unique true label $t_n \in \{-1, 1\}$, $\forall n \in \mathcal{N}$. All true labels in this data set are stored in a vector $\boldsymbol{t} = \{t_1, t_2, ..., t_N\}^T$ in $\{-1, 1\}^N$ and the data set is represented by $\mathcal{D}_{\boldsymbol{t}}$.

In machine learning, the central task is to "learn" from the data, and refine some useful information (or parameters). Then we can make efficient "queries" from our model, this is known as *inference.* In supervised learning problems, each data item consists of features and a unique label, when we learn a model from our algorithm, the *predicted labels* of this model is called an *assignment.* Similarly for the clustering learning problem, although there are no labels for each data item, we implicitly assign a " cluster label" to each data item, and this cluster label is called an *assignment* either.

Recall that, in linear classification problem, a linear model predicts all data points on the positive side of the decision boundary with a label of 1, and assigns a label of -1 to points on the negative side (with an implicit label of 0 for points lying exactly on the decision boundary). We can store the *prediction labels for all data items* in a sign vector (assignment) $\boldsymbol{y} = (y_1, y_2, \ldots, y_N) \in \{1, -1\}^N$.

Similarly, for the K-clustering problem, the continuous variables that we need to specify are called *centroids.* We denote the set of centroids by $\mathcal{U} = \{\boldsymbol{\mu}_{\mathcal{K}}\}$, $k \in \{1, \ldots, K\} = \mathcal{K}$, the centroids $\boldsymbol{\mu}_k$, $\forall k \in \mathcal{K}$ lies in the feature space $\mathbb{R}^D$. Thus, the space of all possible centroids is isomorphic to $\mathbb{R}^{D \times K}$. Each centroid $\boldsymbol{\mu}_k$ is associated with a unique subset of data points. These data points are the closest data points to this centroid than other centroids (in terms of distance), this subset is called *cluster* C_k, $\mathcal{K}$ is then called the *cluster labels.* Therefore, we can assign a unique label $\alpha_n \in \mathcal{K}$ for each data point $\boldsymbol{x}_n$ depending on which cluster they lie. Together, they forms a N-tuple $s = (\alpha_1, \alpha_2, \ldots, \alpha_N) \in \mathcal{K}^N$.

III.2 Classification problem

Algorithms for solving the linear classification problem have a long history, dating back at least to 1936 with linear discriminant analysis. The original objective of the linear classification problem is to find a hyperplane decision boundary that minimizes the number of misclassified data points. In other words, we aim to solve the linear classification problem with a *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 or logistic loss) or approximate combinatorial search, none of which can be guaranteed to solve the problem exactly. Finding 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.

In this chapter, we provide a detailed analysis of the combinatorial essence 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, we will discuss the generalization of our methods to address 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.2.1 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, yet 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. 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 hinge loss, lacks robustness in handling 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 a 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 involving optimizing the *integer coefficient linear classifier* 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 we optimize convex surrogates. Therefore, Ustun and Rudin [2019] presented a cutting-plane algorithm to learn an optimal risk score, or solve 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 (126). Nguyen and Sanner [2013] also constructed a polynomial-time combinatorial search algorithm which is similar to our algorithm, but gave no proof of correctness. Hence, previous work on this problem of solving (126) is either computationally intractable, i.e. worse case exponential run-time complexity, or uses inefficient, off-the-shelf MIP solvers, or is not provably correct [Nguyen and Sanner, 2013].

Previously, we designed and implemented in Python, three other novel, algorithms (E01-ICG, E01-ICG-purge and E01-CE) for 0-1 loss linear classification problem [Xi and Little, 2023], and another algorithm called "E01-ICE" [He and Little, 2023].

Although our previous discussions did not include correctness proofs and formal algorithm derivations for the E01-ICG and the E01-ICG-purge algorithm, we conducted small-scale experiments to compare the wall-clock runtime 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.2.2 Problem specification

The 0-1 loss linear classification problem has a long history in machine learning. The definition of this problem is very simple, given data set $\mathcal{D}$ and its associated binary label vector $\boldsymbol{t} = (t_1, \ldots, t_N)^T$, we need to find a linear hyperplane that minimizes the number of misclassification data points. In other words, we need to solve an optimization problem that minimizes the following objective

$$E_{0\text{-}1}(\boldsymbol{w}) = \sum_{n \in \mathcal{N}} \mathbf{1}\left[\operatorname{sign}\left(\boldsymbol{w}^T \bar{\boldsymbol{x}}_n\right) \neq t_n\right], \tag{124}$$

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 is 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,

$$E_{\text{test}} \leq E_{0\text{-}1}(\boldsymbol{w}) + O\left(\sqrt{\frac{\log\left(N/\left(D+1\right)\right)}{N/\left(D+1\right)}}\right), \tag{125}$$

where E_{test}, $E_{0\text{-}1}(\boldsymbol{w})$ are the *test 0-1 loss* and the *empirical 0-1 loss* of on training data set, respectively [Mohri et al., 2018]. Equation (125) motivates finding the exact 0-1 loss on the training data, since, among all possible linear hyperplane classifiers, with high probability, none has a better worst-case test 0-1 loss than the exact classifier.

Therefore, the 0-1 loss linear classification problem is to find an optimal $\boldsymbol{w}^*$ that minimizes equation (124), which is defined as

$$\boldsymbol{w}^* = \underset{\boldsymbol{w} \in \mathbb{R}^{D+1}}{\operatorname{argmin}} E_{0\text{-}1}(\boldsymbol{w}). \tag{126}$$

Although apparently simple, this is a surprisingly challenging optimization problem. Considered a continuous optimization problem, the standard ML optimization technique, gradient descent, is not applicable (since the gradients of $E_{0\text{-}1}(\boldsymbol{w})$ 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 are only guaranteed to find one of these local 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. However, this can only find a sub-optimal decision function corresponding to an upper bound on the globally optimal value of the objective $E_{0\text{-}1}(\boldsymbol{w})$.

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*, $s = \left(\operatorname{sign}\left(\boldsymbol{w}^T \boldsymbol{x}_1\right), \operatorname{sign}\left(\boldsymbol{w}^T \boldsymbol{x}_2\right), \ldots, \operatorname{sign}\left(\boldsymbol{w}^T \boldsymbol{x}_N\right)\right) \in \mathcal{S}_{\text{basgn}}$, 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 $E_{0\text{-}1}(\boldsymbol{w})$, and we need to find the assignment that can obtain the optimal 0-1 loss $\hat{E}_{0\text{-}1}$. 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 we can not 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 known as (*linear*) *dichotomies*, and while there are 2^N possible assignments, there are only $O\left(N^D\right)$ dichotomies for data set in D dimensions [Cover, 1965]. Thus, the

problem (126) can instead be treated as a combinatorial optimization problem of finding the best such dichotomies and their implied assignments, from which an optimal parameter $\boldsymbol{w}^*$ can be obtained.

III.2.3 The combinatorial essence of the linear classification problem

In Section II.3.3, we 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 essence of the two hyperplane representations mentioned above.

III.2.3.1 Hyperplane-based (H-based) algorithm

Through our exposition in Section II.3.2, we 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. We 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 10, it follows immediately that we can solve the 0-1 loss linear classification problem in $O\left(N^{D+1}\right)$ given $O\left(N\right)$ time for evaluating the 0-1 loss objective.

Nonetheless, 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.

However, 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. In this section, we will demonstrate that this is indeed the case for the linear classification problem with a 0-1 loss objective. We 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. We can 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 13. *C*onsider a dataset $\mathcal{D}$ of N data points of dimension D in general position, along with their associated labels. All globally optimal solutions to problem (126), 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, we transform a dataset $\mathcal{D}$ to its dual arrangement. According to Lemma 7 and Lemma 8, each dichotomy has a corresponding dual cell and if we evaluate the sign vectors for all possible cells in the dual arrangement and their reverse signs, we can obtain the optimal solution for the 0-1 loss classification problem. Assume the optimal cell is f, we need 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 10, 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 we can evaluate all vertices (and the reverse sign vector for these vertices) and choose the best one, which, according to Lemma 8, 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 that we 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 lies on it.

Moreover, in the next section, we will explain 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.2.3.2 Linear programming-based (LP-based) algorithm

The linear programming-based method characterizes a hyperplane as a binary assignment (prediction labels for data items in $\mathcal{D}$), this 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, where we introduced two cell enumeration algorithms: the *reverse search algorithm* (103) and the *incremental sign construction algorithm* (105). 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 (105) is more suitable than the reverse search algorithm (103). We will explain why this is the case in the discussion of this section.

Consider an 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 our previous research [Xi and Little, 2023], we have introduced the E01-ICG (short for, exact 0-1 incremental combinatorial generation) algorithm, which is the dual version of the (105). 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

$$\begin{aligned}&\min_{\boldsymbol{w}}\ \mathbf{0}^T\boldsymbol{w}\\&\text{s.t. }\operatorname{diag}\left(\boldsymbol{y}\right)^T\overline{\boldsymbol{X}}\boldsymbol{w}\geq\mathbf{1}.\end{aligned}\tag{127}$$

Dually, the linear program (127) is essentially the same as the cell feasibility test (102).

Since the E01-ICG algorithm is essentially the same as the recursion (105), except for the additional recursive update process for the 0-1 loss. Let's consider how the 0-1 loss should be updated directly in the recursion (105). In each recursion of (105), 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 increase monotonically 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 increasing* manner. It follows immediately that the dominance relations that we introduced in Subsection II.2.6.3, such as global upper bound technique can be easily incorporated into the recursion (105), allowing any partial assignments with a 0-1 loss worse than the global upper bound to be discarded without further extension.

In 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 increasing with respect to the objective, as long as the true label vector $\boldsymbol{t}$ is not all positive.

The monotonic increasing 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 a complexity (107) 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, we need $O\left(N\times\operatorname{Cover}\left(N,D\right)\right)$ operations to evaluate the 0-1 loss for each dichotomy. Thus, the E01-ICG algorithm has a complexity of

$$O\left(\sum_{n=0}^{N}\left(\operatorname{LP}\left(n,D\right)\times\operatorname{Cover}\left(n-1,D\right)\right)+N\times\operatorname{Cover}\left(N,D+1\right)\right)=O\left(\operatorname{LP}\left(N,D\right)\times N^D+N^{D+1}\right).\tag{128}$$

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 $O\left(\sum_{n=0}^{N}\operatorname{LP}\left(n,D\right)\right)$ 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 (128) is upper-bounded by $O\left(N^{D+3}\right)$.

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.

III.2.4 Further discussions

III.2.4.1 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. In this Subsection, we will explore their differences in terms of the acceleration techniques applicable to each, as well as their respective advantages and limitations.

Acceleration in H-based algorithm 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 14. *Symmetry fusion theorem.* Consider a dataset $\mathcal{D}$ of N data points of dimension D in general position, along with their associated labels. Given hyperplane h 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 are misclassified in $\mathcal{D}^+$ and $\mathcal{D}^-$, thus 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^- is $N - D - l$. □

Acceleration in LP-based algorithm The two dominance relations, *finite dominance relation* (68) and *global upper bound* (66), are readily applicable in this context. As demonstrated in Subsection II.2.6.3, utilizing 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 has been empirically shown to be extremely powerful, as it can yield nearly linear time complexity in the best-case.

Comparison between H-based algorithm and LP-based algorithm 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.*

Firstly, H-based algorithms often outperform LP-based methods in low-dimensional, small-scale problems for two main reasons: First, H-based algorithms can obtain the exact solution by exploring all possible $\binom{N}{D}$ 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 $\mathcal{D}$; Second, H-based algorithms generate hyperplanes through matrix inversion, which has a worst-case time complexity of $O\left(D^3\right)$. In 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.

Secondly, the incremental hyperplane generation process is crucial for applications that require a balance between accuracy and computational time. In 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 $O\left(N\right)$ space for each configuration, whereas representing a hyperplane by the data points that lie on it requires only $O\left(D\right)$ space.

Despite the disadvantages of LP-based algorithms, they possess unique advantages that cannot be easily replaced by H-based algorithm 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 large number of candidate solutions before extending them to completion. In 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.2.4.2 Non-linear (polynomial hypersurface) classification

Following our discussion about *Veronese embedding* in Subsection II.3.2.3, it is straightforward to generalize our algorithms 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.

Similar to Thm. 13, the special combinatorial property of the 0-1 loss allows us to solve the problem more efficiently, which is described in the following theorem.

Theorem 15. *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 (126) with objective

$$E_{0\text{-}1}\left(\boldsymbol{w}\right) = \sum_{n \in \mathcal{N}} \mathbf{1}\left[\text{sign}\left(\boldsymbol{w}^T \tilde{\boldsymbol{x}}_n\right) \neq t_n\right], \tag{129}$$

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.2.4.3 Margin loss linear classifier

The Linear Classification Theorem 10 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, we will briefly explain how the same algorithmic process of exhaustively generating cells can be applied to the margin loss linear classification problem, thereby allowing us to incorporate a hyperparameter to mitigate overfitting.

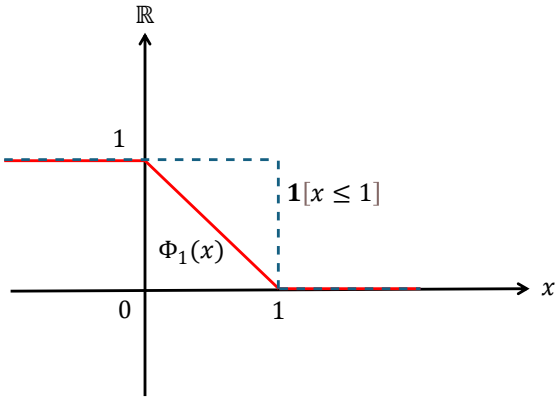


Figure III.2.1: The margin loss $\Phi_\rho(x)$ (red), defined with respect to margin parameter $\rho = 1$ is upper bounded by the shifted 0-1 loss $\mathbf{1}[x \leq \rho]$ (blue dotted line), i.e., $\Phi_\rho(x) \leq \mathbf{1}[x \leq \rho]$, where $x = y_n h(n)$ in the classification problem.

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.

Definition 32. For any $\rho > 0$, the the ρ-margin loss is the function $\Phi : \mathbb{R} \times \mathbb{R} \to \mathbb{R}^+$ defined by

$$\Phi_\rho(x) = \min\left(1, \max\left(0, 1 - \frac{x}{\rho}\right)\right) = \begin{cases} 1 & \text{if } x < 0 \\ 1 - \frac{x}{\rho} & \text{if } 0 \leq x \leq \rho \\ 0 & \text{if } \rho < x. \end{cases} \tag{130}$$

As shown in Fig. III.2.1, the ρ-margin loss is upper bound by *shifted 0-1 loss*, i.e.,

$$\Phi_\rho(y_n h(n)) \leq \mathbf{1}[y_n h(n) \leq \rho], \forall n \in \mathcal{N}, \tag{131}$$

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 margin*s. 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

$$E_{\text{margin}}(\boldsymbol{w}) = \sum_{n \in \mathcal{N}} \Phi_\rho\left(y_n\left(\boldsymbol{w}^T \bar{\boldsymbol{x}}_n\right)\right). \tag{132}$$

From a result in learning theory [Mohri et al., 2018], given a data set $\mathcal{D}$, 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 binary linear classification task with margin loss objective is upper bounded by

$$E_{\text{test}} \leq E_{\text{margin}}(\boldsymbol{w}) + 2\sqrt{\frac{r^2 \Lambda^2 / \rho^2}{N}} + \sqrt{\frac{\log \frac{2}{\delta}}{2N}}. \tag{133}$$

Compared to the VC-dimension bound (125), which is defined in terms of 0-1 loss, the generalization bound based on the ρ-margin loss offers two significant advantages. First, the generalization bound (133) 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 h a higher confidence margin. Second, the generalization bound (133) 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

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}\in\mathcal{H}}{\operatorname{argmin}}\ E_{\text{margin}}\left(\boldsymbol{w}\right) \tag{134}$$

Since the margin loss is discrete, the margin loss linear classification problem (134) cannot be directly optimized using convex optimization methods.

From the definition of the margin loss objective (132), for data points that are correctly classified, we 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: First, 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 $1 - \frac{y_i\left(\boldsymbol{w}^T\bar{\boldsymbol{x}}\right)}{\rho}$; Second, 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) $\boldsymbol{y} = (y_1, y_2, \ldots y_N)$, to optimize the margin loss, we need to solve the following problem

$$\begin{aligned}
&\min_{w} \sum_{i\in I^+} \frac{\xi_i}{\rho} + \left|I^-\right| \\
&\text{s.t. } y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i + b_i\right) \geq \rho - \xi_i, \forall i \in I^+ \\
&\quad\ y_n\left(\boldsymbol{w}^T\boldsymbol{x}_n + b_n\right) \geq 0, , \forall n \in \mathcal{N} \\
&\quad\ 0 \leq \xi_i \leq \rho,
\end{aligned} \tag{135}$$

where $y_n\left(\boldsymbol{w}^T\boldsymbol{x}_n + b_n\right) \geq 0$ is the linear feasibility constrains to make sure $\boldsymbol{y}$ is a dichotomy, and ξ_i, $i \in I^+$ are the slack variables, when $\xi_i = 0$, it represents $\boldsymbol{x}_i$ 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, we have $\xi_i = \rho$ for data points lying on the decision boundary.

Problem (135) is a linear programming problem that can be optimally solved using standard linear programming solvers. However, this linear program (135) is based on *functional margin*, thus we can always scale the size of $\boldsymbol{w}_i$ and b_i to make constraints $y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i + b_i\right) \geq \rho - \xi_i$ feasible. Therefore, we need to modify (135) to optimize the *geometric margin*

$$\begin{aligned}
&\min_{w,b,\xi_i} \frac{1}{2}\left\|w\right\|^2 + \sum_{i\in I^+} \xi_i \\
&s.t.\ y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i + b_i\right) \geq \rho - \xi_i, \forall i \in I^+. \\
&\quad\ y_n\left(\boldsymbol{w}^T\boldsymbol{x}_n + b_n\right) \geq 0, , \forall n \in \mathcal{N} \\
&\quad\ 0 \leq \xi_i \leq \rho.
\end{aligned} \tag{136}$$

Therefore, the margin loss linear classification problem can 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 10, we can solve the margin loss linear classification problem in $O\left(qp\left(N, D\right)N^D\right)$ time.

III.3 Empirical risk minimization for ReLU network

In recent years, neural networks have emerged as a classical supervised learning technique, developed from the perceptron learning algorithm for classification problems. This model has revolutionized nearly every scientific field involving data analysis and has become one of the most widely used machine learning techniques today.

In this section, we address 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). Optimizing even a 2-layer network with ReLU activation is known to be NP-hard. We analyze the combinatorial essence of this problem and present a practical approach for solving it in polynomial time, assuming that the dimension and the number of hidden nodes are fixed. Results from complexity theory demonstrate that our approach achieves optimal efficiency in terms of worst-case complexity.

III.3.1 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 our LP-based algorithm for the linear classification problem. 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, in this chapter, we propose 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. Our approach can be easily generalized to the multilayer case by training it greedily, with the worst-case complexity remaining the same. Unlike prior methods, our algorithm does not require all possible hyperplanes to be pre-stored. Instead, the feasible ReLU networks are generated recursively by the generator that we design.

III.3.2 Problem specification

We extend the ReLU activation function to vectors $\boldsymbol{x} \in \mathbb{R}^D$ through entry-wise operation:
$\sigma\left(\boldsymbol{x}\right) = \left(\max\left(0, x_1\right), \max\left(0, x_2\right), \ldots, \max\left(0, x_D\right)\right)$.

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 $\boldsymbol{W}_1^T = \left(\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_K\right)$. 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 row of $\boldsymbol{W}_1$. Similarly, we can define a linear transformation $f_{\boldsymbol{W}_2} : \mathbb{R}^K \to \mathbb{R}$, such that $\boldsymbol{W}_2 = \left(\alpha_1, \alpha_2, \ldots, \alpha_K\right)$ corresponds to weights of the hidden layer. Thus the decision function f for a 2-layer feedforward ReLU neural network can be defined as

$$f_{\boldsymbol{W}_1, \boldsymbol{W}_2} = f_{\boldsymbol{W}_2} \cdot \sigma \cdot f_{\boldsymbol{W}_1}. \tag{137}$$

Although the two parameters $\boldsymbol{W}_1, \boldsymbol{W}_2$ in (137) are hyperplanes in a continuous space, we have already very familiar that hyperplanes can be characterized combinatorially through our discussion above. We denote the number combinatorial search space for $\boldsymbol{W}_1, \boldsymbol{W}_2$ as $\mathcal{S}_{\text{ReLUNN}}$. Therefore, the empirical risk minimization problem for the 2-layer feedforward ReLU neural network model can be defined as the following optimization problem

$$\left(\boldsymbol{W}_1^*, \boldsymbol{W}_2^*\right) = \underset{\boldsymbol{W}_1, \boldsymbol{W}_2 \in \mathcal{S}_{\text{ReLUNN}}}{\operatorname{argmin}} E_{0\text{-}1}\left(\boldsymbol{W}_1, \boldsymbol{W}_2\right), \tag{138}$$

where $E_{0\text{-}1}\left(\boldsymbol{W}_1, \boldsymbol{W}_2\right) = \sum_{n\in\mathcal{N}} \mathbf{1}\left[\operatorname{sign}\left(f_{\boldsymbol{W}_1, \boldsymbol{W}_2}\left(\bar{\boldsymbol{x}}_n\right)\right) \neq t_n\right]$.

III.3.3 The combinatorial essence 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 we introduced two algorithms that characterize the combinatorics of hyperplanes based on either data point combinations or prediction labels, 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 powerful algorithms for constructing an exact ReLU network model. We will discuss these two characterizations in detail in this section.

III.3.3.1 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 (137) can be written explicitly as

$$f_{\boldsymbol{W}_1, \boldsymbol{W}_2}(\boldsymbol{x}) = \sum_{k \in \mathcal{K}} \alpha_k \max(0, \boldsymbol{w}_k \bar{\boldsymbol{x}}) = \sum_{k \in \mathcal{K}} z_k \max(0, \tilde{\boldsymbol{w}}_k \bar{\boldsymbol{x}}), \tag{139}$$

where $\tilde{\boldsymbol{w}}_k = |\alpha_k| \boldsymbol{w}_k$, and $z_k \in \{1, -1\}$.

Equation (139) 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 *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

$$|\mathcal{S}_{\text{ReLU}}(N, D)| = 2^K \times \left(\begin{pmatrix} N \\ D \end{pmatrix} \right) = O\left(N^{DK}\right). \tag{140}$$

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 we explained in the previous Chapter, we need to run a matrix inversion for each D-combination of data items in order to get the representation for each hyperplane, this will take $O\left(D^3 \times \begin{pmatrix} N \\ D \end{pmatrix}\right)$ time. Therefore, the time complexity for the exact ReLU neural network algorithm will have a complexity

$$O\left(2^K \times \left(\begin{pmatrix} N \\ D \end{pmatrix} \right) + D^3 \times \begin{pmatrix} N \\ D \end{pmatrix}\right) = O\left(N^{DK}\right). \tag{141}$$

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, such as the one-by-one enumeration method proposed by Arora et al. [2016]. Through our discussion in Chapter II.3, we know that hyperplanes in $\mathbb{R}^D$ can be represented by D-combination of data items. Therefore, the K-combinations of hyperplanes are simply K-combinations of D-combination of data items.

Instead of using a one-by-one enumeration approach, we can alternatively use the `kcombs` generator, introduced in Subsection II.2.3.3 of Part II, as a foundation. The *nested combination-combination* generator is then constructed as the composition of two combination generators. In Haskell, this generator can be specified as follows

```
dcombsKcombs :: Int -> Int -> [Int] -> (Css, NCss)
dcombsKcombs d k = <setEmpty d, kcombs k.(!!d)> . kcombs d
```

which 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 `(Css, 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]
type NCss = [[Ncomb]]
type Css = [[Comb]]
```

The `setEmpty d` function is defined as follows

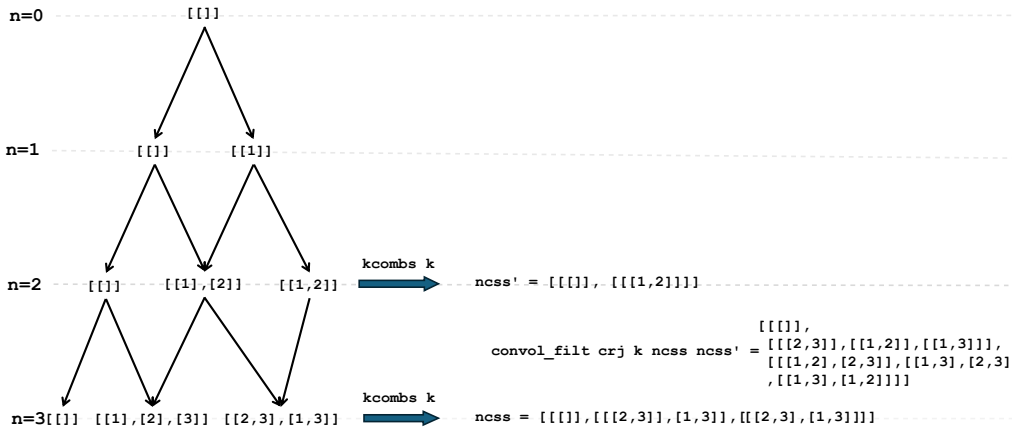


Figure III.3.1: A combination-combination nested generator, where hyperplanes are represented as 2-combinations of data items, and the 2-layer ReLU networks with three hidden neurons are defined as 2-combinations of hyperplanes, i.e., `k=2`, `d=2`. The generation tree on the left of the figure illustrates 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.

```
setEmpty d x
 | (length x) <= d = x
 | otherwise = (init x) ++ [[]]
```

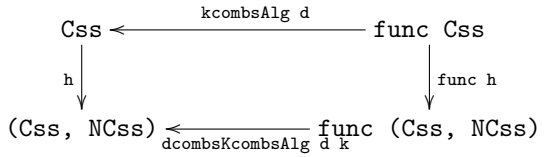
which sets the `d`-th element of the input list `x` to empty. It is used to eliminate D-combinations of data items. Once these D-combinations (hyperplanes) are used to generate new hyperplane combinations, they immediately become redundant, so we eliminate them to save memory.

Although this specification is provably correct, 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, we aim 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* (43), we need to develop an algebra `dcombsKcombsAlg`, that satisfies the following fusion condition

$$\texttt{h . kcombsAlg d = dcombsKcombsAlg d k. func h}, \tag{142}$$

where `h = <setEmpty d, kcombs k.(!!d)>` and `<f,g> = pair f g` is the categorical notation for the pairing operator. In other words, the following diagram commute



Consider the case where `func` is the join-list algebra. It can be verified that the third pattern of `dcombsKcombsAlg d k` is defined as

```
<setEmpty d.fst,cvcrj k.curry.(cross (kcombs d.!!d) id)>.(cvcrj d.fst×cvcrj k.snd).tuple,
```
(143)

satisfies the fusion condition, where `cvcrj k = convol_filt crj k` defines the third pattern of the `kcombs` generator, and `tuple (Join x y) = (x,y)` function turns two configuration joined by `Join` constructor into a tuple. Recall that `f×g = cross f g` is the cross operator introduced in Subsection II.2.2.2.

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)) = (setEmpty d css, ncss)
 where
  css = cvcrj d css1 css2
  ncss
   | null (css!!d) = [[[]]]
   | otherwise = cvcrj k (cvcrj k ncss1 ncss2) (kcombs k (css!!d))
```

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 we merge two partial configurations, `(css1 , ncss1)` and `(css2, ncss2)`, we first need 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, we merge the combinations stored in the first element of the tuple by using `css = cvcrj d css1 css2`. Since `css` generates a new list of D-combinations, we then create a new list of K-combinations 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. III.3.1.

III.3.3.2 Linear programming-based method

Combinatorial complexity In Section III.2.3, we have introduced that the 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 (137) implies that a data item $\boldsymbol{x}$ is predicted to 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 negative class. In other words, if we denote $\boldsymbol{y}^+$ and $\boldsymbol{y}^-$ as the positive and negative predictions of $\boldsymbol{y}$ respectively, then we have

$$\begin{aligned} \boldsymbol{y}^+_{\text{ReLU}} &= \bigcup_{k \in \mathcal{K}} \boldsymbol{y}^+_k \\ \boldsymbol{y}^-_{\text{ReLU}} &= \mathcal{D} \backslash \boldsymbol{y}^+_{\text{ReLU}}, \end{aligned} \tag{144}$$

where $\backslash$ is the set difference and $\bigcup_{k \in \mathcal{K}} \boldsymbol{y}^+_k$ denote the union of $\boldsymbol{y}^+_k$, $k \in \mathcal{K}$. For instance, $\boldsymbol{y}_1 = (1, 1, -1, -1)$ 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, we do not need 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 $2\sum_{d=0}^{D} \binom{N-1}{d}$ possible dichotomies, then the prediction labels for ReLU network $\boldsymbol{y}_{\text{ReLU}}$ can be calculated from (144), this operation will take $O(N \times K)$ time for each K-combination of assignments. Thus the total complexity of the LP-base method will have a complexity of

$$O\left(\sum_{n=0}^{N} \left(\text{LP}(n, D) \times \text{Cover}(n-1, D)\right) + N \times K \times \binom{\text{Cover}(N, D+1)}{K}\right) = O\left(N^{DK}\right). \tag{145}$$

Assignment-combination nested generator Similar to the previous case, the ReLU network model comprises a combination of hyperplanes. The key difference from H-based methods is that the representation of the hyperplanes now becomes assignments. Consequently, the combinatorial generator for LP-based methods is an *combination-assignment nested generator*.

III.3.4 Further discussion

III.3.4.1 Acceleration methods

Symmetry fusion in hyperplane based method In our previous discussion on the hyperplane-based algorithm for solving the 0-1 loss linear classification problem, we established the Symmetry Fusion Theorem 14, 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.

Assuming that we reverse 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 (144), the prediction labels of the 2-layer neural network, $\boldsymbol{y}_{\mathrm{ReLU}}$, 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. If we reverse the orientation of all hyperplanes, these data points will still be classified as positive.

Therefore, we only need 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.

Acceleration techniques for linear programming based method Similarly to the linear classification problem, both the global upper bound and finite dominance relation can be easily incorporated into the LP-based method for solving the ReLU network problem. Specifically, if we have a partial ReLU network assignment $\boldsymbol{y}'_{\mathrm{ReLU}} \in \{1, -1\}^n$, $1 \leq n < N$, it can be immediately discarded if its 0-1 loss is worse than the global upper bound. This upper bound can be obtained by training the network using a standard gradient descent method.

However, when using a K-combination generator, the evaluators are typically only partially fusable, which limits the effectiveness of the upper bound. Therefore, rather than relying on an upper bound that is only partially fusable, it is often more efficient to directly fuse the selector.

Nonetheless, if the method relies *solely* on the assignment generator to solve this problem, which includes a feasibility test to verify that the data partition is indeed the result of the combination of K hyperplanes, then the use of the global upper bound technique would be highly effective.

III.3.4.2 Applying integer SDP generator to save memory

III.4 Decision tree problems

A decision tree is a tree-like model used in machine learning to make decisions based on data. Imagine a flowchart or a series of "yes" or "no" questions that guide you to a final decision. Geometrically, each question or condition at a node splits the 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 the 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 axis-parallel hyperplanes $x_d = 0$. Due to the unparalleled simplicity and interpretability of the decision tree model, algorithms

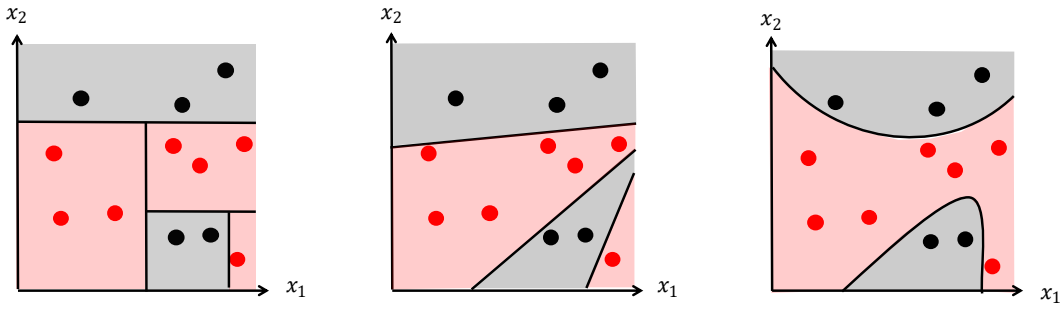


Figure III.4.1: An axis-parallel decision tree model (left), a hyperplane (oblique) decision tree model (middle), and a hypersurface (defined by degree-2 polynomials) decision tree model (right). As the complexity of the splitting functions increases, the tree's complexity decreases (involving fewer splitting nodes), while achieving higher accuracy.

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 forest [Breiman, 2001a]—have achieved significant success across various fields. Breiman [2001b] aptly noted, "On interpretability, trees rate an A+."

In this Chapter, we show 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 process.

III.4.1 Related studies

The classical approximate/heuristic algorithms for creating decision trees, such as CART and C4.5, usually employ a top-down, greedy approach. Therefore, these approximate algorithms can only lead to solutions that are suboptimal.

To improve the accuracy of the decision tree model, there are two common approaches: First, we can solve the axis-parallel decision tree problem to *global optimal*; Second, 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 employs hyperplane decision boundaries to potentially simplify boundary structures. The axis-parallel tree model is very restricted 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. In contrast, the tree generated by 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. Whereas the axis-parallel tree model can only produce *hyper-rectangles* decision regions.

As we can see from Fig. III.4.1, we apply 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 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."

Due to the formidable combinatorial complexity of decision tree problems, the studies on decision tree problems focus on either designing robust 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 binary-split decision trees [Mazumder et al., 2022, Bennett, 1992, Bennett and Blue, 1996]. In particular, a sheer amount of studies are focused on optimal decision tree algorithms for datasets with binary features [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], for which a dynamic programming solution exists. On the other hand, algorithms addressing the axis-parallel decision tree problem in full generality primarily focus on using MIP solvers [Aghaei et al., 2019, 2021, Bertsimas and Dunn, 2017, Günlük et al., 2021].

As with the other problems we've encountered, empirical risk minimization (ERM) for decision trees has been proven to be NP-hard in its full generality. However, the problem defined by (146) is not NP-hard if the number of leaves K and the number of features D are fixed. In this paper, we demonstrate how the axis-parallel split, hyperplane, and polynomial hypersurface decision tree problems can be addressed using the same algorithmic process.

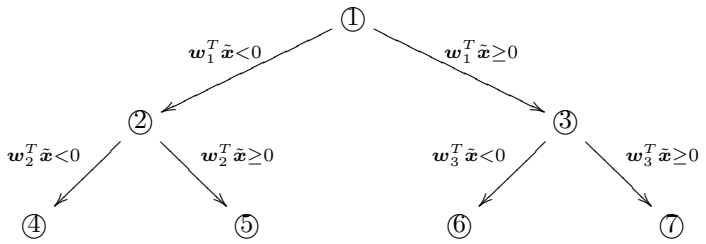
III.4.2 Problem specification

Definition of the decision tree Analogue to the procedure of classical decision tree algorithms in literature, these algorithms attempt to create a tree with binary splits. We denote a tree as T with type $T : \mathbb{T}$, and a hyperplane h with type $h : \mathbb{H}$. The nodes of the tree can be divided into two sets:

> **Branch nodes**: The branch nodes apply a split of the form $\boldsymbol{w}^T \tilde{\boldsymbol{x}} < 0$ (where $\tilde{\boldsymbol{x}} = \rho_W(\bar{\boldsymbol{x}})$ is the W-tuple embedding of $\bar{\boldsymbol{x}}$). Data points that satisfy the split condition follow the left branch in the tree, and those that do not follow the right branch. We denote the set of all branch nodes for a tree T as branch(T).
>
> **Leave nodes**: The leaf nodes make predictions for each point that falls into them. We denote the set of all leaf nodes for a tree T as leave(T). We refer to the connected region in $\mathbb{R}^D$ defined by a leaf node l as the *decision region* associated with leaf l.

When branch nodes are defined by hyperplanes, i.e., $W = 1$, $\tilde{\boldsymbol{x}} = \bar{\boldsymbol{x}}$, and the axis-parallel split involves only a single variable at each split, this can be achieved by enforcing the normal vector $\boldsymbol{w}$ to have only one element equal to one, with all other elements set to zero.

A tree of depth two can be illustrated as follows



MIP objective The optimal decision tree problem seeks to identify a tree T that minimizes the number of misclassifications. This problem involves two key hyperparameters. The trade-off between accuracy and tree complexity is managed either by fixing the number of branching nodes $|T|$ (with the number of leaves being $|T|+1$) or by adding a penalty term $\lambda|T|$ to the objective function. Both approaches are equivalent because a fixed λ corresponds uniquely to a tree with a fixed size $|T|$. In this exposition, we consider the case where $|T| = K$. Another important hyperparameter controls the minimum number of data items required in each leaf, denoted as $N_{\min}$. Given these parameters, we can formulate the optimal decision tree problem as follows

$$\begin{aligned} T^* = \underset{T\in\mathcal{S}_{\text{DTree}}}{\operatorname{argmin}}\ & E_{0\text{-}1}(T) \\ \text{s.t.}\ \ & |l| \geq N_{\min}, \forall l \in \text{leaves}(T) \\ & |T| = K \end{aligned} \tag{146}$$

where $E_{0\text{-}1}(T)$ denotes the number of misclassifications of tree T, $\mathcal{S}_{\text{DTree}}$ is the combinatorial search space consists of all possible decision trees, and leave(T) are all leaves for tree T.

III.4.3 The combinatorial essence of 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 at each branch node through an

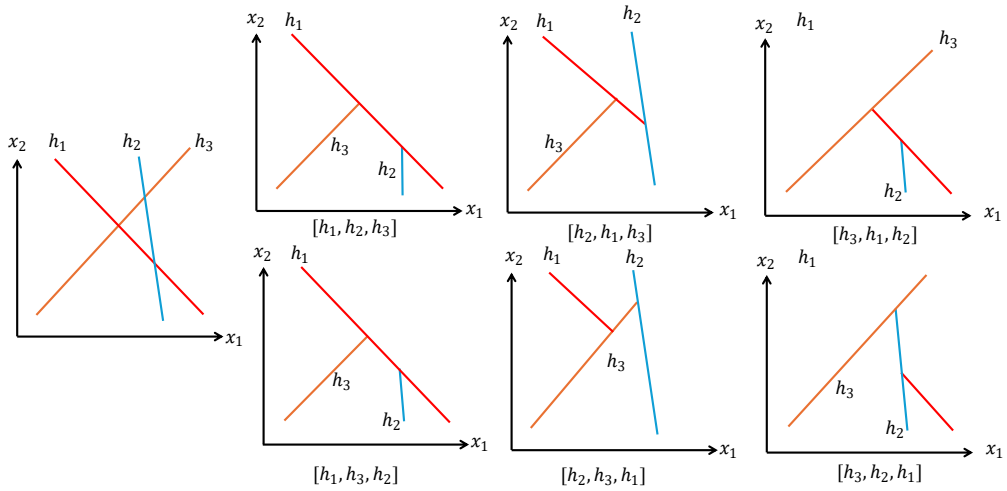


Figure III.4.2: This example illustrates how the ordering (permutation) of hyperplanes affects the partitioning in a decision tree model. The six possible permutations of three hyperplanes h_1, h_2, h_3 resulting different partitions of the space. Once a hyperplane is placed in the first position, the 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 of 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 $\mathrm{O}\big(N^K\big)$. 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 $K = 2, 3$.

Next, we will analyze 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 are required to partition the feature space. However, unlike the deep ReLU network model, the *ordering* of hyperplanes within a set of combinations is critically important for the decision tree model. In Fig. III.4.2, we illustrate how the ordering (permutation) of hyperplanes affects the partitioning in the decision tree model. Each permutation of the 3-combinations of hyperplanes results in a different partition. We have shown only a subset of the possible partitions involving three hyperplanes. When we choose the first hyperplane, the second hyperplane can belong to either side of the first hyperplane, thus there exist 2^{K-1} possibilities for each permutation of hyperplanes. By characterizing a decision tree as a K-permutation of hyperplanes, the type decision tree $\mathbb{T}$ is equivalent to a list of hyperplane $[\mathbb{H}]$.

Following from the 0-1 loss Linear Classification Theorem 13, since decision tree models are essentially PWL functions, Thm. 13 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, we can safely 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 hyperplanes in $\mathbb{R}^D$„ with respect to a dataset of size N in general position. The size of $\mathcal{S}_{\mathrm{KPH}}\left(K, D\right)$ is

$$\left|\mathcal{S}_{\mathrm{KPH}}\left(N, D\right)\right| = K! \times \begin{pmatrix} \begin{pmatrix} N \\ D \end{pmatrix} \\ K \end{pmatrix}. \tag{147}$$

Denote $\mathcal{S}_{\mathrm{HDtree}}\left(K, D\right)$ as the combinatorial search space of all possible hyperplane decision trees with K branch

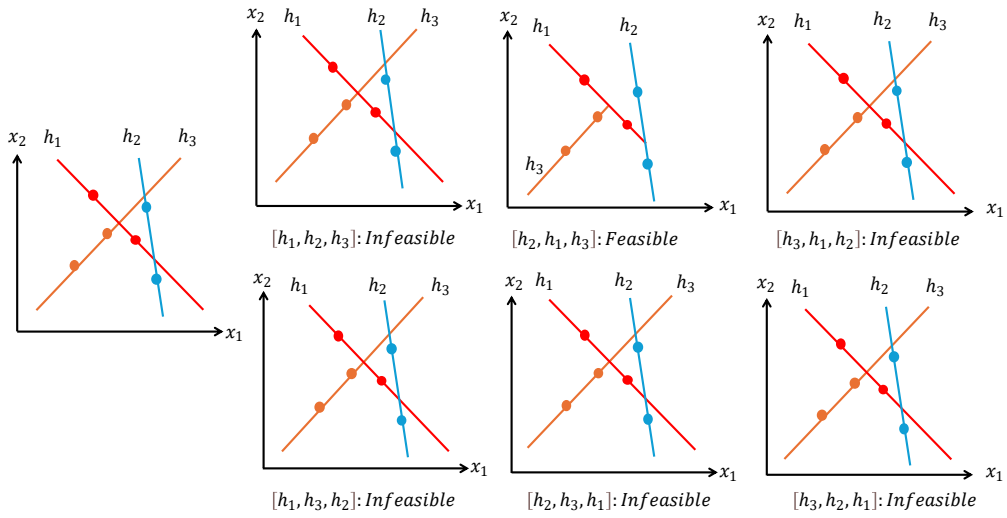


Figure III.4.3: When representing hyperplanes as combinations of data items, the data points used to construct subsequent hyperplanes can only be selected from the sub-regions defined by the previous hyperplane. For instance, if the data points used to construct the second hyperplane h_2 come from two 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 $[h_2, h_1, h_3]$.

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

$$\left|\mathcal{S}_{\text{HDtree}}\left(D, K\right)\right| = 2^{K-1} \times \left|\mathcal{S}_{\text{KPH}}\left(N, D\right)\right| = O\left(N^{DK}\right), \tag{148}$$

From our previous discussion, we know that hypersurface decision boundaries are isomorphic to hyperplanes in a higher-dimensional embedding space. To calculate the combinatorial complexity of a hypersurface decision tree, we simply count the number of possible hyperplanes in the embedding space $G = \binom{N+W}{D}$, 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 (148) 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 (148) 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 what we have given in (148). 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 $\mathcal{S}_{\text{HDtree}}\left(N, D, K\right)$ 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. III.4.3, although six 3-permutations of hyperplanes are shown, the only feasible configuration is $[h_2, h_1, h_3]$.

III.4.4 Efficient hyperplane decision tree generators

III.4.4.1 Difficulties in constructing a hyperplane decision tree (K-permutation of hyperplanes) generator

In this section, we describe an efficient recursive generator for producing hyperplane decision trees. This generator can be easily adapted to solve both the axis-parallel decision tree and the hypersurface decision tree problems. To simplify the discussion, we will focus on the hyperplane decision tree generator.

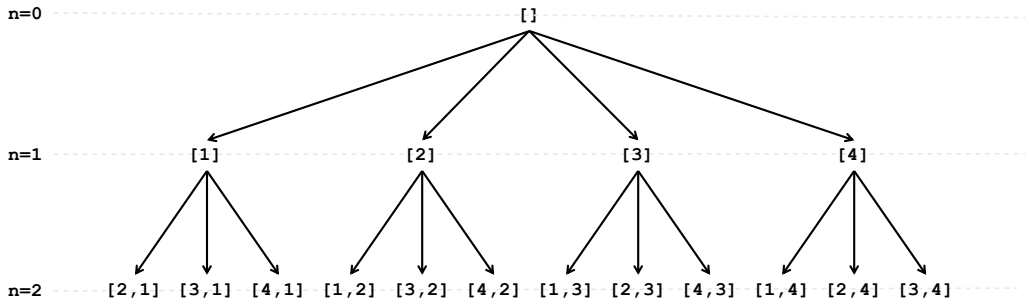


Figure III.4.4: The classical K-permutation generator ($K = 2$) based on sequential decision process (catamorphism over cons-list datatype). We begin with an empty configuration [ ] and recursively select elements from the input sequence $[1, 2, 3, 4]$ of length $N = 4$. At each recursive stage n, there are $N - n$ possible selection choices.

In the analysis of the combinatorial essence of the hyperplane decision tree problem presented earlier, we demonstrated that the key to solving the hyperplane decision tree problem is constructing a generator for generating K-permutation of hyperplanes. Analogue to the ReLU neural network problem, where a combination-combination nested generator is employed. The K-permutation of hyperplanes generator is the same as a *combination-permutation nested generator*. It consists of two separate sub-generators: one for generating D-combinations of data points (hyperplanes), and another for K-permutations, which recursively takes the hyperplanes generated by the first generator and uses them to produce K-permutations of hyperplanes.

Similar to the case of the combination-combination nested generator, the ordinary K-permutation generator based on sequential decision process (catamorphism over cons-list datatype), cannot be used to construct a combination-permutation nested generator. For instance, the classical K-permutation generator, as depicted in Fig. III.4.4, is designed based on the definition of K-permutations. It enumerates all possible K-permutations by recursively selecting one element from the input list. This generator has been applied to solve the *rule list problem* [Angelino et al., 2018], where Angelino et al. [2018] execute the generator using a depth-first approach.

However, this classical K-permutation generator is not well-suited for this problem. Since executing the classical K-permutation generator requires storing the all input data. In the decision tree problem, this input sequence consists of all possible hyperplanes in $\mathbb{R}^D$, which has a size of $O\left(N^D\right)$ in size. Generating all these hyperplanes is both time-consuming and memory-intensive.

III.4.4.2 Haskell implementation of the combination-permutation nested generator

To address the limitations posed by the classical K-permutation generator based on the cons-list datatype, we need to determine how to effectively "merge" newly generated hyperplanes with existing trees. For problems with a combinatorial structure based on K-permutations, at least two approaches can be employed to address this issue, which will be explored in the following discussion.

Join-list K-permutation generator Analogous to the combination-combination nested generator introduced for solving the ReLU network problem, the combination-permutation nested generator discussed here is nearly identical, with the only difference being that K-combinations of hyperplanes are replaced by K-permutations. This generator can be defined based on the `kperms` generator introduced in Section II.2.3.

The inefficient Haskell specification for the combination-permutation nested generator is defined as follows

```
dcombsKperms :: Int -> Int -> [Int] -> (Css, Tss)
dcombsKperms d k = <setEmpty d, kperms k.(!!d)> . kcombs d
```

where `Tss` represents a list of lists of decision trees, and each decision tree is a permutation of hyperplanes. In Haskell, `Tss` is defined as follows

```
type T = [Comb]
type Tss = [[T]]
```

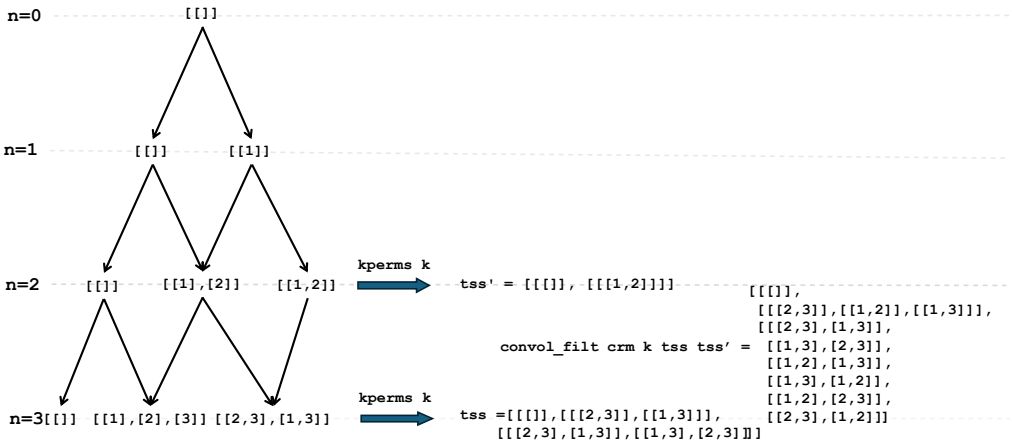


Figure III.4.5: A combination-permutation nested generator, where hyperplanes are represented as 2-combinations of data items, and decision trees are defined as 2-permutations of hyperplanes, i.e., `k=2`, `d=2`. The generation tree on the left of the figure illustrates the incremental generation process 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-permutations of hyperplanes (decision trees with K branch nodes) `tss` by using `kperms` function, a process implicitly indicated by the larger arrow. These newly generated K-permutations of hyperplanes `tss` are merged with the previous K-permutations of hyperplanes (decision trees) `tss'` by `convol_filt crm 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 decision trees of a specific size.

The fusion condition here is almost identical to the fusion condition that we discussed in Chapter III.3, so we will not repeat it. The fused combination-permutation nested algebra can be defined in Haskell as follows

```
dcombsKpermsAlg :: Int->Int-> ListFj Int (Css, Tss)-> (Css, Tss)
dcombsKpermsAlg d k Nil = ([[[]]], [[[]]])
dcombsKpermsAlg d k (Single a) = ([[[]],[[a]]],[[[]]])
dcombsKpermsAlg d k (Join (css1, tss1) (css2, tss2)) = (setEmpty d css, tss)
 where
  css = cvcrj d css1 css2
  tss
   | null (css!!d) = [[[]]]
   | otherwise = cvcrm k (cvcrm k tss1 tss2) (kperms k (css!!d))
```

Evaluating both `(snd (dcombsKperms 2 2 [1,2,3]))!!2` and `(snd (cata (dcombsKpermsAlg 2 2) [1,2,3]))!!2` give us the same result `[[[1,3],[1,2]],[[1,2],[1,3]],[[2,3],[1,2]],[[1,2],[2,3]],[[2,3],[1,3]],[[1,3],[2,3]]]`. The incremental generation process for the decision tree generator is illustrated in Fig. III.4.5.

Recursive interleave As mentioned earlier, K-permutations are simply the permutations of K-combinations, meaning they can be generated by exhaustively reorganizing each K-combination.

Generating K-permutations based on K-combinations offers two main advantages. First, the number of possible K-permutations is $K!$ times larger than the number of K-combinations, so storing only the K-combinations during the algorithm runtime can significantly reduce memory usage. Second, this method involves building each decision tree recursively, allowing us to discard infeasible partial trees before completing them. In contrast, the direct K-permutation generation method requires merging two partial trees of different sizes, making the construction of complete decision trees less efficient.

This alternative approach for generating K-permutations is called "*recursive interleave*" method. Generating K-permutation based on K-combinations closely related to the `adds` function that we have introduced in Section

II.2.3 of Part II. The `adds` function is also known as "*interleave*" function [Bird and Wadler, 1988], which is defined as

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

Intuitively, the ways of interleaving `a` with `x` can be done by appending `a` to the beginning of the list `b:x`, or starting with `b` and then interleaving `a` with `x`. Indeed, the `merge` operator used in `kpermsAlg` and `permsAlg` is based on the join-list version of the interleave function [Bird and Gibbons, 2020].

Similarly, when recursively inserting a list of elements `x` into another list `y` ,we can recursively apply the `interleave` function to each element in `x`, defined as follows

```
recInterleave :: [a] -> [a] -> [[a]]
recInterleave x y = foldr f [y] x
 where f a = concat. map (interleave a)
```

For instance, Evaluating `recInterleave [2,3] [1]` gives us
`[[2,3,1],[3,2,1],[3,1,2],[2,1,3],[1,2,3],[1,3,2]]`.

For any combinatorial structures with a *size constrains*, such as K-combination and K-permutation, using the *filtered convolution product* `convol_filt` that we introduced can lead to a more efficient and elegant definition of the generator.Designing a generator based on `convol_filt` requires determining the combining pattern for merging a list of size k configurations with the $K-k$-size configurations to satisfy the size constraint.

In the case of designing a K-permutation generator based on K-combinations, we need to consider how to "merge" a list of k-combinations, denoted as `cs` with the $K-k$-size permutations `ts` (in this problem, permutations are decision trees, so we denote a list of permutations as `ts`), for all $0 \leq k \leq K$ in a list of lists of permutations `tss'` generated in the previous recursive step. This can be accomplished by recursively interleaving each element `c` in `cs` with each element `t` in `ts`. This function can be defined using the cross product operator, named `cr_intlv` (short for "cross interleave"), as following

```
cr_intlv :: [[a]] -> [[a]] -> [[a]]
cr_intlv cs ts = concat $ crp recInterleave cs ts
```

Finally, the combination-permutation generator for generating K-permutations of hyperplanes can be defined as follows

```
dcombsKpermsAlg' :: Int->Int-> ListFr Int (Css, Tss)->(Css, Tss)
dcombsKpermsAlg' d k Nil = ([[[]]], [[[]]])
dcombsKpermsAlg' d k (Cons a (css', tss')) = (setEmpty d css, tss)
 where
  css = cvcrj d [[[]], [[a]]] css'
  ncss = kcombs k (css!!d)
  tss
   | null (css!!d) = [[[]]]
   | otherwise = convol_filt cr_intlv k ncss tss'
```

Denote the combinations and decision trees generated in the previous recursive step as `css'` and `tss'`. This recursive interleave method operates as follows: in each recursive step of the D-combination generator, there is a list of new hyperplanes (D-combinations) are generated. Instead of creating a new list of trees `tss` by running a K-permutations generator, we generate all possible *nested combinations* (i.e., K-combinations of hyperplanes) `ncss` with respect to all size D combinations, i.e., the elements in the list `css!!d` using the `kcombs` generator. We then interleave each combination of hyperplanes in `ncss` with the trees `tss'` generated in the previous step.

Evaluating `cata (dcombsKpermsAlg' 2 3) [1,2,3]` produces the same result as using `dcombsKpermsAlg`, result in
`([[]],[[3],[2],[1]],[]],[[]],[[[2,3]],[[1,2]],[[1,3]]],[[[1,2],[2,3]],[[2,3],[1,2]]...`

III.4.5 Further discussion

III.4.5.1 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 continuous methods, such as CART or C4.5, struggle to incorporate such constraints. In 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 our algorithm more efficient.

Hyperplanes lies 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 33. 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 we add 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 the 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 leave 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.

Fact 6. Adding more hyperplanes to a decision region 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:

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 $\left|M_1^-\right| + \left|M_2^-\right| = |M^-|$.

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 we assign the label of 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.5 The K-clustering problems

III.5.1 Related studies

Clustering is the grouping of similar objects and clustering of a set is a partition of elements that is chosen to minimize some measure of similarity, 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 $p = 1, 2$ is known as taxicab distance or Manhattan distance and 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].

Fayed and Atiya [2013] use a mixed breadth-depth first strategy BnB algorithm to solve the K-center problem, and Du Merle et al. [1999] combine the *interior point* algorithm with BnB to solve the K-means problem. Meanwhile, Peng and Xia [2005] use a cutting-plane algorithm for solving the K-means problem.

The use of the BnB method predominates 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.5.2 Problem specification

From (108), we have seen the 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, we can define the objective function of the K-clustering problem over continuous variables as

$$E_{\text{kcluster}}\left(\overrightarrow{\boldsymbol{\mu}}\right) = \sum_{\boldsymbol{\mu}_k \in \mathcal{U}} \sum_{\boldsymbol{x}_n \in C_k} d\left(\boldsymbol{x}_n, \boldsymbol{\mu}_k\right) \quad . \tag{149}$$

Then the *K-clustering problem* requires finding an optimal centroids vector $\overrightarrow{\boldsymbol{\mu}} \in \mathbb{R}^{DK}$ which minimize the K-clustering objective function,

$$\overrightarrow{\boldsymbol{\mu}}^* = \underset{\overrightarrow{\boldsymbol{\mu}} \in \mathbb{R}^{DK}}{\operatorname{argmin}} E_{\text{kcluster}}\left(\overrightarrow{\boldsymbol{\mu}}\right) = \sum_{\boldsymbol{\mu}_k \in \mathcal{U}} \sum_{\boldsymbol{x}_n \in C_k} d\left(\boldsymbol{x}_n, \boldsymbol{\mu}_k\right) \quad . \tag{150}$$

Similarly, the K-clustering problem defined over combinatorial variable $s = (\alpha_1, \alpha_2, \ldots, \alpha_N) \in \mathcal{S}_{\text{kasgns}}$ is rendered as

$$s^* = \underset{s \in \mathcal{S}_{\text{kasgns}}}{\operatorname{argmin}} \; E_{\text{kcluster}}(s) = \sum_{k \in \mathcal{K}} \sum_{n \in \mathcal{N}} \mathbf{1}[s_n = k] \, d(\boldsymbol{x}_n, \boldsymbol{\mu}_k)^2 \quad , \tag{151}$$

where function $\mathbf{1}[\,]$ returns 1 if the Boolean argument $s_n = k$ is true, and 0 if false.

As we mentioned, both the K-clustering and the K-medoids problems attempt to minimize the sum of the within-cluster distances for arbitrary distance functions. In contrast to the K-clustering problem, the K-medoids choose 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 we can define the K-medoids problem as

$$\begin{aligned} \mathcal{U}^* = \underset{\mathcal{U}}{\operatorname{argmin}} \; & E_{\text{kmedoids}}(\mathcal{U}) \\ \text{s.t. } & \mathcal{U} \subseteq \mathcal{D}, |\mathcal{U}| = K, \end{aligned} \tag{152}$$

where $E(\mathcal{U}) = \sum_{k \in \mathcal{K}} \sum_{x_n \in C_k} d(\boldsymbol{x}_n, \boldsymbol{\mu}_k)$ is the *objective function* for the K-medoids problem, and $\mathcal{U}^*$ is a set of centroids that optimize the objective function $E(\mathcal{U})$.

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. In 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.5.3 The combinatorial essence of the 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 applying to constrain to 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 K-clustering problem, by the MIP specification of the K-medoids problem, $C(\boldsymbol{z}_c)$ constraints the choice of our centroids to be distinct and must be chosen from data. Following the exhaustive search paradigm that we 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 $N \times (N-1) \times \cdots \times (N-K) = \begin{pmatrix} N \\ K \end{pmatrix}$ 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 14 demonstrate that the optimal solution to the K-means clustering problem must be a Voronoi partition. At the same time, Thm. 12 shows that the all possible Voronoi partition 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 product of k-combinations subset 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

$$|\mathcal{S}_{\text{kmeans}}| = \sum_{d=1}^{K+(K-1)D-1} 2^d \begin{pmatrix} (K-1)N \\ d \end{pmatrix} = O\left(N^{K+(K-1)D-1}\right). \tag{153}$$

2-means problem In the special case of $K = 2$, we have proved 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.6 Time-space complexity trade-off in designing exact algorithms

In all our previous discussions, we focused primarily on the time efficiency of the algorithms, as it has been the central concern in designing combinatorial optimization and generator algorithms. However, as we have noted many times, achieving superior time efficiency often comes with 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, we must navigate the balance between time and space complexities, using specific techniques that mitigate excessive memory usage while maintaining reasonable execution times. Here, we explore three key aspects that can help guide us in managing this trade-off.

Selection of generators Our algorithm design framework is based on deriving an efficient algorithm from an initially inefficient exhaustive search specification. As 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 our context, different generators for generating the same combinatorial structures will serve as definitions for different purposes. In Chapter II.1 of Part II, we introduce four classes of combinatorial generators. Below, we summarize the advantages and limitations of each generator in the context of combinatorial optimization.

First, the **lexicographical generation** method is inefficient for exhaustive generation and is non-recursive. As a result, it does not benefit from acceleration techniques that we introduced, 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 aspects: First, we can integrate backtracking, 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.

The **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.

Lastly, the **integer sequential decision process** falls between SDP generation and Gray code generation. It may be more efficient than classical SDP generators for generating 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 generator This trade-off is primarily observed in generators that are only *partially fusable* with the evaluator. By partially fusable generators, we refer to cases 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, which we have encountered frequently, where an incomplete configuration (i.e., partial configurations that satisfy the relaxed predicate but not the final predicate) cannot be evaluated.

In such cases, we often face 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, we only need 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, we can choose to evaluate configurations *incrementally* throughout the recursion. We have demonstrated that this approach can lead to significant speed-ups, particularly in solving the 0-1 loss linear classification problem [Xi and Little, 2023]. However, this method requires more memory than the previous approach and is more difficult to implement in parallel.

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: Comparison between the catamorphism with backtracking technique and ordinary catamorphism over join-list datatype.

Search strategies The search strategies are discussed in detail in Subsection II.2.3.4, and we summarize them 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. In contrast, the classical catamorphism approach, which does not employ 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

In this final part, we present two Haskell implementations of the algorithms for solving the *0-1 loss linear classification problem* and the *K-medoids problem*. We have discussed the definitions of these two problems in detail in Chapter III.2 and Chapter III.5 in Part III.

From our 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, we empirically demonstrate that our polynomial-time complexity predictions hold true. Moreover, we show 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, our empirical analysis on UCI datasets shows that our exact algorithm consistently achieves the best 0-1 loss among other algorithms for predictions. Similarly, EKM always obtains the best objective value compared with existing approximate algorithms on both UCI datasets and synthetic datasets.

IV.1 0-1 loss linear classification algorithm

In our previous research [He and Little, 2023], we presented an end-to-end implementation of an exact 0-1 loss linear classification algorithm, E01-ICE, short for "exact 0-1 loss incremental cell enumeration algorithm," which was developed based on the 0-1 loss Linear Classification Theorem 13. This algorithm was constructed using a catamorphism over the snoc-list datatype through the `foldl` operator.

Although our algorithm is referred to as a "cell enumeration algorithm," it actually enumerates only the "vertices" of the dual arrangement $\mathcal{H}_{\mathcal{D}}$. We refer to it as a cell enumeration algorithm to emphasize the fact that it implicitly enumerates all possible "cells" $\mathcal{H}_{\mathcal{D}}$.

In this chapter, we revisit this problem and demonstrate 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. 13, 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, we need to pair each size-D sublist 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, we are guaranteed to test every possible assignment, ensuring that we find the optimal solution.

In Chapter II.2 Section II.2.3, we 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 = cata (cpalg (ksubsAlg d) seqnAlg)
```

Evaluating `dcombsseqn 2 [1,2,3]` gives us 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],[1,2,3])]`.

Alternatively, the generator `dcombsseqn` can be defined explicitly as 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))
```

Evaluating `dcombsseqn' = cata dcombsseqnAlg` will return the same result given by `dcombsseqn`.

IV.1.2 Exhaustive, incremental cell enumeration based on join-list

We now have all the ingredients to construct our algorithm, which will enumerate all these linear classification decision hyperplanes and thus solve (126). We will need some basic linear algebra 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 for completeness.

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 comprises 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 Haskell function `take :: Int -> [a] -> [a]` simply truncates a given list to the given number of elements, and the list comprehension `[1.0,1.0..]` is the infinite list of $+1.0$ real values[16]. 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, we can apply it 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, we can obtain the corresponding predicted assignment in $\{0, 1, -1\}$ (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
```

[16] Haskell is a *lazy* language, in that terms are only evaluated when required. This allows for the specification of non-terminating structures like this, which on evaluation will turn out to be finite.

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` just 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, we want to be able to compute the corresponding term in the 0-1 loss. This makes use of Haskell "case" syntax statements

```
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, we can compute the 0-1 loss, $E_{0\text{-}1}$ 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 Our recursive combination-sequence SDP requires a partial configuration datatype which is updated by application of the decisions. For computational efficiency, we package up the linear classification hyperplane defined by the size-D combination, with the 0-1 loss for its corresponding sequence, which together we define as 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`:

```
type Config = ([Item], [Item], Maybe Model)

comb :: Config -> [Item]
comb (c,s,m) = c

seqn :: Config -> [Item]
seqn (c,s,m) = s

model :: Config -> Maybe Model
model (c,s,m) = m
```

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, which is roughly equivalent to allowing variables to take `None` values in Python. Indeed, the initial configuration has empty combination-sequence pairs, and a `Nothing`-valued model

```
empty :: Config
empty = ([], [], 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 We are now in a position to give the main recursion `e01gen`, which is defined by a join-list algebra `iveAlg`

```
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 (126), 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 `ivealg` stands for "incremental vertices enumeration algebra", inspired by its nature of enumerating the vertices of the dual arrangement $\mathcal{H}_{\mathcal{D}}$.

In the pattern defined by `Join` constructor, we merge all partial configurations `c1` in `x` and partial configurations `c2` in `y`. 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`) 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 steps.

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 `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. We have 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, we can give the program for solving problem (126). It generates all positive and negatively-oriented decision boundaries (which are viable with respect to the approximate upper bound `ub`) and selecting 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 our previous discussion of the linear classification problem, the Symmetry Fusion Thm. 14 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, we can solve the 0-1 loss linear classification problem by enumerating only the positive or negative-oriented hyperplanes, rather than both.

As a result, we can save half of the computation by modifying the `iveAlg` 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, we analyze the computational performance of our novel ICE algorithm on both synthetic and real-world data sets. Our 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, and (c) viability filtering using the approximate upper bound leads to polynomial decrease in wall-clock run-time.[17]

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]. We compare our exact algorithm ICE, against approximate algorithms: support vector machine (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

In this section, we test this prediction by analyzing the performance of the proposed algorithm using cross-validation (see Table 5), where the out-of-sample predictions use the *maximum margin representative* of the equivalence class of the exact hyperplane [Vapnik, 1999].

IV.1.3.3 Run-time complexity analysis

In the worst case situation, `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 $E_{0-1} \geq N/2$, we can get its negative part by reversing the direction of the normal vector; the resulting model will have the 0-1 loss

[17] For practical purposes, all our results are obtained using a direct, efficient C++ translation of the Haskell code given in this paper.

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 our novel incremental cell enumeration (ICE) algorithm, against approximate methods (support vector machine, 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.

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 our novel incremental cell enumeration (ICE) algorithm, against approximate methods (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.

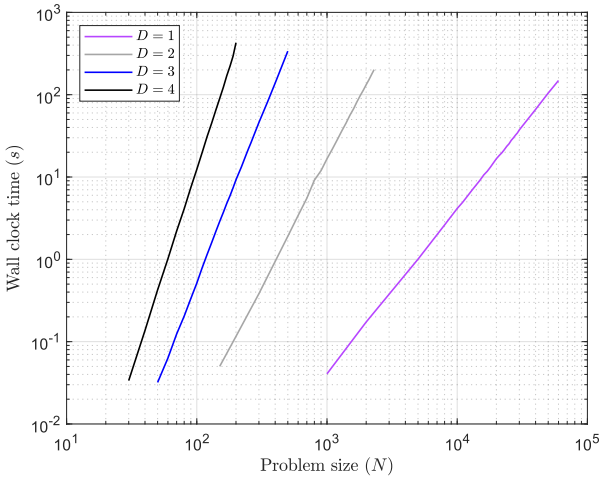


Figure IV.1.1: Log-log wall-clock run time (seconds) for the ICE algorithm in $1D$ to $4D$ synthetic datasets, against dataset size N, where the approximate upper bound is disabled (by setting it to N). The run-time curves from left to right (corresponding to $D = 1, 2, 3, 4$ respectively), have slopes 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.

smaller than $N/2$, both models represented by the same hyperplane). 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.

We test the wall clock time of our novel ICE algorithm 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. IV.1.1).

Viability filtering using the approximate global upper bound `ub` is a powerful technique which can substantially speed up our algorithm Next, we will evaluate the effectiveness of the upper bound (see Fig. IV.1.2). We generate five synthetic datasets with dimension ranging from $D = 1$ to $D = 4$, and varying `ub` 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. IV.1.2 shows polynomial degree decrease in run time as `ub` is decreased from $N/2$ to $E^*_{0\text{-}1}$, and it remains stable when `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 time complexity. Here, we compare the wall-clock run time of our ICE algorithm with the exact *branch-and-bound* (BnB) 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. IV.1.3), predicting for instance that for the $N = 150$ data size with $D = 3$, ICE would take 1.2 seconds worst-case whereas BnB would take approximately 10^{10} seconds (nearly 317 years), demonstrating the clear superiority of our approach. Similar findings would be expected to hold for other implementations such as the use of generic MIP solvers such as GLPK.

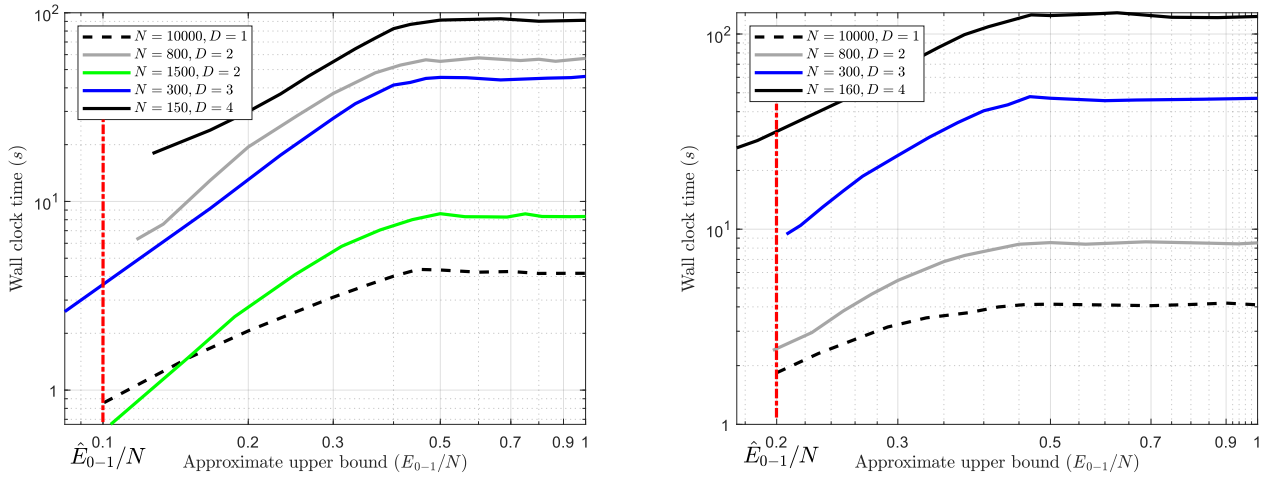


Figure IV.1.2: Log-log wall-clock run time (seconds) of the ICE algorithm on synthetic data, as the approximate upper bound viability is varied, $E^*_{0\text{-}1} \leq \texttt{ub} \leq N$, for $E^*_{0\text{-}1}$ approximately $0.1N$ (left), and approximately $0.2N$ (right). It can be seen that the empirical run-time decreases polynomially as `ub` tends towards the exact $E^*_{0\text{-}1}$ of the dataset.

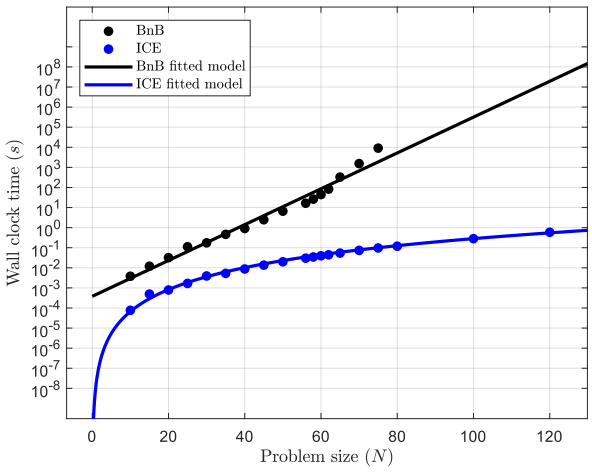


Figure IV.1.3: Log-linear wall-clock run time (seconds) plot comparing the ICE algorithm against the branch-and-bound (BnB) algorithm of Nguyen and Sanner [2013] (MATLAB implementation provided by the authors) on three dimensional synthetic data. On this log-linear scale exponential run time appears 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.

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 `Item`s.

Squared distance evaluation As we have discussed, the K-medoids problem is defined over arbitrary objective function, the most common choice is the squared Euclidean distance function $d_2(\boldsymbol{x}, \boldsymbol{\mu}) = \|\boldsymbol{x} - \boldsymbol{\mu}\|_2^2$. 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 elements 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. These require 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 we compute the distance matrix 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 Analogue to the 0-1 loss linear classification problem, the SSE of a configuration is just a floating point `type SSE = Double`, and we define `Maybe SSE` to represent the existence of the SSE in a particular configuration. Combining this with the combination-sequence datatype, we define the configuration datatype as follows

```
type Config = ([Item], [Item], Maybe SSE)

comb :: Config-> [Item]
comb(c,i,s) = c

seqn :: Config -> [Item]
seqn(c,i,s) = i

sse :: Config -> Maybe SSE
sse(c,i,s) = s
```

We refer to the prediction labels of a given set of centroids as *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 (ms!!j) | (x,i) <- zip xs asgn, i == j]
   | j <- [0..((length ms)-1)]]
  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 We are now ready to define the join-list algebra for solving the K-medoids problem

```
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, we now provide a more generic form of filter-fused algebra. Any segment-closed predicate `p` can be fused into `kmedAlg`. For instance, we can define the following filter-fused algebra with a segment-closed predicate, 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` test and `feasible` test that we used in the 0-1 loss linear classification problem, we introduce an additional segment-closed predicate, `minData`. This predicate checks whether the number of data items in each cluster is greater than `nmin`. We can define `minData` as following

```
minData :: Int -> Config -> Bool
minData nmin c
 | (length (comb c) == 0) = True
 | otherwise = lstElem (countData (length (comb c)) (getasgn (seqn c) (comb c))) <= nmin

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 `counData` 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 we introduce new medoids, 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[18]; (b) wall-clock run-time matches the worst-case polynomial time complexity analysis; (c) modern off-the-shelf MIP solvers (GLPK) for the same problem will have exponential time complexity. 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.

IV.2.2.1 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)[19] on 18 datasets from the UCI Machine Learning Repository, two open-source datasets from

[Wang et al., 2022, Padberg and Rinaldi, 1991, Ren et al., 2022], and two synthetic datasets. We show that, as expected, no other algorithms can achieve better objective function values (see Table 6).

To compare EKM's performance against the BnB algorithm proposed by Ren et al. [2022], we processed most of the real-world datasets tested therein. We discovered through our experiments that all these datasets (except IRIS) can be solved exactly using either the PAM or Faster-PAM algorithms, this indeed provides an extremely tight upper bound in the analysis of Ren et al. [2022]. Additionally, 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.

IV.2.2.2 Time complexity analysis for serial implementation

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

[18]The squares Euclidean distance function was chosen for the experiments, any other proper metrics could also be used.
[19]We set the maximum number of neighbors examined as 4, and the number of iteration as 5.

UCI dataset	N	D	EKM (ours)	PAM	Faster-PAM	CLARANS
LM	338	3	$\mathbf{3.96 \times 10^1}$ (6.82)	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	$\mathbf{8.36 \times 10^1}$ (1.21×10^1)	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	$\mathbf{3.31 \times 10^5}$ (6.98)	3.56×10^5 (4.11×10^{-3})	$\mathbf{3.31 \times 10^5}$ (3.87×10^{-3})	4.68×10^5 (3.40)
Energy	768	8	$\mathbf{2.20 \times 10^6}$ (1.01×10^2)	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	$\mathbf{3.13 \times 10^5}$ (4.98)	$\mathbf{3.13 \times 10^5}$ (3.15×10^{-3})	3.58×10^5 (5.36×10^{-3})	5.27×10^5 (2.58)
Wine	178	13	$\mathbf{2.39 \times 10^6}$ (1.11)	$\mathbf{2.39 \times 10^6}$ (1.06×10^{-3})	2.63×10^6 (2.34×10^{-3})	6.86×10^6 (5.56×10^{-1})
Yeast	1484	8	$\mathbf{8.37 \times 10^1}$ (1.10×10^3)	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	$\mathbf{6.9063 \times 10^9}$ (2.46×10^4)	6.9105×10^9 (8.68×10^{-1})	$\mathbf{6.9063 \times 10^9}$ (1.91×10^{-1})	1.44×10^{10} (2.70×10^1)
WDG	5000	21	$\mathbf{1.67 \times 10^5}$ (2.49×10^5)	$\mathbf{1.67 \times 10^5}$ (1.34)	$\mathbf{1.67 \times 10^5}$ (1.97×10^{-1})	2.77×10^5 (5.32×10^3)
IRIS	150	4	$\mathbf{8.40 \times 10^1}$ (7.32×10^{-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	$\mathbf{5.98 \times 10^2}$ (1.71)	$\mathbf{5.98 \times 10^2}$ (1.14×10^{-3})	$\mathbf{5.98 \times 10^2}$ (3.59×10^{-3})	1.12×10^3 (7.82×10^{-1})
GLASS	214	9	$\mathbf{6.29 \times 10^2}$ (1.81)	$\mathbf{6.29 \times 10^2}$ (1.01×10^{-3})	$\mathbf{6.29 \times 10^2}$ (1.62×10^{-3})	1.04×10^3 (2.27)
BM	249	6	$\mathbf{8.63 \times 10^5}$ (2.64)	8.76×10^5 (4.12×10^{-3})	$\mathbf{8.63 \times 10^5}$ (1.61×10^{-3})	1.33×10^6 (1.02×10^1)
HF	299	12	$\mathbf{7.83 \times 10^{11}}$ (4.57)	$\mathbf{7.83 \times 10^{11}}$ (1.00×10^{-3})	$\mathbf{7.83 \times 10^{11}}$ (4.87×10^{-3})	1.88×10^{12} (6.55×10^{-1})
WHO	440	7	$\mathbf{8.33 \times 10^{10}}$ (1.67×10^1)	$\mathbf{8.33 \times 10^{10}}$ (5.62×10^{-3})	$\mathbf{8.33 \times 10^{10}}$ (2.81×10^{-3})	1.21×10^{11} (8.12)
ABS	740	19	$\mathbf{2.32 \times 10^6}$ (1.04×10^2)	$\mathbf{2.32 \times 10^6}$ (2.11×10^{-2})	2.38×10^6 (5.00×10^{-3})	2.96×10^6 (7.80×10^1)
TR	980	10	$\mathbf{1.13 \times 10^3}$ (2.16×10^2)	$\mathbf{1.13 \times 10^3}$ (5.14×10^{-2})	$\mathbf{1.13 \times 10^3}$ (1.14×10^{-2})	1.38×10^3 (2.59×10^2)
SGC	1000	21	$\mathbf{1.28 \times 10^9}$ (2.20×10^2)	$\mathbf{1.28 \times 10^9}$ (1.71×10^{-1})	$\mathbf{1.28 \times 10^9}$ (4.22×10^{-2})	2.52×10^9 (2.24)
HEMI	1995	7	$\mathbf{9.91 \times 10^6}$ (3.06×10^3)	$\mathbf{9.91 \times 10^6}$ (3.64×10^{-1})	$\mathbf{9.91 \times 10^6}$ (6.99×10^{-2})	1.66×10^7 (9.53)
PR2392	2392	2	$\mathbf{2.13 \times 10^{10}}$ (1.38×10^4)	$\mathbf{2.13 \times 10^{10}}$ (3.66×10^{-1})	$\mathbf{2.13 \times 10^{10}}$ (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) for $K = 3$, in terms of sum-of-squared errors (E), smaller is better. Best performing algorithm marked **bold**. Wall clock execution time in brackets (seconds). Datasets are from the UCI repository.

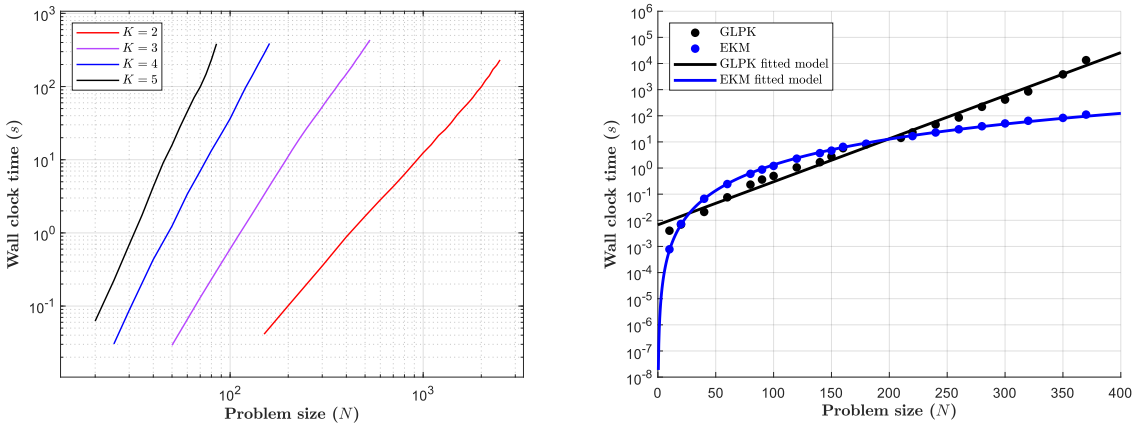


Figure IV.2.1: Log-log wall-clock run time (seconds) for our algorithm (EKM) tested on synthetic datasets (left panel). The run-time curves from left to right (corresponding to $K = 2, 3, 4, 5$ respectively), have slopes 3.005, 4.006, 5.018, and 5.995, an excellent match to the predicted worst-case run-time 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 a classical MIP (BnB) solver (GLPK) on synthetic datasets 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.

from 25 to 160, and $K = 5$ ranges from 30 to 200. The worst-case predictions are well-matched empirically (Fig. IV.2.1, left panel). As predicted, the off-the-shelf BnB-based MIP solver (GLPK) exhibits worst-case exponential time complexity (Fig. IV.2.1, right panel).

IV.3 Discussion

In this chapter, we introduce two polynomial-time algorithms for solving the 0-1 loss classification problem and the K-medoids problem. Our empirical analysis demonstrates that these two exact algorithms outperform approximate methods on various UCI datasets, and the predicted worst-case complexity matches the observed empirical time complexity. While our algorithms are embarrassingly parallelizable, our current implementation only executes the incremental version.

Both algorithms are based on the combination (sublist) generator we previously introduced. As noted, the consequence of using the combination generator is that the evaluator is only *partially fusable* with the generator, which significantly restricts the effectiveness of the global upper bound techinique in such cases. As illustrated in Fig. IV.1.2, even in the best-case case, the use of the global upper bound achieves only a modest magnitude of speed-up. Consequently, we adopted an alternative evaluation strategy for the EKM algorithm, as discussed in III.6, which directly evaluate objective of a configuration with respect to all input data sequence instead of incrementally evaluating it.

Besides presenting two novel algorithms, we aim to prompt researchers to reconsider the metric for assessing the efficiency of exact algorithms to account for subtleties beyond simple problem scale. In discussing the "goodness" of exact algorithms for ML, it is critical to recognize that focusing solely on the scalability of these algorithms—for instance, their capacity to handle large datasets—does not provide a comprehensive assessment of their utility. This inclination to prioritize scalability when assessing exact algorithms arises from the perceived intractable combinatorics of many ML problems. However, for many ML problems, the problems specified for proving NP-hardness are not the same as their original definitions used in practical ML application. If a polynomial-time algorithm does exist for these seemingly intractable problems, overemphasizing scalability can mislead scientific development, diverting attention from important measures such as memory usage, worst-case time complexity, and the practical applicability of the algorithm in real-world scenarios.

Therefore, judging an algorithm implementation solely by the scale of the dataset it can process is not an adequate measure of its effectiveness. Indeed, for large datasets, the use of exact algorithm may often be unnecessary as many high-quality approximate algorithms provide very good results, supported by solid theoretical assurances. If the clustering model closely aligns with the ground truth, the discrepancy between approximate and exact solutions should not be significant, provided the dataset is sufficiently large. For the study of the K-medoids problem, while algorithms presented in [Ren et al., 2022, Ceselli and Righini, 2005, Elloumi, 2010] are exact in principle,

experiments reported by the authors do not demonstrate the actual computation of exact solutions, nor do they provide any theoretical guarantee on the computational time required to achieve satisfactory approximate solutions. If the application of the problem is concerned with only the approximate solution, it may be more beneficial to concentrate on developing more efficient or more robust heuristic algorithms. This could potentially offer more practical value in scenarios where approximate solutions are adequate.

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. 1983.

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.

Saugata Basu, Richard Pollack, and MF Roy. A new algorithm to find a point in every cell de ned by a family of polynomials. 1995.

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.

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, Jean Pauphilet, and Bart Van Parys. Sparse regression. *Statistical Science*, 35(4):555–578, 2020.

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.

Richard S. Bird and Philip L. Wadler. *An Introduction to Functional Programming*. Prentice-Hall, 1988.

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.

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. URL https://books.google.co.uk/books?id=JwQx-WOmSyQC.

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.

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.

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.

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.

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.* Citeseer, 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.

Thomas Gerstner and Markus Holtz. Algorithms for the cell enumeration and orthant decomposition of hyperplane arrangements. 2006.

Surbhi Goel, Adam Klivans, Pasin Manurangsi, and Daniel Reichman. Tight hardness results for training depth-2 relu networks. *ArXiv preprint ArXiv:2011.13550*, 2020.

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.

Robin Hartshorne. *Algebraic geometry*, volume 52. Springer Science & Business Media, 2013.

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. An efficient, provably exact algorithm for the 0-1 loss linear classification problem. *ArXiv preprint ArXiv:2306.12344*, 2023.

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 Special Interest Group on Programming Languages Notices*, 48(9):209–220, 2013.

Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Conjugate hylomorphisms–or: The mother of all structured recursion schemes. *ACM Special Interest Group on Programming Languages Notices*, 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 Notices*, 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*. 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.

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. URL https://doi.org/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. First steps towards the theory of rose trees. *Centrum Wiskunde Informatica, Amsterdam*, 1988.

LGLT Meertens. Algorithmics: Towards programming as a mathematical activity. 1986.

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 Special Interest Group on Programming Languages Notices*, 52(10):2–13, 2017.

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.

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.

The Univalent Foundations Program. Homotopy type theory: univalent foundations of mathematics. *ArXiv preprint ArXiv:1308.0729*, 2013.

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.

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.

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, 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.

H 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.

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

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 Proofs

Corollary 5. 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]`

$$\texttt{concat . map (ΛalgR . (Cons a)) = alg . (Cons a)} \tag{154}$$

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)
≡ ER = 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

$$\texttt{concat . P (ΛalgR . F) = alg . F.} \tag{155}$$

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))
≡ Pf = Λ (f . ∈), 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 ∈ . F is a function, and Λ-fusion law
 concat . Λ(ΛalgR . fmap ∈ . F))
≡ definition of the cross product crp (f . F) = Λ(f . fmap ∈ . F)
 concat . crp (ΛalgR . F)
```

□