Predictive and Prescriptive Trees for Optimization and Control Problems

by

Cheol Woo Kim

B.A., Seoul National University, 2019

Submitted to the Sloan School of Management
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN OPERATIONS RESEARCH

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2024

Authored by: Cheol Woo Kim
Sloan School of Management
Aug 9, 2024

Certified by: Dimitris J. Bertsimas
Boeing Leaders for Global Operations Professor of Management
Associate Dean for Business Analytics, Thesis Supervisor

Accepted by: Patrick Jaillet
Dugald C. Jackson Professor
Department of Electrical Engineering and Computer Science
Co-director, Operations Research Center

Predictive and Prescriptive Trees for Optimization and Control Problems

by

Cheol Woo Kim

Submitted to the Sloan School of Management
on Aug 9, 2024 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN OPERATIONS RESEARCH

ABSTRACT

This thesis introduces novel methods to expedite the solution of a broad range of optimization and control problems using machine learning, specifically decision tree algorithms. In many practical settings, similar optimization and control problems often need to be solved repeatedly. We propose methods to leverage patterns from pre-solved problem instances using machine learning, leading to drastically faster solutions once training is complete.

The thesis is structured into four parts, each tackling different class of optimization or control problems. In Chapter 2, we propose a machine learning approach to the optimal control of multiclass fluid queueing networks (MFQNETs). We prove that a piecewise constant optimal policy exists for MFQNET control problems, with segments separated by hyperplanes passing through the origin. We use Optimal Classification Trees with hyperplane splits (OCT-H) to learn an optimal control policy for MFQNETs.

In Chapter 3, we study fluid restless multi-armed bandits (FRMABs), deriving fundamental properties and designing efficient numerical algorithms. Using these results, we learn state feedback policies with OCT-H and introduce a novel feature augmentation technique to handle nonlinearities.

In Chapter 4, we propose a machine learning framework for solving two-stage linear adaptive robust optimization problems with binary here-and-now decisions and polyhedral uncertainty sets. We also introduce novel methods to expedite training data generation and reduce the number of different target classes the machine learning algorithm needs to be trained on.

In Chapter 5, we introduce a prescriptive machine learning approach to speed up the process of solving mixed integer convex optimization (MICO) problems. We use a prescriptive machine learning algorithm, Optimal Policy Trees (OPT), instead of more commonly used classification algorithms. We demonstrate that OPT-based methods have a significant advantage in finding feasible solutions compared to classification algorithms.

We test our approach on various synthetic and real-world problems. Using the proposed methods, we can obtain high-quality solutions to a broad range of large-scale optimization and control problems in real-time – within milliseconds.

Thesis supervisor: Dimitris J. Bertsimas
Title: Boeing Leaders for Global Operations Professor of Management
Associate Dean for Business Analytics

Acknowledgments

First, I would like to express my utmost gratitude to my advisor, Dimitris Bertsimas. When I first arrived at MIT in August 2019, I knew nothing about the topics in this thesis or research in general. Reflecting on how clueless I was then, I am still amazed by his patience, strength, and leadership. He has taught me how to think critically, approach complex problems, and effectively communicate results to others.

Most importantly, I learned through him the importance of fearlessness and perseverance in taking action on what matters. What I noticed about Dimitris is that he doesn't get disappointed by external forces, always looks in a positive direction, and takes immediate action without looking back if he is convinced that something is important to achieve. It has been a privilege for me to witness and share such energy, decisiveness, conviction, and fearlessness over the past five years. He has always been, and will continue to be, a source of inspiration to me.

I would also like to express my gratitude to my committee members, José Niño-Mora, Alexandre Jacquillat, and David Gamarnik, for their guidance and support. Ever since I started working with José, I have consistently and quite frequently asked for meetings and advice. He has been incredibly patient, kind, and generous with all my requests. His warmth and gentleness have always made me feel more confident and comfortable. Alex was also my committee member for the second-year general exam. I am always amazed by his kindness, passion, compassion, and professionalism. His positive feedback has been a great source of confidence for me. David has also given me precious advice and help for my future career. They are the best mentors I could have ever asked for.

My sincere thanks go to Ernest Ryu for his incredible career advice, which has significantly shaped my future.

I am forever grateful to two of my best friends who are also pursuing PhD degrees: Kyuseong Choi at Cornell and Jay Yoo at UCLA (alphabetical order). They have been the best mentors and friends, both in my personal and academic lives. Without them, none of this would have been possible.

I would also like to thank Seung Hyon Lee, Sangwook Kong, Bomi Huh, and Seunghwan Paik in Korea, who always help me think in a different, more positive way.

I am very fortunate to have many wonderful Korean friends in Boston: Bumsoo Kim, Joonhyuk Cho, Seok Hee Han, Sungyun Yang, Byunghun Lee, Wooseok Lee, Hyunwon Chu, Seungwook Han, JeongHyun Yoon, Eugene Park, Junyoung Hong, Dongwon Lee, Suhan Kim, Jungmin Kim, Jay Kim, Eunjin Jung, Seunghun Han, Seongwon Kim, Joohee Kim, Insang Yoo, Yujin Oak, Doyoon Kim, Jayoung Ryu, Ousama, Jiwook Kim, Joonpyo Sohn, Jooli Han, Sun Kim, Soyoon Yang, Saebyeok Shin, Jaeyoon Song, Jimin Park, Ukjin Kwon, Albert Shin,

Jaedong Hwang, Kihyun Kim, Kwanwoo Hahn, and Hyemin Koo (random order). Thanks to these friends, I have countless wonderful memories in Boston.

I would also like to thank all the friends I have met at the ORC: Evan Yao, Leonard Boussioux, Cynthia Zeng, Thodoris Koukouvinos for our chat times that helped me adapt to living overseas; Zhen Lin, who has been my fellow TA and class project teammate multiple times; Jean Pauphilet and Arthur Delarue for giving me precious advice during my hardest times in the PhD journey. I would also like to extend my gratitude to Moise Blanchard, Amine Bennouna, Shuvomoy Das Gupta, Matthew Yuan, Kevin Hu, Sabrina Zhai, Victor Gonzalez, Leann Thayaparan, Kimberly Villalobos Carballo, Irra Na, Yu Ma, Kayhan Behdin, Haoyue Wang, Benjamin Boucher, Angelos Georgio Koulouras, and Alex Paskov.

Finally, I owe everything to my family. To my parents and brother, Kyunghee Shin, Jae-il Kim, and Kwang Woo Kim. To my aunt in New York, Sunghee Shin. To my uncle and aunt in Korea, Hoyoung Choi and Sookhee Shin. This thesis is dedicated to their unwavering support and love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Advances in algorithms and hardware capabilities have enabled us to solve optimization and control problems on a scale that was once considered unimaginable just a couple of decades ago. However, the computational burden associated with solving large-scale optimization and control problems, particularly in real-time scenarios, remains an ongoing challenge. Furthermore, computational complexity theory has revealed that numerous problems of practical significance are inherently difficult to solve. As a result, researchers have focused on designing either approximation algorithms with provable guarantees, or heuristics with good empirical performance [19], [36], [48], [66].

This thesis presents a unified approach to overcome this difficulty by harnessing the power of machine learning, specifically through decision tree algorithms. In many practical settings, similar optimization and control problems often need to be solved repeatedly. The overarching goal of this thesis is to propose methods to expedite the solution of a broad range of problems, outperforming conventional algorithms by leveraging patterns from previous instances. By doing so, we can solve large-scale optimization and control problems that were traditionally viewed as intractable, all in real-time – a matter of milliseconds. This paradigm shift has the potential to yield substantial enhancements in a wide array of real-world decision systems, including machine learning, healthcare, power systems, design, simulation, synthesis and numerous others.

This area of research has received substantial attention within the academic community in recent years. For instance, [2], [63] have proposed techniques for learning efficient branching rules to solve mixed-integer optimization problems, while [10], [59] have employed machine learning to automatically fine-tune hyperparameters in optimization algorithms. Reinforcement learning methods have also played a pivotal role in solving stochastic control problems with the aid of machine learning techniques [44], [72], [95]. For a comprehensive overview of this subject, please refer to [18], [73].

However, despite these advancements, the potential of this research direction remains under-explored. Key areas for improvement include selecting the most suitable machine learning algorithms for solving specific optimization and control problems. While deep neural networks are commonly chosen for their powerful approximation capabilities, theoretical considerations often suggest alternative algorithms that could yield superior empirical results.

Another important area lies in expanding the scope of optimization and control problems amenable to machine learning. This involves identifying recurring parameters as features and

selecting relevant prediction targets within a problem class. Moreover, each problem class presents inherent challenges that must be addressed to successfully apply a machine learning framework.

In this thesis, we explore the application of decision tree algorithms to a wide range of problems and propose appropriate approaches tailored to each task. These dual contributions—identifying the most effective machine learning algorithms and expanding the application scope—complement each other, collectively advancing the integration of machine learning into optimization and control problems.

1.1 Outline and Main Contributions

This thesis consists of two primary topics. The first two chapters focus on using decision trees to learn state feedback policies for continuous-time optimal control problems. The final two chapters focus on accelerating the solution of static mixed-integer programming problems and their two-stage robust optimization extensions using decision trees.

Chapter 2 We propose a machine learning approach to the optimal control of multiclass fluid queueing networks (MFQNETs) that provides explicit and insightful control policies. We prove that a piecewise constant optimal policy exists for MFQNET control problems, with segments separated by hyperplanes passing through the origin. We use Optimal Classification Trees with hyperplane splits (OCT-H) to learn an optimal control policy for MFQNETs. We use numerical solutions of MFQNET control problems as a training set and apply OCT-H to learn explicit control policies. We report experimental results with up to 33 servers and 99 classes that demonstrate that the learned policies achieve 100% accuracy on the test set. While the offline training of OCT-H can take days in large networks, the online application takes milliseconds.

Chapter 3 We propose a machine learning approach to the optimal control of fluid restless multi-armed bandits (FRMABs) with state equations that are either affine or quadratic in the state variables. By deriving fundamental properties of FRMAB problems, we design an efficient numerical algorithm. Using this algorithm, we solve multiple instances with varying initial states to generate a comprehensive training set. We then learn a state feedback policy using Optimal Classification Trees with hyperplane splits (OCT-H). We test our approach on machine maintenance, epidemic control and fisheries control problem. Our method yields high-quality state feedback policies and achieves a speed-up of more than 26 million times compared to a direct numerical algorithm for fluid problems.

Chapter 4 We propose an approach based on machine learning to solve two-stage linear adaptive robust optimization (ARO) problems with binary here-and-now variables and polyhedral uncertainty sets. We encode the optimal here-and-now decisions, the worst-case scenarios associated with the optimal here-and-now decisions, and the optimal wait-and-see decisions into what we denote as the strategy. We solve multiple similar ARO instances in advance using the column and constraint generation algorithm and extract the optimal

strategies to generate a training set. We train machine learning models that predict high-quality strategies for the here-and-now decisions, the worst-case scenarios associated with the optimal here-and-now decisions, and the wait-and-see decisions. The models can be applied to problems with varying dimensions. We also introduce novel methods to expedite training data generation and reduce the number of different target classes the machine learning algorithm needs to be trained on. We apply the proposed approach to the facility location, the multi-item inventory control and the unit commitment problems. Our approach solves ARO problems drastically faster than the state-of-the-art algorithms with high accuracy.

Chapter 5 We introduce a prescriptive machine learning approach to speed up the process of solving mixed integer convex optimization (MICO) problems. We solve multiple optimization instances and train a machine learning model in advance, which we use to solve new instances. Previous works [24], [25] have shown that the predictions of classification algorithms enable us to solve optimization problems much faster than commercial solvers. What distinguishes this paper from the previous work is that we use a prescriptive algorithm, Optimal Policy Trees (OPT) [3], instead of classification algorithms. While classification algorithms aim to predict the correct label and consider all other labels equally undesirable, a prescriptive approach takes into account all the available decision options and their counterfactuals. We first introduce an algorithm that is purely based on OPT, and also its extension. We compare their performance with Optimal Classification Trees (OCT) [20], [21] on various MICO problems. Test problems include transportation optimization, portfolio optimization, facility location and hybrid vehicle control. We also experiment on real-world instances taken from MIPLIB [53]. OPT-based methods have a significant edge on finding feasible solutions, while OCT-based methods have a slight edge on the degree of suboptimality. The proposed extension of the pure OPT algorithm improves on the suboptimality of the solutions the algorithm produces.

Chapter 2

Optimal Control of Multiclass Fluid Queueing Networks: A Machine Learning Approach

2.1 Introduction

Multiclass queueing networks (MQNETs) are complex systems that model the behavior of multiple classes of jobs, each with their own arrival and service rates, routing paths and holding costs. These networks find numerous applications in diverse fields, including manufacturing [67], healthcare [41] and communication networks [102] among many. The control of MQNETs is of great importance in improving system efficiency, optimizing resource allocation, and reducing operational costs. However, the inherent complexity of these systems makes their analysis and control a challenging task.

Multiclass fluid queueing networks (MFQNETs) have been developed as a deterministic, continuous approximation of MQNETs, primarily to provide a tractable method for analyzing the stability of the underlying MQNETs. [43], [104] demonstrate that the stability of MFQNETs implies the stability of underlying MQNETs. Several related studies including [47], [51], [80] have also explored the topic. See [27] for a comprehensive review.

MFQNETs also provide a useful way to construct control policies for MQNETs as the optimal control of MFQNETs is often much more tractable than the optimal control of underlying MQNETs. To this end, several approaches have been proposed in the literature. [76], [77] propose discrete review policies and show that they achieve asymptotic optimality and stability under fluid scailing. [31] provide a robust formulation of MFQNET control problem and translate the resulting policy to the underlying MQNET. [32], [45] propose methods to approximately minimize make-span based on the associated fluid models. For a comprehensive review of the topic, see [79] and [27].

Mathematically, optimal control of MFQNETs falls into a subclass of infinite dimensional linear optimization models known as separated continuous linear programs (SCLPs). Several researchers have investigated the theoretical properties of SCLPs, such as [4], [90], [92], [93]. [7] find closed-form optimal policies for specific MFQNETs using optimality conditions from optimal control theory. Other works have proposed numerical algorithms for solving

SCLPs. [74], [91], [94] develop algorithms based on discretization, while [101], [109] propose simplex-like methods. [14], [49] propose polynomial-time approximation algorithms.

Despite significant efforts in the field and its practical applications, optimal control of MFQNETs remains a challenging computational task. Moreover, current algorithms typically provide only numerical solutions, making it difficult to gain insight into the underlying structure of the optimal policy.

Recently, there has been growing interest in applying machine learning techniques to solve challenging optimization and control problems. For instance, [2], [24], [25], [28], [39], [63] propose machine learning-based approaches to mixed-integer optimization. [30] develop a method to solve two-stage adaptive robust optimization problems using machine learning. Machine learning has been used for hyperparameter tuning in optimization algorithms as well [12], [59]. For queueing network control, reinforcement learning methods are proposed in [44], [72], [95]. Although these approaches have shown to be effective in addressing computational challenges, it can be difficult to provide theoretical guarantees that the machine learning methods lead to optimal solutions.

In this paper, we present a novel approach that leverages machine learning to solve MFQNET control problems. The MFQNET control problem we consider is the fluid analog of the sequencing problem in MQNETs. The sequencing problem in MQNETs is a stochastic and discrete control problem that involves deciding which class of jobs to process at each server at any given time, with the aim of minimizing the expected total cost. We formulate the fluid analog of this problem as a SCLP problem and propose a machine learning-based algorithm to address it.

We solve multiple MFQNET control problems and use the resulting numerical solutions to learn an optimal state feedback policy. The machine learning algorithm we use is Optimal Classification Trees with hyperplane splits (OCT-H) proposed by [20], [21]. OCT-H is a classification algorithm that partitions the feature space using hyperplanes and assigns a prediction to each region. We prove that OCT-H can learn exact optimal policies for the MFQNET control problems.

The contributions of the paper are as follows.

1. We prove the existence of a piecewise constant optimal policy for MFQNET control problems, with segments separated by hyperplanes passing through the origin. This result was previously proven only for special cases.

2. Based on the theoretical findings, we propose an efficient algorithm that can learn an exact optimal control policy for MFQNETs using OCT-H. We report experimental results with up to 33 servers and 99 classes that demonstrate that the learned policies achieve 100% accuracy on the test set.

3. Once a policy is learned offline, it can be directly applied online to unseen states in milliseconds, leading to a significant speed-up compared to solving the problem numerically.

4. The high interpretability of decision trees allows us to gain insights into the structure of the optimal policy, which is a significant advantage that numerical optimization algorithms often lack. By providing the actual decision trees learned by OCT-H, we develop a deeper understanding of MFQNETs and their optimal policy.

The structure of this paper is as follows. Section 2.2 provides the definition of MFQNET control, along with the associated optimality conditions. We also provide a brief review of OCT-H. Section 2.3 provides our theoretical results on the structure of optimal policy for MFQNET control. We then develop a learning algorithm based on OCT-H and provide a small example to illustrate the method. Section 2.4 reports the results of computational experiments, where we analyze the accuracy, speed, and interpretability of our approach.

Notational conventions Throughout this paper, we use lower case boldface letters to denote vectors and upper case boldface letters to denote matrices. The i_{th} entry of a vector $\boldsymbol{x}$ is denoted x_i, and the entry in the i_{th} row and j_{th} column of a matrix $\boldsymbol{A}$ is denoted a_{ij}. Division between two vectors is always assumed to be entry-wise. We use $\boldsymbol{e}$ to denote the vector of all ones and $\boldsymbol{0}$ to denote the vector of zeros. We use $x(\cdot)$ to denote a real-valued function, and $\boldsymbol{x}(\cdot)$ to denote a vector whose entries are real-valued functions. We use $\boldsymbol{x}$ instead of $\boldsymbol{x}(\cdot)$ when it is clear from the context that $\boldsymbol{x}$ is referring to a vector of functions.

2.2 Background

In this section, we first define the optimal control problem for MFQNETs. Then, we review its key theoretical properties and provide a brief overview of OCT-H.

2.2.1 Optimal Control of MFQNETs

Consider a queueing network with m servers and n job classes. Each job class $i \in [n]$ is processed by a single server $s(i) \in [m]$ with service rate μ_i. After jobs of class i are processed, they either leave the system or change to a different class in a deterministic manner. Jobs may arrive from either another server or from outside the system with external arrival rate λ_i. If there is no external arrival for class i, then $\lambda_i = 0$. The cost per unit time for holding a job of class i is denoted c_i.

For each class $i \in [n]$, the control variable $u_i(t)$ denotes the fraction of effort the server $s(i)$ spends processing class i jobs at time t. The state variable $x_i(t)$ is the number of jobs of class i at time t. The dynamics of the system can be expressed using a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, where $a_{ii} = -\mu_i$ and $a_{ij} = \mu_j$ if class i receives arrivals from class j, $j \neq i$. The rest of the entries of $\boldsymbol{A}$ are zero. Then, the dynamics of the system is

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{u}(t) + \boldsymbol{\lambda}.$$

In addition, the sum of the control variables for all classes that are processed at the same server should be less than or equal to one. This constraint can be expressed as

$$\boldsymbol{D}\boldsymbol{u}(t) \leq \boldsymbol{e},$$

where $\boldsymbol{D} \in \{0,1\}^{m \times n}$ is a binary matrix with $d_{ij} = 1$ if $s(j) = i$ and $d_{ij} = 0$, otherwise.

The MFQNET control problem aims to find a control $\boldsymbol{u}$ that minimizes the total holding cost of the jobs in the system over the time interval $[0, T]$. We define the MFQNET control

problem with an initial state $\boldsymbol{x}_0$ as the following:

$$\begin{aligned} \min_{\boldsymbol{u}(\cdot),\boldsymbol{x}(\cdot)} \quad & \int_0^T \boldsymbol{c}^\top \boldsymbol{x}(t)\, dt \\ s.t. \quad & \dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{u}(t) + \boldsymbol{\lambda}, \quad \forall t \in [0,T], \\ & \boldsymbol{D}\boldsymbol{u}(t) \leq \boldsymbol{e}, \quad \forall t \in [0,T], \\ & \boldsymbol{u}(t), \boldsymbol{x}(t) \geq \boldsymbol{0}, \quad \forall t \in [0,T], \\ & \boldsymbol{x}(0) = \boldsymbol{x}_0. \end{aligned} \tag{2.1}$$

We use $\boldsymbol{u}^*_{\boldsymbol{x}_0}$ and $\boldsymbol{x}^*_{\boldsymbol{x}_0}$ to denote the optimal control and the associated state trajectory of problem (2.1) with the initial state $\boldsymbol{x}_0$. We use $\boldsymbol{u}^*$ and $\boldsymbol{x}^*$ to denote the optimal control and the associated state trajectory of general MFQNET control problems when the initial state is not specified.

We define the vector load $-\boldsymbol{D}\boldsymbol{A}^{-1}\boldsymbol{\lambda} \in \mathbb{R}^m$, and assume that all the entries of the workload vector are strictly smaller than 1 for stability. We further assume T is large enough, so that the system can be emptied by time T [79]. Under this setting, identifying the optimal initial control $\boldsymbol{u}^*_{\boldsymbol{x}_0}(0)$ for any initial state $\boldsymbol{x}_0$ is equivalent to identifying the optimal stationary feedback control $\boldsymbol{u}^*(t)$ for any state $\boldsymbol{x}(t)$.

For each job class $i \in [n]$, we define the depletion time $T_i = \inf\{t \in (0,T] : x_i^*(t) = 0\}$ under an optimal state trajectory $\boldsymbol{x}^*$, assuming that $x_i^*(0) > 0$. We define the value function $V(\boldsymbol{x}_0)$ as the optimal objective value of Problem (2.1) associated with the initial state $\boldsymbol{x}_0$.

2.2.2 Properties of MFQNET Control problems

We present several theoretical properties of MFQNET control problems, which will be used to derive our main results. The Pontryagin Maximum Principle [89], [100] provides necessary optimality conditions for general optimal control problems. Due to the non-negativity constraints on the state variable in Problem (2.1), the conditions that we provide are tailored for the optimal control problems with pure state constraints.

We define the Hamiltonian of Problem (2.1) as

$$H(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{y}, t) = \boldsymbol{c}^\top \boldsymbol{x}(t) + \boldsymbol{y}(t)^\top [\boldsymbol{A}\boldsymbol{u}(t) + \boldsymbol{\lambda}],$$

where $\boldsymbol{y}(t)$ is known as the costate variable.

Lemma 1 (Pontryagin Maximum Principle [89], [100])**.** *If the feasible control $\boldsymbol{u}^*$ and the state trajectory $\boldsymbol{x}^*$ is optimal for Problem (2.1), there exists $\boldsymbol{y}(t)$ for any $t \in [0,T]$ that satisfies the following conditions.*

(a) *$H(\boldsymbol{x}^*, \boldsymbol{u}^*, \boldsymbol{y}, t) \leq H(\boldsymbol{x}^*, \boldsymbol{u}, \boldsymbol{y}, t)$ for all $\boldsymbol{u}(t)$ satisfying $\boldsymbol{u}(t) \geq \boldsymbol{0}$, $\boldsymbol{D}\boldsymbol{u}(t) \leq \boldsymbol{e}$.*

(b) *Whenever $\boldsymbol{u}^*(t)$ is continuous, $\dot{\boldsymbol{y}}(t) = -\boldsymbol{c} + \boldsymbol{\pi}(t)$, where $\boldsymbol{\pi}(t) \geq \boldsymbol{0}, \boldsymbol{\pi}(t)^\top \boldsymbol{x}^*(t) = 0$.*

(c) *$\boldsymbol{y}(T) = \boldsymbol{0}$.*

Proof. See [100]. □

The following two lemmas describe the property of the costate variable in Lemma 1 and the optimal feedback policy, respectively.

Lemma 2 ([6]). *$\boldsymbol{y}(t)$ is piecewise linear and continuous function of t. Furthermore, slope changes can only occur at the depletion times $\{T_1, \ldots, T_n\}$.*

Proof. See [6]. □

Lemma 3 ([15]). *Consider two initial states $\boldsymbol{x}_0$ and $\alpha\boldsymbol{x}_0$, where α is some positive scalar. Then, $\boldsymbol{u}^*_{\boldsymbol{x}_0}(0) = \boldsymbol{u}^*_{\alpha\boldsymbol{x}_0}(0)$*

Proof. See [15]. □

2.2.3 Optimal Classification Trees with Hyperplane Splits

Optimal Classification Trees (OCT) is an algorithm to learn near-optimal decision trees for classification tasks. Classification and Regression Trees (CART) [34], an earlier algorithm to learn decision trees for prediction tasks, learns a decision tree in a greedy manner using recursive partitioning of the feature space at each child node. However, OCT aims to learn a globally optimal decision tree using mixed-integer optimization and local heuristics.

Similar to CART and other classification algorithms, OCT takes N data inputs $\{(\boldsymbol{\theta}_i, z_i)\}_{i=1}^N$, where $\boldsymbol{\theta}_i$ is the feature vector and z_i is the label for the i_{th} data point. Given this data set, OCT learns a decision tree that uses a single feature for the split at each node and assigns a label to each node of the tree. Given a new data point $\boldsymbol{\theta}_0$, it traverses the decision tree until it reaches a leaf node. The prediction of the tree for $\boldsymbol{\theta}_0$ is the label assigned to the leaf node.

OCT-H, a generalization of OCT, can use an arbitrary linear combination of the features for splits at the nodes. This means that OCT-H can use general hyperplanes for splits, whereas OCT is confined to use hyperplanes that are perpendicular to the axes in the feature space. Essentially, OCT-H partitions the feature space with hyperplanes, and assigns a prediction to each region. This observation is the key to our work to solve Problem (2.1) using OCT-H. Compared to OCT, OCT-H generally shows higher prediction accuracy and learns shallower trees.

In OCT-H, it is possible to limit the number of features that can be used for splits, which can result in a more interpretable tree. This version of OCT-H is denoted as OCT-H with sparsity throughout the remainder of the paper. We simply use OCT-H to denote regular OCT-H, where the entire features can be used. For a more detailed explanation on OCT and OCT-H, we refer readers to [20], [21].

2.3 OCT-H for the Optimal Control of MFQNETs

In this section, we first prove that OCT-H can learn an optimal policy of Problem (2.1). Based on this result, we then proceed to develop an efficient algorithm to learn an optimal policy of Problem (2.1) using OCT-H.

2.3.1 Theoretical Results

The following Lemma 4 will be used to prove our main theorem.

Lemma 4. *The value of $\boldsymbol{y}(0)$ can be expressed as a linear function of $(T_1, \ldots, T_n)$.*

Proof. We fix an index $i \in [n]$ and prove the stated property for $y_i(0)$. By Lemma 2, $y_i(t)$ is piecewise linear and continuous, with potential breakpoints at the depletion times $\{T_1, \ldots, T_n\}$. Although the exact ordering of the depletion times is unknown, the following argument applies to any ordering. Hence, assume a fixed ordering. By working backwards in time from $t = T$ (where $y_i(T) = 0$ by Lemma 1), we can determine the value of $y(t)$ at the breakpoints $\{T_1, \ldots, T_n\}$, starting from the breakpoint that is closest to T. At any breakpoint, the value of $y_i(t)$ can be expressed as a linear combination of $T_1, \ldots, T_n$ that are greater than or equal to t. Once the value of $y_i(t)$ at a breakpoint is determined, the value of $y_i(t)$ at the next earlier breakpoint can be determined, as we know that $y_i(t)$ is piecewise linear and continuous. We recursively follow this procedure until $t = 0$, where the value of $y_i(0)$ can be expressed as a linear combination of $(T_1, \ldots, T_n)$. □

The following theorem is our main theoretical result that generalizes the results by [7] to general MFQNETs.

Theorem 1. *For Problem (2.1), there exists a piecewise constant optimal policy, with segments separated by hyperplanes passing through the origin.*

Proof. Our proof is based upon the algorithm by [6] to solve Problem (2.1) with any given initial state. We prove that the solution this algorithm finds is a a piecewise constant optimal policy, with segments separated by hyperplanes passing through the origin.

By condition (a) of Lemma 1, the optimal control at each time t is decided by the priority index $r_i(t)$ defined for each job class $i \in [n]$, where $r_i(t) = [\boldsymbol{y}(t)^\top \boldsymbol{A}]_i$. Lemma 2 indicates that $r_i(t)$ is a continuous, piecewise linear function and its slope can only change at $\{T_1, \ldots, T_n\}$. At each server, the optimal policy is to put maximum effort to the job class with the smallest priority index, and put zero effort to the rest, without violating the non-negativity constraint on the state variables. When all the job classes at the server have positive priority indices, the optimal policy is to idle. This case can be captured by considering idling as a job class with the constant priority index 0. Hence, Lemma 1 implies that as long as the rank of the priority indices does not change, the optimal control is a constant vector.

The condition under which a server transfers effort from one class to another is defined by the equalities of the form $r_j(t) = r_k(t)$, given that class j jobs and class k jobs are processed by the same server. If this equality holds, then it is indifferent whether the server prioritizes class j or k. If this equality becomes inequality, then it would be beneficial to prioritize one class over the other. This description indicates that the switching between job classes are defined by the equalities between the priority indices.

We derive the condition that the priority is switched from class j jobs to class k jobs, starting from an initial state $\boldsymbol{x}_0$. Depending on the parameters $\boldsymbol{c}$ and $\boldsymbol{A}$, certain switches might not be always possible. Furthermore, the order of the depletion times $\{T_1, \ldots, T_n\}$ and the future switches associated with the trajectory should be adequately decided as well (Specific examples of how $\boldsymbol{A}$, $\boldsymbol{c}$ and the order of $\{T_1, \ldots, T_n\}$ can make a switch possible or

not are given in [6], [7]). We assume that $\boldsymbol{A}$, $\boldsymbol{c}$, the order of $\{T_1, \ldots, T_n\}$ and the switches associated with the trajectory are appropriately fixed. The specific order of $\{T_1, \ldots, T_n\}$ leads to a collection of equalities between the indices $r_i(t)$ during the entire trajectory, and completely determines the optimal solution of the problem [6], [7].

Without loss of generality, we assume that the switch from class j to k happens at $t = 0$. The switching condition that we would like to derive is then $r_j(0) = r_k(0)$, where $r_j(0)$ and $r_k(0)$ are both linear functions of $(T_1, \ldots, T_n)$ due to Lemma 4. We now prove that $(T_1, \ldots, T_n)$ is a linear function of $\boldsymbol{x}_0$, which verifies that $r_j(0) = r_k(0)$ represents a hyperplane passing through the origin in the state space.

By definition, T_i can be computed from the equation of the form $\int_0^{b_1} [\boldsymbol{A}\boldsymbol{u}(t) + \boldsymbol{\lambda}]_i \, dt + \cdots + \int_{b_q}^{T_i} [\boldsymbol{A}\boldsymbol{u}(t) + \boldsymbol{\lambda}]_i \, dt = -x_i$, where $b_l, l \in [q]$, represents a breakpoint in $[\boldsymbol{A}\boldsymbol{u}(t) + \boldsymbol{\lambda}]_i$. The control $[\boldsymbol{A}\boldsymbol{u}(t) + \boldsymbol{\lambda}]_i$ is constant between the breakpoints, and the breakpoints b_i are always the intersections between two indices. Any time of intersection between two indices can be expressed as a linear function of the vector $(T_1, \ldots, T_n)$ due to Lemma 4. Hence, the above equation leads to an equality between x_i and a linear function of $(T_1, \ldots, T_n)$. Likewise, the collection of equalities that we have all lead to equalities between $\boldsymbol{x}_0$ and linear functions of $(T_1, \ldots, T_n)$. Rearranging these equalities leads to the expression of $(T_1, \ldots, T_n)$ as a linear function of $\boldsymbol{x}_0$. □

The proof above demonstrates that the state space is separated by hyperplanes, according to the relative ordering of the priority indices of each job class. Within each region, a constant control vector is optimal. These hyperplanes are referred to as switching curves [79], [100].

The following Corollary 1 is the building block to develop a learning algorithm in Section 2.3.2.

Corollary 1. *OCT-H can learn an optimal policy of Problem* (2.1).

Proof. By Theorem 1, there exist switching curves that are hyperplanes passing through the origin. As described in Section 2.2.3, OCT-H learns a decision tree that partitions the feature space with hyperplanes, and assigns a label to each region. Hence, it can naturally learn the switching curves and the optimal control at each region partitioned by the switching curves.

Another condition to consider is whether $x_i(t) = 0$ for some class $i \in [n]$. If $x_i = 0$, then server splitting might occur to satisfy the non-negativity constraint on the state vector. We first note that the condition $x_i = 0$ is also a hyperplane in the state space passing through the origin.

In general, decision trees are confined to use inequalities for node splits. In our context, however, equality conditions such as $x_i(t) = 0$ can be learned as the condition $x_i \leq 0$. Since the state vector is always non-negative, these two conditions are equivalent for Problem (2.1). Hence, OCT-H can learn the optimal policy of Problem (2.1) both in the interior and the boundary of the state space. □

The following Theorem 2, along with Lemma 3 will be used in Section 2.3.2 to develop a more efficient learning algorithm.

Theorem 2. *Consider a pair of optimal control and the associated state trajectory*

$$\{(\boldsymbol{u}^*_{\boldsymbol{x}_0}(t), \boldsymbol{x}^*_{\boldsymbol{x}_0}(t)) : t \in [0, T]\}$$

and a positive scalar α*. Then,* $\{(\boldsymbol{u}^*_{\boldsymbol{x}_0}(\frac{t}{\alpha}), \alpha \boldsymbol{x}^*_{\boldsymbol{x}_0}(\frac{t}{\alpha})) : t \in [0, \alpha T]\}$ *is optimal for Problem* (2.1) *with the initial state* $\alpha \boldsymbol{x}_0$*.*

Proof. The proof of this theorem follows from the proof of Theorem 3 in [15]. By Theorem 3 in [15], we know $V(\alpha \boldsymbol{x}_0) = \alpha^2 V(\boldsymbol{x}_0)$ and also that $\{(\boldsymbol{u}^*_{\boldsymbol{x}_0}(\frac{t}{\alpha}), \alpha \boldsymbol{x}^*_{\boldsymbol{x}_0}(\frac{t}{\alpha})) : t \in [0, \alpha T]\}$ is feasible. The objective cost associated with the pair $\{(\boldsymbol{u}^*_{\boldsymbol{x}_0}(\frac{t}{\alpha}), \alpha \boldsymbol{x}^*_{\boldsymbol{x}_0}(\frac{t}{\alpha})) : t \in [0, \alpha T]\}$ is

$$\alpha \int_0^{\alpha T} \boldsymbol{c}^\top \boldsymbol{x}^*_{\boldsymbol{x}_0}(\frac{t}{\alpha})\, dt = \alpha^2 \int_0^{T} \boldsymbol{c}^\top \boldsymbol{x}^*_{\boldsymbol{x}_0}(t)\, dt = \alpha^2 V(\boldsymbol{x}_0).$$

As this solution achieves the optimal objective cost and is also feasible, it is optimal. □

2.3.2 Algorithm

We present an algorithm that utilizes OCT-H to learn an optimal policy for Problem (2.1). To ensure a more comprehensive and efficient learning process, we discuss several key considerations that have been taken into account while developing the algorithm.

Given Problem (2.1) with the initial state $\boldsymbol{x}_0$, we can solve it to optimality using the algorithm proposed by [101]. Once we solve it, we obtain the optimal control $\boldsymbol{u}^*_{\boldsymbol{x}_0}(t)$ and the optimal state trajectory $\boldsymbol{x}^*_{\boldsymbol{x}_0}(t)$ for the entire time interval $t \in [0, T]$. We choose $N \in \mathbb{N}$ elements $t_1, \ldots, t_N$ from the interval $[0, T]$, and extract the corresponding state values $\boldsymbol{x}^*(t_1), \ldots, \boldsymbol{x}^*(t_N)$ and the control values $\boldsymbol{u}^*(t_1), \ldots, \boldsymbol{u}^*(t_N)$. The training data that we obtain from this procedure is $\{(\boldsymbol{x}^*_{\boldsymbol{x}_0}(t_i), \boldsymbol{u}^*_{\boldsymbol{x}_0}(t_i))\}_{i=1}^N$. As in the usual supervised learning literature, $\boldsymbol{x}^*_{\boldsymbol{x}_0}(t_i)$ is the feature vector and $\boldsymbol{u}^*_{\boldsymbol{x}_0}(t_i)$ is the target for the i_{th} data point.

To ensure a comprehensive coverage of the state space, we generate multiple initial states and solve the associated Problem (2.1) for each initial state. By considering multiple instances with different initial states, we can obtain a more diverse set of state trajectories. Instead of arbitrarily generating the initial states, we develop a more systematic approach. Assuming that there are n job classes, there are $\binom{n}{1} + \cdots + \binom{n}{n} = 2^n - 1$ possible cases of which classes among n are non-empty (excluding the trivial case that the entire system is empty). We let $\mathcal{S} = \{s_1, s_2, \ldots, s_{2^n-1}\}$ be the set of such cases, where each element $s_i, i \in [2^n - 1]$, represents a set of non-empty classes. For example, if $n = 2$, then $\mathcal{S} = \Big\{\{1\}, \{2\}, \{1, 2\}\Big\}$. For each $s \in \mathcal{S}$, we generate values for the non-zero entries of the initial state specified in s and fix the remaining entries to zero. This systematic approach ensures that the training data covers the state space seamlessly, including both the interior and the boundary regions.

Another consideration is that as n increases, the number of hyperplanes required to describe the optimal policy can get prohibitively large. To address this issue, we propose training multiple decision trees, if necessary. Each data point in the training set corresponds to an element of $\mathcal{S}$, depending on which entries of the state vector are non-zero. Hence, once we define a partition of $\mathcal{S}$, this partition can also be used to partition the training set. Then, we train a decision tree for each partition of the training set. For example, if we define a partition of the set $\mathcal{S} = \Big\{\{1\}, \{2\}, \{1, 2\}\Big\}$ to be $\mathcal{P} = \Bigg\{\Big\{\{1\}, \{2\}\Big\}, \Big\{\{1, 2\}\Big\}\Bigg\}$, we train two decision trees. The first decision tree is trained using the state vectors where

either the first or the second entry is zero. The second decision tree is trained using the state vectors that are strictly positive. Essentially, we are dividing the state space into multiple regions and learning the optimal policy for each region. This approach allows us to distribute the learning process across multiple decision trees, reducing the computational burden and enabling efficient training even when dealing with a large number of job classes.

Finally, we use Lemma 3 and Theorem 2 for a more efficient data generation. According to Theorem 2, solving Problem (2.1) with the initial state $\boldsymbol{x}_0$ and solving it again with $\alpha\boldsymbol{x}_0$ would be redundant. This observation allows us to streamline the data generation process. Instead of generating initial states arbitrarily, we sample them uniformly at random from the unit sphere in the non-negative orthant. This approach ensures that we cover a diverse range of initial states while avoiding unnecessary repetitions. Furthermore, Lemma 3 suggests that we can augment the training data $\{(\boldsymbol{x}^*_{\boldsymbol{x}_0}(t_i), \boldsymbol{u}^*_{\boldsymbol{x}_0}(t_i))\}_{i=1}^N$ by including additional data points $\{(\alpha\boldsymbol{x}^*_{\boldsymbol{x}_0}(t_i), \boldsymbol{u}^*_{\boldsymbol{x}_0}(t_i))\}_{i=1}^N$, possibly multiple times with varying α values.

Algorithm 1 outlines the entire procedure more rigorously. We let $\mathcal{A}$ denote the set of α that we use to augment data. We let $\mathcal{P}$ denote the partition of $\mathcal{S}$. We use M to denote the number of initial states we sample for each element in $\mathcal{S}$. We use $\boldsymbol{x}_{[s]}$ to denote the entries of $\boldsymbol{x}$ in s. For a set $K = \{(\boldsymbol{x}^*_{\boldsymbol{x}_0}(t_i), \boldsymbol{u}^*_{\boldsymbol{x}_0}(t_i))\}_{i=1}^N$, we use αK to denote $\{(\alpha\boldsymbol{x}^*_{\boldsymbol{x}_0}(t_i), \boldsymbol{u}^*_{\boldsymbol{x}_0}(t_i))\}_{i=1}^N$. Without loss of generality, we assume that the order of the cells in the partition $\mathcal{P}$ is fixed and $\mathcal{P}_{[j]}$ is the j_{th} cell of $\mathcal{P}$.

2.3.3 Example

We provide two small examples to illustrate Algorithm 1. For both examples, the closed-form expressions of the optimal policies are already known. We compare the policy learned by Algorithm 1 with the closed-form optimal policy to demonstrate that it can learn near-optimal policies. The first example is to demonstrate that Algorithm 1 can learn the optimal switching curve in the interior of the state space, and the second example is to demonstrate that it can learn the optimal server splitting policy when some job classes are empty.

The first example is the criss-cross network considered by [57]. The criss-cross network is composed of three classes and two servers. Server 1 processes Class 1 and 2 jobs, and Server 2 processes Class 3 jobs. Class 1 and 2 jobs take external arrivals. After Class 1 jobs are processed at Server 1, they become Class 3 jobs and move to Server 2. After Class 2 and 3 jobs are processed, they leave the system. Its graphical representation is given in Figure 2.1. It is clear that $u_3^*(t) = 1$ as long as Server 2 is not empty. The problem is to choose which class to process at Server 1. For this example, we only demonstrate the case in which none of the classes are empty. In other words, we learn the optimal policy for the case $s = \{1, 2, 3\}$. We let $\boldsymbol{c} = \boldsymbol{e}$, $\lambda_1 = \lambda_2 = 0.5$, $\mu_1 = 1.5$, $\mu_2 = 1$ and $\mu_3 = 2$. Under this set of parameters, the switching curve and the corresponding optimal policy of this network derived by [7] are given in Table 2.1. The closed-form expression of the switching curve is $x_1(t) = 6x_3(t)$. Other parameters we used for Algorithm 1 are $N = 1, t_1 = 0, \mathcal{A} = \{0.5, 1.5\}, \mathcal{P} = \left\{\{\{s_1, s_2, s_3\}\}, \dots\right\}$ and $M = 1000$.

Figure 2.2 displays the decision tree learned by OCT-H, where each node contains the prediction made on that node. The decision tree that OCT-H learned predicts $\boldsymbol{u}^*(t) = (0, 1, 1)$ if $x_1(t) \leq 5.93x_3(t) + 0.01$ and predicts $\boldsymbol{u}^*(t) = (1, 0, 1)$ if $x_1(t) \geq 5.93x_3(t) + 0.01$. This

Algorithm 1: OCT-H for MFQNET control.

Input: $\boldsymbol{c}, \boldsymbol{A}, \boldsymbol{\lambda}, \boldsymbol{D}, \{t_1, \ldots, t_N\}, \mathcal{A}, \mathcal{S}, \mathcal{P}, M$
Output: $|\mathcal{P}|$ classification trees with hyperplane splits.
Initialization: $K_{\mathcal{S}}, K_1, \ldots, K_{|\mathcal{P}|} \leftarrow \emptyset$

1. Data Generation
for $s \in \mathcal{S}$ **do**
  $j \leftarrow 1$
  while $j \leq M$ **do**
    $\boldsymbol{x}_0 \leftarrow \mathbf{0} \in \mathbb{R}^n$
    Sample a positive vector $\hat{\boldsymbol{x}}$ from the $|s|$ dimensional unit sphere.
    $\boldsymbol{x}_{0[s]} \leftarrow \hat{\boldsymbol{x}}$
    Solve Problem (2.1) with the initial state $\boldsymbol{x}_0$.
    $K_{\mathcal{S}} \leftarrow K_{\mathcal{S}} \cup \{(\boldsymbol{x}^*_{\boldsymbol{x}_0}(t_i), \boldsymbol{u}^*_{\boldsymbol{x}_0}(t_i))\}_{i=1}^N$
    $j \leftarrow j + 1$
for $(\boldsymbol{x}, \boldsymbol{u}) \in K_{\mathcal{S}}$ **do**
  $\hat{s} \leftarrow \{i \in [n] : x_i > 0\}$
  for $j \in [|\mathcal{P}|]$ **do**
    for $s \in \mathcal{P}_{[j]}$ **do**
      if $s = \hat{s}$ **then**
        $K_j \leftarrow K_j \cup (\boldsymbol{x}, \boldsymbol{u})$

2. Data Augmentation
for $K \in \{K_1, \ldots, K_{|\mathcal{P}|}\}$ **do**
  for $\alpha \in \mathcal{A}$ **do**
    $K \leftarrow K \cup \alpha K$

3. Training
for $K \in \{K_1, \ldots, K_{|\mathcal{P}|}\}$ **do**
  Use OCT-H to train a classification tree on K.

Conditions	$\boldsymbol{u}^*(t)$
$\frac{x_1(t)}{x_3(t)} \geq \frac{c_2\mu_1}{c_1\mu_1 - c_3\mu_3} \times \frac{\mu_1 - \lambda_1}{\mu_2 - \mu_1}$	$(1, 0, 1)$
$\frac{x_1(t)}{x_3(t)} \leq \frac{c_2\mu_1}{c_1\mu_1 - c_3\mu_3} \times \frac{\mu_1 - \lambda_1}{\mu_2 - \mu_1}$	$(0, 1, 1)$

Table 2.1: Optimal policy for the criss-cross network when $\boldsymbol{x}(t) > \mathbf{0}$.

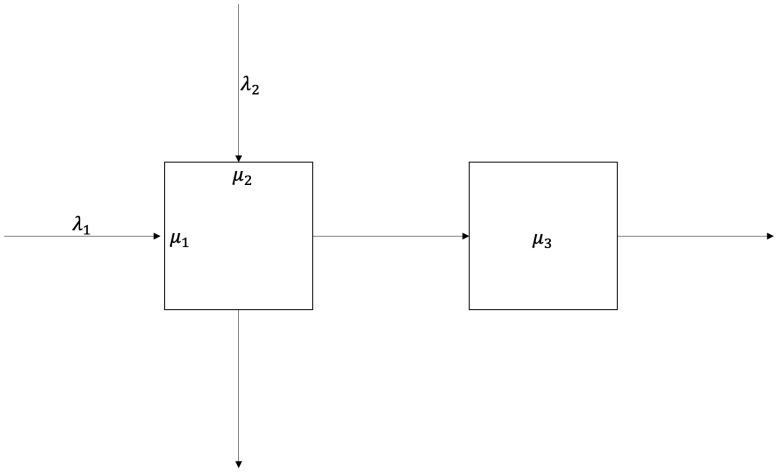

Figure 2.1: Criss-cross network.

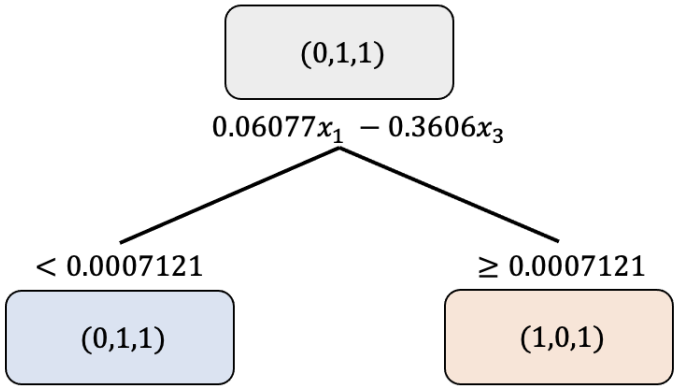

Figure 2.2: The decision tree learned by OCT-H for the criss-cross network.

closely resembles the optimal policy, exhibiting only minor numerical differences.

The second example is the Rybko-Stolyar network studied in [96]. This network is composed of four classes and two servers. Server 1 processes Class 1 and 4, and Server 2 processes Class 2 and 3 jobs. Class 1 and 3 jobs take external arrivals. After Class 1 jobs are processed, they become Class 2 job and move to Server 2. After Class 3 jobs are processed, they become Class 4 jobs and move to Server 1. After Class 2 and 4 jobs are processed, they exit the system. Its graphical representation is given in Figure 2.3. We let $\boldsymbol{c} = \boldsymbol{e}$, $\lambda_1 = \lambda_3 = 1$, $\mu_1 = \mu_3 = 6$ and $\mu_2 = \mu_4 = 1.5$. Under this set of parameters, the optimal policy is to prioritize Class 2 and 4 jobs unless either one of them is empty. If any one of them is empty, server splitting occurs. For this example, we train a single decision tree to learn the optimal policy that covers the entire state space. The parameters we used for Algorithm 1 are $N = 1, t_1 = 0, \mathcal{A} = \{0.5, 1.5\}, \mathcal{P} = \{\mathcal{S}\}$ and $M = 1000$.

Figure 2.4 displays the decision tree learned by OCT-H. Since decision trees are confined to use inequalities for node splits, we can observe that the condition $x_i = 0$ for some $i \in [4]$ is learned as $x_i \leq \epsilon$ for a number ϵ with small absolute value. As the state vectors are always

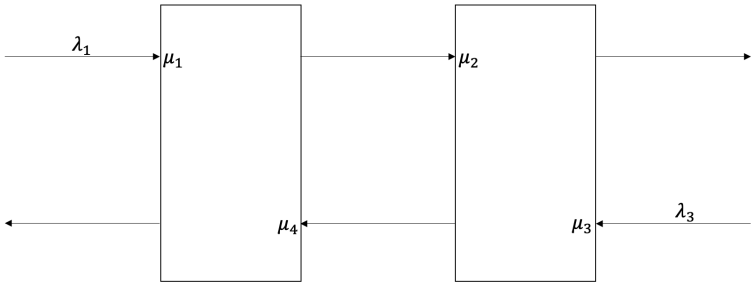

Figure 2.3: Rybko-Stolyar network.

non-negative, these two conditions are effectively equivalent. Additionally, when both Class 2 and 4 are non-empty, the decision tree prioritizes them. When either one of them is empty, server splitting occurs. This policy aligns with the description provided earlier. To assess the quality of this policy when server splitting occurs, we generated a test set following the same procedure as the training set but with $M = 200, \mathcal{A} = \{5\}$. This guarantees that the test set consists of state-control pairs that OCT-H has not seen during training. We then computed the classification accuracy of the learned policy on this test set. The classification accuracy was 100%, implying that OCT-H learned a high-quality policy that is empirically optimal. Furthermore, we compared the optimal objective cost with the cost achieved by applying the learned policy to the network. We generated 100 random initial points and applied the decision tree to each instance. When implementing the policy, we discretized the time steps to simulate continuous dynamics and compute integral values. After calculating the objective cost, we subtracted the optimal objective cost and divided the resulting value by the optimal objective cost to compute the suboptimality. We observed that for all 100 instances, the suboptimality was smaller than 0.0001.

2.4 Computational Experiments

This section presents the findings of computational experiments conducted on MFQNETs with varying sizes. We analyze the accuracy of the policy learned by Algorithm 1 and compare its online application speed with that of the algorithm by [101]. In addition, we provide insights on the optimal policy of MFQNET control problems by presenting some of the actual decision trees. We also apply OCT-H with sparsity on several MFQNET problems and analyze the impact of sparsity on the performance and the resulting decision tree. The networks in this section are taken from [31] and [101].

2.4.1 Experiment Setting

We consider a reentrant network with m servers and $3m$ classes of jobs. Each Server $i \in [m]$ processes jobs of Classes $3(i-1)+1, 3(i-1)+2$ and $3(i-1)+3$. Only Class 1 jobs take external arrivals with the arrival rate λ_1. Class $3(i-1)+1$ jobs become Class $3i+1$ until they become Class $3(m-1)+1$. After Class $3(m-1)+1$ jobs are processed, they change to Class 2 and enter Server 1. Class $3(i-1)+2$ jobs become Class $3i+2$ jobs until they

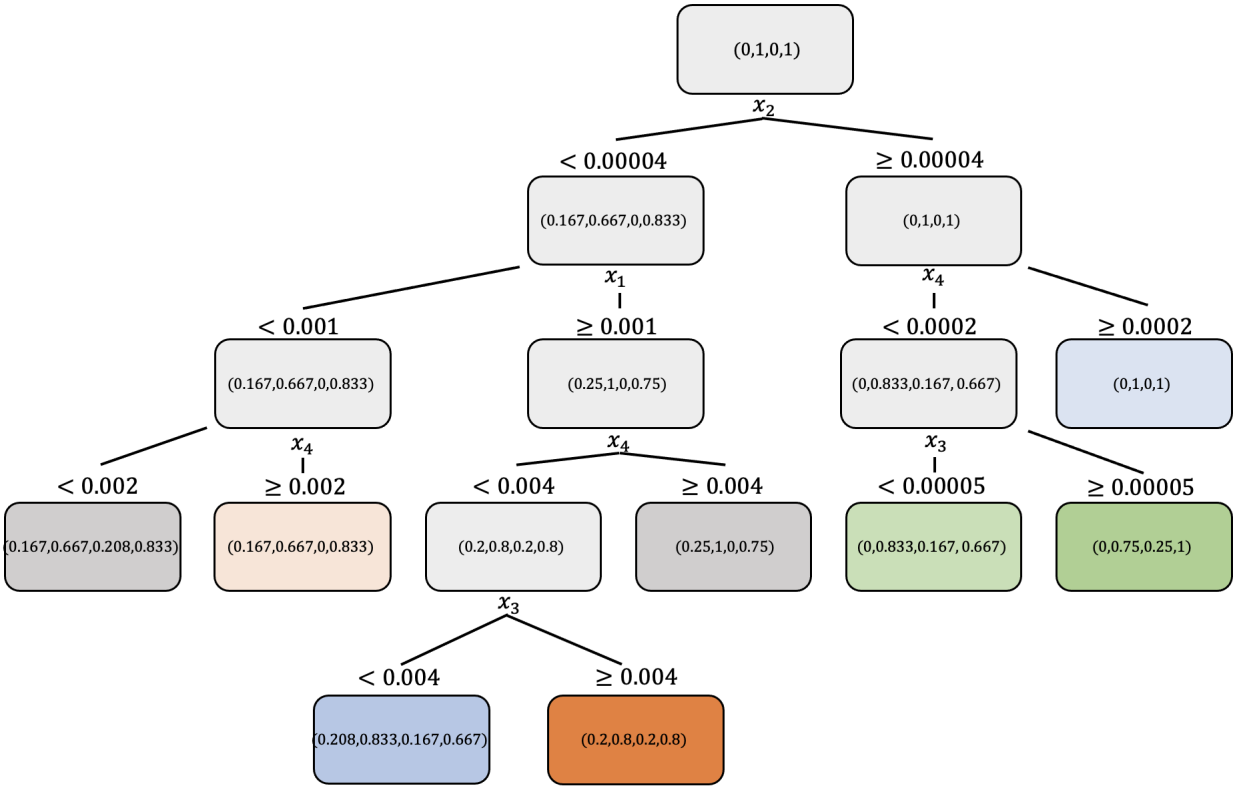

Figure 2.4: The decision tree learned by OCT-H for the Rybko-Stolyar network.

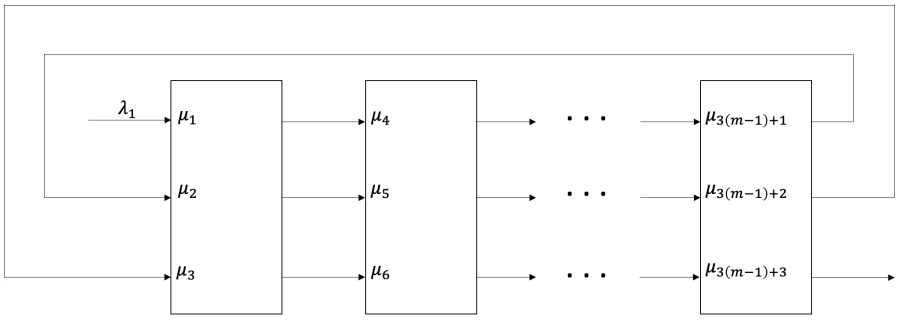

Figure 2.5: Reentrant network.

become Class $3(m-1)+2$. After Class $3(m-1)+2$ jobs are processed, they change to Class 3 to enter Server 1. Class $3(i-1)+3$ jobs become $3i+3$ jobs, until they become Class $3m$ and exit the system after processed. We provide a graphical representation in Figure 2.5. The parameters $\boldsymbol{\lambda}, \boldsymbol{\mu}, \boldsymbol{c}$ are randomly generated using the software by [101].

We test Algorithm 1 on the reentrant networks with varying m. We treat all $2^{3m}-1$ cases of non-zero entries separately. Using the formalism in Section 2.3.2, we let $\mathcal{P} = \big\{\{s_1\}, \{s_2\}, \ldots, \{s_{2^{3m}-1}\}\big\}$. Instead of exhaustively demonstrating our approach on the entire cells of $\mathcal{P}$, we randomly choose three cells and learn the optimal policy for each cell. We always include the case where none of the classes are empty. After we fix some $s \in \mathcal{S}$, we generate a training set with $N = 1, t_1 = 0, \mathcal{A} = \{0.5, 0.75, 5\}$ and $M = 10000$. We generate a test set with $N = 1, t_1 = 0, \mathcal{A} = \{10\}$ and $M = 2000$. Again, this ensures that the test set consists of state-control pairs that are completely distinct from the training set. We report the classification accuracy of OCT-H on the test set. For each fluid control instance that is used to generate test set, we also measure the time it takes to solve the instance using the algorithm by [101], and divide it by the time it takes for the trained decision tree to make a prediction. We report the mean of the ratios rounded to the nearest integer as the relative speed-up of Algorithm 1.

Software for OCT-H is available at [60]. We tune the maximum depth of the tree by grid searching over the list [3,5,10]. Public implementation of the algorithm by [101] is available at https://github.com/IBM/SCLPsolver. When we use this implementation, we set the zero entries in $\boldsymbol{\lambda}$ to a small number 10^{-6} instead of 0, as we have observed that this results in better numerical stability. The experiments were executed on a MacBook Pro with 2.6 GHz Intel Core i7 CPU and 16GB of RAM, except for the training part. We trained decision trees on MIT Engaging Computing Cluster with Dell C6300, 2 socket Intel E5-2690v4 processor, 14 Cores per CPU and 128 GB RAM.

2.4.2 Speed and Accuracy

In Table 2.2, we report the results of numerical experiments, focusing on the speed and accuracy of Algorithm 1. In the second column, we report the number of non-zero entries of the state vector, denoted by $|s|$. In the third column, we report the number of distinct labels for the classification task, denoted by $|\{\boldsymbol{u}^*\}|$. In the rest of the columns we report m, the training time for OCT-H, the relative speed-up of Algorithm 1 and the out-of-sample

m	$\lvert s\rvert$	$\lvert\{\boldsymbol{u}^*\}\rvert$	Training Time (hours:minutes)	Speed-up	Accuracy (%)
	9	3	00:30	153	100
3	7	3	00:38	196	100
	5	3	00:48	125	100
	21	4	01:26	255	100
7	9	4	00:34	151	100
	7	4	00:34	245	100
	24	4	00:28	296	100
8	12	4	00:44	278	100
	7	4	00:23	313	100
	27	4	00:30	364	100
9	24	2	00:33	360	100
	7	2	00:40	343	100
	42	6	03:50	781	100
14	36	6	04:34	790	100
	7	4	04:20	661	100
	60	9	46:20	700	100
20	30	9	44:10	599	100
	7	9	40:24	628	100
	99	9	48:20	6014	100
33	51	9	47:30	994	100
	45	9	42:20	982	100

Table 2.2: Experiment results for the reentrant network.

classification error on the test set, rounded to the third decimal place.

Observations from Table 2.2

- Algorithm 1 achieves perfect accuracy regardless of the size of the network, the number of unique labels and the number of non-zero entries.
- The training time for OCT-H takes at most 48 hours in our experiment, suggesting that data generation and training might take hours to days in practice.
- Once a policy is learned, Algorithm 1 is significantly faster than the algorithm by [101], with a speed-up ranging from hundreds to thousands of times faster in our experiment. In general, this relative speed-up becomes even greater as the dimension of the state space gets higher.
- As the dimension of the state space gets higher, the number of distinct labels do not increase significantly. This observation suggests that even for high dimensional problems, the structure of the optimal policy might be simple enough to be learned by OCT-H with shallow decision trees.

2.4.3 Interpretability

We now provide the decision tree for a problem solved in Section 2.4.2 and develop insights on the structure of the learned policy. Although not all of the node splits have straightforward interpretations, we highlight a few splits that make intuitive sense.

Due to space concerns, we display the prediction targets in the tree figures as the list of job classes that are prioritized, rather than the optimal control vector $\boldsymbol{u}^*$ itself. The job classes that are prioritized receive effort 1, and the rest of the job classes receive effort 0. In addition, we assign a number to each node split and provide a separate table that contains information on the hyperplane for each split.

Furthermore, we introduce a vector $\boldsymbol{c}/\boldsymbol{\mu}$ that offers an interesting interpretation on the learned policy. This vector captures the relative cost of holding each job class in terms of their service rate. A higher value in this vector indicates that the corresponding job class poses a greater challenge to the fluid network controller.

In Figure 2.6, we provide the decision tree for the problem with $m = 7$, confined to strictly positive state vectors ($|s| = 21$). In Table 2.3, we provide the node split information associated with the decision tree. For this problem, the parameters rounded to the third decimal place are

$\boldsymbol{\mu} = (0.143, 0.253, 0.002, 0.287, 0.169, 0.278, 0.22, 0.11, 0.207, 0.216, 0.299, 0.004, 0.185, 0.205, 0.25, 0.268, 0.027, 0.028, 0.245, 0.168, 0.248)$,

$\boldsymbol{c} = (0.705, 0.235, 0.972, 0.968, 0.719, 0.107, 1.484, 1.395, 0.493, 0.746, 1.584, 1.512, 0.07, 0.892, 1.255, 0.305, 1.941, 1.496, 0.643, 1.021, 1.975)$.

After we compute $\boldsymbol{c}/\boldsymbol{\mu}$ and sort it in descending order, the resulting indices in the sorted order is

$$(3, 12, 17, 18, 8, 21, 7, 20, 11, 15, 1, 14, 5, 10, 4, 19, 9, 16, 2, 6, 13).$$

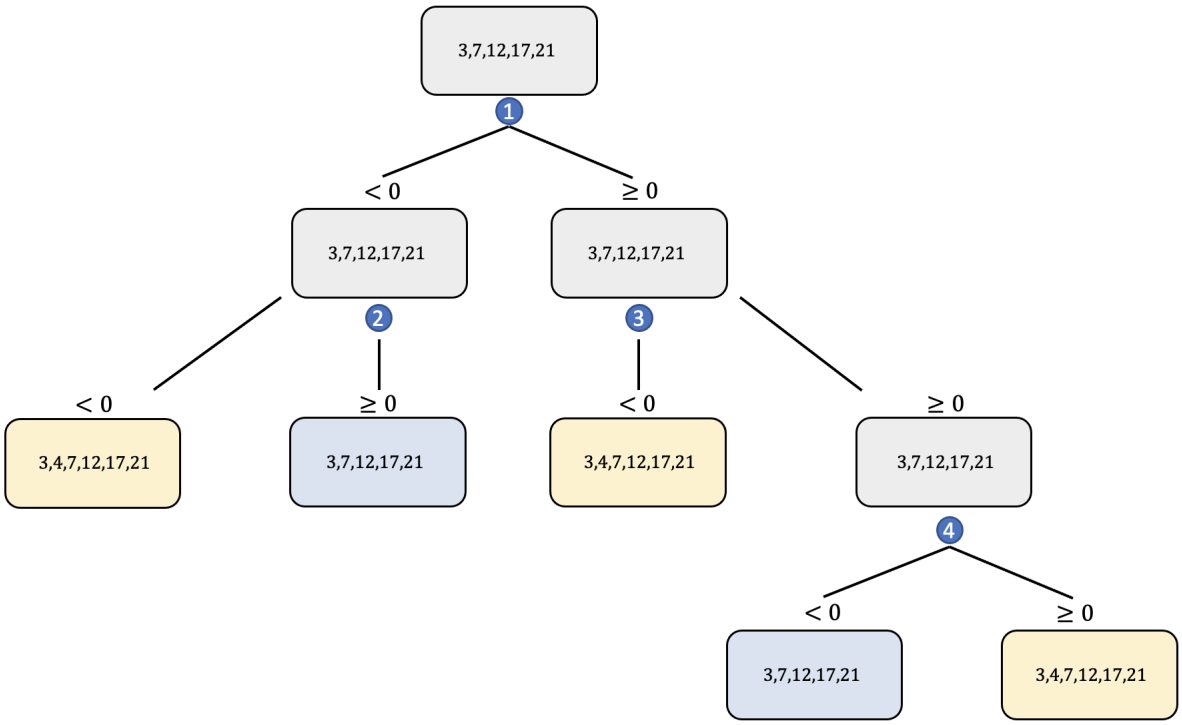

Figure 2.6: The decision tree learned by OCT-H for the reentrant network with $m = 7$ and $|s| = 21$.

Node split number	Hyperplane
1	$0.144x_1 - 0.002x_4 + 0.17x_7 - 0.05x_{10} - 0.0005$
2	$-0.096x_4 + 1.036x_7$
3	$-732.7x_1 - 34.84x_2 - 5.311x_3 - 1989.5x_4 + 19.1x_5$ $+14304.6x_7 + 4.175x_8 + 2939.7x_{10} + 33.72x_{11} + 62.25x_{12}$ $-6.67x_{13} - 12.42x_{14} + 1.382x_{15} + 8.84x_{16} - 2.794x_{19} + 12.92x_{20}$
4	$-2711.2x_3 + 0.5262x_{12} - 0.0001$

Table 2.3: Node split information on the decision tree in Figure 2.6.

Node split number	Hyperplane
1	$-0.316x_4 + 3.398x_7$
2	$-0.034x_1 - 0.103x_4 + 0.753x_7 + 0.152x_{10} + 0.0003x_{11}$

Table 2.4: Node split information on the decision tree in Figure 2.7.

Observations from Figure 2.6

- Class 3,7,12,17,21 jobs are always prioritized, regardless of the node. These job classes are often the highest ranking classes within their respective server in terms of the value in the vector $\boldsymbol{c}/\boldsymbol{\mu}$. The only exception is Class 8, as Class 8 is not prioritized even though it is ranked the highest in its server. This observation implies that the learned policy for this network is to drain the "toughest" job classes from the system first.

- The only difference in the nodes is whether to process Class 4 jobs or idle Server 2. See split 1 and 2, for example. If x_4 is relatively large compared to a linear combination of x_1, x_7, x_{10}, and if x_4 is again relatively large compared to x_7, the decision is to process Class 4 jobs instead of idling Server 2. However, after traversing the left edge in split 1, if x_7 turns out to be too large compared to x_4, Class 4 jobs are not processed. A possible explanation is that as Class 4 jobs become Class 7 after processed, it might be beneficial to idle Server 2 in case there are too many Class 7 jobs waiting in the queue.

- See split 3. If we focus on the terms associated with x_4 and x_7, again the decision is to process Class 4 jobs if x_4 is relatively large compared to x_7. The same interpretation as above can be applied to this decision.

2.4.4 OCT-H with sparsity

In this experiment, we apply OCT-H with sparsity instead of OCT-H in Algorithm 1 on a subset of the problems solved in Section 2.4.2. As mentioned in Section 2.2.3, OCT-H with sparsity often results in more interpretable decision trees compared to OCT-H. The purpose of this experiment is to analyze the price we have to pay in order to gain more interpretability. We vary the proportion of the total number of states allowed to be used for splits, denoted by sparsity parameter. We analyze how the sparsity parameter affects the training time and the classification accuracy on the test set. Table 2.5 provides the experiment results, where the same notations as Table 2.2 are used. We summarize our findings in the following.

Observations from Table 2.5

- In general, classification accuracy slightly degrades as sparsity parameter gets smaller. However, classification accuracy never gets below 94% in our experiment, suggesting that OCT-H with sparsity can still learn high-quality policies.

- Training becomes faster as the sparsity parameter gets smaller. For the sparsity parameter 0.25, training can be around 4 times faster than OCT-H.

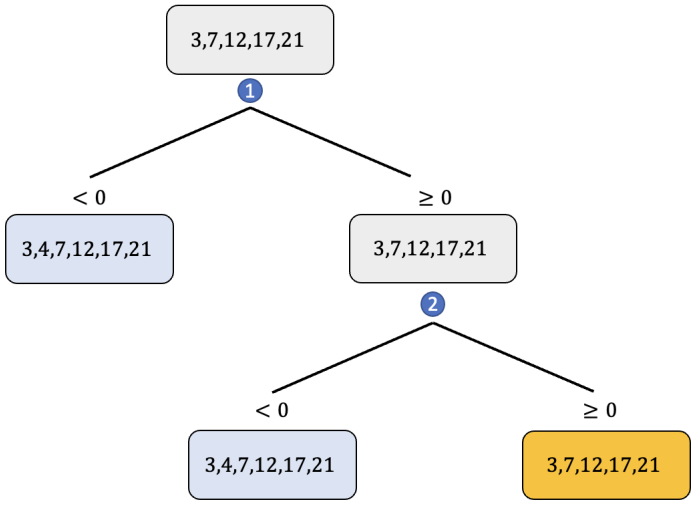

Figure 2.7: The decision tree learned by OCT-H with the sparsity parameter 0.25 for the reentrant network with $m = 7$ and $|s| = 21$.

m	$\|s\|$	Sparsity Parameter	Training Time (hours:minutes)	Accuracy (%)
7	21	0.5	00:48	99.8
		0.25	00:35	99.8
9	27	0.5	00:08	100
		0.25	00:07	100
14	42	0.5	02:52	98.3
		0.25	00:58	97
20	60	0.5	27:28	95.3
		0.25	14:20	94
33	99	0.5	26:51	94
		0.25	14:14	94

Table 2.5: Experiment results for the reentrant network using OCT-H with sparsity.

We provide the decision tree for the problem with $m = 7, s = 21$ and the sparsity parameter 0.25 in Figure 2.7. We compare this tree with the tree in Figure 2.6, which is learned by OCT-H on the same problem. Note that OCT-H with sparsity achieves 99.8 % accuracy on this problem, which is only 0.2 % decrease compared to OCT-H. Node split information is given in Table 2.4.

Observations from Figure 2.7

- The states used for the splits are a strict subset of the states used for the splits in Figure 2.6.
- The learned policy is also qualitatively similar to the policy learned with OCT-H. For example, in split 1, if x_4 is relatively large compared to x_7, the decision is to process Class 4 jobs. In split 2, if we focus on the terms associated with x_4 and x_7, again the decision is to process Class 4 jobs if x_4 is relatively large compared to x_7. Else, we idle Server 2 so that the queue on Class 7 jobs do not increase.

2.5 Conclusions

We presented an approach to solve MFQNET control problems using OCT-H. We proved that MFQNET control problems have piecewise constant optimal policy, where the segments are separated by hyperplanes passing through the origin. Based on this result, we developed an algorithm to use OCT-H to learn the optimal policy of MFQNET control problems. Computational experiments demonstrate that OCT-H can learn empirically optimal policies of MFQNET control problems with varying sizes. Once the policy is learned, we can solve MFQNET control problems considerably faster than the state-of-the-art algorithm by [101]. Furthermore, we demonstrated that the simple decision tree structure enables us to develop insights on large dimensional MFQNET control problems.

Chapter 3

Optimal Control of Fluid Restless Multi-armed Bandits: A Machine Learning Approach

3.1 Introduction

We study the continuous-time deterministic formulation of restless multi-armed bandit problems, referred to as fluid restless multi-armed bandit (FRMAB) problems. Restless multi-armed bandits, introduced by [110], are stochastic control problems that model sequential resource allocation problems across multiple projects, where each project's state evolves stochastically even when no effort is allocated to it. This model has numerous real-world applications, including healthcare [78], machine maintenance [1], and wireless communication [82], among many others. See also the recent survey [85].

To address the complexity of solving stochastic control problems, their deterministic approximations are commonly explored. A notable example is in the control and analysis of multiclass queueing networks. It has been shown that the stability of a fluid queueing network implies the stability of its stochastic counterpart [43], [104]. Fluid queueing networks also provide a useful framework for building control policies for their stochastic counterparts, often demonstrating strong empirical performance and asymptotic optimality properties [31], [76], [77]. See also the book by [27].

In the restless multi-armed bandit literature, [68] explores fluid approximations for a specific class of problems. They demonstrate that the fluid policy shows strong empirical performance for the associated stochastic problems. However, their analysis relies on strong assumptions about the problem structure and the number of projects. They also relax the coupling resource constraint, a fundamental aspect of restless bandits. Generally, solving optimal control problems without such assumptions or relaxations is computationally challenging. This challenge is particularly relevant in the scenarios where optimal control problems with varying initial states need to be solved repeatedly, a common situation in real-world applications.

Recently, many learning-based approaches have been proposed to overcome the complexity of solving optimal control problems. [64], [69], [75] develop reinforcement learning

methods, while [29] propose using a decision tree algorithm, Optimal Classification Trees with Hyperplane Splits (OCT-H) [20], to solve fluid queueing network control problems.

In this work, we propose a machine learning approach to solve FRMAB problems using OCT-H. Classification tree algorithms are particularly appealing for learning optimal policies in many continuous-time optimal control problems. These problems often have piecewise constant optimal policies, which decision trees with hyperplane splits can effectively learn [Chapters 4 & 8 in 29], [79].

Our approach is similar to the class of methods known as imitation learning in the reinforcement learning literature. We generate multiple control instances and solve them using numerical algorithms to generate training data. Then, we use supervised learning algorithm, OCT-H, to imitate the state-control mapping in the optimal trajectories.

Notational Conventions Throughout this paper, we use boldface letters to denote vectors and matrices. The i_{th} entry of a vector $\boldsymbol{x}$ is denoted x_i. We use $\mathbf{0}$ to denote the vector of zeros. We use $x(\cdot)$ to denote a real-valued function, and $\boldsymbol{x}(\cdot)$ to denote a vector whose entries are real-valued functions. We use $\boldsymbol{x}$ instead of $\boldsymbol{x}(\cdot)$ when it is clear from the context that $\boldsymbol{x}$ is referring to a vector of functions.

3.1.1 Problem Formulation

We consider a FRMAB model with n projects with finite time horizon $T < \infty$. Project $i \in [n]$ has state $x_i(t)$ at time $t \geqslant 0$, moving over the open state space $\mathcal{X}_i \triangleq (0, H_i)$ with $H_i \leqslant \infty$. We write the system state as $\boldsymbol{x}(t) = (x_i(t))_{i=1}^n$, which belongs to the state space $\mathcal{X} \triangleq \prod_{i=1}^n \mathcal{X}_i$. At each time t the system controller chooses a control $\boldsymbol{u}(t) = (u_i(t))_{i=1}^n \in [0,1]^n$ where $u_i(t) \in [0,1]$, which is required to be piecewise continuous, models the level of effort allocated to project i. The values 1 and 0 represent "full effort" and "least effort" levels, respectively. At most $m < n$ projects can be set at each time t to the "full effort" level, so we have the coupling resource constraints $\sum_{i=1}^n u_i(t) \leqslant m$.

The state evolution of project i follows first-order autonomous ordinary differential equation (ODE) referred to as the state equation: at all times t where $\boldsymbol{u}(\cdot)$ is continuous, for given continuously differentiable functions $\phi_{0,i}(\cdot)$ and $\phi_{1,i}(\cdot)$,

$$\dot{x}_i(t) = u_i(t)\phi_{1,i}(x_i(t)) + (1 - u_i(t))\phi_{0,i}(x_i(t)).$$

$\phi_{1,i}(\cdot)$ and $\phi_{0,i}(\cdot)$ represent the state equations when $u_i(t) = 1$ and $u_i(t) = 0$, respectively. Similarly, the instantaneous reward rate earned by project i at each time t depends on its current state and control, and is given by $u_i(t)R_{1,i}(x_i(t)) + (1 - u_i(t))R_{0,i}(x_i(t))$ for given continuously differentiable functions $R_{0,i}(\cdot)$ and $R_{1,i}(\cdot)$.

For a given initial state $\boldsymbol{x}_0$, the general FRMAB problem described above can be formulated

as the following optimal control problem:

$$
\begin{aligned}
\max_{\boldsymbol{u}(\cdot),\boldsymbol{x}(\cdot)} \quad & \int_0^T \sum_{i=1}^n \Big[u_i(t) R_{1,i}(x_i(t)) + (1-u_i(t)) R_{0,i}(x_i(t)) \Big] \, dt \\
\text{s.t.} \quad & \dot{x}_i(t) = u_i(t)\phi_{1,i}(x_i(t)) + (1-u_i(t))\phi_{0,i}(x_i(t)), \quad \forall i \in [n], \forall t \in [0,T], \\
& 0 < x_i(t) < H_i, \quad \forall i \in [n], \forall t \in [0,T], \\
& \boldsymbol{x}(0) = \boldsymbol{x}_0, \\
& 0 \leqslant u_i(t) \leqslant 1, \quad \forall i \in [n], \forall t \in [0,T], \\
& \sum_{i=1}^n u_i(t) \leqslant m, \quad \forall t \in [0,T].
\end{aligned}
\tag{3.1}
$$

Specifically, in this work, we focus on two fundamental cases that capture many important real-world problems. In the first case, we assume that both the state equations and the reward functions are affine in the state variable: $\phi_{u,i}(x) = \alpha_{u,i} + \beta_{u,i}x, R_{u,i}(x) = r_{u,i}x - c_{u,i}, u \in \{0,1\}$, In the second case, we assume that the state equations are quadratic (without the intercept) and the reward functions are affine in the state variable: $\alpha_{u,i}x + \beta_{u,i}x^2, \alpha_{u,i} \neq 0, \beta_{u,i} \neq 0, , R_{u,i}(x) = r_{u,i}x - c_{u,i}, u \in \{0,1\}, i \in [n]$. For ease of reference, we will term the first case as the "affine system" and the second as the "quadratic system" throughout the remainder of the paper.

Here, we assume that the state equations for the quadratic system do not include intercepts in $\phi_{u,i}$. Our derivation relies on closed-form expressions of state trajectories within intervals where the control variable is fixed to a constant vector. This assumption simplifies these expressions, allowing us to present the key ideas and results more concisely. However, the principles of our approach can be extended to more general state equations. We also add the following assumption:

Assumption 1. *$\phi_{u,i}(\cdot)$ is a concave function for all $u \in \{0,1\}, i \in [n]$.*

Note that the affine system automatically satisifes this assumption. In the quadratic case, this assumption requires $\beta_{u,i} < 0, u \in \{0,1\}, i \in [n]$. This condition ensures that the algorithm developed in Section 3.3.2 finds optimal solutions for Problem 3.1.

In general, enforcing the state constraints such as $0 < x_i(t) < H_i$ makes the problem considerably more challenging to solve. Fortunately, many important problems automatically satisfy these upper and lower bound constraints without explicit enforcement. This is because the state equation often guarantees that the state trajectory remains within a bounded interval, due to the existence of equilibrium points in dynamical systems [100, Chapter 5-11]. Therefore, we focus on the class of problems where we can treat the problem as if state constraints do not exist.

A solution to Problem (3.1) is given by a pair of optimal state and control trajectories $\boldsymbol{x}^*(\cdot)$ and $\boldsymbol{u}^*(\cdot)$ starting from a fixed initial state $\boldsymbol{x}_0$, where $\boldsymbol{x}^*(\cdot)$ is piecewise continuously differentiable and $\boldsymbol{u}^*(\cdot)$ is piecewise continuous over $[0,T]$. However, in practice, one often needs to resolve Problem (3.1) with different initial states repeatedly. Therefore, the mapping from any state $\boldsymbol{x} \in \mathcal{X}$ to its associated optimal control $\boldsymbol{u} \in [0,1]^n$ is more useful than a single pair of trajectories from a specific initial state. This mapping is referred to as a state feedback policy, which is typically time-dependent because Problem (3.1) is a finite horizon

problem and an optimal stationary policy usually does not exist. Computing such a policy, however, is generally considered very challenging. The goal of this work is to propose using OCT-H to learn a time-dependent state feedback policy

$$\pi : \mathcal{X} \times [0, T] \mapsto [0, 1]^n.$$

3.1.2 Contributions

The contributions of the paper are as follows.

1. We initiate the study of fluid restless multi-armed bandits where the state equations are either affine or quadratic in the state. We derive fundamental properties of these systems and use them to efficiently implement a numerical solution algorithm known as the shooting method [103].

2. We propose the use of the decision tree algorithm, OCT-H, to learn a state feedback policy. To address potential nonlinearities in the training data, we leverage the structural properties of FRMAB problems, developing an efficient technique for nonlinear feature augmentation.

3. We test our approach on machine maintenance, epidemic control, and fisheries control problems, demonstrating that it produces high-quality feedback policies for these applications.

4. We show that once a policy is learned, it leads to a significant speed-up compared to solving a problem from scratch using the shooting method.

3.1.3 Paper Structure

Section 3.2 provides the optimality conditions for general FRMAB problems and includes a brief review of OCT-H. Section 3.3 analyzes affine and quadratic systems. The derived results enable efficient implementation of the shooting method. In Section 3.4, we develop a learning approach based on OCT-H. Section 3.5 reports the results of computational experiments, analyzing the accuracy and speed of our approach. In Section 3.6, we include our conclusions.

3.2 Background

In this section, we review Pontryagin's Maximum Principle [55, Theorem 3.4] to derive the optimality conditions for Problem (3.1). Following this, we provide a brief overview of OCT-H.

3.2.1 Optimality Conditions of FRMAB Problems

The Pontryagin's Maximum Principle gives necessary optimality conditions for general optimal control problems [55, Theorem 3.4]. Due to Assumption 1, these conditions also become

sufficient [100, Chapter 2]. To apply Pontryagin's maximum principle to Problem (3.1), we formulate the *Hamiltonian* which involves the *costate* variable $\boldsymbol{y}(\cdot)$, given by

$$
\begin{aligned}
&H(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{y}, t) \\
&= \sum_{i=1}^{n} u_i(t) R_{1,i}(x_i(t)) + (1 - u_i(t)) R_{0,i}(x_i(t)) + y_i(t) \Big[u_i(t) \phi_{1,i}(x_i(t)) + (1 - u_i(t)) \phi_{0,i}(x_i(t)) \Big].
\end{aligned}
$$

The Pontryagin's Maximum Principle applied to Problem (3.1) can be formulated as the following lemma.

Lemma 5 (Pontryagin Maximum Principle [55], [89]). *Under Assumption 1, $\boldsymbol{x}^*(\cdot)$ and $\boldsymbol{u}^*(\cdot)$ are optimal state and control trajectories for Problem (3.1), if and only if there exists a continuous and piecewise continously differentiable costate variable $\boldsymbol{y}(\cdot)$, such that*

a) *For all $i \in [n]$, at every time t where $\boldsymbol{u}(\cdot)$ is continuous,*

$$
\begin{aligned}
\dot{y}_i(t) &= -\mathcal{H}_{x_i}(\boldsymbol{x}^*, \boldsymbol{u}^*, \boldsymbol{y}, t) \\
&= -\dot{R}_{0,i}(x_i^*(t)) - y_i(t) \dot{\phi}_{0,i}(x_i^*(t)) \\
&- \big[\dot{R}_{1,i}(x_i^*(t)) - \dot{R}_{0,i}(x_i^*(t)) \ + y_i(t) \big(\dot{\phi}_{1,i}(x_i^*(t)) - \dot{\phi}_{0,i}(x_i^*(t)) \big) \big] u_i^*(t)
\end{aligned} \tag{3.2}
$$

b) *The following* transversality condition *holds:*

$$
\boldsymbol{y}(T) = 0. \tag{3.3}
$$

c) *At each time t,*

$$
\mathcal{H}(\boldsymbol{x}^*, \boldsymbol{u}^*, \boldsymbol{y}, t) \geqslant \mathcal{H}(\boldsymbol{x}^*, \boldsymbol{u}, \boldsymbol{y}, t) \text{ for all feasible controls } \boldsymbol{u}. \tag{3.4}
$$

By Lemma 5 (c), it is straightforward that at any time t, a control $\boldsymbol{u}^*(t)$ satisfying (3.4) can be computed by solving the following linear optimization (LO) problem:

$$
\begin{aligned}
\max_{\boldsymbol{u}} \quad & \sum_{i=1}^{n} \gamma_i^*(t)\, u_i \\
s.t. \quad & 0 \leqslant u_i \leqslant 1, \quad \forall i \in [n], \\
& \sum_{i=1}^{n} u_i \leqslant m,
\end{aligned} \tag{3.5}
$$

where $\gamma_i^*(t) \triangleq R_{1,i}(x_i^*(t)) - R_{0,i}(x_i^*(t)) + y_i(t) \big[\phi_{1,i}(x_i^*(t)) - \phi_{0,i}(x_i^*(t)) \big]$. That is, the optimal control at time t follows an index policy, determined by ranking the index functions $\gamma_i^*(t)$.

3.2.2 Optimal Classification Trees with Hyperplane Splits

Optimal Classification Trees (OCT) trains a near-optimal decision tree for classification tasks. Unlike classical decision tree algorithms such as CART [34], which rely on a greedy algorithm, OCT aims to learn a globally optimal decision tree using advanced optimization techniques. OCT uses a single feature in the node splits, meaning it partitions the feature space with hyperplanes that are perpendicular to the axis and assigns a prediction to each region. This often results in improved accuracy, robustness to noise, and shallower trees compared to CART.

OCT-H, introduced in [20], is a generalization of OCT, where arbitrary linear combinations of the features are used for splits. Unlike OCT, OCT-H can partition the feature space with arbitrary hyperplanes, enabling it to capture more complex patterns in the data and often leading to better prediction accuracy. For more details on this technique, refer to [21].

3.3 Fluid Restless Multi-Armed Bandits

In this section, we first outline the basic properties of general FRMAB problems. Then, we analyze two distinct cases: one where the state equations are affine in the state and another where they are quadratic. Finally, we describe the shooting method [103], a classical numerical algorithm used to solve optimal control problems. The version we present is tailored to solve Problem (3.1) using the results we derive.

3.3.1 Basic Properties

The next result elucidates the structure of the optimal control $\boldsymbol{u}^*(\cdot)$ to Problem (3.1) using Lemma 5.

Proposition 1. *There exists an optimal control $\boldsymbol{u}^*(\cdot)$ that is piecewise constant in t, with each entry being either 1 or 0.*

Proof. As mentioned in Section 3.2.1, at each time t, the optimal control $\boldsymbol{u}^*(t)$ is determined by solving Problem (3.5). $\boldsymbol{u}^*(t)$ is always binary, which follows straightforwardly from the analysis of the dual LO for Problem (3.5):

$$\begin{aligned} \min_{\boldsymbol{v},w} \quad & mw + \sum_{i=1}^{n} v_i \\ s.t. \quad & v_i \geqslant 0, \quad w \geqslant 0, \\ & v_i + w \geqslant \gamma_i^*(t), \quad i \in [n]. \end{aligned}$$

Furthermore, the index function $\gamma_i^*(t)$ is continuous in t in Lemma 5. As long as the rank between the index functions does not change, the control vector remains constant. Therefore, the optimal control $\boldsymbol{u}^*(t)$ is piecewise constant in t and always a binary vector. □

Proposition 1 indicates that in the optimal trajectories of $\boldsymbol{u}^*(\cdot)$ and $\boldsymbol{x}^*(\cdot)$, the time interval is divided into multiple subintervals. Within each subinterval, the optimal control $\boldsymbol{u}(\cdot)$ is a constant binary vector. In the subsequent sections, we derive the closed-form trajectories of

the state and the costate variables in a subinterval with a constant control vector. These closed-form expressions will be used to develop an efficient version of the shooting method, as well as the machine learning approach.

For convenience, we introduce the following notations for the remainder of the paper, defined for all $i \in [n]$:

$$\begin{aligned}\alpha_i(u) &\triangleq \alpha_{0,i} + u(\alpha_{1,i} - \alpha_{0,i}),\\ \beta_i(u) &\triangleq \beta_{0,i} + u(\beta_{1,i} - \beta_{0,i}),\\ r_i(u) &\triangleq r_{0,i} + u(r_{1,i} - r_{0,i}).\end{aligned}$$

Affine Systems

We next apply the above framework to the case where both the state equations and reward functions are affine in the state, given by $\phi_{u,i}(x_i) = \alpha_{u,i} + \beta_{u,i}x_i$ and $R_{u,i}(x_i) = r_{u,i}x_i - c_{u,i}$ for $u = 0, 1, i \in [n]$.

Suppose we are given a finite partition of the time interval $[0, T]$ that consists of S subintervals $[t_s, t_{s+1})$, along with corresponding binary controls $\boldsymbol{u}_s \in \{0,1\}^n$, for $s = 0, 1, \ldots, T-1$, with $t_0 = 0$ and $t_S = T$. Given an initial state $\boldsymbol{x}(0) = \boldsymbol{x}_0$ and a costate $\boldsymbol{y}(0) = \boldsymbol{y}_0$, we can consider the resulting trajectories $\{(\boldsymbol{x}(t), \boldsymbol{y}(t)) : t \in [0, T]\}$ obtained by taking control $\boldsymbol{u}_s$ on time interval $[t_s, t_{s+1})$ for each s. Recall that both the state and the costate trajectories are continuous in t, which allows us to build the entire trajectory with the given information. Hence, without loss of generality, we focus on an interval $[t_s, t_{s+1})$ with constant control.

The ODEs giving the state and costate evolution for project i are: for $t \in [t_s, t_{s+1})$,

$$\begin{aligned}\dot{x}_i(t) &= \alpha_i(u_{s,i}) + \beta_i(u_{s,i})x_i(t),\\ \dot{y}_i(t) &= -r_i(u_{s,i}) - \beta_i(u_{s,i})y_i(t).\end{aligned} \tag{3.6}$$

The solution to (3.6), given $x_i(t_s)$ and $y_i(t_s)$, can be obtained in closed form, as shown in the next result. The proof is not included, as it is straightforward from elementary ODE theory.

$$\begin{aligned}x_i(t) &= \begin{cases} x_i(t_s) + \left[-\dfrac{\alpha_i(u_{s,i})}{\beta_i(u_{s,i})} - x_i(t_s)\right]\left[1 - e^{\beta_i(u_{s,i})(t-t_s)}\right], & \text{if } \beta_i(u_{s,i}) \neq 0,\\ x_i(t_s) + \alpha_i(u_{s,i})(t - t_s), & \text{if } \beta_i(u_{s,i}) = 0,\end{cases}\\ y_i(t) &= \begin{cases} y_i(t_s) + \left[-\dfrac{r_i(u_{s,i})}{\beta_i(u_{s,i})} - y_i(t_s)\right]\left[1 - e^{-\beta_i(u_{s,i})(t-t_s)}\right], & \text{if } \beta_i(u_{s,i}) \neq 0,\\ y_i(t_s) - r_i(u_{s,i})(t - t_s), & \text{if } \beta_i(u_{s,i}) = 0.\end{cases}\end{aligned} \tag{3.7}$$

Quadratic Systems

We proceed similarly to the case where the state equations and the reward functions are quadratic and affine in the state, respectively. That is, the reward function for each project $i \in [n]$ is given by $R_{1,i}(x_i) = r_{u,i}x_i - c_{u,i}$ and the state equation is $\phi_{u,i}(x_i) = \alpha_{u,i}x_i + \beta_{u,i}x_i^2, \beta_{u,i} \neq 0, \alpha_{u,i} \neq 0$, for all $u \in \{0,1\}, i \in [n]$. Although not addressed in this work, generalizing the reward functions to quadratic forms is straightforward. Again, we focus

on the fixed interval $[t_s, t_{s+1})$ with constant control $\boldsymbol{u}(t) = \boldsymbol{u}_s$. In this interval, the ODEs governing the state and costate evolution for project i are:

$$\begin{aligned} \dot{x}_i(t) &= \alpha_i(u_{s,i})x_i(t) + \beta_i(u_{s,i})x_i^2(t), \\ \dot{y}_i(t) &= -r_i(u_{s,i}) - y_i(t)\big[\alpha_i(u_{s,i}) + 2\beta_i(u_{s,i})x_i(t)\big]. \end{aligned} \tag{3.8}$$

The solution to (3.8) is given as follows. Unlike the case of affine systems, we do not use expressions involving the initial conditions in the subinterval $x_i(t_s)$ and $y_i(t_s)$, as this leads to considerably more complicated expressions. Instead, we introduce constants K and G for simplicity.

$$\begin{aligned} x_i(t) &= \frac{K\alpha_i(u_{s,i})e^{\alpha_i(u_{s,i})t}}{1 - K\beta_i(u_{s,i})e^{\alpha_i(u_{s,i})t}}, \\ y_i(t) &= Ge^{-\alpha_i(u_{s,i})t}(1 - K\beta_i(u_{s,i})e^{\alpha_i(u_{s,i})t})^2 - \frac{r_i(u_{s,i})}{K\alpha_i(u_{s,i})\beta_i(u_{s,i})}(1 - K\beta_i(u_{s,i})e^{\alpha_i(u_{s,i})t})e^{-\alpha_i(u_{s,i})t}. \end{aligned} \tag{3.9}$$

3.3.2 The Shooting Method

We use the results derived above to design an algorithm for computing the optimal trajectories for Problem (3.1) with a fixed initial state $\boldsymbol{x}_0$. The algorithm is based on the well-known *shooting method*, and further exploits the structure of the optimal trajectories identified previously.

The algorithm aims to find an initial costate value $\boldsymbol{y}_0$ that achieves the terminal condition $\boldsymbol{y}(T) = 0$. Once an initial state $\boldsymbol{x}_0$ and initial costate $\boldsymbol{y}_0$ are fixed, the associated trajectory $\boldsymbol{x}(\cdot), \boldsymbol{u}(\cdot), \boldsymbol{y}(\cdot)$ up to time T can be computed. This computation follows the index policy that satisfies (3.4). Specifically, at each time t, given $\boldsymbol{x}(t)$ and $\boldsymbol{y}(t)$, we rank the index functions $\gamma_i(t)$ for all projects to determine the associated control vector. As long as the control vector remains constant, we can roll out the trajectories using the expressions derived in Section 3.3.1. A change in the control vector indicates the start of a new subinterval. According to Lemma 5, if the resulting trajectories also satisfy $\boldsymbol{y}(T) = 0$, they are guaranteed to be optimal.

Formally, let $\boldsymbol{g} : \mathbb{R}^n \mapsto \mathbb{R}^n$ be the function that takes an initial costate value $\boldsymbol{y}_0$ as an input, and the resulting terminal costate value $\boldsymbol{y}(T)$ as an output. The shooting method is an iterative algorithm to solve the $n \times n$ root finding problem $\boldsymbol{g}(\boldsymbol{y}) = \boldsymbol{0}$ numerically.

In the k_{th} iteration of the shooting method, the algorithm starts with a guess $\boldsymbol{y}_{k,0}$ for $\boldsymbol{y}(0)$, computes corresponding state and costate trajectories $\boldsymbol{x}(\cdot)$ and $\boldsymbol{y}(\cdot)$ up to time T following the index policy structure. This automatically partitions of the time interval $[0, T]$ into subintervals $[t_s, t_{s+1})$ with a constant control u^s, for $s = 0, \ldots, S-1$, with $t_0 = 0$ and $t_S = T$, where the number S of intervals is also determined. Then, the algorithm checks whether $\boldsymbol{y}(T) \approx 0$ within a given tolerance level ϵ. If such is the case, the algorithm stops. Otherwise, the initial costate value is updated and the process is repeated. The update follows Broyden's method [35], [52], a well-established derivative-free quasi-Newton's method.

In the k_{th} iteration, $k \geqslant 1$, Broyden's method computes iterates $\boldsymbol{y}_{k,0}$ and $\boldsymbol{J}_k$, a surrogate

for the Jacobian of $\boldsymbol{g}$, following

$$\begin{aligned}\boldsymbol{y}_{k,0} &= \boldsymbol{y}_{k-1,0} - \big(\boldsymbol{J}_{k-1}\big)^{-1}\boldsymbol{g}(\boldsymbol{y}_{k-1,0}),\\ \boldsymbol{J}_k &= \boldsymbol{J}_{k-1} + \frac{\boldsymbol{g}(\boldsymbol{y}_{k,0}) - \boldsymbol{g}(\boldsymbol{y}_{k-1,0}) - \boldsymbol{J}_{k-1}(\boldsymbol{y}_{k,0} - \boldsymbol{y}_{k-1,0})}{\|\boldsymbol{y}_{k,0} - \boldsymbol{y}_{k-1,0}\|^2}\big(\boldsymbol{y}_{k,0} - \boldsymbol{y}_{k-1,0}\big)'.\end{aligned} \tag{3.10}$$

Typically, the method starts with $\boldsymbol{J}^0 = \boldsymbol{I}$, the identity matrix. Also, when we compute the trajectory starting from $t = 0$, we choose a small step size δ and proceed by incrementally adding δ to t until reaching T. The details of the algorithm are provided in Algorithm 2.

Remark. In general, shooting methods are known to be vulnerable to numerical instability, primarily because they involve computing the trajectories of $\boldsymbol{x}(\cdot), \boldsymbol{y}(\cdot)$, which often require numerical approximations of the solutions to differential equations [46], [62], [103]. However, for the affine and quadratic systems discussed in this work, no approximation is required because closed-form expressions of $\boldsymbol{x}(\cdot), \boldsymbol{y}(\cdot)$ exist, given some initial values. This advantage reduces the risk of numerical errors and enhances the reliability and accuracy of the shooting method.

Algorithm 2: Shooting Method

Input: $\epsilon, \delta, R_i^0(\cdot), R_i^1(\cdot), \phi_i^0(\cdot), \phi_i^1(\cdot), T, m, n, \boldsymbol{x}_0$

Output: $\Big\{\big(\boldsymbol{x}^*(t), \boldsymbol{u}^*(t), \boldsymbol{y}(t)\big) : t \in [0, T], \boldsymbol{x}(0) = \boldsymbol{x}_0\Big\}$

Initialization: $\boldsymbol{y}_{0,0} \in \mathbb{R}^n, \boldsymbol{J}_0 = \boldsymbol{I}, k = 0$

repeat

- Fix the initial state to $\boldsymbol{x}_0$ and the initial costate to $\boldsymbol{y}_{k,0}$.
- Compute the trajectories of $\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{y}(t)$ starting from $t = 0$ up to $t = T$.
- $k \leftarrow k + 1$.
- Update $\boldsymbol{y}_{k,0}$ and $\boldsymbol{J}_k$ as in (3.10).

until $\|\boldsymbol{g}(\boldsymbol{y}_{k-1,0})\|_\infty \le \epsilon$;

Return $\Big\{\big(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{y}(t)\big) : t \in [0, T], \boldsymbol{x}(0) = \boldsymbol{x}_0\Big\}$.

3.4 A Machine Learning Approach

In this section, we present a machine learning approach to learn a state feedback policy for Problem (3.1). We begin by providing an overview of the algorithm, then introduce a nonlinear feature augmentation technique to incorporate nonlinearity into the learned policy. Finally, we present an example of our approach, focusing on the case of admission and routing to parallel infinite-server queues.

3.4.1 Algorithm Overview

Our approach uses OCT-H to imitate the time-dependent state-control mappings in the optimal trajectories. To achieve this, we generate multiple initial states for Problem (3.1)

and solve each instance using Algorithm 2. For each instance with an initial state $\boldsymbol{x}_0$, we obtain the optimal state and control trajectories $\Big\{(\boldsymbol{x}^*(t), \boldsymbol{u}^*(t)) : t \in [0, T], \boldsymbol{x}(0) = \boldsymbol{x}_0\Big\}$. We select $N \in \mathbb{N}$ distinct time points $t_1, \ldots, t_N$ from the interval $[0, T]$, and extract the corresponding state values $\boldsymbol{x}^*(t_1), \ldots, \boldsymbol{x}^*(t_N)$ and the control values $\boldsymbol{u}^*(t_1), \ldots, \boldsymbol{u}^*(t_N)$. The training data extracted from this process consists of $\left\{\Big(\big(\boldsymbol{x}^*(t_\ell), t_\ell\big), \boldsymbol{u}^*(t_\ell)\Big)\right\}_{\ell=1}^{N}$, where the tuple $\big(\boldsymbol{x}^*(t_\ell), t_\ell\big)$ is the feature vector and $\boldsymbol{u}^*(t_\ell)$ is the target for the ℓ_{th} data point. The time t_ℓ is included as part of the feature vector to incorporate the time-dependency of the state feedback policy.

As discussed in Section 3.2.2, OCT-H partitions the state space using hyperplanes and assigns predictions to each region. Consequently, OCT-H may struggle to capture complex nonlinear patterns in the data. One straightforward solution could be to use deep neural networks, leveraging their strong approximation capabilities. However, deep neural networks are black box algorithms that obscure how decisions are made, as they automatically learn nonlinearities embedded in their layers. Moreover, simply feeding raw data into a deep neural network makes it challenging to fully exploit the inherent structures of FRMAB problems.

Instead of using deep neural networks, we adhere to OCT-H and address potential nonlinearity in the decision boundaries by augmenting the feature vector with nonlinear transformations of the state variables. Specifically, for each feature vector $\big(\boldsymbol{x}^*(t_\ell), t_\ell\big)$ in the training data, we add nonlinear transformations of the state vector $\boldsymbol{x}^*(t_\ell)$ as additional entries. Then, we apply OCT-H to the augmented data set. This approach ensures that the resulting policy remains interpretable, maintaining the decision-making process within the framework of decision trees.

3.4.2 Addressing Nonlinearity by Feature Augmentation

We propose a heuristic approach to choosing nonlinear transformations on the state variables. As discussed in Section 3.2.1, the optimal control at time t is determined by the relative ordering of the priority indices $\gamma_i(t), i \in [n]$. This implies that switches in effort (i.e., changes in the control vector) occur either when $\gamma_i(t) = \gamma_j(t)$ for some projects $i \neq j$, or when $\gamma_i(t) = 0$. At these points, it becomes indifferent whether we invest effort in project i or j, or whether we invest effort in project i or idle, respectively. The set of points in the state space where such switching of efforts occur are known as the switching curves in the optimal control literature [100]. In other regions of the state space, there exists a unique constant optimal control vector. Switching curves can be thought of as discriminating lines in the feature space that separate data points with different labels in a supervised learning setting. However, identifying switching curves or their functional forms is challenging. Switching curves can be found by expressing the equalities $\gamma_i(t) = \gamma_j(t)$ only with respect to the state variables $\boldsymbol{x}(t)$. Since $\gamma_i(t)$ involves the costate variable $y_i(t)$, we need a way to express $y_i(t)$ in terms of $\boldsymbol{x}(t)$. We propose a heuristic to approximate the relation between $y_i(t)$ and $\boldsymbol{x}(t)$.

Affine Systems

Consider an interval $[t_s, t_{s+1})$ with a constant control $\boldsymbol{u}_s$. We first express the index function $\gamma_i(t)$ only in terms $x_i(t)$ in this specific interval.

Proposition 2. *For affine systems, in an interval $[t_s, t_{s+1})$ with a constant control $\boldsymbol{u}_s$, $\gamma_i(t)$ can be expressed as an affine combination of the terms*

$$t, x_i(t), x_i^2(t), \frac{1}{x_i(t) + \frac{\alpha_{0,i}}{\beta_{0,i}}}, \frac{1}{x_i(t) + \frac{\alpha_{1,i}}{\beta_{1,i}}}. \tag{3.11}$$

Proof. Using the results in (3.7), the relation between $x_i(t)$ and $y_i(t)$ can be described as follows:

$$y_i(t) = \begin{cases} y_{i,1}(t, u_{s,i}) \triangleq -\dfrac{r_i(u_{s,i})}{\beta_i(u_{s,i})} + \dfrac{\left(x_i(t_s) + \frac{\alpha_i(u_{s,i})}{\beta_i(u_{s,i})}\right)\left(y_i(t_s) + \frac{r_i(u_{s,i})}{\beta_i(u_{s,i})}\right)}{x_i(t) + \frac{\alpha_i(u_{s,i})}{\beta_i(u_{s,i})}}, \\ \qquad \text{if } \beta_i(u_{s,i}) \neq 0, \\ y_{i,2}(t, u_{s,i}) \triangleq y_i(t_s) - r_i(u_{s,i})\dfrac{x_i(t) - x_i(t_s)}{\alpha_i(u_{s,i})}, \\ \qquad \text{if } \beta_i(u_{s,i}) = 0, \alpha_i(u_{s,i}) \neq 0, r_i(u_{s,i}) \neq 0, \\ y_{i,3}(t, u_{s,i}) \triangleq x_i(t) - x_i(t_s) + y_i(t_s) - r_i(u_{s,i})(t - t_s), \\ \qquad \text{if } \beta_i(u_{s,i}) = \alpha_i(u_{s,i}) = 0, r_i(u_{s,i}) \neq 0, \\ y_{i,4}(t, u_{s,i}) \triangleq y_i(t_s), \\ \qquad \text{if } \beta_i(u_{s,i}) = r_i(u_{s,i}) = 0. \end{cases}$$

Given that the optimal control vector is always binary, as stated in Proposition 1, this relation leads to the unified expression:

$$y_i(t) = \sum_{j=1}^{4} \Big(C_{1,j} y_{i,j}(t, 1) + C_{0,j} y_{i,j}(t, 0) \Big), \tag{3.12}$$

where only one of the coefficients $C_{u,j}, u \in \{0, 1\}, j \in [4]$ is non-zero, depending on the values of $\beta_i(u_{s,i}), \alpha_i(u_{s,i}), r_i(u_{s,i})$.

Subsequently, we substitute the expression (3.12) into the index function

$$\gamma_i(t) = r_{1,i} x_i(t) - c_{1,i} - r_{0,i} x_i(t) + c_{0,i} + y_i(t)[\alpha_{1,i} + \beta_{1,i} x_i(t) - \alpha_{0,i} - \beta_{0,i} x_i(t)],$$

resulting in an expression for $\gamma_i(t)$ that involves an affine combination of the terms only with respect to $x_i(t)$ given in (3.11). Here, the terms $\frac{1}{x_i(t)+\frac{\alpha_{0,i}}{\beta_{0,i}}}$ and $\frac{1}{x_i(t)+\frac{\alpha_{1,i}}{\beta_{1,i}}}$ correspond to $y_{i,1}(t, u_{s,i})$, while $x_i^2(t)$ corresponds to $y_{i,2}(t, u_{s,i})$ and $y_{i,3}(t, u_{s,i})$, and t corresponds to $y_{i,3}(t, u_{s,i})$. □

The initial values $y_i(t_s)$ and $x_i(t_s)$ that appear in the coefficients of the affine combination in Proposition 2 should vary depending on the specific time interval and the initial state $\boldsymbol{x}(0)$ that led to that interval. In other words, these coefficients remain constant only within

the specific interval considered. Thus, these values should generally be treated as time and state-dependent, and the relation between $\gamma_i(t)$ and $x_i(t)$ described in Proposition 2 does not globally apply across the entire state space.

We propose simply approximating $\gamma_i(t)$ with the affine combination of the terms in (3.11), regardless of time and state. This approach treats the coefficients of the affine combination as constants to be learned, and allows us to approximate any switching curve using the affine combination of the terms in (3.11), defined for all $i \in [n]$. Therefore, for each $i \in [n]$, we augment the original feature vectors with the terms in (3.11). Applying OCT-H to these augmented feature vectors enables the learning of nonlinear switching curves within the state space.

Note that while our approach assumes implicit time-dependence of switching curves through the inclusion of time as part of the feature vector in each original data point, explicit time dependencies are not separately modeled in the approximation of $\gamma_i(t)$.

Quadratic Systems

For the quadratic case, we apply the same principle as in the affine systems, leading to the following proposition.

Proposition 3. *For quadratic systems, in an interval $[t_s, t_{s+1})$ with a constant control $\boldsymbol{u}_s$, $\gamma_i(t)$ can be expressed as an affine combination of the terms*

$$x_i(t), \frac{1}{x_i(t)}, \frac{1}{x_i(t) + \frac{\alpha_i^0}{\beta_i^0}}, \frac{1}{x_i(t) + \frac{\alpha_i^1}{\beta_i^1}}. \tag{3.13}$$

Proof. The results in (3.9) lead to the following expression:

$$y_i(t) = \left(\frac{K(\alpha_i(u) + x_i(t)\beta_i(u))}{x_i(t)} - K\beta_i(u) \right) \left(G - G\beta_i(u) \frac{x_i(t)}{\alpha_i(u) + x\beta_i(u)} - \frac{r_i(u)}{K\alpha_i(u)\beta_i(u)} \right)$$

We plug this expression into the index function

$$\gamma_i(t) = u(r_i^1 x_i(t) - c_i^1) + (1-u)(r_i^0 x_i(t) - c_i^0) + y_i(t) \left(u(\alpha_i^1 x_i + \beta_i^1 x_i^2) + (1-u)(\alpha_i^0 x_i + \beta_i^0 x_i^2) \right),$$

to get the results. □

Following the approach used for affine systems, we augment the original feature vector in each data point with the terms in (3.13), for all $i \in [n]$.

Data-driven Feature Augmentation

In practice, not all of the terms listed above are needed, as the expression of $y_i(t)$ with respect to $x_i(t)$ depends on the problem parameters $\beta_{u_{s,i},i}, \alpha_{u_{s,i},i}, r_{u_{s,i},i}$ and whether $u_{s,i}$ is 1 or 0. This means some of the expressions in (3.11) and (3.13) may not be used at all. After we have generated the training data, we can augment the feature vector in a data-driven way, by investigating the control variables that occurred in the generated data. For example, in

the affine system, assume $u_i = 1$ for all data points generated. If $\beta_{1,i} \neq 0$, then the terms resulting from the cases $y_{i,1}(t, 0)$ and $y_{i,j}(t, u_i), j = 2, 3, 4$, are unnecessary because only $C_{1,1}$ in (3.12) will be non-zero. Hence, we augment the feature vector only with $\frac{1}{x_i(t)+\frac{\alpha_{1,i}}{\beta_{1,i}}}$. This principle generalizes to data-driven feature augmentation process, provided in Algorithm 3. We describe the entire proposed approach, beginning from data generation to decision tree training, in Algorithm 4.

Algorithm 3: Data-Driven Feature Augmentation

Input: $\mathcal{D}$,$\mathcal{U}, n$
Output: Augmented feature vectors $\mathcal{D}'$
Initialization: $\mathcal{D}' = \mathcal{D}$

Identify $v_i = \{u_i \in \mathbb{R} : \boldsymbol{u} \in \mathcal{U}\},\ \forall i \in [n]$.
Affine Systems
for $i \in [n]$ **do**
  for $u_i \in v_i$ **do**
    for $\big(\boldsymbol{x}, t\big) \in \mathcal{D}'$ **do**
      if $\beta_i(u_i) \neq 0$ **then**
        Add $\frac{1}{x_i+\frac{\alpha_i(u_i)}{\beta_i(u_i)}}$ as an additional entry to the feature vector $\big(\boldsymbol{x}, t\big)$.
      else if $r_i(u_i) \neq 0$ **then**
        Add x_i^2 as an additional entry to the feature vector $\big(\boldsymbol{x}, t\big)$.

Quadratic Systems
for $i \in [n]$ **do**
  for $\big(\boldsymbol{x}, t\big) \in \mathcal{D}'$ **do**
    Add $\frac{1}{x_i}$ as an additional entry to the feature vector $\big(\boldsymbol{x}, t\big)$.
    for $u_i \in v_i$ **do**
      Add $\frac{1}{x_i+\frac{\alpha_i(u_i)}{\beta_i(u_i)}}$ as an additional entry to the feature vector $\big(\boldsymbol{x}, t\big)$.

3.4.3 Example: Optimal Control of Admission and Routing to Parallel Infinite-Server Queues

We study the optimal control of admission and routing to parallel infinite-server queues as a special case of the affine systems discussed above. Due to its simple structure, we can derive closed-form expressions for the index functions $\gamma_i(t), i \in [n]$. For small n, this allows us to obtain a simple closed-form optimal policy. Then, we use Algorithm 4 to learn a decision tree policy and compare it with the closed-form optimal policy.

Algorithm 4: OCT-H for Fluid Restless Multi-armed Bandits.

Input: $M, \{t_1, \ldots, t_N\}, \epsilon, \delta, R_i^0(\cdot), R_i^1(\cdot), \phi_i^0(\cdot), \phi_i^1(\cdot), T, m, n, \boldsymbol{x}_0$
Output: $\pi : \mathcal{X} \times [0, T] \mapsto [0, 1]^n$
Initialization: $\mathcal{D} = \emptyset, \mathcal{U} = \emptyset, j = 1$

1. Data Generation
for $j = 1, \ldots, M$ **do**
  Sample an initial state $\boldsymbol{x}_0$ from the state space $\mathcal{X}$.
  Solve Problem (3.1) with the initial state $\boldsymbol{x}_0$ using Algorithm 2.
  $\mathcal{D} \leftarrow \mathcal{D} \cup \left\{ \left(\boldsymbol{x}^*(t_\ell), t_\ell \right) \right\}_{\ell=1}^{N}$
  $\mathcal{U} \leftarrow \mathcal{U} \cup \left\{ \boldsymbol{u}^*(t_\ell) \right\}_{\ell=1}^{N}$
  $j \leftarrow j + 1$

2. Feature Augmentation
Use Algorithm 3 to augment the feature vectors in $\mathcal{D}$ and denote the augmented feature vectors as $\mathcal{D}'$.

3. Training
Use OCT-H to train a classification tree. The feature vectors are $\mathcal{D}'$ and the target vectors are $\mathcal{U}$.

Deriving the Closed-form Index Functions

Consider a system with n parallel fluid queues with infinite buffers. Fluid arrives to the system at rate λ. The controller chooses the proportion $u_i(t) \in [0, 1]$ to be routed to each queue i at each time t. The system equation for the buffer contents $x_i(t)$ of queue i is

$$\dot{x}_i(t) = \lambda u_i(t) - \mu_i x_i(t),$$

which corresponds to the fluid analog of an infinite-server queue. The remaining proportion, $1 - \sum_{i=1}^n u_i(t)$, is rejected, incurring a cost rate R. Note that $\sum_{i=1}^n u_i(t) \leqslant 1$. Furthermore, queue i incurs holding costs at rate C_i.

The goal is to minimize the following cost objective over a finite horizon $[0, T]$:

$$\min_{\boldsymbol{u}(\cdot), \boldsymbol{x}(\cdot)} \int_0^T \left[R\lambda(1 - \sum_i u_i(t)) + \sum_i C_i x_i(t) \right] dt,$$

which in turn is reformulated in maximization form as

$$\max_{\boldsymbol{u}(\cdot), \boldsymbol{x}(\cdot)} \int_0^T \sum_i [R\lambda u_i(t) - C_i x_i(t)] \, dt.$$

The upper bound in the state space of each queue i is $H_i \triangleq \infty$. We define the optimal control of admission and routing to parallel infinite-server queues as the following:

$$
\begin{aligned}
\max_{\boldsymbol{u}(\cdot),\boldsymbol{x}(\cdot)} \quad & \int_0^T \sum_{i=1}^n [R\lambda u_i(t) - C_i x_i(t)]\, dt \\
\text{s.t.} \quad & \dot{x}_i(t) = \lambda u_i(t) - \mu_i x_i(t), \quad \forall i \in [n], \forall t \in [0,T], \\
& x_i(t) > 0, \quad \forall i \in [n], \forall t \in [0,T], \\
& \boldsymbol{x}(0) = \boldsymbol{x}_0, \\
& 0 \leqslant u_i(t) \leqslant 1, \quad \forall i \in [n], \forall t \in [0,T], \\
& \sum_{i=1}^n u_i(t) \leqslant 1, \quad \forall t \in [0,T].
\end{aligned} \tag{3.14}
$$

We first prove that the constraints $x_i(t) > 0, \forall i \in [n]$, are automatically satisfied regardless of the control trajectory.

Proposition 4. *For Problem (3.14), if $\boldsymbol{x}_0 > 0$, then $\boldsymbol{x}(t) > 0, \forall t \in [0,T]$ regardless of the control trajectory.*

Proof. For $t \in [t_s, t_{s+1})$ with a constant control vector $\boldsymbol{u}_s \in [0,1]^n$,

$$
\begin{aligned}
x_i(t) &= x_i(t_s) + \left[\frac{\lambda u_{s,i}}{\mu_i} - x_i(t_s)\right]\left[1 - e^{-\mu_i(t-t_s)}\right] \\
&= \frac{\lambda u_{s,i}}{\mu_i}(1 - e^{-\mu_i(t-t_s)}) + x_i(t_s)e^{-\mu_i(t-t_s)}
\end{aligned}
$$

Assuming $x_i(t_s) > 0$, it is straightforward that $x_i(t) > 0$ in the interval $[t_s, t_{s+1})$. It is also clear that $x_i(t_{s+1}) > 0$, again implying $x_i(t) > 0$ in the interval $[t_{s+1}, t_{s+2})$. Hence, due to mathematical induction, $\boldsymbol{x}_0 > 0$ guarantees that $\boldsymbol{x}(t) > 0,\ \forall t \in [0,T]$. □

Now, we derive a closed-form expression for the index function.

Proposition 5. *The index function for Problem (3.14) is*

$$
\gamma_i(t) = R - C_i \frac{\lambda}{\mu_i}\left[1 - e^{-\mu_i(T-t)}\right], \quad i \in [n], t \in [0,T]. \tag{3.15}
$$

Proof. First, note that the costate $y_i(t)$ satisfies the following ODE in this model:

$$
\dot{y}_i(t) = C_i + \mu_i y_i(t), \quad t \in [0,T].
$$

The index function derived from Lemma 5 is $\gamma_i(t) = R + \lambda y_i(t)$.

For $t \in [t_s, t_{s+1})$,

$$
\begin{aligned}
x_i(t) &= x_i(t_s) + \left[\frac{\lambda u_{s,i}}{\mu_i} - x_i(t_s)\right]\left[1 - e^{-\mu_i(t-t_s)}\right] \\
y_i(t) &= y_i(t_s) + \left[-\frac{C_i}{\mu_i} - y_i(t_s)\right]\left[1 - e^{\mu_i(t-t_s)}\right].
\end{aligned}
$$

Applying the boundary condition $y_i(T) = 0$, $y_i(t)$ is given by

$$
y_i(t) = -\frac{C_i}{\mu_i}\left[1 - e^{-\mu_i(T-t)}\right], \quad t \in [0,T]. \tag{3.16}
$$

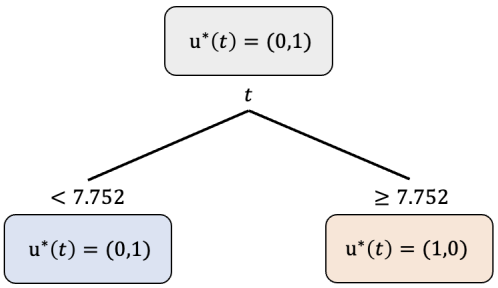

Figure 3.1: The decision tree OCT-H learned for the infinite server routing problem with $n = 2$.

Therefore, substituting (3.16) into the index function $\gamma_i(t)$, we obtain the closed-form index expression. □

By Proposition 5, solving Problem (3.14) is significantly simplified. At each time t, we compute the index functions for all projects and allocate efforts accordingly.

Learning Feedback Policy using OCT-H

In this section, we present a state feedback policy for Problem (3.14) learned by OCT-H. We consider the problem with $n = 2, m = 1$. For this problem, at each time t, we only need to compare $\gamma_1(t), \gamma_2(t)$ and 0 to determine the optimal control. From the derivation above, it is straightforward that the optimal feedback policy depends only on the time t and not on the state variable. Given the parameters $\mu_1 = 0.5, \mu_2 = 1, C_1 = 1, C_2 = 1.5, \lambda = 1, R = 3, T = 10$, the optimal feedback policy is:

$$\pi(\boldsymbol{x}, t) = \begin{cases} (0,1), & \text{if } t < 10 - \log 9 \approx 7.802, \\ (1,0), & \text{if } t \geq 10 - \log 9 \approx 7.802. \end{cases} \tag{3.17}$$

To generate a training data, we sample 1000 initial states uniformly from the interval $(0, 10)^2$ and solve each instance. For each solved instance, we extract 10 feature vectors along with their associated control vectors from each subinterval with constant control. The policy learned by OCT-H is given in Figure 3.1. We observe that this policy is almost identical to the optimal policy in (3.17), with only slight numerical differences.

3.5 Computational Experiments

In this section, we present the results of computational experiments. We conduct experiments on three distinct problems with varying sizes and time horizons. The first problem, the machine maintenance problem, belongs to affine systems. The other two problems, the epidemic control and the fisheries control problems, belong to quadratic system. We evaluate the quality of the policy learned using Algorithm 4, and also assess the relative speed-up compared to Algorithm 2. For all problems considered, the state trajectories automatically

satisfy the state constraints $x_i(t) \in (0, H_i), \forall i \in [n]$, regardless of the control trajectories. The proof is provided in the Appendix A.

3.5.1 Problem Description

Machine Maintenance

We consider the classic machine maintenance problem studied in [61]. In this problem, the state variable $x_i(t), i \in [n]$ represents the cumulative probability that machine i has failed by time t, while $u_i(t)$ represents the preventive maintenance rate at time t. The natural failure rate, cost of maintenance, junk value, revenue of machine i are denoted as h_i, C_i, L_i, R_i, respectively. The primary objective is to maximize the total profit generated by the machines, adjusted for any junk values if a machine fails prematurely. The objective is to maximize $\int_0^T \sum_{i=1}^n \Big[R_i - C_i h_i u_i(t) + L_i(h(1-u_i(t))(1-x_i(t)))\Big]\, dt$ subject to the state equations $\dot{x}_i(t) = h(1-u_i(t))(1-x_i(t))$. To align this problem with the formulation in Section 3.3.1, we can set the parameters as $\alpha_{1,i} = \beta_{1,i} = 0$, and $\alpha_{0,i} = h_i, \beta_{0,i} = -h_i$. In this model, the state trajectory stays in $(0,1)^n$ regardless of the control policy.

To generate a problem, we sample the parameters $\boldsymbol{h}, \boldsymbol{C}, \boldsymbol{L}, \boldsymbol{R}$ uniformly at random from the intervals $[0, 0.5]^n, [1,3]^n, [2,4]^n, [2,4]^n$, respectively.

Epidemic Control

This problem is based on the SIS epidemic model studied in [88]. It has been shown that the fraction of infected individuals in a stochastic version of the SIS epidemic model, following a continuous-time Markov chain, converges in probability to the solution of a continuous-time deterministic differential equation. The epidemic control problem we consider is derived from this continuous-time deterministic differential equation.

In this problem, the state variable $x_i(t)$ represents the fraction of infected people in subpopulation i and $u_i(t)$ represents the intervention effort for subpopulation i at time t. Let C_i be the unit social cost per fraction of infected population, and P_i be the unit intervention cost. The transmission rates $\lambda_{1,i}$ and $\lambda_{0,i}$ correspond to active and inactive intervention cases in subpopulation i, respectively. Similarly, $\mu_{1,i}$ and $\mu_{0,i}$ denote the recovery rates under active and inactive intervention, respectively. The objective is to minimize the total cost $\int_0^T \sum_{i=1}^n [C_i(t)x_i(t) + P_i u_i(t)]\, dt$ subject to the state equation $\dot{x}_i(t) = u_i(t)\left[\lambda_{1,i}x(t)\left(1 - \frac{\mu_1}{\lambda_{1,i}} - x_i(t)\right)\right] + (1-u_i(t))\left[\lambda_{0,i}x_i(t)\left(1 - \frac{\mu_0}{\lambda_{0,i}} - x(t)\right)\right]$. In this model, the state trajectory always stays in $(0,1)^n$.

To generate a problem, we make two assumptions. First, we assume that in each subpopulation i, the intervention cost is always lower than the cost of infection. Second, $\lambda_{1,i} < \mu_{1,i}$ and $\lambda_{0,i} > \mu_{0,i}$, to reflect the effects of active intervention. To implement these assumptions, we first sample $\boldsymbol{C}$ uniformly at random from the interval $[0,1]^n$. We then multiply $\boldsymbol{C}$ element-wise with another vector sampled uniformly at random from $[0,1]^n$ to compute $\boldsymbol{P}$. Next, we sample $\boldsymbol{\lambda}_1$ and $\boldsymbol{\mu}_0$ uniformly at random from $[2,4]^n$. Finally, we

randomly sample two vectors uniformly from $[0, 0.5]^n$ and add them to $\boldsymbol{\lambda}_1$ and $\boldsymbol{\mu}_0$ to obtain $\boldsymbol{\mu}_1$ and $\boldsymbol{\lambda}_0$, respectively.

Fisheries Control

This problem is based on the classic logistic model of population growth [8], extended to the optimal control of fisheries studied by [97], [98]. In this example, $x_i(t)$ and $u_i(t)$ represent the size of population i at time t and the fishing effort on population i at time t, respectively. r_i denotes the intrinsic growth rate of population i, and H_i is the maximum sustainable population size. For each population i, its catchability coefficient, the unit price of landed fish, and the unit cost of effort are denoted q_i, p_i, and C_i, respectively. The objective is to maximize $\int_0^T \sum_{i=1}^n (p_i q_i x_i(t) - C_i) u_i(t)\, dt$ subject to the state equation:

$$\dot{x}_i(t) = r_i \left(1 - \frac{x_i(t)}{H_i}\right) x_i(t) - q_i x_i(t) u_i(t).$$

To align this problem with the formulation in Section 3.3.1, we can set the parameters as $\beta_{0,i} = \beta_{1,i} = -\frac{r_i}{H_i}$, $\alpha_{0,i} = r_i$, and $\alpha_{1,i} = r_i - q_i$. In this model, the state trajectory always stays in $\prod_{i=1}^n (0, H_i)$.

To generate a problem, we sample the parameters $\boldsymbol{r}, \boldsymbol{H}, \boldsymbol{q}, \boldsymbol{p}, \boldsymbol{C}$ uniformly at random from the intervals $[0, 0.15]^n, [1, 6]^n, [0, 0.15]^n, [0, 2]^n, [0, 0.1]^n$, respectively.

3.5.2 Experimental Setups

We randomly generate problem instances with varying initial states and solve them using Algorithm 2 to generate training data. We sample 3000 initial states uniformly at random from $\prod_{i=1}^n (0, H_i)$, where H_i represents the bounds specific to each problem. For each solved instance, we divide each subinterval with constant control into 10 different time grids, extracting the associated feature vectors and the control vectors. For Algorithm 2, we fix $m = \lfloor 0.3n \rfloor$, $\epsilon = 0.00001$, and $\delta = 0.0001$.

We evaluate our approach in two different ways. First, we measure simply how well the learned policy imitates the optimal trajectory generated by Algorithm 2. We evaluate the out-of-sample test accuracy on 1000 data points consisting of state-control pairs that the decision tree has not seen during training.

Second, to evaluate the ultimate quality of the learned policies, we apply them to problems with varying initial states and compute the associated objective cost. We compare this cost with the optimal objective cost acquired by Algorithm 2. To compute the objective cost associated with a policy, we discretize the dynamics to approximate the continuous-time trajectory and the integral objective values. We measure the suboptimality of a policy by subtracting the associated objective cost from the optimal objective cost, and then dividing the result by the absolute value of the associated objective cost. As the goal is maximization, this value is guaranteed to be non-negative. We report the maximum of these values across the test instances to analyze the potential suboptimality of the learned policies for unseen problems. The number of test instances is 100.

When measuring the speed-up compared to the shooting method, we divide the time it takes to solve an instance from scratch using Algorithm 2 by the time it takes for the trained

n	T	Training Time (min:s)	Speed-up	Accuracy	Max Suboptimality
5	1	2:23	47625	1.00	0.0000
10		4:50	93751	1.00	0.0000
5	5	6:49	4.90×10^5	1.00	0.0000
10		19:27	3.23×10^6	0.98	0.0181

Table 3.1: Experiment results for the machine maintenance problem.

decision tree to make an inference. The number of test instances is 100 for this. Finally, we report the time required to train a decision tree using OCT-H, with training times reported in minutes and seconds.

All experiments in this section were conducted on a MacBook Pro with an Apple M2 Pro chip and 16GB of memory. Software for OCT-H is available at [60]. We tune the maximum depth of the tree by grid searching over the list [5,10,15].

3.5.3 Experimental Results

In Tables 3.1, 3.2, and 3.3, we report the experimental results for the machine maintenance, the epidemic control, and the fisheries control problems, respectively. We draw the following conclusions.

Observations

- The out-of-sample classification accuracy is consistently high, never falling below 98% and frequently achieving 100%. This indicates that the policy learned using Algorithm 4 imitates the optimal trajectory very well.
- Even when the out-of-sample test accuracy falls below 100%, the maximum suboptimality remains very low, never exceeding 1.8%.
- The proposed approach significantly outspeeds Algorithm 2, with speed-ups reaching up to more than 26 million times. We also note that the relative speed-up increases with larger n and T. This enhancement is attributed to the increased computational demand of solving problems with larger number of projects and longer time horizons.
- The training times are typically short, up to several hours in a personal laptop. demonstrating the practicality of our approach.

3.6 Conclusions

We have proposed a machine learning approach to learn a state feedback policy for fluid restless multi-armed bandit problems with affine and quadratic state equations. Instead of relying on black-box algorithms that automatically learn nonlinear patterns, we introduce a feature augmentation technique and use OCT-H. Our computational experiments demonstrate

n	T	Training Time (min:s)	Speed-up	Accuracy	Max Suboptimality
5	1	7:33	2.10×10^5	0.99	0.0011
10		12:33	1.08×10^6	1.00	0.0000
5	5	7:04	8.60×10^5	0.99	0.0000
10		18:07	2.65×10^7	1.00	0.0000

Table 3.2: Experiment results for the epidemic control problem.

n	T	Training Time (min:s)	Speed-up	Accuracy	Max Suboptimality
5	1	34:24	90653	0.98	0.0000
10		51:29	1.59×10^5	0.99	0.0000
5	5	12:40	3.90×10^5	0.99	0.0011
10		143:20	2.07×10^6	0.98	0.0111

Table 3.3: Experiment results for the fisheries control problem.

that this approach effectively learns high-quality policies for a variety of problems across different sizes and time horizons.

Chapter 4

A Machine Learning Approach to Two-Stage Adaptive Robust Optimization

4.1 Introduction

Robust optimization (RO) has become increasingly popular as a method to account for parameter uncertainty. Compared to more conventional methods such as stochastic optimization, which can be computationally intensive in high dimensions, RO offers a significant computational advantage [17], [22].

Adaptive robust optimization (ARO) is an important extension of RO that allows certain decision variables, referred to as the wait-and-see variables, to be determined after the uncertainty is revealed. In ARO, the wait-and-see decisions are mathematically modeled as functions of uncertain parameters, enabling them to adapt to the realization of those parameters. ARO is particularly useful in multi-stage decision-making problems, where decision-makers may be uncertain about future parameter values, and where decisions may need to be made sequentially over time. Compared to RO, ARO provides greater modeling flexibility and often results in superior solutions that are better able to adapt to changing conditions [16]. Application areas include energy [23], [106], inventory management [5], [99], portfolio management [50], machine scheduling [42] among many others.

Despite its many benefits, ARO poses significant computational challenges that distinguish it from RO. One of the primary challenges arises from the fact that ARO consists of infinite-dimensional optimization problems, as the wait-and-see variables are functional variables. To overcome this issue, approximation methods have been proposed that restrict the wait-and-see variables to a limited set of functions, such as affine functions [22, Section 7]. However, while these methods may be able to reformulate ARO into RO, there is no guarantee that the resulting approximation will be near-optimal or even feasible [111, Section 5]. Other methods have been developed that can ensure near-optimal or even optimal solutions for ARO, including Benders Decomposition [23], Column and Constraint Generation (CCG) [112], scenario reduction [54], [108], branch-and-bound [70] and Fourier-Motzkin Elimination [114]. These methods, however, may not scale well in high dimensions or assume specific structure on the uncertainties. Given the substantial computational burden of ARO, the application of ARO may be limited particularly in real-time settings where time and computational

resources are severely constrained. In these domains where decisions need to be made within seconds or even milliseconds, opting for ARO can be often unviable.

To address this challenge, we propose a novel machine learning approach that can significantly reduce the computational burden associated with ARO. To generate a training set, we solve multiple ARO instances in advance using the CCG algorithm. Then, we train machine learning models to predict high-quality strategies for ARO problems that can simplify their solution process (exact definition of strategies will be presented in Section 4.3.1). To the best of our knowledge, our work is the first to harness the power of machine learning to tackle ARO. While our approach might involve heavy computational burden in the training phase, this investment enables us to attain remarkable speed-ups once the training is completed, outperforming state-of-the-art algorithms by several orders of magnitude. In practical terms, this means that ARO can now be solved in a matter of milliseconds.

While previous work by [24], [25], [28] explored machine learning techniques for solving mixed-integer convex optimization (MICO) problems, our work takes a leap by extending these methods to ARO. Their approach is to train a machine learning model that predicts the optimal strategy of a MICO problem, where the optimal strategy of a MICO instance is defined as the set of tight constraints and the set of binary variables that are equal to one.

However, adapting these methods to the realm of ARO is not straightforward. ARO presents distinct mathematical structures and computational challenges compared to MICO. First, while MICO deals with finite-dimensional problems, ARO deals with infinite-dimensional ones. Consequently, it requires a different definition of the optimal strategies to encode the optimal solutions. Second, ARO, involving dynamic optimization problems, requires a comprehensive solution including not only here-and-now decisions but also worst-case scenarios associated with these decisions and subsequent wait-and-see decisions. Third, the computational complexity of solving ARO is substantially higher than MICO, posing challenges in training data generation. Finally, in the realm of ARO, the number of distinct target classes can grow significantly depending on the sampling strategy and the size of the uncertainty sets, making the prediction task difficult. Our proposed techniques effectively address these challenges inherent in ARO. Furthermore, unlike previous approaches, the machine learning models we train can handle problems with varying dimensions, thereby enhancing their versatility.

Our contributions are summarized as follows.

1. We propose a machine learning approach to solve two-stage linear ARO with binary here-and-now variables and polyhedral uncertainty sets. Our approach provides a comprehensive solution including high-quality here-and-now decisions, worst-case scenarios associated with these decisions and subsequent wait-and-see decisions. Moreover, the machine learning models can be applied to problems with varying dimensions.

2. We propose a method to expedite the training data generation process, enhancing our approach's adaptability to shifting environments.

3. We propose a method to reduce the number of distinct target classes the machine learning model is trained on. This technique enables our approach to effectively address high-dimensional problems and accommodate large uncertainty sets.

4. We conduct a series of computational experiments involving both synthetic and real-world problems. The examples we test on include the facility location, the multi-item inventory control and the unit commitment problems. We demonstrate that we can obtain high-quality solutions using the proposed method. Notably, despite potentially lengthy training periods, the real-time application of our methodology dramatically outspeeds the state-of-the-art algorithms. In our experiments, we demonstrate a speed-up of more than 10 million times.

The structure of this paper is as follows. In Section 4.2, we briefly introduce ARO and explain how we solve a two-stage linear ARO problem with polyhedral uncertainty sets. We also demonstrate that this method may not scale to problems with high dimension. In Section 4.3, we develop a machine learning approach to solve two-stage ARO with binary here-and-now variables and polyhedral uncertainty sets. In Section 4.4, we present a technique to accelerate training data generation. In Section 4.5, we present an algorithm to reduce the number of different target classes. In Section 4.6, we present the results of numerical experiments.

Notational Conventions Throughout this paper, we use boldface letters to denote vectors and matrices. The i_{th} entry of a vector $\boldsymbol{x}$ is denoted x_i or $[\boldsymbol{x}]_i$, unless otherwise noted. For a positive integer N, we use $[N]$ to denote the set $\{i \in \mathbb{Z} : 1 \leq i \leq N\}$. We use $\boldsymbol{x}(\cdot)$ to denote a vector whose entries are real-valued functions.

4.2 Two-stage Linear Adaptive Robust Optimization

In this section, we review two-stage linear ARO. We describe how to obtain the optimal here-and-now decisions, the worst-case scenarios associated with the optimal here-and-now decisions, and the optimal wait-and-see decisions using the CCG algorithm. We demonstrate numerically that this algorithm may not scale well with dimension. Additional computational analysis of the algorithm, including the impact of tolerance parameters and the choice of different initial points can be found in the Appendix B.

4.2.1 Problem Formulation

Consider the two-stage ARO

$$\begin{aligned} \min_{\boldsymbol{x}, \boldsymbol{y}(\cdot)} \quad & \left(\max_{\boldsymbol{d} \in \mathcal{D}} \boldsymbol{c}(\boldsymbol{d})^\top \boldsymbol{x} + \boldsymbol{b}^\top \boldsymbol{y}(\boldsymbol{d}) \right) \\ \text{s.t.} \quad & \boldsymbol{A}(\boldsymbol{d})\boldsymbol{x} + \boldsymbol{B}\boldsymbol{y}(\boldsymbol{d}) \leq \boldsymbol{g}, \quad \forall \boldsymbol{d} \in \mathcal{D}, \end{aligned} \tag{4.1}$$

where $\boldsymbol{d}$ is a vector of uncertain parameters and $\mathcal{D}$ is a polyhedral uncertainty set. We also use the term scenario to refer to a specific realization of the uncertain parameter. $\boldsymbol{x}$ is the vector of here-and-now variables that represents the decisions that have to be made before the uncertainty is revealed. $\boldsymbol{y}(\cdot)$ is the vector of wait-and-see variables, which is a function of $\boldsymbol{d}$. $\boldsymbol{A}(\boldsymbol{d})$ and $\boldsymbol{c}(\boldsymbol{d})$ are affine in $\boldsymbol{d}$. We assume $\boldsymbol{B}$ and $\boldsymbol{b}$ do not involve uncertainty, a condition

commonly known as the fixed-recourse assumption. Without this assumption, solving an ARO instance becomes considerably more challenging [22, Chapter 6&7].

The wait-and-see variables represent the decisions that can be made after the uncertainty is revealed. This flexibility leads to less conservative and more realistic solutions compared to RO. ARO results in better objective values because the wait-and-see variables can be decided based on actual realizations of the uncertain parameters, whereas in RO, conservative decisions must be made in advance. Moreover, ARO tolerates larger uncertainty levels than RO. In some cases, a RO problem can be infeasible if the uncertainty set is too large. However, by switching some of the decision variables to wait-and-see variables, the resulting ARO problem may become feasible [16, Section 1&5].

As the wait-and-decision variable $\boldsymbol{y}(\cdot)$ is an arbitrary function, it represents an infinite-dimensional variable. To manage this challenge, a common approach is to constrain $\boldsymbol{y}(\cdot)$ to a family of parametric functions. Among the popular choices is the affine decision rule, where we assume that $\boldsymbol{y}(\boldsymbol{d}) = \boldsymbol{P}\boldsymbol{d} + \boldsymbol{z}$ for some $\boldsymbol{P}$ and $\boldsymbol{z}$. By substituting this expression back into problem (4.1), $\boldsymbol{P}$ and $\boldsymbol{z}$ become finite vectors of decision variables alongside $\boldsymbol{x}$.

Another approximation method is to consider only a limited number of key scenarios from $\mathcal{D}$. In this approach, $\mathcal{D}$ is replaced with its finite subset in problem (4.1). For each scenario $\boldsymbol{d}$ in the subset, we define a wait-and-see variable $\boldsymbol{y_d}$, representing the action to be taken if scenario $\boldsymbol{d}$ is realized. The CCG algorithm falls into this class of methods.

4.2.2 Column and Constraint Generation Algorithm

The CCG algorithm is an iterative algorithm to solve problem (4.1) to near optimality. The first step of this algorithm is to reformulate the objective function as a function of here-and-now variables. Problem (4.1) can be reformulated as

$$\min_{\boldsymbol{x}} \quad \left(\max_{\boldsymbol{d}\in\mathcal{D}} \min_{\boldsymbol{y}\in\Omega(\boldsymbol{x},\boldsymbol{d})} \boldsymbol{c}(\boldsymbol{d})^\top \boldsymbol{x} + \boldsymbol{b}^\top \boldsymbol{y} \right), \tag{4.2}$$

with $\Omega(\boldsymbol{x}, \boldsymbol{d}) = \{\boldsymbol{y} : \boldsymbol{A}(\boldsymbol{d})\boldsymbol{x} + \boldsymbol{B}\boldsymbol{y} \leq \boldsymbol{g}\}$. We also define

$$\mathcal{Q}(\boldsymbol{x}) = \max_{\boldsymbol{d}\in\mathcal{D}} \min_{\boldsymbol{y_d}\in\Omega(\boldsymbol{x},\boldsymbol{d})} \boldsymbol{c}(\boldsymbol{d})^\top \boldsymbol{x} + \boldsymbol{b}^\top \boldsymbol{y_d}, \tag{4.3}$$

which is the objective value corresponding to a here-and-now decision $\boldsymbol{x}$. The solution to the outer maximization problem in (4.3) is the worst-case scenario associated with $\boldsymbol{x}$.

Since a worst-case scenario is an extreme point of the uncertainty set, in the inner maximization of problem (4.2), it suffices to optimize over the extreme points of $\mathcal{D}$ rather than the entire set. Hence, problem (4.2) is equivalent to the following problem.

$$\begin{aligned} \min_{\boldsymbol{x},\boldsymbol{\alpha},\{\boldsymbol{y_d}\}_{\boldsymbol{d}\in\mathcal{E}}} \quad & \alpha \\ \text{s.t.} \quad & \alpha \geq \boldsymbol{c}(\boldsymbol{d})^\top \boldsymbol{x} + \boldsymbol{b}^\top \boldsymbol{y_d}, \quad \forall \boldsymbol{d} \in \mathcal{E}, \\ & \boldsymbol{y_d} \in \Omega(\boldsymbol{x}, \boldsymbol{d}), \quad \forall \boldsymbol{d} \in \mathcal{E}, \end{aligned} \tag{4.4}$$

where $\mathcal{E}$ is the set of all extreme points of $\mathcal{D}$ and $\boldsymbol{y_d}$ is the wait-and-see variable associated with $\boldsymbol{d}$.

Without using the entire set $\mathcal{E}$, CCG solves (4.4) by iteratively adding a new extreme point $\boldsymbol{d}$ and the associated wait-and-see variable $\boldsymbol{y_d}$ until a convergence criterion is met. In the initial iteration, where no extreme point is identified yet ($\mathcal{E}_0 = \emptyset$), we solve (4.3) with any initial here-and-now decision $\boldsymbol{x}_0$ to find the associated worst-case scenario and then add this scenario to $\mathcal{E}_0$. Iteration i of the CCG algorithm involves i extreme points identified so far. Denoting the set of extreme points at iteration i as $\mathcal{E}_i$, the so-called restricted master problem at iteration i is

$$\begin{aligned} \min_{\boldsymbol{x},\boldsymbol{\alpha},\{\boldsymbol{y_d}\}_{\boldsymbol{d}\in\mathcal{E}_i}} \quad & \alpha \\ \text{s.t.} \quad & \alpha \geq \boldsymbol{c}(\boldsymbol{d})^\top \boldsymbol{x} + \boldsymbol{b}^\top \boldsymbol{y_d}, \quad \forall \boldsymbol{d} \in \mathcal{E}_i, \\ & \boldsymbol{y_d} \in \Omega(\boldsymbol{x}, \boldsymbol{d}), \quad \forall \boldsymbol{d} \in \mathcal{E}_i. \end{aligned} \tag{4.5}$$

The objective value of (4.5) is lower than the objective value of (4.4), as only a subset of the constraints are imposed. Once we solve (4.5) and obtain its solution $\boldsymbol{x}_i$, we calculate $\mathcal{Q}(\boldsymbol{x}_i)$ and also obtain the worst-case scenario $\boldsymbol{d}_i$ associated with $\boldsymbol{x}_i$. If the gap between $\mathcal{Q}(\boldsymbol{x}_i)$ and the objective value of (4.5) is larger than a convergence criterion, $\boldsymbol{d}_i$ is added to $\mathcal{E}_i$ from the next iteration.

In every iteration, we have to evaluate $\mathcal{Q}(\boldsymbol{x}_i)$, which is a non-convex max-min problem as shown in (4.3). Several methods have been proposed to solve this problem, such as converting it to a mixed integer linear optimization problem [105], [112]. In our implementation, we use a heuristic called the Alternating Direction method due to its computational efficiency and strong empirical performance.

Using the strong duality in linear optimization, the inner minimization problem in (4.3) can be converted to a maximization problem. Now the problem (4.3) is recast into the following maximization problem.

$$\begin{aligned} \max_{\boldsymbol{d},\boldsymbol{\pi}} \quad & \boldsymbol{\pi}^\top (\boldsymbol{A}(\boldsymbol{d})\boldsymbol{x} - \boldsymbol{g}) + \boldsymbol{c}(\boldsymbol{d})^\top \boldsymbol{x} \\ \text{s.t.} \quad & -\boldsymbol{\pi}^\top \boldsymbol{B} = \boldsymbol{b}^\top, \\ & \boldsymbol{\pi} \geq \boldsymbol{0}, \\ & \boldsymbol{d} \in \mathcal{D}. \end{aligned} \tag{4.6}$$

The Alternating Direction method to solve problem (4.6) is described in Algorithm 6. For conciseness, we use $\mathcal{T} = \{\boldsymbol{\pi} \mid -\boldsymbol{\pi}^\top \boldsymbol{B} = \boldsymbol{b}^\top, \boldsymbol{\pi} \geq 0\}$ in the algorithm description. Theoretically, CCG can output suboptimal solutions precisely because problem (4.3) is non-convex. Hence, in each iteration, we are in fact computing an approximation of $\mathcal{Q}(\boldsymbol{x}_i)$, which we denote as $\tilde{\mathcal{Q}}(\boldsymbol{x}_i)$. To ensure the quality of the solution, in our implementation we try three different initial points for problem (4.3) and choose the best solution found. For more detail on CCG method and the Alternating Direction method see [105], [106]. Algorithms 5 and 6 describe the CCG and the Alternating Direction method, respectively.

4.2.3 Obtaining the solutions

Given problem (4.1) and a scenario $\bar{\boldsymbol{d}}$, we can find a near-optimal here-and-now decision $\tilde{\boldsymbol{x}}^*$ and an associated worst-case scenario $\tilde{\boldsymbol{d}}^*$ using Algorithm 5. We can find a near-optimal

Algorithm 5: Column and Constraint Generation

Input: Problem (4.1), ϵ_1
Output: $\tilde{\boldsymbol{x}}^*$,$\tilde{\boldsymbol{d}}^*$
Initialization: $i = 0$, $\boldsymbol{x}_0$, $\mathcal{E}_0 = \emptyset$, UB $= \infty$, LB $= -\infty$
while *UB -LB* $\geq \epsilon_1$ **do**
  if $i = 0$ **then**
    Evaluate $\mathcal{Q}(\boldsymbol{x}_i)$ to get $\tilde{\mathcal{Q}}(\boldsymbol{x}_i)$ and a solution $\boldsymbol{d}_i$
    $\mathcal{E}_{i+1} \leftarrow \mathcal{E}_i \bigcup \{\boldsymbol{d}_i\}$
    $i \leftarrow i + 1$
  else
    Solve (4.5) with the extreme points in $\mathcal{E}_i$. Denote the solutions as $\boldsymbol{x}_i$ and $\boldsymbol{\alpha}_i$.
    LB $\leftarrow \boldsymbol{\alpha}_i$
    Evaluate $\mathcal{Q}(\boldsymbol{x}_i)$ to get $\tilde{\mathcal{Q}}(\boldsymbol{x}_i)$ and a solution $\boldsymbol{d}_i$.
    UB $\leftarrow \tilde{\mathcal{Q}}(\boldsymbol{x}_i)$
    $\mathcal{E}_{i+1} \leftarrow \mathcal{E}_i \bigcup \{\boldsymbol{d}_i\}$
    $i \leftarrow i + 1$
$\tilde{\boldsymbol{x}}^* \leftarrow \boldsymbol{x}_i$
$\tilde{\boldsymbol{d}}^* \leftarrow \boldsymbol{d}_i$

Algorithm 6: Alternating Direction Method

Input: Problem (4.3), $\boldsymbol{x}_i, \epsilon_2$
Output: $\tilde{\mathcal{Q}}(\boldsymbol{x}_i)$, $\boldsymbol{d}_i$
Initialization: Some $\boldsymbol{d}_0 \in \mathcal{D}, t = 0$, UB $= \infty$, LB $= -\infty$
while *UB -LB* $\geq \epsilon_2$ **do**
  LB $\leftarrow (a)\ \max_{\boldsymbol{\pi} \in \mathcal{T}} \boldsymbol{\pi}^\top (\boldsymbol{A}(\boldsymbol{d}_t)\boldsymbol{x}_i - \boldsymbol{g}) + \boldsymbol{c}(\boldsymbol{d}_t)^\top \boldsymbol{x}_i$
  Denote the solution of (a) as $\boldsymbol{\pi}_t$.
  UB $\leftarrow (b)\ \max_{\boldsymbol{d} \in \mathcal{D}} \boldsymbol{\pi}_t^\top (\boldsymbol{A}(\boldsymbol{d})\boldsymbol{x}_i - \boldsymbol{g}) + \boldsymbol{c}(\boldsymbol{d})^\top \boldsymbol{x}_i$
  Denote the solution of (b) as $\boldsymbol{d}_t$.
  $t \leftarrow t + 1$
$\tilde{\mathcal{Q}}(\boldsymbol{x}_i) \leftarrow \frac{\text{UB+LB}}{2}$
$\boldsymbol{d}_i \leftarrow \boldsymbol{d}_t$

wait-and-see decision $\tilde{\boldsymbol{y}}^*(\bar{\boldsymbol{d}})$ by fixing $\boldsymbol{x} = \tilde{\boldsymbol{x}}^*$, $\boldsymbol{d} = \bar{\boldsymbol{d}}$ and solving the deterministic version of (4.1), which is the following problem.

$$\begin{aligned} \min_{\boldsymbol{y}} \quad & \boldsymbol{c}(\bar{\boldsymbol{d}})^\top \tilde{\boldsymbol{x}}^* + \boldsymbol{b}^\top \boldsymbol{y} \\ s.t. \quad & \boldsymbol{A}(\bar{\boldsymbol{d}})\tilde{\boldsymbol{x}}^* + \boldsymbol{B}\boldsymbol{y} \leq \boldsymbol{g}. \end{aligned}$$

Note that the decision variable $\boldsymbol{y}$ is no longer a functional variable but a finite vector of decision variables, because the specific scenario $\bar{\boldsymbol{d}}$ has been realized.

4.2.4 Scalability

We demonstrate that solving ARO problems using Algorithm 5 can be computationally demanding by considering the unit commitment problem from power systems literature. This problem involves minimizing energy production costs while satisfying energy demand over m time steps for a power system with n generators. We should decide which generators to start up, shut down, and how much energy each generator should produce at each time. Whether we should start up or shut down each generator at each time, referred to as the commitment decisions, are modeled as binary variables. In the ARO version, these variables represent the here-and-now decisions made before the demand is realized. The demand at each time is the uncertain parameter. After the demand is realized, the wait-and-see decisions determine how much energy each generator should produce. Appendix B provides the complete formulation of the deterministic version, and the data used is from [37]. We use the budget uncertainty set defined as $\mathcal{D} = \{\boldsymbol{d} \mid \sum_i \left|\frac{d_i - \bar{d}_i}{0.1 \times \bar{d}_i}\right| \leq 2, \left|d_i - \bar{d}_i\right| \leq 0.1 \times \bar{d}_i\}$, where $\bar{\boldsymbol{d}}$ is from the data in [37].

For varying values of n, we compare the solve times of ARO and the deterministic version of the unit commitment problem. We keep $m = 24$ fixed for all experiments. For each n, we generate 100 deterministic instances and solve them with Gurobi [56] using the optimality gap of 0.01. Then, we solve the ARO versions of these problems using Algorithm 5, with tolerance set to $\epsilon_1 = 0.01$ for fair comparison. For Algorithm 6 implemented within Algorithm 5, three random initial points are used and the tolerance is set to $\epsilon_2 = 0.01$. The experiment in this section was executed in Julia 1.4.1 on a MacBook Pro with 2.6 GHz Intel Core i7 CPU and 16GB of RAM.

We present the experiment results in Figure 4.1, where we report the mean solve times for each n. It is evident that compared to solving the deterministic version of the unit commitment problem, considerably more time is required to solve its ARO counterpart using Algorithm 5. The solve time for ARO increases drastically at $n = 5$, while the solve time for the deterministic version remains relatively consistent across all n values. This finding suggests that solving ARO problems using Algorithm 5 can be computationally challenging, especially for large scale problems.

4.3 A Machine Learning Approach to ARO

In this section, we develop a machine learning approach to two-stage ARO with binary here-and-now variables and polyhedral uncertainty sets. In the context of MICO, [24], [25] use classification algorithms, while [28] use a prescriptive machine learning algorithm, Optimal

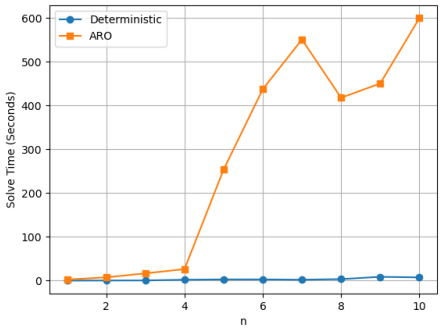

Figure 4.1: Solve time for the unit commitment problem.

Policy Trees (OPT) [3]. [28] demonstrate that OPT has an edge over a classification algorithm, particularly in its ability to avoid infeasible or highly suboptimal solutions. In our work, we provide an integrated perspective by introducing both classification and prescriptive approaches in the context of ARO. We begin by presenting a comprehensive explanation of our foundational approach from Section 4.3.1 to 4.3.5. Following this, we introduce a generalization that extends the applicability of machine learning models to problems with varying dimensions in Section 4.3.6.

4.3.1 Optimal Strategy

We consider the following ARO problem, where $\boldsymbol{\theta}$ is the key parameter used to generate instances.

$$\begin{aligned} \min_{\boldsymbol{x},\boldsymbol{y}(\cdot)} \quad & \left(\max_{\boldsymbol{d}\in\mathcal{D},} \boldsymbol{c}(\boldsymbol{d},\boldsymbol{\theta})^\top \boldsymbol{x} + \boldsymbol{b}(\boldsymbol{\theta})^\top \boldsymbol{y} \right) \\ \text{s.t.} \quad & \boldsymbol{A}(\boldsymbol{d},\boldsymbol{\theta})\boldsymbol{x} + \boldsymbol{B}(\boldsymbol{\theta})\boldsymbol{y}(\boldsymbol{d}) \le \boldsymbol{g}(\boldsymbol{\theta}), \quad \forall \boldsymbol{d} \in \mathcal{D}, \\ & \boldsymbol{x} \text{ is binary.} \end{aligned} \tag{4.7}$$

We denote the deterministic version of problem (4.7) with fixed $\boldsymbol{x} = \boldsymbol{x}^*, \boldsymbol{d} = \boldsymbol{d}^*$ as $Det(\boldsymbol{\theta}, \boldsymbol{x}^*, \boldsymbol{d}^*)$, which is the following problem.

$$\begin{aligned} \min_{\boldsymbol{y}} \quad & \boldsymbol{c}(\boldsymbol{d}^*,\boldsymbol{\theta})^\top \boldsymbol{x}^* + \boldsymbol{b}(\boldsymbol{\theta})^\top \boldsymbol{y} \\ \text{s.t.} \quad & \boldsymbol{A}(\boldsymbol{d}^*,\boldsymbol{\theta})\boldsymbol{x}^* + \boldsymbol{B}(\boldsymbol{\theta})\boldsymbol{y} \le \boldsymbol{g}(\boldsymbol{\theta}). \end{aligned} \qquad (Det(\boldsymbol{\theta}, \boldsymbol{x}^*, \boldsymbol{d}^*))$$

We denote the optimal objective cost of $Det(\boldsymbol{\theta}, \boldsymbol{x}^*, \boldsymbol{d}^*)$ as $V(\boldsymbol{\theta}, \boldsymbol{x}^*, \boldsymbol{d}^*)$ and assume that $Det(\boldsymbol{\theta}, \boldsymbol{x}^*, \boldsymbol{d}^*)$ has m constraints.

Given the ARO instance (4.7) with a parameter $\bar{\boldsymbol{\theta}}$ and a scenario $\bar{\boldsymbol{d}} \in \mathcal{D}$, we define the optimal strategy for the here-and-now decisions, the worst-case scenarios associated with the optimal here-and-now decisions, and the wait-and-see decisions associated with the scenario $\bar{\boldsymbol{d}}$. We denote these strategies as $s_{\boldsymbol{x}}(\bar{\boldsymbol{\theta}}), s_{\boldsymbol{d}}(\bar{\boldsymbol{\theta}}), s_{\boldsymbol{y}}(\bar{\boldsymbol{\theta}}, \bar{\boldsymbol{d}})$, respectively. We use $s_{\boldsymbol{x}}, s_{\boldsymbol{d}}, s_{\boldsymbol{y}}$ instead when we are not referring to a specific instance or scenario. These strategies serve as the prediction targets for the trained machine learning models. In the following description, we

use $\boldsymbol{x}^*, \boldsymbol{d}^*$ to denote the optimal here-and-now decision and the worst-case scenario associated with the optimal here-and-now decision, respectively.

Here-and-now Decisions The here-and-now variables in problem (4.7) are binary. Therefore, we define the optimal strategy for the here-and-now decisions as the optimal here-and-now decision itself, meaning that $s_{\boldsymbol{x}}(\bar{\boldsymbol{\theta}}) = \boldsymbol{x}^*$. Once a machine learning model is trained, it can directly predict a here-and-now decision given a new parameter $\hat{\boldsymbol{\theta}}$.

Worst-case Scenarios In general, it is hard to define a single worst-case scenario of an ARO instance. However, if we fix some here-and-now decision, the worst-case scenario that corresponds to this specific decision can be defined. We define the optimal strategy for the worst-case scenarios as $s_{\boldsymbol{d}}(\bar{\boldsymbol{\theta}}) = (\boldsymbol{x}^*, \boldsymbol{d}^*)$. Note that since a worst-case scenario is one of the extreme points of $\mathcal{D}$, there can only be a finite number of worst-case scenarios possible. Once a machine learning model is trained, it can directly predict a here-and-now decision and a worst-case scenario given a new parameter $\hat{\boldsymbol{\theta}}$.

Wait-and-see Decisions Given $\boldsymbol{x}^*$ and the scenario $\bar{\boldsymbol{d}}$, we solve $Det(\bar{\boldsymbol{\theta}}, \boldsymbol{x}^*, \bar{\boldsymbol{d}})$ to identify the optimal solution $\boldsymbol{y}^*$ and the set of constraints that are satisfied as equality at optimality. These constraints are referred to as the tight constraints and are denoted as $\tau_{\boldsymbol{y}}(\bar{\boldsymbol{\theta}}, \bar{\boldsymbol{d}})$. Formally, they are defined as

$$\tau_{\boldsymbol{y}}(\bar{\boldsymbol{\theta}}, \bar{\boldsymbol{d}}) = \{j \in [m] \mid [\boldsymbol{A}(\bar{\boldsymbol{d}}, \bar{\boldsymbol{\theta}})\boldsymbol{x}^* + \boldsymbol{B}(\bar{\boldsymbol{\theta}})\boldsymbol{y}^*]_j = [\boldsymbol{g}(\bar{\boldsymbol{\theta}})]_j\}.$$

Identifying the tight constraints simplifies the linear programming problem, as unnecessary constraints can be removed. We define the optimal strategy as $s_{\boldsymbol{y}}(\bar{\boldsymbol{\theta}}, \bar{\boldsymbol{d}}) = (\boldsymbol{x}^*, \tau_{\boldsymbol{y}}(\bar{\boldsymbol{\theta}}, \bar{\boldsymbol{d}}))$. Given a new parameter $\hat{\boldsymbol{\theta}}$ and a scenario $\hat{\boldsymbol{d}}$, a model predicts a here-and-now decision $\hat{\boldsymbol{x}}$ and a set of tight constraints. A wait-and-see decision for the scenario $\hat{\boldsymbol{d}}$ can be computed by solving $Det(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{d}})$ imposing only the predicted tight constraints.

Notice that the optimal strategies for the worst-case-scenarios and the wait-and-see decisions already contain the optimal here-and-now decisions. Therefore, it might seem redundant to train a separate model to predict the optimal here-and-now decisions. However, we demonstrate in Section 4.6 that the prediction accuracy for the here-and-now decisions is generally higher than the other prediction targets. Hence, if one is only interested in predicting the optimal here-and-now decisions, training a separate model might be beneficial.

4.3.2 Suboptimality and Infeasibility

We explain how we evaluate the quality of the strategies applied to an ARO instance associated with $\hat{\boldsymbol{\theta}}$ and a scenario $\hat{\boldsymbol{d}}$. We denote the strategies that we would like to evaluate as $\hat{\boldsymbol{x}}, (\hat{\boldsymbol{x}}, \hat{\boldsymbol{d}}'), (\hat{\boldsymbol{x}}, \hat{\tau}_{\boldsymbol{y}})$, respectively.

Here-and-now Decisions Suboptimality and infeasibility of the strategy $\hat{\boldsymbol{x}}$ are measured by comparing $\tilde{\mathcal{Q}}(\hat{\boldsymbol{x}})$ and $\tilde{\mathcal{Q}}(\tilde{\boldsymbol{x}}^*)$ of the instance associated with $\hat{\boldsymbol{\theta}}$. We consider $\hat{\boldsymbol{x}}$ infeasible if $\tilde{\mathcal{Q}}(\hat{\boldsymbol{x}}) = \infty$. If it is feasible, we define its suboptimality as

$$sub(\hat{\boldsymbol{x}}) = (\tilde{\mathcal{Q}}(\hat{\boldsymbol{x}}) - \tilde{\mathcal{Q}}(\tilde{\boldsymbol{x}}^*))/\left|\tilde{\mathcal{Q}}(\tilde{\boldsymbol{x}}^*)\right|.$$

Worst-Case Scenarios Measuring the quality of the strategy $(\hat{\boldsymbol{x}}, \hat{\boldsymbol{d}}')$ consists of two stages. First, we evaluate the $\hat{\boldsymbol{x}}$ part following the procedure described above for the here-and-now decisions. If $\hat{\boldsymbol{x}}$ is infeasible, the strategy $(\hat{\boldsymbol{x}}, \hat{\boldsymbol{d}}')$ is considered infeasible in the first place. Otherwise, it is considered feasible. If it is feasible, then now we check if $\hat{\boldsymbol{d}}'$ is the worst-case scenario for $\hat{\boldsymbol{x}}$. To do so, we solve $Det(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{d}}')$ and compare the optimal cost with $\tilde{\mathcal{Q}}(\hat{\boldsymbol{x}})$. We define the suboptimality as

$$sub(\hat{\boldsymbol{x}}, \hat{\boldsymbol{d}}') = \max\left\{sub(\hat{\boldsymbol{x}}), \left(\tilde{\mathcal{Q}}(\hat{\boldsymbol{x}}) - V(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{d}}')\right) / \left|\tilde{\mathcal{Q}}(\hat{\boldsymbol{x}})\right|\right\}.$$

Wait-and-see Decisions Measuring the quality of the strategy $(\hat{\boldsymbol{x}}, \hat{\tau}_{\boldsymbol{y}})$ also consists of two stages. First, we evaluate the $\hat{\boldsymbol{x}}$ part following the procedure described above for the here-and-now decisions. If it is infeasible, then $(\hat{\boldsymbol{x}}, \hat{\tau}_{\boldsymbol{y}})$ is considered infeasible in the first place. If it is feasible, then we evaluate the $\hat{\tau}_{\boldsymbol{y}}$ part. We solve $Det(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{d}})$ imposing only the constraints included in $\hat{\tau}_{\boldsymbol{y}}$. If this leads to infeasibility, then again $(\hat{\boldsymbol{x}}, \hat{\tau}_{\boldsymbol{y}})$ is considered infeasible. If a feasible solution $\hat{\boldsymbol{y}}$ is found, we compute the suboptimality of the $\hat{\tau}_{\boldsymbol{y}}$ part defined as

$$sub(\hat{\tau}_{\boldsymbol{y}}) = \left(\left(\boldsymbol{c}(\hat{\boldsymbol{d}}, \hat{\boldsymbol{\theta}})^\top \hat{\boldsymbol{x}} + \boldsymbol{b}(\hat{\boldsymbol{\theta}})^\top \hat{\boldsymbol{y}}\right) - V(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{d}})\right) / V(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{d}}),$$

For a feasible $(\hat{\boldsymbol{x}}, \hat{\tau}_{\boldsymbol{y}})$, we define its suboptimality as

$$sub(\hat{\boldsymbol{x}}, \hat{\tau}_{\boldsymbol{y}}) = \max\left\{sub(\hat{\boldsymbol{x}}), sub(\hat{\tau}_{\boldsymbol{y}})\right\}.$$

4.3.3 A Classification Approach

In this section, we develop an approach to solve ARO problems using classification algorithms. The proposed approach consists of three phases. In Phase 1, we generate $N \in \mathbb{N}$ parameters $\{\boldsymbol{\theta}_i\}_{i\in[N]}$ and solve the associated ARO instances using Algorithm 5. For each $\boldsymbol{\theta}_i$, we also sample a scenario $\boldsymbol{d}_i$ from the uncertainty set. Then, we identify near-optimal or slightly suboptimal strategies for the here-and-now decisions, the worst-case scenarios associated with the here-and-now decisions, and the wait-and-see decisions associated with the scenario $\boldsymbol{d}_i$. In Phase 2, we use classification algorithms to train three machine learning models that predict each of these strategies. In Phase 3, given a new parameter $\hat{\boldsymbol{\theta}}$ and a scenario $\hat{\boldsymbol{d}}$, we use the predictions of these models to compute a here-and-now decision, a worst-case scenario, and a wait-and-see decision associated with the scenario $\hat{\boldsymbol{d}}$. Algorithm 7 provides a comprehensive overview of the entire procedure with detailed steps.

Remark 1 As mentioned in Section 4.2.2, solving ARO problems to exact optimality is hard in general. Hence, we use the outputs of Algorithm 5 to compute near-optimal or slightly suboptimal strategies instead in Phase 1. The suboptimalities of the strategies depend on the tolerance ϵ_1 in Algorithm 5.

Remark 2 In this work, we sample parameters $\boldsymbol{\theta}_i, i \in [N]$, uniformly at random from the ball with a predefined radius r. We sample scenarios $\boldsymbol{d}_i$ from the uncertainty set, also uniformly at random. Depending on the application area or any prior knowledge, the sampling scheme may vary.

Algorithm 7: Classification Approach to ARO

Input: $\bar{\boldsymbol{\theta}}$, N, r, Problem (4.7), ϵ_1, ϵ_2

Phase 1

1.1 **for** $i = 1, \ldots, N$ **do**

Sample a point $\boldsymbol{\theta}_i$ from the ball $\mathcal{B}(\bar{\boldsymbol{\theta}}, r)$ uniformly at random.
Fix $\boldsymbol{\theta} = \boldsymbol{\theta}_i$ and solve problem (4.7) with Algorithm 5 to obtain a here-and-now decision $\tilde{\boldsymbol{x}}_i^*$ and an associated worst-case-scenario $\tilde{\boldsymbol{d}}_i^*$. The tolerances for Algorithm 5 and 6 are set to ϵ_1 and ϵ_2, respectively.
Sample a point $\boldsymbol{d}_i$ from $\mathcal{D}$ uniformly at random.
Solve $Det(\boldsymbol{\theta}_i, \tilde{\boldsymbol{x}}_i^*, \boldsymbol{d}_i)$ to obtain $\tilde{\tau}_{\boldsymbol{y}}(\boldsymbol{\theta}_i, \boldsymbol{d}_i)$.
$s_{\boldsymbol{x}}(\boldsymbol{\theta}_i) \leftarrow \tilde{\boldsymbol{x}}_i^*$
$s_{\boldsymbol{d}}(\boldsymbol{\theta}_i) \leftarrow (\tilde{\boldsymbol{x}}_i^*, \tilde{\boldsymbol{d}}_i^*)$
$s_{\boldsymbol{y}}(\boldsymbol{\theta}_i, \boldsymbol{d}_i) \leftarrow (\tilde{\boldsymbol{x}}_i^*, \tilde{\tau}_{\boldsymbol{y}}(\boldsymbol{\theta}_i, \boldsymbol{d}_i))$

Phase 2

2.1 Train a machine learning model $\mathcal{L}_{\boldsymbol{x}}$ using $(\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_N)$ as the feature matrix and $\Big(s_{\boldsymbol{x}}(\boldsymbol{\theta}_1), \ldots, s_{\boldsymbol{x}}(\boldsymbol{\theta}_N)\Big)$ as the target vector.

2.2 Train a machine learning model $\mathcal{L}_{\boldsymbol{d}}$ using $(\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_N)$ as the feature matrix and $\Big(s_{\boldsymbol{d}}(\boldsymbol{\theta}_1), \ldots, s_{\boldsymbol{d}}(\boldsymbol{\theta}_N)\Big)$ as the target vector.

2.3 Train a machine learning model $\mathcal{L}_{\boldsymbol{y}}$ using $\Big((\boldsymbol{\theta}_1, \boldsymbol{d}_1), \ldots, (\boldsymbol{\theta}_N, \boldsymbol{d}_N)\Big)$ as the feature matrix and $\Big(s_{\boldsymbol{y}}(\boldsymbol{\theta}_1, \boldsymbol{d}_1), \ldots, s_{\boldsymbol{y}}(\boldsymbol{\theta}_N, \boldsymbol{d}_N)\Big)$ as the target vector.

Phase 3

3.1 For a new instance $\hat{\boldsymbol{\theta}}$, $\mathcal{L}_{\boldsymbol{x}}$ predicts $\hat{s}_{\boldsymbol{x}}(\hat{\boldsymbol{\theta}}) = \hat{\boldsymbol{x}}$.
3.2 For a new instance $\hat{\boldsymbol{\theta}}$, $\mathcal{L}_{\boldsymbol{d}}$ predicts $\hat{s}_{\boldsymbol{d}}(\hat{\boldsymbol{\theta}}) = (\hat{\boldsymbol{x}}, \hat{\boldsymbol{d}}')$.
3.3.1 For a new instance $\hat{\boldsymbol{\theta}}$ and $\hat{\boldsymbol{d}}$, $\mathcal{L}_{\boldsymbol{y}}$ predicts $\hat{s}_{\boldsymbol{y}}(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{d}}) = (\hat{\boldsymbol{x}}, \hat{\tau}_{\boldsymbol{y}}(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{d}}))$.
3.3.2 Solve $Det(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{d}})$ using $\hat{\tau}_{\boldsymbol{y}}(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{d}})$ to compute a wait-and-see decision.

4.3.4 A Prescriptive Approach

In this section, we present a prescriptive approach to ARO using OPT. We begin with a baseline approach and then generalize it. In the following explanation, we focus on training a machine learning model for the here-and-now variables, but the cases of worst-case scenarios and the wait-and-see decisions are straightforward extensions.

The baseline approach is similar to the classification approach provided in Algorithm 7, except for Phase 2. In Phase 2 of the prescriptive approach, we need to compute what we refer to as the reward matrices.

Assume that after Phase 1 of Algorithm 7, we have solved $N \in \mathbb{N}$ ARO instances and identified $Q \in \mathbb{N}$ different strategies in the training set. Note that $Q \leq N$, as the optimal

strategies of different instances might overlap. We let $\mathcal{S}_{\boldsymbol{x}} = \{s_{\boldsymbol{x},1}, \ldots, s_{\boldsymbol{x},Q}\}$ be the set of optimal strategies identified, where $s_{\boldsymbol{x},i} \neq s_{\boldsymbol{x},j}$ if $i \neq j$. The reward matrix $\boldsymbol{R}_{\boldsymbol{x}} \in \mathbb{R}^{N \times Q}$ is then defined such that its entry in the i_{th} row and j_{th} column corresponds to the suboptimality of the strategy $s_{\boldsymbol{x},j}$ applied to the i_{th} ARO instance. If the strategy is infeasible, we assign an arbitrary large number to its entry. Using the reward matrix, we train a decision tree by solving the optimization problem

$$\min_{v(\cdot),\boldsymbol{z}} \quad \sum_{i=1}^{N} \sum_{\ell} \mathbb{1}\{v(\boldsymbol{\theta}_i) = \ell\} \cdot R_{iz_\ell},$$

where $v(\boldsymbol{\theta}_i)$ is the leaf of the tree $\boldsymbol{\theta}_i$ is assigned to, $\boldsymbol{z}_\ell$ is the strategy assigned to the points in leaf ℓ, and R_{iz_ℓ} is the suboptimality of the instance i under the strategy z_ℓ. This optimization problem determines the structure of the decision tree using the decision variable $v(\cdot)$ and assigns strategies to each leaf using the decision variable $\boldsymbol{z}$. The objective is to train a decision tree that prescribes a strategy to an ARO instance, so that the resulting suboptimality is minimized. Once a decision tree is trained, given a feature vector $\hat{\boldsymbol{\theta}}$, we traverse the tree using the feature until we reach the leaf node. The strategy assigned to this leaf is the prediction for the instance $\hat{\boldsymbol{\theta}}$. For a more detailed explanation, please refer to [3].

Now, we introduce a generalization of the baseline approach just explained. In our computational experiments, we have observed that the number Q can get prohibitively large, especially for large scale problems. While we propose a method to address this issue when training a model for the wait-and-see decisions in Section 4.5, this method does not extend to other prediction targets.

In the generalization we propose, we randomly select $Q_1 \leq Q$ strategies from $\mathcal{S}_{\boldsymbol{x}}$, and compute the corresponding reward matrix. Using this reward matrix, we train a decision tree. Algorithm 8 outlines the entire procedure.

4.3.5 Example

In this section, we apply Algorithm 8 to a small sized example and present the actual decision trees learned with OPT. We consider the following facility location problem formulated as an ARO problem.

$$\begin{aligned}
\min_{\boldsymbol{y}(\cdot),\boldsymbol{x}} \max_{\boldsymbol{d}\in\mathcal{D}} \quad & \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij} y_{ij}(\boldsymbol{d}) + \sum_{i=1}^{n} f_i x_i & \\
\text{s.t.} \quad & \sum_{i=1}^{n} y_{ij}(\boldsymbol{d}) \geq d_j, & \forall \boldsymbol{d} \in \mathcal{D},\ \forall j \in [m], \\
& \sum_{j=1}^{m} y_{ij}(\boldsymbol{d}) \leq p_i x_i, & \forall \boldsymbol{d} \in \mathcal{D},\ \forall i \in [n], \\
& y_{ij}(\boldsymbol{d}) \geq 0, & \forall \boldsymbol{d} \in \mathcal{D},\ \forall i \in [n],\ \forall j \in [m], \\
& \boldsymbol{x} \in \{0,1\}^n. &
\end{aligned}$$

Let $i \in [n]$ denote a possible location to build facilities, and $j \in [m]$ denote a delivery destination. Let c_{ij} be the cost of transporting goods from location i to destination j and p_i

Algorithm 8: OPT for ARO

Input: $\bar{\boldsymbol{\theta}}$, N, r, Problem (4.7), ϵ_1, ϵ_2, M, Q_1, Q_2, Q_3

Phase 1

1.1 Identical to 1.1 of Algorithm 7.

1.2 $\mathcal{S}_{\boldsymbol{x}} \leftarrow \left\{ s_{\boldsymbol{x}}(\boldsymbol{\theta}_1), \ldots, s_{\boldsymbol{x}}(\boldsymbol{\theta}_N) \right\}$

$\mathcal{S}_{\boldsymbol{d}} \leftarrow \left\{ s_{\boldsymbol{d}}(\boldsymbol{\theta}_1), \ldots, s_{\boldsymbol{d}}(\boldsymbol{\theta}_N) \right\}$

$\mathcal{S}_{\boldsymbol{y}} \leftarrow \left\{ s_{\boldsymbol{y}}(\boldsymbol{\theta}_1, \boldsymbol{d}_1), \ldots, s_{\boldsymbol{y}}(\boldsymbol{\theta}_N, \boldsymbol{d}_N) \right\}$

Phase 2

2.1.1 Choose Q_1 distinct strategies from $\mathcal{S}_{\boldsymbol{x}}$ at random and compute the reward matrix $\boldsymbol{R}_{\boldsymbol{x}} \in \mathbb{R}^{N \times Q_1}$ using those strategies.

2.1.2 Train a decision tree $\mathcal{T}_{\boldsymbol{x}}$ using $\boldsymbol{R}_{\boldsymbol{x}}$.

2.2.1 Choose Q_2 distinct strategies from $\mathcal{S}_{\boldsymbol{d}}$ at random and compute the reward matrix $\boldsymbol{R}_{\boldsymbol{d}} \in \mathbb{R}^{N \times Q_2}$ using those strategies.

2.2.2 Train a decision tree $\mathcal{T}_{\boldsymbol{d}}$ using $\boldsymbol{R}_{\boldsymbol{d}}$.

2.3.1 Choose Q_3 distinct strategies from $\mathcal{S}_{\boldsymbol{y}}$ at random and compute the reward matrix $\boldsymbol{R}_{\boldsymbol{y}} \in \mathbb{R}^{N \times Q_3}$ using those strategies.

2.3.2 Train a decision tree $\mathcal{T}_{\boldsymbol{y}}$ using $\boldsymbol{R}_{\boldsymbol{y}}$.

Phase 3

3.1 For a new instance $\hat{\boldsymbol{\theta}}$, $\mathcal{T}_{\boldsymbol{x}}$ predicts $\hat{s}_{\boldsymbol{x}}(\hat{\boldsymbol{\theta}}) = \hat{\boldsymbol{x}}$.

3.2 For a new instance $\hat{\boldsymbol{\theta}}$, $\mathcal{T}_{\boldsymbol{d}}$ predicts $\hat{s}_{\boldsymbol{d}}(\hat{\boldsymbol{\theta}}) = (\hat{\boldsymbol{x}}, \hat{\boldsymbol{d}}')$.

3.3.1 For a new instance $\hat{\boldsymbol{\theta}}$ and $\hat{\boldsymbol{d}}$, $\mathcal{T}_{\boldsymbol{y}}$ predicts $\hat{s}_{\boldsymbol{y}}(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{d}}) = (\hat{\boldsymbol{x}}, \hat{\tau}_{\boldsymbol{y}}(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{d}}))$.

3.3.2 Solve $Det(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{d}})$ using $\hat{\tau}_{\boldsymbol{y}}(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{d}})$ to compute a wait-and-see decision.

be the capacity of the facility built on location i. The construction cost to build a facility on location i is denoted as f_i. The demand at destination j is denoted as d_j, which is the uncertain parameter. The demand is assumed to be realized after the construction decisions and before the delivery decisions are made. Binary variable x_i is the here-and-now variable representing whether we build facility on location i or not. y_{ij} is the amount of goods to transport from i to j, which is the wait-and-see variable.

The parameter we use to generate instances is the coefficient vector $\boldsymbol{c}$ of the cost function. The vector $\boldsymbol{c}$ is sampled uniformly from the ball $B(\bar{\boldsymbol{c}}, 1)$, where each entry of $\bar{\boldsymbol{c}}$ is drawn from $U(2, 4)$. Capacities p_i are sampled from $U(8, 18)$ and f_i is sampled from $U(3, 5)$. The uncertainty set is defined as $\mathcal{D} = \left\{ \boldsymbol{d} \mid \sum_i d_i \leq 16, 4 \leq d_i \leq 6 \right\}$.

For the purpose of illustration, we use a small sized example with $n = m = 3$. We set $Q_1 = |\mathcal{S}_{\boldsymbol{x}}|, Q_2 = |\mathcal{S}_{\boldsymbol{d}}|$ and $Q_3 = |\mathcal{S}_{\boldsymbol{y}}|$. In other words, the entire set of strategies found is used for training. The tolerances are set to $\epsilon_1 = \epsilon_2 = 0.001$, and the penalty for infeasible predictions are set to $M = 1000000$. We limit the maximum depth of the tree to two in order to develop intuition on the learned models. Furthermore, we assign a number to each constraint in the deterministic version of the problem to clarify which constraint we are referring to in the following description. The demand satisfaction constraint at destination $j, j \in [3]$, is denoted constraint j. The capacity constraint at location $i, i \in [3]$, is denoted

constraint $i+3$. The non-negativity constraint on the amount of goods to transport from i to j is denoted constraint $6+3(i-1)+j$.

In Figure 4.2, we show the decision tree for the here-and-now decisions. Each node contains the predicted here-and-now decision. We can observe how the the cost vector is used to make a construction decision. For example, if c_{22} is smaller than 3.2 and c_{13} is smaller than 2.8, we should build facility both on location 1 and 2 (Note that we always build facility on location 3 regardless of the cost). This makes sense as small value of c_{22} and c_{13} indicates that transporting goods from location 1 and 2 is generally cheap. Likewise, if c_{22} is smaller than 3.2 and c_{13} is larger than 2.8, then we should build facility on location 2 but not on location 1.

In Figure 4.3, we show the decision tree for the worst-case scenarios. Each node contains the predicted here-and-now decision and the associated worst-case scenario. The cost vector is used to make a construction decision, and also predict a worst-case demand that can happen for the construction decision. For example, if c_{32} is larger than 2.662 and c_{12} is larger than 3.411, the worst-case scenario is the scenario in which d_2 gets as large as possible within the uncertainty set. A possible interpretation is that large value of c_{32} and c_{12} indicates it is costly to transport goods to destination 2.

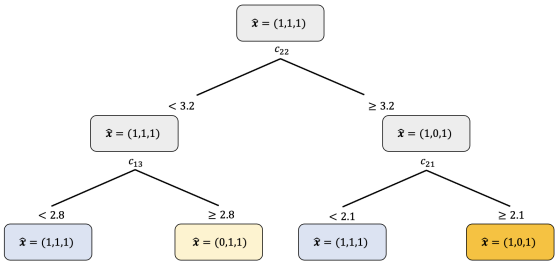

Figure 4.2: Decision tree to predict the optimal strategies for the here-and-now decisions.

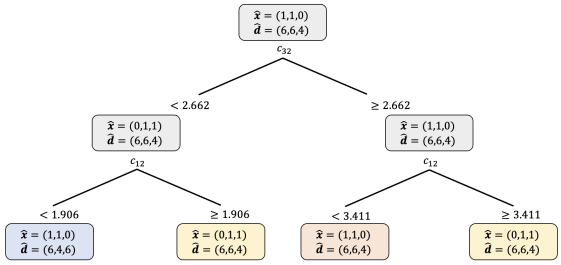

Figure 4.3: Decision tree to predict the optimal strategies for the worst-case scenarios.

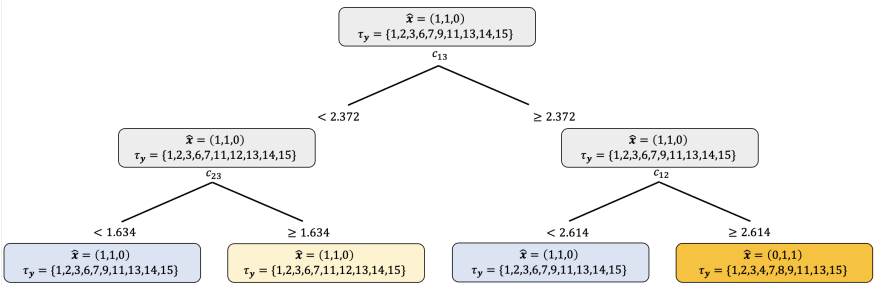

Figure 4.4: Decision tree to predict the optimal strategies for the wait-and-see decisions.

In Figure 4.4, we show the decision tree for the wait-and-see decisions. Each node contains the predicted here-and-now decision and the indices of the tight constraints. The demand satisfaction constraints are always tight, as we need to minimize cost while satisfying the demand. If c_{13} is larger than 2.372 and c_{12} is larger than 2.614, we should not build facility on location 1. This might be because transporting goods from location 1 is too expensive. Then, constraint 4 is tight, as the right-hand-side of the capacity constraint on location 1 is

zero. Constraints 7,8,9 are also tight, as $y_{11} = y_{12} = y_{13} = 0$. If c_{13} is larger than 2.372 and c_{12} is smaller than 2.614, then we should build facility on location 1 but not on location 3. Then, constraint 6 is tight, as the right-hand-side of the capacity constraint on location 3 is zero. Likewise, constraints 13,14,15 are tight, as $y_{31} = y_{32} = y_{33} = 0$.

4.3.6 Machine Learning Model for Varying Dimensions

In the approach presented earlier, each model is trained for instances with a fixed number of variables and constraints. Now, we introduce a generalized approach so that the trained models can be applied to problems with varying dimensions.

In practice, decision-makers often anticipate encountering a number of contingencies. A contingency refers to a situation where specific decision variables or constraints become irrelevant at the time of here-and-now decision-making. To accommodate this setting, we formally define contingency as a set of decision variables and constraints to be excluded.

Consider an ARO problem with n_1 here-and-now variables, n_2 wait-and-see variables, and m constraints for its deterministic version. Then, a contingency can be represented as a triplet $(\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3)$, where $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ are subsets of $[n_1]$, $[n_2]$, and $[m]$, respectively. They contain the indices of the here-and-now variables, the wait-and-see variables and the constraints to be excluded. An ARO instance can now be associated with both a key parameter $\boldsymbol{\theta}$ and a contingency. Solving an ARO instance involves fixing the key parameter to $\boldsymbol{\theta}$ and excluding the here-and-now variables, the wait-and-see variables and the constraints whose indices are in $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$, respectively. Then, we apply Algorithm 5.

To integrate this generalization into our framework, we assume that decision-makers have a predefined list of contingencies they expect to encounter, which we refer to as the contingency list. This list serves as an additional input for Algorithm 7 and 8. Given a contingency list, several adjustments are required for these algorithms. In Phase 1, for each contingency in the contingency list, we vary the key parameter to generate multiple instances. This results in training data with diverse combinations of contingencies and key parameters. In Phase 2, the contingencies are integrated into the feature matrix as categorical features. This means that the type of contingency is included as part of the input features for the machine learning models along with the key parameters $\boldsymbol{\theta}$. In Phase 3, given a new parameter and a contingency, we remove the variables and constraints specified in the contingency, and then apply the predictions of the machine learning models.

In Phase 2, encoding contingencies as categorical features can be done in various ways. For instance, suppose one must consider all possible combinations of whether to remove or retain ℓ different here-and-now variables $x_i, i \in [\ell]$. In this case, the contingency list can be represented as $\left\{(\mathcal{C}_1, \emptyset, \emptyset)\right\}_{\mathcal{C}_1 \in 2^{[\ell]}}$. The simplest way to encode contingencies as categorical features is to introduce a single categorical feature representing the type of contingency. Each unique $(\mathcal{C}_1, \emptyset, \emptyset)$ in the contingency list would then correspond to a distinct categorical value. However, this method results in a categorical feature with 2^ℓ distinct values, reflecting the 2^ℓ different contingencies in the contingency list. In such cases, it may be more practical to introduce ℓ categorical features. Here, each feature $i \in \ell$ indicates whether x_i is removed or not.

4.4 Accelerating Training Data Generation

In this section, we introduce a method to expedite the process of generating training data (Phase 1 in Algorithms 7 and 8). As demonstrated in Section 4.2.4, generating training data in ARO can be computationally intensive. This computational demand may limit the practical applicability of our approach, particularly when frequent retraining of models under various parameter settings is necessary. We address this challenge by drawing inspiration from the principles of online learning. The notation used in this section follows the descriptions provided in Section 4.2.2.

4.4.1 Motivation

Conceptually, our work can be viewed as solving a sequence of ARO instances. From this perspective, Algorithm 7 and 8 are divided two distinct phases: pure exploration (Phase 1) and pure exploitation (Phase 3). In Phase 1, each ARO instance is solved independently from scratch. We focus solely on collecting data on ARO instances and their solutions without any learning component. Conversely, in Phase 3, we rely entirely on the learned model. This workflow shares similarities with prior works such as [24], [28], as well as various other learning-based methods to optimization algorithms [2], [12], [39], [83]. Our proposed method enhances this process by introducing a more fine-grained approach. Instead of strict divisions between exploration and exploitation, we update prediction models more frequently, gradually reducing the level of exploration over time.

4.4.2 Algorithms

We partition Phase 1 into three subphases. The first subphase is the pure exploration stage dedicated to data collection. Using this (potentially very small-sized) data, we train three intermediate models, denoted as $\mathcal{I}_1, \mathcal{I}_2$ and $\mathcal{I}_3$ that are updated throughout Phase 1. These models are utilized to expedite the solution process of Algorithm 5, and the specific manner in which they are utilized distinguishes subphases two and three. First, we outline how Algorithm 5 can be expedited using $\mathcal{I}_1, \mathcal{I}_2$, and $\mathcal{I}_3$, followed by a description of the training process.

The goal of $\mathcal{I}_1$ is to expedite the solution process for Problem (4.5). In each iteration i of Algorithm 5, given a set $\mathcal{E}_i$, Problem (4.5) is solved to determine the optimal here-and-now decision $\boldsymbol{x}_i$ that is robust against the scenarios in $\mathcal{E}_i$. Using the key parameter vector and the latest scenario $\boldsymbol{d}_i$ added to $\mathcal{E}_i$ as inputs, $\mathcal{I}_1$ outputs a probability vector indicating the likelihood of each entry of $\boldsymbol{x}_i$ being one. The goal of $\mathcal{I}_2$ ($\mathcal{I}_3$) is to provide the initial point $\boldsymbol{x}_0$ ($\boldsymbol{d}_0$) for Algorithm 5 (6), respectively.

In the first subphase, we solve ARO instances independently from scratch. This stage focuses solely on gathering data, using random initial points $\boldsymbol{x}_0$ for Algorithm 5 and three random initial points $\boldsymbol{d}_0$ for Algorithm 6. We then train three intermediate models.

In the second subphase, given an ARO instance with the key parameter $\hat{\boldsymbol{\theta}}$, we use the predictions of $\mathcal{I}_2$ and $\mathcal{I}_3$ as initial points for Algorithm 5 and 6, respectively. In each iteration of Algorithm 5, $\mathcal{I}_1$ predicts the probability of each entry of the here-and-now decision being one. We then set a warm-starting point for the binary variables where the predicted probability is

greater than a threshold p to be one, and smaller than $1-p$ to be zero. The threshold p is set very close to 1, indicating certainty in the model's predictions. This version of Algorithm 5 is denoted as Algorithm 9. The intermediate models and the value of p can be updated multiple times as we gather more training data.

After the second subphase, with more training data, we anticipate improved accuracy in the intermediate models. In the third subphase, we use the predictions of $\mathcal{I}_1$ to partially fix (distinct from warm-starting) the here-and-now variables in each iteration of Algorithm 5. Specifically, we fix variables where the predicted probability exceeds a threshold p or falls below $1-p$. By fully fixing certain binary variables, our aim is to further expedite Algorithm 9. $\mathcal{I}_2$ and $\mathcal{I}_3$ are utilized in the same manner as in Algorithm 9. This modified version of Algorithm 5 is denoted as Algorithm 10.

The value of p must be chosen carefully, as it directly influences the trade-off between prediction accuracy and the proportion of binary variables that can be fixed or warm-started. A larger p leads to more accurate predictions, but at the expense of being able to fix or warm-start a smaller portion of the binary variables. To determine the value of p, we utilize a validation set consisting of 200 data points. We select the smallest p such that the misclassification rate on the validation set, computed only on the entries with probability outputs greater than p or smaller than $1-p$, is less than 0.00001.

Now, we elaborate on the training process for $\mathcal{I}_1, \mathcal{I}_2$, and $\mathcal{I}_3$. Assume Algorithm 5, 9 or 10 has been applied to the ARO instance associated with a parameter $\bar{\boldsymbol{\theta}}$, and it terminated after J iterations. Then, the training data extracted from this single instance for $\mathcal{I}_1$ is $\left((\bar{\boldsymbol{\theta}}, \boldsymbol{d}_j), \boldsymbol{x}_j\right)_{j=1}^{J}$. The training data for $\mathcal{I}_2$ is $\left(\bar{\boldsymbol{\theta}}, \boldsymbol{x}_J\right)$, while for $\mathcal{I}_3$ it is $\left(\bar{\boldsymbol{\theta}}, \boldsymbol{d}_J\right)$. $\mathcal{I}_1$ and $\mathcal{I}_2$ are binary classifiers that predict whether each entry of the here-and-now decision is one, while $\mathcal{I}_3$ is a multiclass classifier similar to the models in Algorithm 7. While the prediction targets of $\mathcal{I}_1$ and $\mathcal{I}_2$ resemble those described in Algorithm 7 and 8, the models in those algorithms predict the entire here-and-now decision vector as a unified bundle. On the contrary, $\mathcal{I}_1$ and $\mathcal{I}_2$ are binary classifiers that predict whether each entry of the here-and-now variable is zero or one individually. Hence, neural networks are specifically chosen due to the high-dimensionality of the prediction target.

Remark In Section 4.6.7, we show that the solution quality of Algorithm 9 and 10 remains practically identical to Algorithm 5. Even if solutions are of poor quality, however, it does not pose a significant challenge to our main approach in Algorithm 8. This is because during the computation of the reward matrix, entries associated with poor solutions will be assigned high suboptimality.

4.5 Partitioning Algorithm

In this section, we propose an algorithm to reduce the number of distinct strategies for the wait-and-see decisions in the training set. In the computational experiments, we have observed that as the size of the uncertainty sets gets large, the number of distinct strategies for the wait-and-see variables in the training set can get prohibitively large. While a similar issue is discussed in [25], there is a subtle difference in our context. For MICO problems, there is no notion of uncertainty set. Therefore, the number of distinct strategies is controlled

Algorithm 9: Column and Constraint Generation with Warm Start

Input: Problem (4.1), $\mathcal{I}_1$, $\mathcal{I}_2$, $\mathcal{I}_3$, ϵ_1, ϵ_2, p
Output: $\tilde{\boldsymbol{x}}^*$,$\tilde{\boldsymbol{d}}^*$
Initialization: $i = 0$, $\mathcal{E}_0 = \emptyset$, UB $= \infty$, LB $= -\infty$

$\mathcal{I}_2$ predicts $\boldsymbol{x}_0$ and $\mathcal{I}_3$ predicts $\hat{\boldsymbol{d}}_0$
while *UB -LB* $\geq \epsilon_1$ **do**
  if $i = 0$ **then**
    Evaluate $\mathcal{Q}(\boldsymbol{x}_i)$ using $\hat{\boldsymbol{d}}_0$ as the initial point in Algorithm 6 to get $\tilde{\mathcal{Q}}(\boldsymbol{x}_i)$ and a solution $\boldsymbol{d}_i$.
    $\mathcal{E}_{i+1} \leftarrow \mathcal{E}_i \bigcup \{\boldsymbol{d}_i\}$
    $i \leftarrow i + 1$
  else
    $\mathcal{I}_1$ outputs a probability vector.
    Solve (4.5) with the extreme points in $\mathcal{E}_i$. The binary variables whose corresponding entries in the probability vector that are greater than p and smaller than $1 - p$ are warm-started at 1 and 0, respectively. Denote the solutions as $\boldsymbol{x}_i$ and $\boldsymbol{\alpha}_i$.
    LB $\leftarrow \boldsymbol{\alpha}_i$
    Evaluate $\mathcal{Q}(\boldsymbol{x}_i)$ using $\hat{\boldsymbol{d}}_0$ as the initial point in Algorithm 6 to get $\tilde{\mathcal{Q}}(\boldsymbol{x}_i)$ and a solution $\boldsymbol{d}_i$.
    UB $\leftarrow \tilde{\mathcal{Q}}(\boldsymbol{x}_i)$
    $\mathcal{E}_{i+1} \leftarrow \mathcal{E}_i \bigcup \{\boldsymbol{d}_i\}$
    $i \leftarrow i + 1$
$\tilde{\boldsymbol{x}}^* \leftarrow \boldsymbol{x}_i$
$\tilde{\boldsymbol{d}}^* \leftarrow \boldsymbol{d}_i$

Algorithm 10: Column and Constraint Generation with Warm Start and Fixed Variables

Input: Problem (4.1),$\mathcal{I}_1$, $\mathcal{I}_2$, $\mathcal{I}_3$, ϵ_1, ϵ_2, p
Output: $\tilde{\boldsymbol{x}}^*$,$\tilde{\boldsymbol{d}}^*$
Initialization: $i = 0$, $\boldsymbol{x}_0$, $\mathcal{E}_0 = \emptyset$, UB $= \infty$, LB $= -\infty$

$\mathcal{I}_2$ predicts $\boldsymbol{x}_0$ and $\mathcal{I}_3$ predicts $\hat{\boldsymbol{d}}_0$
while UB -$LB \geq \epsilon_1$ **do**
  if $i = 0$ **then**
    Evaluate $\mathcal{Q}(\boldsymbol{x}_i)$ using $\hat{\boldsymbol{d}}_0$ as the initial point in Algorithm 6 to get $\tilde{\mathcal{Q}}(\boldsymbol{x}_i)$ and a solution $\boldsymbol{d}_i$.
    $\mathcal{E}_{i+1} \leftarrow \mathcal{E}_i \bigcup \{\boldsymbol{d}_i\}$
    $i \leftarrow i + 1$
  else
    $\mathcal{I}_1$ outputs a probability vector.
    Solve (4.5) with the extreme points in $\mathcal{E}_i$. The binary variables whose corresponding entries in the probability vector that are greater than p and smaller than $1 - p$ are fixed at 1 and 0, respectively. Denote the solutions as $\boldsymbol{x}_i$ and $\boldsymbol{\alpha}_i$.
    LB $\leftarrow \boldsymbol{\alpha}_i$
    Evaluate $\mathcal{Q}(\boldsymbol{x}_i)$ using $\hat{\boldsymbol{d}}_0$ as the initial point in Algorithm 6 to get $\tilde{\mathcal{Q}}(\boldsymbol{x}_i)$ and a solution $\boldsymbol{d}_i$.
    UB $\leftarrow \tilde{\mathcal{Q}}(\boldsymbol{x}_i)$
    $\mathcal{E}_{i+1} \leftarrow \mathcal{E}_i \bigcup \{\boldsymbol{d}_i\}$
    $i \leftarrow i + 1$
$\tilde{\boldsymbol{x}}^* \leftarrow \boldsymbol{x}_i$
$\tilde{\boldsymbol{d}}^* \leftarrow \boldsymbol{d}_i$

by the support of the training distribution. If the number becomes too large for a distribution of interest, one can partition its support into multiple smaller regions and train a machine learning model for each region. However, this approach does not directly translate to ARO, as the number of distinct strategies depends on both the training distribution and the size of the uncertainty set. We cannot simply reduce or partition the uncertainty set, because this leads to less robust solutions. Moreover, the pruning algorithm described in [25, Section 4.3] is often insufficient to handle the large number of strategies encountered in our numerical experiments with large uncertainty sets. However, the algorithm we develop in this section can reduce the number effectively.

The high-level idea is that instead of trying to identify the tight constraints, we try to identify a subset of the redundant constraints. We can optimize excluding these constraints and still get the optimal solution to the original problem.

Before giving a formal description of the algorithm, we first provide a small motivating example. Consider the following hypothetical setting. We are given a deterministic continuous optimization problem with four constraints, each denoted as constraint 1,2,3,4, respectively. In Phase 1, we generate four training parameters, $\boldsymbol{\theta}_i, i \in [4]$, and solve the associated instances to optimality. The tight constraints (which also defines the optimal strategy as the problem of interest is continuous) for each instance is $\tau(\boldsymbol{\theta}_i) = \{i\}$. That is, the optimal strategies of the training instances are all different, resulting in four distinct target classes. Learning in this setting is challenging, as the number of distinct target classes is equal to the number of training instances. Using $\boldsymbol{\tau}$ to denote the set of tight constraints found in the training set, we get $\boldsymbol{\tau} = \Big\{\{1\},\{2\},\{3\},\{4\}\Big\}$.

In the algorithm we propose, we first need to define a partition of the set $\boldsymbol{\tau}$. In this example, we define the partition as $\mathcal{P} = \left\{\Big\{\{1\},\{2\}\Big\},\Big\{\{3\},\{4\}\Big\}\right\}$. For each cell in $\mathcal{P}$, we compute the union of its elements. The unions are $\{1,2\}$ and $\{3,4\}$ under the partition we defined. Then, for the parameters $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$, we redefine their prediction targets as $\{1,2\}$. We can still compute the optimal solutions of the the parameters $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ by imposing the constraints $\{1,2\}$ only. Likewise, for $\boldsymbol{\theta}_3$ and $\boldsymbol{\theta}_4$, we redefine their prediction targets as $\{3,4\}$. Once we redefine the prediction targets following this procedure, the number of distinct prediction targets is reduced to two. A downside might be that given a new instance, prediction of a trained model now contains two constraints instead of one. This can undermine the computational efficiency we could have gained by imposing just a single constraint. The partitioning algorithm we propose is a generalization of this procedure to two-stage ARO with binary here-and-now variables.

Assume we have N ARO instances and the corresponding optimal strategies $(s_1, \ldots, s_N)$, where $s_i = (\boldsymbol{x}_i^*, \tau_{\boldsymbol{y},i}), i \in [N]$. Let $\boldsymbol{\tau} = \{\tau_{\boldsymbol{y},1}, \tau_{\boldsymbol{y},2}, \ldots, \tau_{\boldsymbol{y},N}\}$ be the set of tight constraints in our training set. Without loss of generality, we assume $\boldsymbol{\tau} = \{\tau_{\boldsymbol{y},1}, \tau_{\boldsymbol{y},2}, \ldots, \tau_{\boldsymbol{y},M}\}$, where $\tau_{\boldsymbol{y},i} \neq \tau_{\boldsymbol{y},j}$ if $i \neq j$, and $\boldsymbol{\tau}$ is sorted in the order such that if $i < j$, then $\tau_{\boldsymbol{y},i}$ occurred more frequently than $\tau_{\boldsymbol{y},j}$ in the training set (ties are broken arbitrarily). Note that $M \leq N$, since the optimal strategies might overlap. We divide $\boldsymbol{\tau}$ into K partitions $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_K\}$ $(K \leq M)$, and compute the union of the elements in each cell. We use u_i to denote the union of the elements in the cell that $\tau_{\boldsymbol{y},i}$ originally belonged to. For the i_{th} instance, we replace its prediction target with $(\boldsymbol{x}_i^*, u_i)$. Algorithm 11 provides a formal description.

In our implementation in Section 4.6, we define the partition of $\boldsymbol{\tau}$ the following way. We let $\mathcal{P}_i = \{\tau_{\boldsymbol{y},i}\}, i \in [K-1]$, and $\mathcal{P}_K = \{\tau_{\boldsymbol{y},K}, \ldots, \tau_{\boldsymbol{y},M}\}$. In other words, we let $K-1$ most frequently occurring tight constraints form their own partitions with a single element. We combine the rest of the tight constraints to form the K_{th} cell, resulting in K cells in total. We denote $\bar{u} = \bigcup_{\tau_{\boldsymbol{y},j} \in \mathcal{P}_K} \tau_{\boldsymbol{y},j}$ as the union constraints.

As mentioned above, using the union of the tight constraints can undermine the computational efficiency that we could have gained by imposing only the exact tight constraints. Another concern might be that as we are artificially redefining the prediction targets in the data set, training an accurate prediction model might become challenging. However, empirically, tight constraints of different instances mostly overlap. Hence, the cardinality of the union constraints is in general similar to the cardinality of the individual tight constraints. Furthermore, even after applying Algorithm 11 to the data set, accurate models can be trained. We demonstrate these points in Section 4.6.

Algorithm 11: Partitioning Algorithm

Output: $\left\{\left(\boldsymbol{\theta_i}, (\boldsymbol{x}_i^*, u_i)\right)\right\}_{i=1}^{N}$

Input: $\left\{\left(\boldsymbol{\theta_i}, (\boldsymbol{x}_i^*, \tau_{\boldsymbol{y},i})\right)\right\}_{i=1}^{N}$, $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_K\}$

```
for P_i ∈ P do
    τ̄_{y,i} ← ⋃_{τ_{y,j} ∈ P_i} τ_{y,j}
end
for i = 1 to N do
    for j = 1 to K do
        if τ_{y,i} ∈ P_j then
            u_i ← τ̄_{y,j}
            break
        end
    end
end
```

4.6 Computational Experiments

In this section, we provide the results of the computational experiments on synthetic and real-world problems. We evaluate the quality of the predicted strategies and also analyze the relative speed-up of our approach compared with Algorithm 5. We also demonstrate the effectiveness of Algorithm 9, 10 and 11. In the Appendix B, we offer further insights into the performance of our approach across a range of scenarios. This includes a report on the offline computation time of our method, as well as an analysis of its performance under varying sizes of uncertainty sets, training data, and distributional shifts. Identical to the experiment in Section 4.2.4, the experiment in this section was executed in Julia 1.4.1 on a MacBook Pro with 2.6 GHz Intel Core i7 CPU and 16GB of RAM. Likewise, all deterministic optimization problems involved are solved with Gurobi. The software for OPT is available from [60].

4.6.1 Problem Description

We describe the synthetic and real-world problems that we test our approach on. We also provide sample generation details and the uncertainty sets used.

Facility Location We consider the facility location problem introduced in Section 4.3.5. We use the polyhedral uncertainty set defined as $\mathcal{D} = \{\boldsymbol{d} \mid \sum_i d_i \leq \Gamma, 4 \leq d_i \leq 6\}$. The feature vector is $\boldsymbol{f}$. For the case with $n = 7$, we sample $\boldsymbol{f}$ from the ball $B(\bar{\boldsymbol{f}}, 3)$, where $\bar{f}_i$ is sampled from $U(2, 12)$ and fixed. For all other cases, we sample $\boldsymbol{f}$ from $B(\bar{\boldsymbol{f}}, 1.5)$, where $\bar{f}_i$ is sampled from $U(2, 22)$ and fixed. We sample p_i from $U(8, 18)$ and c_i from $U(2, 4)$.

Inventory Control Consider the multi-item inventory control problem, where ordering decisions can be partially made after the demand is realized. There are n different items to order, with three different ways to order each item. For each item $i, i \in [n]$, we can order a fixed lot size of l_i at the unit cost of c_i^1 or order a fixed lot size of l_i at the unit cost of c_i^2 before the demand is realized. After we see the demand, we can order any amount y_i at the unit cost of c_i^3. We must also pay storage and disposal cost of c^4 for the remaining stock after the demand is satisfied. We set $c_i^1, c_i^2 \leq c_i^3 \leq c_i^4$ to avoid trivial solutions. The here-and-now binary variables to decide whether we order fixed lot sizes with the cost c_i^1 and c_i^2 before seeing the demand are denoted x_i^1 and x_i^2, respectively. The wait-and-see variable is y_i. The exact formulation is as follows.

$$
\begin{array}{lll}
\min\limits_{\boldsymbol{y}(\cdot), \boldsymbol{x}^1, \boldsymbol{x}^2} & \max\limits_{\boldsymbol{d} \in \mathcal{D}} & \sum\limits_{i=1}^{n} c_i^1 l_i x_i^1 + \sum\limits_{i=1}^{n} c_i^2 l_i x_i^2 + \sum\limits_{i=1}^{n} c_i^3 y_i(\boldsymbol{d}) + c^4 \sum\limits_{i=1}^{n} [l_i x_i^1 + l_i x_i^2 + y_i(\boldsymbol{d}) - d_i] \\
\text{s.t.} & l_i x_i^1 + l_i x_i^2 + y_i(\boldsymbol{d}) - d_i \geq 0, & \forall \boldsymbol{d} \in \mathcal{D}, \quad \forall i \in [n], \\
& y_i(\boldsymbol{d}) \geq 0, & \forall \boldsymbol{d} \in \mathcal{D}, \quad \forall i \in [n], \\
& \boldsymbol{x}^1, \boldsymbol{x}^2 \in \{0, 1\}^n.
\end{array}
$$

We use the uncertainty set defined as $\mathcal{D} = \{\boldsymbol{d} \mid \|\boldsymbol{d} - 50\|_1 \leq \Gamma\}$. For the problem with $n = 25$, feature vectors are $\boldsymbol{c}^2$ and $\boldsymbol{c}^3$. We sample $\boldsymbol{c}^2$ from $B(\bar{\boldsymbol{c}^2}, 5)$, where $\bar{c_i^2}$ is sampled from $U(40, 60)$ and fixed. We sample $\boldsymbol{c}^3$ from $B(\bar{\boldsymbol{c}^3}, 5)$, where $\bar{c_i^3}$ is sampled from $U(60, 80)$ and fixed. For the larger problems, the feature vector is $\boldsymbol{c}^3$. We sample $\boldsymbol{c}^3$ from $B(\bar{\boldsymbol{c}^3}, 2)$, where $\bar{c_i^3}$ is sampled from $U(60, 80)$ and fixed. We sample l_i from $U(20, 30)$, c_i^1 from $U(40, 60)$ and fix $c_4 = 60$.

Unit Commitment We consider the unit commitment problem described in Section 4.2.4. We give the complete formulation of the deterministic version in the Appendix B, and the data is taken from [37]. This problem is analogous to the facility location problem, but with more complicated constraints. We use the budget uncertainty set defined as $\mathcal{D} = \{\boldsymbol{d} \mid \sum_i \left|\frac{d_i - \bar{d}_i}{0.1 \times \bar{d}_i}\right| \leq \Gamma, \left|d_i - \bar{d}_i\right| \leq 0.1 \times \bar{d}_i\}$, where $\bar{\boldsymbol{d}}$ is the original data. The feature vector is the coefficient vector $\boldsymbol{b}$ of the production cost function. The parameters are sampled from the ball with radius 1.5, and the center of the ball is the original data.

4.6.2 Experimental Design

We conduct two sets of experiments. In the first set of experiments, we generate and solve ARO instances to near-optimality in Phase 1 using tight tolerances for Algorithm 5 and 6. Then, we use XGBOOST [40] for the classification approach in Algorithm 7, and compare its performance with OPT (Algorithm 8). There are two main reasons behind this experimental design. First, we aim to demonstrate the effectiveness of our approach regardless of the machine learning method used. Second, we aim to evaluate the trade-off between interpretability and prediction accuracy. XGBOOST is known for its high performance on various prediction tasks but lacks interpretability compared to OPT. In contrast, OPT is highly interpretable due to its simple decision tree structure but may have weaker prediction accuracy compared to XGBOOST. By comparing these two methods, we aim to analyze the cost we have to pay to gain interpretability. We also analyze the effectiveness of Algorithm 11 and the generalization described in Section 4.3.6. We provide the results in Section 4.6.3,4.6.4 and 4.6.5.

In the second set of experiments, we generate suboptimal strategies in Phase 1 to solve large scale unit commitment problems. As shown in Section 4.2.4, solving such problems can be computationally challenging. As a result, generating a training set in Phase 1 can be a significant computational burden. To overcome this issue, we use more relaxed tolerances for Algorithm 5 and 6 to terminate earlier. In Section 4.6.6, we demonstrate that Algorithm 8 can still find high quality solutions for large scale unit commitment problems. In Section 4.6.7, we demonstrate the effectiveness of Algorithm 9 and 10 to further expedite training set generation.

Furthermore, [24], [25] propose using multiple predictions of the trained model in Phase 3. Classification algorithms generate a likelihood vector where each entry represents the likelihood of a particular label being the true label for a data point. Hence, multiple most promising predictions can be identified using this vector. Similarly, OPT can output multiple best strategies [28]. We can evaluate all of these predictions in parallel by computing their infeasibilities and suboptimalities to choose the best one. The drawback of this approach is that the evaluation process requires additional computation in Phase 3. For both experiments, we use multiple predictions of OPT, only if using just a single prediction does not result in perfect accuracy. In this case, we provide separate tables to analyze the performance improvement and the additional computational burden. We use k to denote the number of predictions we use in Phase 3.

4.6.3 Solving ARO with Near-Optimal Strategies

In this section, we generate and solve ARO instances to near-optimality in Phase 1. Throughout the entire experiment, we set $\epsilon_1 = 0.001$ and $\epsilon_2 = 0.001$ for Algorithm 5 and 6, respectively. The optimality gap of Gurobi is fixed at its default value of 0.0001. When using OPT, we use the entire set of strategies found to ensure a fair comparison with XGBoost. In other words, we set $Q_1 = |\mathcal{S}_{\boldsymbol{x}}|, Q_2 = |\mathcal{S}_{\boldsymbol{d}}|$ and $Q_3 = |\mathcal{S}_{\boldsymbol{y}}|$. For both XGBoost and OPT, we minimize the hyperparameter tuning process and grid search over the maximum depths 5 and 10.

Table 4.1, 4.2, 4.3 contain the experiment results on the facility location, the inventory control and the unit commitment problem, respectively. For the experiments reported in

Target	n	m	Γ	Learner	Accuracy	Infeasibility	sub_{max}	$\lvert\mathcal{S}\rvert$	N	t_{ratio}
$s_{\boldsymbol{x}}$					1.00	0	0.0070	2	20000	1666
$s_{\boldsymbol{d}}$	7	7	38	OPT	1.00	0	0.0000	4	20000	1538
$s_{\boldsymbol{y}}$					0.93	0	0.0070	23	20000	34
$s_{\boldsymbol{x}}$					1.00	0	0.0000	2	20000	34
$s_{\boldsymbol{d}}$	7	7	38	XGB	1.00	0	0.0000	4	20000	32
$s_{\boldsymbol{y}}$					0.97	0	0.0010	23	20000	17
$s_{\boldsymbol{x}}$					0.99	0	0.0004	10	20000	33333
$s_{\boldsymbol{d}}$	80	60	241	OPT	0.99	0	0.0004	10	20000	36363
$s_{\boldsymbol{y}}$					0.98	0	0.0004	22	20000	21
$s_{\boldsymbol{x}}$					1.00	0	0.0000	10	20000	276
$s_{\boldsymbol{d}}$	80	60	241	XGB	0.99	0	0.0004	10	20000	278
$s_{\boldsymbol{y}}$					0.95	0	0.0004	22	20000	21
$s_{\boldsymbol{x}}$					1.00	0	0.0002	10	25000	3.75×10^5
$s_{\boldsymbol{d}}$	200	150	601	OPT	0.99	0	0.0002	10	25000	4.06×10^5
$s_{\boldsymbol{y}}$					0.99	0	0.0002	10	25000	37
$s_{\boldsymbol{x}}$					1.00	0	0.0002	10	25000	1036
$s_{\boldsymbol{d}}$	200	150	601	XGB	0.98	0	0.0002	10	25000	880
$s_{\boldsymbol{y}}$					0.97	0	0.0002	10	25000	35
$s_{\boldsymbol{x}}$					1.00	0	0.0000	42	25000	3.09×10^6
$s_{\boldsymbol{d}}$	200	150	751	OPT	1.00	0	0.0000	42	25000	3.37×10^6
$s_{\boldsymbol{y}}$					1.00	0	0.0000	42	25000	186
$s_{\boldsymbol{x}}$					0.99	0	0.0020	42	25000	9568
$s_{\boldsymbol{d}}$	200	150	751	XGB	0.99	0	0.0020	42	25000	12783
$s_{\boldsymbol{y}}$					0.99	0	0.0020	42	25000	183

Table 4.1: Numerical results for the facility location problem with $k = 1$.

these tables, we only use a single prediction in Phase 3 ($k = 1$). Table 4.4, 4.5, 4.6 contain the experiment results using multiple predictions of OPT in Phase 3 ($k \geq 1$). As mentioned above, we experiment with $k \geq 1$ only if the prediction accuracy with $k = 1$ is not perfect. We report how the performance changes as we increase k.

Table Notations Total N ARO instances are generated, which are randomly split into the training set (70%) and the test set (30%). Columns n and m contain the parameters that define the problem size and column Γ contains the parameter that determines the size of the uncertainty sets. In the Accuracy column, we report the percentage of accurate predictions on the test set, rounded up to the second decimal place. For all three prediction targets, we consider a prediction accurate if it is feasible and the suboptimality is smaller than 0.0001. In the Infeasibility column, we report the percentage of infeasible predictions on the test set. We report the maximum suboptimality among the feasible predictions in the column sub_{max}. In the $|\mathcal{S}|$ column, we report the number of distinct strategies found in the training set. In case we used Algorithm 11 to reduce this number, we report the number we get by applying Algorithm 11, not the original number. We provide the original number of strategies and further analysis of Algorithm 11 in Section 4.6.4. In the t_{ratio} column, we report the computation time it takes to obtain the solution from scratch using Algorithm 5, divided by the computation time of our approach. It is rounded up to the nearest integer.

Target	n	Γ	Learner	Accuracy	Infeasibility	sub_{max}	$\|\mathcal{S}\|$	N	t_{ratio}
$s_{\boldsymbol{x}}$				1.00	0	0.0004	12	40000	1220
$s_{\boldsymbol{d}}$	25	10	OPT	1.00	0	0.0004	28	40000	1126
$s_{\boldsymbol{y}}$				0.99	0	0.0004	30	40000	38
$s_{\boldsymbol{x}}$				1.00	0	0.0000	12	40000	13
$s_{\boldsymbol{d}}$	25	10	XGB	1.00	0	0.0000	28	40000	14
$s_{\boldsymbol{y}}$				0.99	0	0.0004	30	40000	8
$s_{\boldsymbol{x}}$				1.00	0	0.0000	17	60000	17241
$s_{\boldsymbol{d}}$	600	10	OPT	1.00	0	0.0000	30	60000	14589
$s_{\boldsymbol{y}}$				1.00	0	0.0000	48	60000	78
$s_{\boldsymbol{x}}$				1.00	0	0.0000	17	60000	21
$s_{\boldsymbol{d}}$	600	10	XGB	1.00	0	0.0000	30	60000	22
$s_{\boldsymbol{y}}$				1.00	0	0.0000	48	60000	14
$s_{\boldsymbol{x}}$				1.00	0	0.0000	9	60000	12838
$s_{\boldsymbol{d}}$	1000	10	OPT	1.00	0	0.0000	27	60000	11851
$s_{\boldsymbol{y}}$				1.00	0	0.0000	11	60000	84
$s_{\boldsymbol{x}}$				1.00	0	0.0000	9	60000	13
$s_{\boldsymbol{d}}$	1000	10	XGB	0.99	0	0.0000	27	60000	13
$s_{\boldsymbol{y}}$				1.00	0	0.0000	11	60000	13
$s_{\boldsymbol{x}}$				1.00	0	0.0000	7	60000	25480
$s_{\boldsymbol{d}}$	1000	45	OPT	1.00	0	0.0000	30	60000	23520
$s_{\boldsymbol{y}}$				1.00	0	0.0000	7	60000	94
$s_{\boldsymbol{x}}$				1.00	0	0.0000	7	60000	21
$s_{\boldsymbol{d}}$	1000	45	XGB	1.00	0	0.0000	30	60000	24
$s_{\boldsymbol{y}}$				1.00	0	0.0000	7	25000	17

Table 4.2: Numerical results for the inventory control problem with $k = 1$.

Target	n	m	Γ	Learner	Accuracy	Infeasibility	sub_{max}	$\|\mathcal{S}\|$	N	t_{ratio}
$s_{\boldsymbol{x}}$					0.97	0	0.0010	17	20000	88137
$s_{\boldsymbol{d}}$	10	24	0.1	OPT	0.97	0	0.0010	24	20000	1.30×10^5
$s_{\boldsymbol{y}}$					0.96	0	0.0010	32	20000	119
$s_{\boldsymbol{x}}$					1.00	0	0.0000	17	20000	5137
$s_{\boldsymbol{d}}$	10	24	0.1	XGB	0.98	0	0.0040	24	20000	6445
$s_{\boldsymbol{y}}$					0.93	0	0.0040	32	20000	277
$s_{\boldsymbol{x}}$					1.00	0	0.0000	9	15000	93318
$s_{\boldsymbol{d}}$	10	24	2	OPT	1.00	0	0.0000	9	15000	87228
$s_{\boldsymbol{y}}$					1.00	0	0.0000	9	15000	296
$s_{\boldsymbol{x}}$					1.00	0	0.0004	9	15000	5300
$s_{\boldsymbol{d}}$	10	24	2	XGB	1.00	0	0.0000	9	15000	6389
$s_{\boldsymbol{y}}$					1.00	0	0.0000	9	15000	293

Table 4.3: Numerical results for the unit commitment problem with $k = 1$.

Target	k	n	m	Γ	Accuracy	Infeasibility	sub_{max}	$\lvert\mathcal{S}\rvert$	N	t_{ratio}
	1				0.93	0	0.0070			34
$s_{\boldsymbol{y}}$	5	7	7	38	0.95	0	0.0070	23	20000	6
	10				1.00	0	0.0000			6
	1				0.99	0	0.0004			33333
$s_{\boldsymbol{x}}$	5	80	60	241	1.00	0	0.0002	10	20000	10
	10				1.00	0	0.0000			10
	1				0.99	0	0.0004			36363
$s_{\boldsymbol{d}}$	5	80	60	241	1.00	0	0.0000	10	20000	10
	10				1.00	0	0.0000			10
	1				0.98	0	0.0004			21
$s_{\boldsymbol{y}}$	5	80	60	241	1.00	0	0.0000	22	20000	7
	10				1.00	0	0.0000			7
	1				1.00	0	0.0002			3.75×10^5
$s_{\boldsymbol{x}}$	5	200	150	601	1.00	0	0.0000	10	25000	8
	10				1.00	0	0.0000			8
	1				0.99	0	0.0002			4.06×10^5
$s_{\boldsymbol{d}}$	5	200	150	601	1.00	0	0.0000	10	25000	8
	10				1.00	0	0.0000			8
	1				0.99	0	0.0002			37
$s_{\boldsymbol{y}}$	5	200	150	601	1.00	0	0.0000	10	25000	7
	10				1.00	0	0.0000			7

Table 4.4: Numerical results for the facility location problem with $k \geq 1$ using OPT.

Target	k	n	Γ	Accuracy	Infeasibility	sub_{max}	$\lvert\mathcal{S}\rvert$	N	t_{ratio}
	1			1.00	0	0.0004			1220
$s_{\boldsymbol{x}}$	5	25	10	1.00	0	0.0000	12	40000	7
	10			1.00	0	0.0000			7
	1			1.00	0	0.0004			1126
$s_{\boldsymbol{d}}$	5	25	10	1.00	0	0.0000	28	40000	7
	10			1.00	0	0.0000			7
	1			0.99	0	0.0004			38
$s_{\boldsymbol{y}}$	5	25	10	0.99	0	0.0002	30	40000	7
	10			0.99	0	0.0002			7

Table 4.5: Numerical results for the inventory control problem with $k \geq 1$ using OPT.

Target	k	n	m	Γ	Accuracy	Infeasibility	sub_{max}	$\lvert\mathcal{S}\rvert$	N	t_{ratio}
	1				0.97	0	0.0010		20000	88137
$s_{\boldsymbol{x}}$	5	10	24	0.1	0.98	0	0.0004	17	20000	27
	10				1.00	0	0.0002		20000	27
	1				0.97	0	0.0010		20000	1.30×10^5
$s_{\boldsymbol{d}}$	5	10	24	0.1	0.98	0	0.0004	24	20000	25
	10				0.99	0	0.0002		25000	25
	1				0.96	0	0.0010		20000	119
$s_{\boldsymbol{y}}$	5	10	24	0.1	0.98	0	0.0004	32	20000	5
	10				0.99	0	0.0002		25000	5

Table 4.6: Numerical results for the unit commitment problem with $k \geq 1$ using OPT.

Results

- Both OPT and XGBoost consistently demonstrate excellent accuracy, never falling below 0.93 and often reaching 0.99 or 1.00. This performance holds true regardless of the problem size or the size of the uncertainty sets. Even when the solutions are not exactly accurate, the maximum suboptimalities remain exceptionally low, at most 0.001. This indicates the high quality of the solutions.
- The predictions are never infeasible for both OPT and XGBoost.
- In general, the prediction accuracy for the here-and-now decisions is the highest, followed by the worst-case scenarios and the wait-and-see decisions.
- The solve times using OPT and XGBoost are significantly faster than Algorithm 5, at times reaching up to 3.37 million times faster. Additionally, OPT tends to outperform XGBoost in terms of speed. This is primarily because the time required for a decision tree to compute its predictions is typically less than a millisecond, whereas XGBoost generally takes slightly longer.
- The speed-up of our approach to compute the wait-and-see decisions is less drastic compared to here-and-now decisions or worst-case scenarios, typically ranging from tens to hundreds of times faster. To compute a here-and-now decision or a worst-case scenario, the only computation needed is to determine the output of the trained model on an input. In order to compute a wait-and-see decision, however, we still need to solve a linear optimization problem, leading to longer computation time.
- OPT and XGBoost show very similar performance in general. This implies that we often do not have to compromise performance too much to gain interpretability.
- As we increase k, the quality of the solutions improves monotonically. At the same time, the relative speed-up of our approach decreases due to the evaluation process required to choose the best strategy.

4.6.4 Analysis of Algorithm 11

In this section, we demonstrate the effectiveness of Algorithm 11. We apply Algorithm 11 in the previously described experiment, in case the number of distinct strategies for the wait-and-see decisions is extremely large. Tables 4.7, 4.8, 4.9 contain the numerical results on the facility location, the inventory control and the unit commitment problem, respectively.

Table Notations We use $|\boldsymbol{\tau}|$ to denote the number of distinct tight constraints found in the training set, before applying Algorithm 11. As before, K denotes the number it is reduced to. We also report how many more constraints the union constraints contain compared to individual tight constraints, denoted as $|\bar{u}| - |\tau_{\boldsymbol{y}}|$. Other columns are given to specify which problem Algorithm 11 is applied to.

n	m	Γ	$\lvert\boldsymbol{\tau}\rvert$	K	$\lvert\bar{u}\rvert - \lvert\tau_{\boldsymbol{y}}\rvert$
7	7	38	22	1	9
80	60	241	65	13	9
200	150	751	17498	1	148
200	150	601	115	1	12

Table 4.7: Numerical results of Algorithm 11 applied to the facility location problem.

n	Γ	$\lvert\boldsymbol{\tau}\rvert$	K	$\lvert\bar{u}\rvert - \lvert\tau_{\boldsymbol{y}}\rvert$
25	10	47	7	9
600	10	90	13	13
1000	10	48	5	8
1000	45	5550	1	24

Table 4.8: Numerical results of Algorithm 11 applied to the inventory control problem.

n	m	Γ	$\lvert\boldsymbol{\tau}\rvert$	K	$\lvert\bar{u}\rvert - \lvert\tau_{\boldsymbol{y}}\rvert$
10	24	0.1	1492	30	430
10	24	2	10482	1	446

Table 4.9: Numerical results of Algorithm 11 applied to the unit commitment problem.

Results

- When the number of distinct tight constraints found in the training instances is excessively large, we can combine the entire tight constraints into a single union constraints. In other words, the value of K is set to 1. See Table 4.7, for example. In the facility location problem with $n = 200, m = 150, \Gamma = 751$, the entire set of 17498 tight constraints are combined to a single union constraints. Nevertheless, the increase in the number of constraints is relatively small, regarding that the total number of constraints in the deterministic version of this problem is 30350. This result applies similarly to other examples with $K = 1$ as well. Therefore, Algorithm 11 may not add too much additional computational burden even in extreme cases.
- We have shown in Section 4.6.3 that the prediction accuracy for the wait-and-see decisions is very high, even after we apply Algorithm 11 to the training instances. This result holds true regardless of the value of K. This implies that even after reassigning the prediction targets of the training data, accurate machine learning models can still be trained.

4.6.5 Solving ARO with Varying Dimensions

We evaluate the performance of the generalized approach discussed in Section 4.3.6 for problems with varying dimensions. We conduct two experiments on the inventory control problem with $n = 25$ and $\Gamma = 10$.

In the first experiment, we randomly generate five distinct contingencies, each removing a certain portion of the here-and-now decision variables. These contingencies simulate situations where specific ordering options are no longer available.

In the second experiment, we randomly generate five different contingencies, each removing certain non-negativity constraints on the wait-and-see variables. These contingencies simulate situations where certain orders can be canceled without incurring additional costs.

Target	k	Accuracy	Infeasibility	sub_{max}	$\lvert\mathcal{S}\rvert$
$s_{\boldsymbol{x}}$	1	0.99	0	0.0005	19
	5	1.00	0	0.0000	
	10	1.00	0	0.0000	
$s_{\boldsymbol{d}}$	1	0.98	0	0.0005	65
	5	0.99	0	0.0005	
	10	1.00	0	0.0003	
$s_{\boldsymbol{y}}$	1	0.99	0	0.0005	19
	5	1.00	0	0.0000	
	10	1.00	0	0.0000	

Table 4.10: Numerical results for the inventory control problem with varying number of decision variables.

Target	k	Accuracy	Infeasibility	sub_{max}	$\lvert\mathcal{S}\rvert$
$s_{\boldsymbol{x}}$	1	0.99	0	0.0004	19
	5	1.00	0	0.0000	
	10	1.00	0	0.0000	
$s_{\boldsymbol{d}}$	1	0.99	0	0.0004	61
	5	0.99	0	0.0004	
	10	1.00	0	0.0004	
$s_{\boldsymbol{y}}$	1	0.99	0	0.0004	19
	5	1.00	0	0.0000	
	10	1.00	0	0.0000	

Table 4.11: Numerical results for the inventory control problem with varying number of constraints.

Other experimental setups are identical to the descriptions in Section 4.6.1 and 4.6.3. For both experiments, we use a single categorical feature to encode the type of contingency. This results in five distinct categorical values, each corresponding to a specific type of contingency. Moreover, the number of strategies for the wait-and-see variables in these experiments is substantially higher compared to the previous experiment on the same inventory control problem: 90 and 102 for the two experiments, respectively. Therefore, we use Algorithm 11 with $K = 1$ in this section. The main results are presented in Tables 4.10 and 4.11.

Results

- The number of unique strategies identified in the training set is larger compared to the previous experiment (refer to Table 4.2 for comparison). While the number of strategies for the wait-and-see variables might seem smaller, this is due to the use of Algorithm 11, as mentioned earlier. This increased diversity results from the existence of multiple contingencies.
- Our approach continues to achieve near-perfect performance, demonstrating its effectiveness for problems with varying contingencies.

Target	k	Γ	Accuracy	Infeasibility	sub_{max}	$\lvert\mathcal{S}\rvert$	N	t_{ratio}
$s_{\boldsymbol{x}}$	1	2	0.18	0	0.0205	3341	10000	8.49×10^7
	5		0.18	0	0.0205			10
	10		0.18	0	0.0205			10
$s_{\boldsymbol{d}}$	1	2	0.18	0	0.0205	3341	10000	7.07×10^7
	5		0.18	0	0.0205			10
	10		0.18	0	0.0205			10
$s_{\boldsymbol{y}}$	1	2	0.18	0	0.0205	3341	10000	4871
	5		0.18	0	0.0205			10
	10		0.18	0	0.0205			10

Table 4.12: Numerical results for the unit commitment problem with $n = 100, m = 24, k \geq 1$ using OPT.

4.6.6 Solving ARO with Suboptimal Strategies

In this section, we apply Algorithm 8 to solve large scale unit commitment problems using suboptimal strategies. The size of the unit commitment problem we consider is $n = 100$ and $m = 24$, which is much larger than the scale we considered in Section 4.6.3. Throughout the experiment, we set the optimality gap of Gurobi to 0.005. When generating a training set in Phase 1 by solving ARO instances, we set $\epsilon_1 = 0.05$ and $\epsilon_2 = 0.01$ for Algorithm 5 and Algorithm 6, respectively. When computing suboptimalities to generate reward matrices and choose the best among $k > 1$ predictions, we set $\epsilon_2 = 0.001$. When evaluating the final output of the decision trees to assess the ultimate effectiveness of our approach on the test set, we set $\epsilon_1 = 0.001, \epsilon_2 = 0.001$, for precise assessment. Moreover, the number of unique strategies in the training set is prohibitively large in this experiment, as we will demonstrate below. Therefore, we set $Q_1 = Q_2 = Q_3 = 40$. We perform a grid search over the maximum depths 5 and 10 for OPT. Table 4.12 contains the main experiment results. The notations are identical to the previous sections.

Results

- The accuracies are much lower compared to the previous results. However, the maximum suboptimalities are still around 0.02, indicating that the predictions are of reasonable quality. As in Section 4.6.3, the predictions are always feasible.
- When $k = 1$, the solve time using OPT can be more than 10 million times faster than Algorithm 5 . This scale of speed-up is much more drastic than the previous results. However, as k increases, the relative speed-up becomes similar to the previous results.
- The accuracies and maximum suboptimalities are identical across different prediction targets, and the performances do not improve as we increase k.
- Overall, even if we use only 40 out of 3341 strategies found, Algorithm 8 can find high-quality solutions.

Algorithm	$N^{'}$	Proportion	sub_{max}	t_{ratio}
Algorithm 9	1000	0.22	0.0000	2.89
Algorithm 10			0.0001	7.56
Algorithm 9	2000	0.51	0.0000	3.25
Algorithm 10			0.0001	8.95
Algorithm 9	3000	0.71	0.0000	3.45
Algorithm 10			0.0000	9.19
Algorithm 9	4000	0.88	0.0000	4.28
Algorithm 10			0.0000	11.72

Table 4.13: Numerical results of Algorithm 9 and 10 applied to the unit commitment problem.

4.6.7 Analysis of Algorithm 9 and 10

In this section, we assess the effectiveness of Algorithms 9 and 10 using the unit commitment problem with $n = 100$ and $m = 24$. We implement $\mathcal{I}_1$, $\mathcal{I}_2$, and $\mathcal{I}_3$ as feedforward neural networks with two hidden layers, each consisting of 32 neurons. These models are trained using the Adam optimizer [65] with a learning rate of 0.001, implemented in PyTorch [86]. We set $\epsilon_1 = 0.05$ and $\epsilon_2 = 0.01$ for all algorithms. We compare the solution outputs and the runtime of Algorithm 5 with those of Algorithm 9 and 10 on 200 test instances. Consistent with our previous implementation, we use a random $\boldsymbol{x}_0$ for Algorithm 5 and three random initial points for Algorithm 6. We vary the size of the training data for the intermediate models $\mathcal{I}_1$, $\mathcal{I}_2$, and $\mathcal{I}_3$ to observe any performance changes. We use relatively smaller training data compared to Algorithms 7 and 8 to demonstrate the effectiveness of Algorithms 9 and 10 even with a limited dataset. We report the experiment results in Table 4.13.

Table Notations In the columns Algorithm and N', we report the type of acceleration algorithm used (Algorithms 9 or 10) and the number of training samples for $\mathcal{I}_1$, $\mathcal{I}_2$, and $\mathcal{I}_3$, respectively. In the Proportion column, we report the proportion of binary variables in the test set that are fixed or warm-started using the p values decided in the validation set (recall that p is the threshold value for the probability output of $\mathcal{I}_1$ to decide which entries of the output will be used). In the t_{ratio} column, we report the relative speed-up compared to Algorithm 5, rounded to the second decimal place. In the sub_{max} column, we report the maximum suboptimality of the solution output compared with the solution output of Algorithm 5. Note that unlike the experiments in Section 4.6.3, the relative speed-up and the suboptimality are computed with respect to Algorithm 5 and 6 with $\epsilon_1 = 0.05$ and $\epsilon_2 = 0.01$, not the near-optimal version with very tight tolerances.

Results

- As expected, Algorithm 10 outperforms both Algorithm 9 and 5 in terms of speed. Specifically, with just 4000 training data, Algorithm 10 achieves more than a 10-fold speedup compared to Algorithm 5, while Algorithm 9 achieves more than a 4-fold speedup.
- The maximum suboptimality of the solutions is practically negligible across all training

data sizes. This indicates that the solutions generated by Algorithm 9 and 10 are virtually identical to those generated by Algorithm 5.

- As the number of training data increases, the proportion of variables that can be fixed or warm-started also increases, indicating an improvement in the accuracy of $\mathcal{I}_1$. For $N^{'} = 4000$, approximately 88% of the binary variables can already be fixed or warm-started on average. Naturally, as this proportion increases, the solution speed of Algorithm 9 and 10 also improves.

4.7 Conclusions

Despite the theoretical advantages of ARO compared to RO, existing solution algorithms generally suffer from heavy computational burden. We proposed a machine learning approach to solve two-stage ARO with polyhedral uncertainty sets and binary here-and-now variables. We generate multiple ARO instances by varying a key parameter of the problem, and solve them with Algorithm 5. Using the parameters as features, we train a machine learning model to predict high-quality strategies for the here-and-now decisions, the worst-case scenarios associated with the here-and-now decisions, and the wait-and-see decisions. We also proposed learning-based algorithms to expedite training data generation, and a partitioning algorithm to reduce the number of distinct target classes to make the prediction task easier. Numerical experiments on synthetic and real-world problems show that our approach can find high quality solutions of ARO problems significantly faster than the state-of-the-art algorithms.

Chapter 5

A Prescriptive Machine Learning Approach to Mixed-Integer Convex Optimization

5.1 Introduction

We propose a novel prescriptive machine learning approach to speed up the process of solving mixed integer convex optimization (MICO) problems. MICO comprises those optimization problems where the objective and constraint functions are convex in the decision variables, and part of the decision variables are constrained to take integer values. Due to its expressive power, MICO is used in numerous important domains, including auction theory [71], power systems [37], hybrid vehicle control [107] and more [26], [84]. Even so, MICO remains a difficult class of problems to be solved, and solving a large dimensional MICO problem is a computationally demanding task.

Recently, there has been significant interest in the research community in solving challenging optimization problems using machine learning. [2], [63] propose learning efficient branching rules for solving mixed integer optimization problems. [12], [59] use machine learning to automatically tune hyperparameters in optimization algorithms. [39] propose a specialized method to solve MICO problems with logical constraints using neural networks, inspired by robotics problems. On the more theoretical side, [11] analyze the sample complexity of learning which cutting planes to use during the branch-and-cut algorithm of integer optimization solvers. [9], [13] analyze the sample complexity of learning high-quality hyperparameters in optimization algorithms. For a more comprehensive overview on the topic, we refer readers to [18], [73].

Our work builds upon the approach proposed by [24], [25] to solve MICO problems using machine learning. Their idea is to train a classification model that predicts an optimal strategy of an optimization problem by solving multiple similar optimization instances off-line. In the case of MICO problems with binary and continuous variables, the optimal strategy is defined as the set of tight constraints and the set of binary variables that are equal to one. By leveraging the predictions of this model, the process of solving a MICO problem gets considerably simpler compared to solving it from scratch.

However, previous works have given less emphasis on selecting the appropriate machine learning algorithm to achieve better performance. Although the use of Optimal Classification Trees (OCT) [20], [21] in [24] is a crucial part of their work, their focus is interpretability rather than performance. We extend the approach of [24], [25], but propose using a prescriptive instead of a predictive machine learning method.

Predictive machine learning methods aim to learn a model that predicts the ground-truth target given a covariate vector. In this setting, each data point consists of a covariate vector and a corresponding ground-truth target. Most standard supervised learning tasks fall into this category, including classification and regression tasks.

On the other hand, prescriptive machine learning methods aim to learn a policy that assigns a decision to a covariate vector in order to optimize certain outcome [58]. In a prescriptive task, each data point consists of a covariate vector, a decision applied to this point and a corresponding outcome. Unlike predictive machine learning tasks, there is often no notion of a ground-truth target in prescriptive tasks. Application areas include personalized medicine [113], revenue management [33] and hiring decisions [87] among many others. One of the main challenges in prescriptive machine learning that distinguishes it from predictive machine learning is that we often do not know the counterfactuals of applying different decision options. This means that historical data is the only source of information, and the goal is to learn the best policy possible from that data.

In the previous approaches to solve optimization using machine learning, predictive algorithms are mainly used. [24] use OCT, a near globally optimal classification tree. They show that OCT outputs high quality solutions, even comparable to black box models like deep neural networks. We propose using a prescriptive algorithm instead and demonstrate that the ground-truth counterfactuals can be calculated. The prescriptive algorithm we use is Optimal Policy Trees (OPT) [3], which is also based upon the optimal-tree framework. Given a data set where each data point is composed of a covariate vector, a decision applied and a corresponding outcome, OPT learns a decision tree that maps a covariate vector to the best decision. We also propose an extension of the approach that further improves the quality of the predicted strategy.

The high-level idea of our approach is to first generate a collection of instances of a MICO problem and find an optimal strategy for every instance. Then, on every instance, we apply all the strategies we generated to evaluate the associated objective cost and assess feasibility. In this way, we can evaluate the strength of each strategy from a cost and feasibility perspective. Based on this evaluation, we train a decision tree that assigns the best strategy to a MICO instance so that the associated objective cost is minimized.

We do not focus on the relative speed-up of our method compared to commercial off-the-shelf solvers such as Gurobi [56]. The way we speed up the solve process of a MICO problem is identical to the previous works by [24], [25], which is by using the strategy output by a machine learning model. Extensive results on how much this approach can speed up the solve process of MICO problems compared to Gurobi are available in these works. Rather, we focus on comparing the quality of the strategies output by our approach versus the previous approach.

The contributions of the paper are the following. First, we propose a novel approach to use a prescriptive algorithm, OPT, instead of classification algorithms to speed up the process of solving MICO problems. By using a prescriptive algorithm to a potential classification

task, we can take into account the evaluations of all the available strategies for each data point. Second, we propose an extension of the pure OPT approach, which further improves on the suboptimality of the solutions. Finally, we demonstrate empirically that OPT-based methods are much more likely to produce feasible solutions than OCT, and the advantage of OPT-based methods increases as the number of distinct strategies grows. Thus, the proposed approach makes OPT-based methods more appropriate than the previous approach for various real-world scenarios in which avoiding infeasible solutions is crucial, while a slight suboptimality is acceptable.

The structure of the paper is as follows. In Section 2, we briefly review previous works that lay the foundation of our approach. We first review OPT, an optimal-tree based algorithm to assign the best decision to a data point using historical data. Second, we briefly review OCT, an optimal-tree based classification algorithm. Then, we review [24], [25], a machine learning approach to solve MICO problems. In Section 3, we present the approach to use a prescriptive algorithm, OPT(k), to speed up the process of solving MICO problems. We also present an extension of this approach that improves on the suboptimality of the solution. In Section 4, we present the results of computational experiments on synthetic data. The class of problems we consider include linear optimization (LO), mixed integer linear optimization (MILO), quadratic optimization (QO) and mixed integer quadratic optimization (MIQO). In Section 5, we present the results of computational experiments on real-world MILO problems taken from MIPLIB [53].

5.2 Foundations

In this section, we review previous works that serve as the basis of our approach. We review OPT[3] and OCT [20], [21]. Then, we review [24], [25], a machine learning approach to MICO.

5.2.1 Optimal Policy Trees

OPT is an optimal-tree based prescriptive method to learn a decision tree that assigns a decision to a data point, either to maximize or minimize certain outcome. It separates counterfactual estimation and policy learning, and utilizes near globally-optimal trees.

The OPT algorithm requires a training set $\{(\boldsymbol{\theta}_i, s_i, y_i)\}_{i\in[N]}$ consisting of N data points, where $\boldsymbol{\theta}_i$ represents the covariate vector, s_i represents the decision applied and y_i represents the outcome related to point i. Let $\mathcal{S} = \{s_1, \ldots, s_p\}$ $(p \leq N)$ be the set of decision options we have. Without loss of generality, we assume $s_i \neq s_j$ if $i \neq j$.

In OPT, we first estimate the outcome for each point i when decision $s \in \mathcal{S}$ is applied. As the counterfactuals are usually unknown, we use causal inference methods, for example the doubly robust algorithm [81]. The goal is to build what we call the reward matrix $\boldsymbol{R} \in \mathbb{R}^{N\times p}$, where each entry $\boldsymbol{R}_{i,j}$ is the estimated outcome of applying decision $s_j \in \mathcal{S}$ to point i.

Using the reward matrix, OPT learns a decision tree $\mathcal{T}$ that assigns $\boldsymbol{\theta}_i$ to a leaf of the tree and selects which decision to assign to each leaf. The optimization problem that we

Table 5.1: First five rows of the reward matrix for the synthetic adverstisement assignment problem.

	Ad1	Ad2
1	409.407	1121.5
2	716.539	774.948
3	1019.76	351.229
4	652.254	726.154
5	391.693	1032.29

solve to learn $\mathcal{T}$ is the following.

$$\min_{v(\cdot),s} \quad \sum_{i=1}^{N} \sum_{\ell} \mathbb{1}\{v(\boldsymbol{\theta}_i) = \ell\} \cdot \boldsymbol{R}_{i,s_\ell}, \tag{5.1}$$

where $v(\boldsymbol{\theta}_i)$ is the leaf of the tree $\boldsymbol{\theta}_i$ is assigned to, s_ℓ is the decision assigned to the points in leaf ℓ and $\boldsymbol{R}_{i,s_\ell}$ is the estimated outcome for point i under the decision s_ℓ. For a more detailed discussion on optimization methods for the optimal-tree framework, see [20], [21], [38].

Given the policy tree $\mathcal{T}$ and a new data point, we traverse the policy tree according to its features until we reach a leaf node. At the leaf node, we have assigned the optimal decision based on the learned policy, which is then assigned to the new data point.

A generalization of OPT outputs a collection of $k \geq 1$ decisions that are most likely to be optimal [60]. Given a data point, we find the leaf of the tree containing this point and then calculate the average outcome in this leaf under each decision. Then, we find k best decisions, using the calculated outcomes. Note that the first best decision among the k best decisions is simply the decision that this leaf is assigned to in $\mathcal{T}$. Hence, setting $k = 1$ makes this equivalent to OPT. We denote this generalization as OPT(k).

To illustrate OPT, we present an example using synthetic data. We generated customer data composed of age, average monthly spending, the advertisement type exposed to each customer in the last month and the resulting revenue in the last month. Here, age and average monthly spending are the covariates, the advertisement type in the last month is the historical decision and the revenue in the last month is the historical outcome. Our goal is to use this historical data to assign an advertisement to the customers in order to maximize revenue in the next month.

We generate the data set in a manner so that Advertisement 1 is better suited for people who are older or spend less and Advertisement 2 is better suited for younger people or people who spend more. We generate 1000 training samples. Ages are integers uniformly sampled from 10 to 60. Spendings are uniformly sampled from 100 to 1100. Advertisement is randomly assigned to each customer. If the assigned advertisement is Advertisement 1, the historical outcome is computed as $(\text{age} \times 5) + (1100 - \text{spending}) + \epsilon$, where $\epsilon \sim \mathcal{N}(0, 50)$. If the assigned advertisement is Advertisement 2, the historical outcome is computed as $((60 - \text{age}) \times 5) + \text{spending} + \epsilon$, where $\epsilon \sim \mathcal{N}(0, 50)$.

First, the reward matrix is estimated using the doubly-robust estimation. The resulting

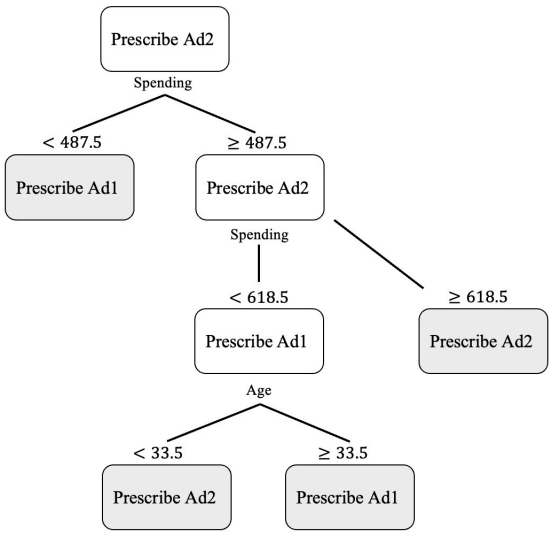

Figure 5.1: Decision tree obtained by OPT for the synthetic advertisement assignment problem.

reward matrix is given in Table 5.1. Column names "Ad1" and "Ad2" are the advertisement options. The $(i, j)_{th}$ entry, $i \in [10], j \in [2]$ is the (estimated) outcome when applying advertisement j to data point i. The trained tree is provided in Figure 5.1. We can see that OPT assigns an optimal advertisement according to a customer's age and spending.

5.2.2 Optimal Classification Trees

OCT is an optimal-tree based classification algorithm [20], [60]. Classification and regression trees (CART) [34] use greedy algorithm to train decision trees. In contrast, OCT aims to find a near globally optimal decision tree for classification tasks. OCT-H is a generalization of OCT, which allows hyperplane splits instead using a single variable for splits. OCT shows lower classification error compared to CART. It is also more interpretable as the resulting tree is less complex than the tree produced by CART [20], [21].

Just like OPT, OCT can output a collection of $k \geq 1$ predictions given a new data point. OCT can output a vector where the i_{th} entry of this vector represents the likelihood that the i_{th} class is the true class for this data point. Using this vector, we can identify multiple most likely predictions. We denote this generalization of OCT as OCT(k), where OCT(1) is equivalent to OCT.

5.2.3 A Machine Learning Approach to MICO

We review a machine learning approach to MICO proposed in [24], [25]. This approach stems from the idea that, in practice, similar optimization problems are solved repeatedly with only a slight variation in key parameters.

We consider the following optimization problem.

$$\begin{aligned} \min \quad & f(\boldsymbol{x}, \boldsymbol{\theta}) \\ s.t. \quad & g_1(\boldsymbol{x}, \boldsymbol{\theta}) \leq \mathbf{0}, \\ & \vdots \\ & g_m(\boldsymbol{x}, \boldsymbol{\theta}) \leq \mathbf{0}, \\ & \boldsymbol{x}_{\mathcal{I}} \in \{0,1\}^d, \end{aligned} \tag{5.2}$$

where $\boldsymbol{x} \in \mathbb{R}^n$ is the vector of decision variables and $\boldsymbol{\theta} \in \mathbb{R}^\ell$ is the vector of key parameters that will be used as covariates. The set of indices for the decision variables constrained to take binary values is denoted $\mathcal{I}$, where $|\mathcal{I}| = d$. Functions $f : \mathbb{R}^n \times \mathbb{R}^\ell \mapsto \mathbb{R}$ and $g_j : \mathbb{R}^n \times \mathbb{R}^\ell \mapsto \mathbb{R}, j \in [m]$, are assumed to be convex in $\boldsymbol{x}$.

The method consists of three phases. In Phase 1, we first generate N parameters $\{\boldsymbol{\theta}_i\}_{i\in[N]}$. For each $\boldsymbol{\theta}_i$, we solve problem (5.2) after fixing $\boldsymbol{\theta} = \boldsymbol{\theta}_i$. Convexity of f and g is important in this phase, as we have to solve problem (5.2) to optimality which is in general harder for nonconvex problems.

After solving the instance associated with $\boldsymbol{\theta}_i$ and acquiring an optimal solution $\boldsymbol{x}^*(\boldsymbol{\theta}_i)$, we extract the optimal strategy associated with the solution, denoted as $s(\boldsymbol{\theta}_i)$. In order to define the optimal strategy, we first define the tight constraints as $\tau(\boldsymbol{\theta}_i) = \{j \in [m] \mid g_j(\boldsymbol{x}^*(\boldsymbol{\theta}_i), \boldsymbol{\theta}_i) = 0\}$. If $\mathcal{I} = \emptyset$, the optimal strategy consists of the tight constraints, that is, $s(\boldsymbol{\theta}_i) = \tau(\boldsymbol{\theta}_i)$. If $\mathcal{I} \neq \emptyset$, the optimal strategy consists of which variables are equal to one as well as the tight constraints. In other words, $s(\boldsymbol{\theta}_i) = (\boldsymbol{x}^*_{\mathcal{I}}(\boldsymbol{\theta}_i), \tau(\boldsymbol{\theta}_i))$ Solving an instance becomes considerably simpler once we know its optimal strategy. For general MICO problems, we can fix the integer variables to the values specified in the optimal strategy and impose only the tight constraints. Then, we solve the resulting continuous optimization problem. For MIQO problems, this procedure can be even simpler, which can be reduced to solving a linear system based on Karush-Kuhn-Tucker (KKT) optimality conditions [25]. Hence, the optimal strategies are the classes we will try to predict in Phase 3.

In Phase 2, we train a classification model that maps the generated parameters to the optimal strategies of associated MICO instances. The training data is $\Big\{\big(\boldsymbol{\theta}_i, s(\boldsymbol{\theta}_i)\big)\Big\}_{i\in[N]}$, where $\boldsymbol{\theta}_i$ is a covariate vector and $s(\boldsymbol{\theta}_i)$ its corresponding target.

In Phase 3, given a new parameter $\boldsymbol{\theta}_0$, we predict a strategy $\hat{s}(\boldsymbol{\theta}_0)$ using the classification model trained in Phase 2. Then we apply $\hat{s}(\boldsymbol{\theta}_0)$ to the instance associated with $\boldsymbol{\theta}_0$ to compute a solution $\hat{\boldsymbol{x}}(\boldsymbol{\theta}_0)$.

To evaluate the quality of the predictions in the numerical experiments, the suboptimality of the strategy $\hat{s}(\boldsymbol{\theta}_0)$ is defined as

$$sub\Big(\hat{s}(\boldsymbol{\theta}_0)\Big) = \Big(f\Big(\hat{\boldsymbol{x}}(\boldsymbol{\theta}_0), \boldsymbol{\theta}_0\Big) - f\Big(\boldsymbol{x}^*(\boldsymbol{\theta}_0), \boldsymbol{\theta}_0\Big)\Big) / \Big|f\Big(\boldsymbol{x}^*(\boldsymbol{\theta}_0), \boldsymbol{\theta}_0\Big)\Big|,$$

if $\hat{\boldsymbol{x}}(\boldsymbol{\theta}_0)$ is feasible.

[24] compare the performance of OCT, OCT-H, and neural networks under this framework. They found that both OCT and OCT-H are highly accurate, comparable to neural networks.

While neural networks are known for their accuracy in many domains, they are often considered as black-box algorithms lacking interpretability. In contrast, OCT has a simple tree structure that is by design interpretable, allowing users to understand how the algorithm made its decision. This result suggests that the interpretability of OCT can be leveraged without sacrificing accuracy. Furthermore, the study demonstrates that the solve time using this approach can be up to three orders of magnitude faster than Gurobi.

Just as OCT, classification algorithms such as neural network output a vector where the i_{th} entry of this vector represents the likelihood that the i_{th} label is the true label. Given a new parameter $\boldsymbol{\theta}_0$, we identify the k most likely optimal strategies using this vector. We then evaluate the objective cost and the infeasibility associated with each strategy to choose the best one. [25] demonstrate in detail that increasing k results in significantly better performance, even when the number of distinct strategies in the training set is very large.

5.3 A Prescriptive Machine Learning Approach to MICO

This section outlines our novel approach for solving MICO problems using OPT(k). First, we introduce the theoretical motivation of the approach and present the algorithm in detail. Then, we provide an example to illustrate its application to the facility location problem. To address one of the key weaknesses of the approach, we introduce an extension to the algorithm as well.

5.3.1 Learning Objective: The Edge of Prescriptive over Predictive Machine Learning

We introduce an abstract learning objective that formalizes our approach. We assume that a key parameter $\boldsymbol{\theta}$ belongs to a set Θ and is drawn from a fixed distribution $Pr(\Theta)$ over Θ. We assume that an optimal strategy belongs to a set $\mathcal{S}$. A policy $h : \Theta \mapsto \mathcal{S}$, which belongs to a hypothesis space $\mathcal{H}$, is a mapping from a key parameter to a strategy. Let $v : \Theta \times \mathcal{S} \mapsto \mathbb{R}$ be the objective cost that we get by applying a strategy to the optimization instance associated with a parameter. Our ultimate learning objective can be expressed as

$$\min_{h \in \mathcal{H}} \mathbb{E}_{\boldsymbol{\theta} \sim Pr(\Theta)} \Big[v(\boldsymbol{\theta}, h(\boldsymbol{\theta})) \Big].$$

Notice that OPT(k) optimizes an empirical version of this objective, where $\mathcal{H}$ is the space of decision trees, expectation is replaced with summation and $v(\boldsymbol{\theta}, h(\boldsymbol{\theta}))$ is replaced with the reward matrix in problem (5.1).

Classification algorithms, however, do not try to optimize the above learning objective. Instead, a learning objective for classification algorithms can be expressed as

$$\min_{h \in \mathcal{H}} \mathbb{E}_{\boldsymbol{\theta} \sim Pr(\Theta)} \Big[\mathbb{1}_{\{h(\boldsymbol{\theta}) \neq \arg\min_{s \in \mathcal{S}} v(\boldsymbol{\theta}, s)\}} \Big].$$

In a sense, classification algorithms optimize a surrogate objective instead of optimizing the actual objective. This observation implies that using prescriptive instead of predictive algorithms might be better in the context of solving MICO problems.

In Section 5.4 and 5.5, we compare the performance of OPT(k) and OCT(k) on various MICO problems. OPT(k) and OCT(k) share the same hypothesis space $\mathcal{H}$, which is the space of decision trees. We believe this fact makes the comparison more informative, as the approximation capabilities of the mappings that OPT(k) and OCT(k) learn are the same. Hence, any performance gap can be attributed to the difference between the learning objectives prescriptive and predictive algorithms try to optimize.

Remark Note that the above formalization of the learning objectives can be generalized to other approaches to incorporate machine learning into optimization algorithms. For instance, consider the setting where we want to learn the best hyperparameter of an optimization algorithm to minimize the running time. Then $\mathcal{S}$ can simply be translated to the set of hyperparameters, and $v(\boldsymbol{\theta}, s)$ is the running time of the algorithm when we choose the hyperparameter s on the instance associated with $\boldsymbol{\theta}$. Similar discussions can be found at [18].

5.3.2 The Prescriptive Algorithm

Algorithm 12: OPT(k) for MICO.

Input: $\bar{\boldsymbol{\theta}}$, $f(\cdot)$, $\{g_i(\cdot)\}_{i\in[m]}$, $\mathcal{I}$, N, r, M, k.
Output: Decision Tree $\mathcal{T}$.

1. We generate parameters $\{\boldsymbol{\theta}_i\}_{i\in[N]}$ uniformly from the Ball $\mathcal{B}(\bar{\boldsymbol{\theta}}, r)$. For each $\boldsymbol{\theta}_i$, we solve problem (5.2) after fixing $\boldsymbol{\theta} = \boldsymbol{\theta}_i$ to get an optimal solution and obtain the associated optimal strategy, $s(\boldsymbol{\theta}_i)$. In the case of multiple optimal solutions, we select arbitrarily one of them and obtain the associated optimal strategy. We denote the set of strategies generated in this step as $\mathcal{S} = \{s_1, \ldots, s_\ell\}$, where $s_k \neq s_j$ if $k \neq j$ ($\ell \leq N$ because optimal strategies of different instances may overlap).
2. For each $\boldsymbol{\theta}_i$, we apply $s_j \in \mathcal{S}, s_j \neq s(\boldsymbol{\theta}_i)$ to the instance associated with $\boldsymbol{\theta}_i$ and recover a solution $\hat{\boldsymbol{x}}_j$. We compute $f(\hat{\boldsymbol{x}}_j, \boldsymbol{\theta}_i)$ and also assess feasibility. If $\hat{\boldsymbol{x}}_j$ is infeasible, then $f(\hat{\boldsymbol{x}}_j, \boldsymbol{\theta}_i) := M$, where M is a large number. In this step, we compute the reward matrix $\boldsymbol{R} \in \mathbb{R}^{N\times\ell}$, where $\boldsymbol{R}_{i,j} := f(\hat{\boldsymbol{x}}_j, \boldsymbol{\theta}_i)$.
3. We construct an optimal tree $\mathcal{T}$ using the reward matrix $\boldsymbol{R}$ computed in step 2.
4. Given a new parameter $\boldsymbol{\theta}_0$, we use the tree $\mathcal{T}$ to find the k-best strategies, and apply all of them to select the best one.

Given problem (5.2), we provide the proposed prescriptive approach in Algorithm 12. A key difference of Algorithm 12 from the OPT(k) described in Section 2.1 is that we do not need to estimate the counterfactuals. Rather, we compute them directly by applying the available set of strategies to each instance in the training set. The outcome of applying a strategy to an instance is measured by the resulting objective cost and feasibility. The way we apply a strategy to a MICO instance is identical to the general description in [24]. We fix the integer variables to the values specified in the strategy and impose only the tight constraints. We solve the resulting continuous optimization problem to recover a solution. For the rest of the paper, we denote Algorithm 12 as OPT(k).

5.3.3 Example

We present an example of the proposed approach applied to the facility location problem. Facility location is a MILO problem that involves determining the optimal locations for facilities and the optimal amount of goods to transport from each location to each destination. The unit cost of delivering goods from location $i \in [n]$ to destination $j \in [m]$ is given by c_{ij}, while the cost of building a facility at location i is represented by f_i. The capacity of a facility at location i is denoted by p_i, and d_j represents the demand from destination j. The facility location problem involves two main decisions: selecting which facility to build among the n possible locations, which is represented by the binary variable x_i, and deciding the optimal amount of goods to transport from each location i to each destination j, which is represented by the continuous variable y_{ij}. The parameter vector (the covariate vector for training) we choose is the demand vector. The exact model is

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}y_{ij} + \sum_{i=1}^{n} f_i x_i \\
s.t. \quad & \sum_{i=1}^{n} y_{ij} \geq d_j, \quad \forall j \in [m], \\
& \sum_{j=1}^{m} y_{ij} \leq p_i x_i, \quad \forall i \in [n], \\
& y_{ij} \geq 0, \quad \forall i \in [n],\ \forall j \in [m], \\
& \boldsymbol{x} \in \{0,1\}^n.
\end{aligned}
$$

First, we describe the computation of the optimal strategies and the reward matrix using a very small sized example with $n = 2$, $m = 1$. Let the indices of the constraints $\sum_{i=1}^{n} y_{i1} \geq d_1, \sum_{j=1}^{1} y_{1j} \leq p_1 x_1, \sum_{j=1}^{1} y_{2j} \leq p_2 x_2, y_{11} \geq 0, y_{21} \geq 0$ be $1, 2, 3, 4, 5$ respectively. We fixed $p_1 = 10, p_2 = 15$, sampled c_i from $U(0, 10)$ and f_{ij} from $U(0, 10)$. We consider only two instances: $d_1 = 1$ and $d_1 = 20$. For the instance with $d_1 = 1$, the optimal solution is $(x_1, x_2, y_{11}, y_{21}) = (0, 1, 0, 1)$ with the objective cost 8.76. That is, we only build the facility on location 2 and satisfy the entire demand from this facility. The constraints that are tight under this solution are $\sum_{i=1}^{2} y_{i1} \geq d_1$, $\sum_{j=1}^{1} y_{1j} \leq p_1 x_1$ and $y_{11} \geq 0$. Thus, the optimal strategy $(\boldsymbol{x}^*, \tau)$ (recall that τ is the indices of the tight constraints) is $((0, 1), \{1, 2, 4\})$. We denote this strategy as s_1 for now. For the instance with $d_1 = 20$, the optimal solution is $(x_1, x_2, y_{11}, y_{21}) = (1, 1, 5, 15)$ with the objective cost 87.02. In this solution, we build facilities on both locations and the facility on location 2 reaches its full capacity to satisfy the demand. The constraints that are tight under this solution are $\sum_{i=1}^{2} y_{i1} \geq d_1$ and $\sum_{j=1}^{1} y_{2j} \leq p_2 x_2$. The optimal strategy is $(\boldsymbol{x}^*, \tau) = ((1, 1), \{1, 3\})$. We denote this strategy as s_2 for now. We now have two different strategies in our data set. The next step is to apply s_1 to the instance associated with $d_1 = 20$ and apply s_2 to the instance associated with $d_1 = 1$ to compute the reward matrix. To apply s_1 to the instance with $d_1 = 20$, we fix $d_1 = 20, x_1 = 0, x_2 = 1$ and impose only the constraints $\{1, 2, 4\}$. The resulting solution $(x_1, x_2, y_{11}, y_{21}) = (0, 1, 0, 20)$ is infeasible to the original problem, as it violates the capacity constraint on location 2. Hence we assign an arbitrary large number M to this case. Following the same procedure, we can

Table 5.2: The reward matrix of the facility location problem with $n = 2$ and $m = 1$.

	$s1$	$s2$
1	8.76	M
2	M	87.02

Table 5.3: First five rows of the reward matrix of the facility location problem with $n = 10$ and $m = 20$.

	$s1$	$s2$	$s3$	$s4$	$s5$	$s6$	$s7$	$s8$
1	36.29	10000.0	37.84	37.26	10000.0	37.82	38.58	10000.0
2	34.67	10000.0	37.70	36.30	10000.0	35.25	35.68	10000.0
3	32.78	10000.0	37.20	37.46	10000.0	35.78	36.74	10000.0
4	36.32	10000.0	38.49	37.41	10000.0	33.39	33.85	10000.0
5	36.37	10000.0	37.47	37.59	10000.0	37.26	38.25	10000.0

find that applying s_2 to the instance with $d_1 = 1$ also leads to an infeasible solution. The reward matrix that we get is given in Table 5.2.

Now, we demonstrate our approach on a bigger sized example with $n = 10$ and $m = 20$. We skip the details on the computation in this part and provide the decision trees that OPT(k) and OCT(k) learned. The maximum depth of the trees is limited to 3 for the sake of simplicity. We sampled 7000 training instances uniformly from the ball of radius 2 and found 8 different strategies. Without loss of generality, let $\mathcal{S} = \{s1, \ldots, s8\}$. Table 5.3 depicts the reward matrix that we computed. We use $M = 10000$ as the penalty to infeasibility.

In step 3, OPT(k) uses the reward matrix illustrated in Table 5.3 to construct a decision tree by solving problem (5.1). As the cost of predicting infeasible solutions is very large, it will try to avoid infeasible solutions.

Figures 5.2 and 5.3 are the decision trees learned with OCT(k) and OPT(k), respectively. We can see that the resulting decision trees are actually very different, although the features that contribute to the splits are identical. In particular, we observe that the number of splits is larger for OCT(k), which might suggest overfitting.

5.3.4 An Extension of OPT(k)

In the computational experiments provided in Section 5.4, we have observed that OPT(k) finds feasible solutions much more frequently than OCT(k). On the other hand, OCT(k) finds slightly better solutions when they are feasible. We propose an extension of OPT(k) that aims to improve on this slight suboptimality. Rather than using the top-k predictions of decision tree learned with OPT, we only use the top-$(k - Q)$ predictions. The remaining Q strategies are replaced by the Q most frequently occurring strategies in the training set. The exact algorithm is given in Algorithm 13. Note that setting $Q = 0$ makes Algorithm 13 equivalent to OPT(k). For the rest of the paper, we denote Algorithm 13 as OPT(k, Q).

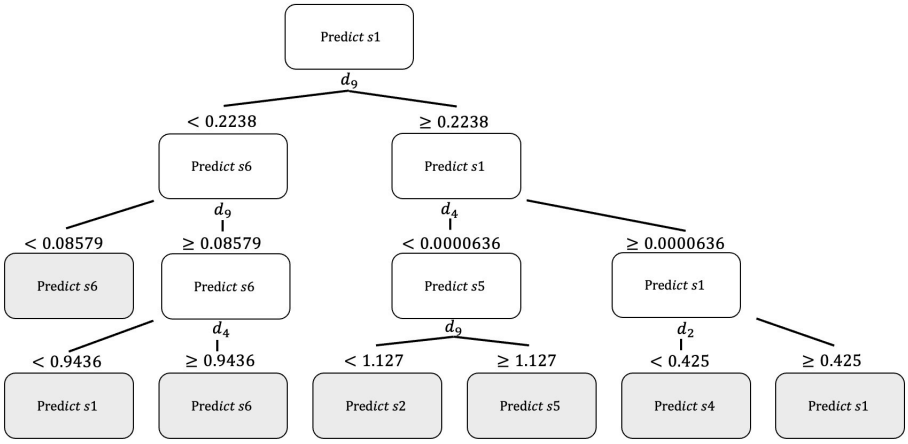

Figure 5.2: Decision tree learned with OCT(k) for the facility location problem. $d9, d4, d2$ are the demands from the destination 9,4,2, respectively.

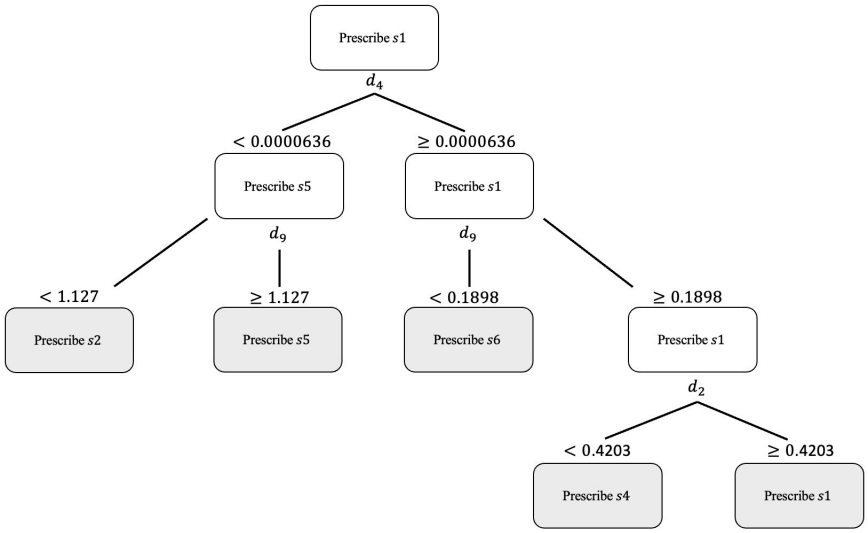

Figure 5.3: Decision tree learned with OPT(k) for the facility location problem. $d9, d4, d2$ are the demands from the destination 9,4,2, respectively.

Algorithm 13: OPT(k, Q) for MICO.

Input: $\bar{\boldsymbol{\theta}}$, $f(\cdot)$, $\{g_i(\cdot)\}_{i\in[m]}$, $\mathcal{I}$, N, r, M, k, Q.
Output: Decision Tree $\mathcal{T}$, $\mathcal{S}_Q$.

1. After step 1 of Algorithm 12, we identify the Q most frequently occurring strategies and denote them as $\mathcal{S}_Q$.
2. We train an optimal tree $\mathcal{T}$ using only the strategies $\mathcal{S} \setminus \mathcal{S}_Q$ and the corresponding reward matrix.
3. Given an instance $\boldsymbol{\theta}_0$, we use the tree $\mathcal{T}$ to find the $k - Q$ best strategies. We apply all of them and also $\mathcal{S}_Q$ to select the best.

5.4 Computational Experiments on Synthetic Data

In this section, we first conduct a comprehensive analysis of the performance of OPT(1) and OCT(1) on synthetic data. The synthetic examples we consider include LO, QO, MILO, and MIQO problems. We also investigate the effect of the penalty M on the performance of OPT(k) and the training time required to learn a decision tree in OPT(k). Additionally, we compare the training times of OPT(k) and OCT(k), as well as the online solve times of OPT(k) and Gurobi. The code for our implementation is available at https://github.com/acwkim/ml_mico.

5.4.1 Experimental Settings and Problem Descriptions

Software for OPT and OCT is available at [60]. We used Gurobi to solve MICO problems and the continuous optimization problems induced by the strategies. For decision tree training, we minimized hyperparameter tuning process by grid searching over the maximum depths 5 and 10.

The number of training instances is 7000 and the number of test instances is 3000, unless noted otherwise. We regard a prediction accurate if it is feasible and the suboptimality is smaller than 0.001. We report the number of accurate, suboptimal, feasible and infeasible predictions as well as the maximum suboptimality among the feasible predictions made on the test set. By definition, the number of feasible predictions is the total of suboptimal and accurate predictions. If the maximum suboptimality is smaller than 0.0001, we simply report it as 0. The duration of the entire training process starting from the hyperparameter tuning is measured to determine the training times. In the following tables, we use $|\mathcal{S}|$ to denote the number of distinct strategies in the training set and sub_{max} to denote the maximum suboptimality.

We now describe the synthetic problems considered in the experiments. We also provide data generation details for the experiments in Section 5.4.2.

Transportation Optimization Transportation optimization is a LO problem to minimize the total cost of transporting goods from warehouses $i \in [n]$ to destinations $j \in [m]$. The unit cost of delivering goods from warehouse i to destination j is denoted as c_{ij}, while p_i and d_j denote the stock in warehouse i and the demand from destination j, respectively. The decision variable x_{ij} denotes the quantity of goods shipped from warehouse i to a destination j. The problem can be formulated as follows:

$$\begin{aligned}
\min \quad & \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}x_{ij} \\
s.t. \quad & \sum_{i=1}^{n} x_{ij} \geq d_j, \quad \forall j \in [m], \\
& \sum_{j=1}^{m} x_{ij} \leq p_i, \quad \forall i \in [n], \\
& x_{ij} \geq 0, \quad \forall i \in [n],\ \forall j \in [m].
\end{aligned}$$

The instances are generated using the key parameter vector $\boldsymbol{d} = (d_1, \ldots, d_m)$, which is drawn uniformly from $\mathcal{B}(\bar{\boldsymbol{d}}, 0.5)$. Here, each entry of $\bar{\boldsymbol{d}}$ is independently drawn from the uniform distribution $U(1, 6)$, and is fixed for all instances in the experiment. c_{ij} is drawn from the uniform distribution $U(0, 10)$, while p_i is drawn from $U(3, 13)$.

Portfolio Optimization

$$\begin{array}{ll} \max & \boldsymbol{\mu}^T \boldsymbol{x} - \gamma(\boldsymbol{x}^T \boldsymbol{\Sigma} \boldsymbol{x}) \\ s.t. & \boldsymbol{e}^T \boldsymbol{x} = 1, \\ & \boldsymbol{x} \geq \boldsymbol{0}. \end{array}$$

The instances are generated using the key parameter vector $\boldsymbol{\mu}$. The parameters are drawn uniformly from $\mathcal{B}(\bar{\boldsymbol{\mu}}, 0.15)$, where each entry of $\bar{\boldsymbol{\mu}}$ is drawn from $U(0, 3)$. The entries of $\boldsymbol{F}$ are chosen to be nonzero with 50 % probability and nonzero entries are drawn from $N(0, 1)$. The entries of $\boldsymbol{D}$ are drawn from $U(0, \sqrt{m})$. The risk-aversion coefficient is fixed to 1. Since this is a maximization problem, we assign $M = 0$ to an infeasible solution.

Facility Location We also test the approach on the facility location problem described in Section 5.3.3. The key parameter vector is again $\boldsymbol{d} = (d_1, \ldots, d_m)$. The parameters are drawn uniformly from $\mathcal{B}(\bar{\boldsymbol{d}}, 0.4)$, where each entry of $\bar{\boldsymbol{d}}$ is drawn from the uniform distribution $U(5, 6)$. c_i is drawn from $U(0, 10)$, p_i from $U(10, 18)$ and f_{ij} from $U(0, 10)$. We assign $M = 10000$ to an infeasible solution.

Hybrid Vehicle Control We consider the hybrid vehicle control problem taken from [107]. Hybrid vehicle control is a MIQO problem to plan the battery and the engine power outputs P_t^{batt} and P_t^{eng} at each time $t \in \{0\} \cup [T-1]$ while satisfying the power demand. The internal energy and the power demand at time t are represented by E_t and P_t^{des}, respectively. The on-off state of engine, and the cost of turning on the engine at time t are represented by z_t and $\delta(z_t - z_{t-1})$, respectively. The stage power cost f is defined as $f(P, z) = \alpha P^2 + \beta P + \gamma z$. The problem can be formulated as follows:

$$\begin{array}{ll} \min & \eta(E_T - E^{\max})^2 + \sum_{t=0}^{T-1} f(P_t^{\text{eng}}, z_t) + \delta(z_t - z_{t-1}) \\ s.t. & E_{t+1} = E_t - \tau P_t^{\text{batt}}, \quad \forall t \in \{0\} \cup [T-1], \\ & 0 \leq E_t \leq E^{\max}, \quad \forall t \in \{0\} \cup [T], \\ & E_0 = E_{\text{init}}, \\ & 0 \leq P_t^{\text{eng}} \leq P^{\max}, \quad \forall t \in \{0\} \cup [T-1], \\ & P_t^{\text{batt}} + P_t^{\text{eng}} \geq P_t^{\text{des}}, \quad \forall t \in \{0\} \cup [T-1], \\ & z_t \in \{0, 1\}, \quad \forall t \in \{0\} \cup [T-1]. \end{array}$$

The key parameter vector $\boldsymbol{P}^{\text{des}} = (P_0^{\text{des}}, \ldots, P_{T-1}^{\text{des}})$ is drawn from $\mathcal{B}(\bar{\boldsymbol{P}}_{1:T}^{\text{des}}, 0.5)$, where $\bar{\boldsymbol{P}}_{1:T}^{\text{des}}$

Table 5.4: Transportation Optimization.

Learner	n	m	Accurate	Suboptimal	Feasible	Infeasible	sub_{max}	$\lvert\mathcal{S}\rvert$
OPT(1)	20	10	3000	0	3000	0	0	12
OCT(1)			2299	0	2299	701	0	
OPT(1)	40	20	2991	0	2991	9	0	5
OCT(1)			2639	0	2639	361	0	
OPT(1)	60	30	3000	0	3000	0	0	3
OCT(1)			3000	0	3000	0	0	
OPT(1)	80	40	2988	0	2988	12	0	6
OCT(1)			2913	0	2913	87	0	

denotes the first T entries of the vector

$$\begin{aligned}\bar{\boldsymbol{P}}^{\text{des}} = (&0.05, 0.30, 0.55, 0.80, 1.05, 1.30, 1.55, 1.80, 1.95, 1.70, 1.45, 1.20, 1.02,\\ &1.12, 1.22, 1.32, 1.42, 1.52, 1.62, 1.72, 1.73, 1.38, 1.03, 0.68, 0.33, -0.02, -0.37, -0.72,\\ &-0.94, -0.64, -0.34, -0.04, 0.18, 0.08, -0.02, -0.12, -0.22, -0.32, -0.42, -0.52).\end{aligned}$$

We let $\alpha = \beta = \gamma = 1, \delta = 0.1$, $\tau = 4$, $E^{\max} = 50, P^{\max} = 1$ and sample E_0 uniformly from $\mathcal{B}(40, 0.5)$. We assign $M = 1000000$ to an infeasible solution.

5.4.2 Comparison of OPT(1) and OCT(1)

We compare the performance of OPT(1) and OCT(1), which is the setting where the difference between the two algorithms can be the most drastic. Tables 5.4, 5.6, 5.5, 5.7 contain the experiment results on transportation optimization, portfolio optimization, facility location and hybrid vehicle control, respectively.

We draw the following conclusions:

- OPT(1) outperforms OCT(1) in terms of finding feasible solutions, as OPT(1) finds feasible solutions much more frequently than OCT(1).
- When the predicted solution is feasible, OPT(1) tends to have slightly larger suboptimality compared to OCT(1).
- Maximum suboptimalities of OPT(1) and OCT(1) are still similar.
- One possible explanation for the above observations is that OPT(1) is designed to place more emphasis on avoiding infeasible solutions. In the reward matrix used for training a policy, infeasible solutions are assigned very large penalty numbers as opposed to only slightly suboptimal solutions. In contrast, OCT(1) is unable to utilize the varying degrees of suboptimalities and infeasibilities of existing strategies.

Table 5.5: Facility Location.

Learner	n	m	Accurate	Suboptimal	Feasible	Infeasible	sub_{max}	$\|\mathcal{S}\|$
OPT(1)	20	10	3000	0	3000	0	0	3
OCT(1)			2963	0	2963	37	0	
OPT(1)	40	20	2992	0	2992	8	0	9
OCT(1)			2790	0	2790	210	0	
OPT(1)	60	60	2984	0	2984	16	0.0010	12
OCT(1)			2814	0	2814	186	0.0007	
OPT(1)	80	40	3000	0	3000	0	0	5
OCT(1)			2917	0	2917	83	0	

Table 5.6: Portfolio Optimization.

Learner	n	m	Accurate	Suboptimal	Feasible	Infeasible	sub_{max}	$\|\mathcal{S}\|$
OPT(1)	100	10	3000	0	3000	0	0	10
OCT(1)			2842	0	2842	158	0	
OPT(1)	200	20	3000	0	3000	0	0	7
OCT(1)			2618	0	2618	392	0	
OPT(1)	300	30	3000	0	3000	0	0	13
OCT(1)			2364	0	2364	636	0	
OPT(1)	400	40	3000	0	3000	0	0	3
OCT(1)			3000	0	3000	0	0	

Table 5.7: Hybrid Vehicle Control.

Learner	T	Accurate	Suboptimal	Feasible	Infeasible	sub_{max}	$\|\mathcal{S}\|$
OPT(1)	10	3000	0	3000	0	0.0004	8
OCT(1)		3000	0	3000	0	0	
OPT(1)	20	2959	9	2968	32	0.0016	10
OCT(1)		2937	0	2937	63	0.0002	
OPT(1)	30	2986	0	2986	14	0.0003	8
OCT(1)		2984	0	2984	16	0	
OPT(1)	40	2988	1	2989	11	0.0011	12
OCT(1)		2986	0	2986	14	0	

Table 5.8: The effect of the penalty M on the performance of OPT(1).

Accurate	Suboptimal	Feasible	Infeasible	sub_{max}	M	obj_{max}
2799	156	2955	45	0.0231	3279	
2798	156	2954	46	0.0231	3500	
2790	161	2951	49	0.0451	4500	
2797	156	2953	47	0.0231	5000	3278.2
2797	156	2953	47	0.0231	10000	
2797	156	2953	47	0.0231	100000	
2797	156	2953	47	0.0231	1000000	

5.4.3 On the Choice of the Penalty M

We analyze how the size of the penalty M with respect to the largest objective cost found in the training points affects the accuracy and the training time of OPT(k). In the following tables, we use obj_{max} to denote the largest objective cost found in the training points.

We consider the hybrid vehicle control with $T = 10$. The parameters are sampled uniformly from $\mathcal{B}(\bar{\boldsymbol{P}}^{\text{des}}_{1:10}, 1)$ to generate a training set. For this single training set, we train multiple decision trees with OPT(k), but using different values of M. We provide the experiment results in Table 5.8. We observe that even though we vary M quite extensively, the resulting trees show very similar results on the test set. This implies that tuning the number M to control the behavior of OPT(k) might not be an effective approach.

Now, we analyze the effect of M on the training time of OPT(k). We vary the time horizon T in the hybrid vehicle control problem and generate twenty different training sets for each time horizon. The parameters are sampled uniformly from $\mathcal{B}(\bar{\boldsymbol{P}}^{\text{des}}_{1:T}, 0.5)$. For each training set, we implement OPT(k) using varying values of M chosen with respect to the largest objective cost found in the training set. Then we compute mean and standard deviation of the training times for each choice of M. Table 5.9 contains the experiment results. In Table 5.9, *avg* and *std* denote the rounded average and the rounded standard deviation of the training times measured in seconds, respectively. We observe that there is no clear correlation between the size of M and the training time. This observation, combined with the previous discussion, suggests that OPT(k) is largely insensitive to the choice of M, as long as M is greater than the largest objective cost in the training set.

5.4.4 Training Time

We analyze how the size of the reward matrix affects the training time of OPT(k), and compare it with OCT(k). To vary the size of the reward matrix, we generate multiple training sets using different values of r, the radius of the ball that we sample instances from. As r gets larger, the number of distinct strategies in the training set will also likely get larger, leading to a larger reward matrix. For each training set, we compare the training times in OPT(k) and OCT(k). The problem we use for this experiment is the facility location problem with fixed size $n = 40, m = 20$. Instance generation details are identical to the description in Section 5.4.2 except for the choice of r.

The results of our experiments are presented in Table 5.10. One observation is that for

Table 5.9: The effect of the penalty M on the training time of OPT(k), measured in seconds.

T	M	$avg(s)$	$std(s)$	T	M	$avg(s)$	$std(s)$
10	obj_{max}	10	4	20	obj_{max}	33	6
	$obj_{max} \times 2$	13	2		$obj_{max} \times 2$	35	5
	$obj_{max} \times 5$	14	2		$obj_{max} \times 5$	33	5
	$obj_{max} \times 10$	13	2		$obj_{max} \times 10$	30	6
	$obj_{max} \times 100$	13	1		$obj_{max} \times 100$	26	4
	$obj_{max} \times 1000$	12	1		$obj_{max} \times 1000$	26	4

T	M	$avg(s)$	$std(s)$
30	obj_{max}	194	34
	$obj_{max} \times 2$	226	53
	$obj_{max} \times 5$	217	48
	$obj_{max} \times 10$	203	39
	$obj_{max} \times 100$	200	41
	$obj_{max} \times 1000$	196	40

small reward matrices, OPT(k) outperforms OCT(k) in terms of training time. However, as the size of the reward matrix increases, the training time of OPT(k) also increases and eventually becomes slower than OCT(k). Another observation is that the training time of OCT(k) is not always directly proportional to $|\mathcal{S}|$, while for OPT(k) it is roughly proportional. As a result, training in OCT(k) can be much slower when applied to training sets with a small $|\mathcal{S}|$.

Table 5.10: Comparison of the training times of OPT(k) and OCT(k), measured in seconds.

Learner	r	$\lvert\mathcal{S}\rvert$	Training time(s)	Learner	r	$\lvert\mathcal{S}\rvert$	Training time(s)
OPT(k)	0.1	3	4	OPT(k)	1	23	161
OCT(k)			120	OCT(k)			242
OPT(k)	0.25	5	3	OPT(k)	1.5	26	293
OCT(k)			154	OCT(k)			193
OPT(k)	0.5	8	7	OPT(k)	2	37	361
OCT(k)			37	OCT(k)			318
OPT(k)	0.75	11	21	OPT(k)	3	127	1704
OCT(k)			139	OCT(k)			460

5.4.5 On-line Solve Time

In the following experiments, we compare the average on-line solve time of OPT(k) with that of Gurobi for solving the facility location and the hybrid vehicle control of varying sizes. Specifically, we measure the time it takes to solve a MICO instance using the decision tree trained with OPT(k), and compare it with the time it takes to solve the same instance from scratch using Gurobi. When we apply a strategy to an instance to measure the solve time using OPT(k), we fix the integer variables to the values specified in the strategy, and impose

(a) Facility Location.

(b) Hybrid Vehicle Control.

Figure 5.4: Comparison of the on-line solve times of OPT(k) and Gurobi, measured in seconds.

only the tight constraints. Then, we solve the resulting continuous optimization problem using Gurobi. This is a general method to solve MICO problems using strategies, proposed in [24]. As mentioned in Section 5.2.3, for MIQO problems it is possible to solve a linear system defined by the KKT conditions instead of using a solver. In our work, we do not use this method but use the general method in order to evaluate the speed-up of OPT(k) without exploiting the structural properties of MIQO. For more details on how exploiting the structural properties of MIQO leads to even faster solve times, see [25].

We provide the experiment results in Figure 5.4. We observe that even without exploiting the structural properties of MIQO, the solve time using OPT(k) can be hundreds of times faster than using Gurobi from scratch. Additionally, as the problem size increases, the speed-up achieved by our approach becomes even more significant.

5.5 Computational Experiments on Real-World Data

In this section, we present the results of the computational experiments on real-world MILO problems taken from MIPLIB [53]. In Section 5.5.1, we compare OPT(1) and OCT(1). In Section 5.5.2, we make the prediction task more challenging by increasing r, the radius of the ball that we generate instances from. Then, we apply $k \geq 1$ strategies to compare OPT(k), OPT(k, Q), and OCT(k) as we increase k.

The models of the problems from MIPLIB are not available in exact form and instead, they are given in generalized matrix formats. That is, MIPLIB provide the data $(\boldsymbol{A}_{eq}, \boldsymbol{b}_{eq}, \boldsymbol{A}_{ineq}, \boldsymbol{b}_{ineq}, \boldsymbol{c}, lb, ub, \mathcal{I})$, so that the objective is to minimize $\boldsymbol{c}^T\boldsymbol{x}$ over the feasible set $\{\boldsymbol{x} : \boldsymbol{A}_{ineq}\boldsymbol{x} \leq \boldsymbol{b}_{ineq}, \boldsymbol{A}_{eq}\boldsymbol{x} = \boldsymbol{b}_{eq}, lb \leq \boldsymbol{x} \leq ub\}$ and $\mathcal{I}$ is the set of indices for the decision variables constrained to take integer values. This poses a challenge in selecting meaningful key parameters for instance generation. Thus, the choice of key parameters for MIPLIB examples is slightly arbitrary and is based upon a few general rules. First, $\bar{\boldsymbol{\theta}}$, the center of the ball is from the original data. Second, $\bar{\boldsymbol{\theta}}$ should not contain zero in its entries. Finally, the entries of $\bar{\boldsymbol{\theta}}$ all have the same sign. Based upon these rules, we choose a part of $\boldsymbol{b}_{eq}$, $\boldsymbol{b}_{ineq}$, or $\boldsymbol{c}$. We denote the part of a vector $\boldsymbol{b}$ from the i_{th} to the j_{th} entry as $\boldsymbol{b}_{i:j}$.

As MIPLIB provides data in a generalized matrix, we simply report the size of the

Table 5.11: MIPLIB problems.

Learner	Problem	N_{con}	$\lvert\mathcal{I}\rvert$	$\lvert\mathcal{C}\rvert$	Accurate	Suboptimal	Feasible	Infeasible	sub_{max}	$\lvert\mathcal{S}\rvert$
OPT(1)	ns1830653	3274	1458	171	2955	44	2999	1	0.0428	15
OCT(1)					2977	16	2993	7	0.0426	
OPT(1)	mas76	14	150	1	3000	0	3000	0	0	10
OCT(1)					2991	0	2991	9	0	
OPT(1)	binkar10_1	3154	170	2128	2992	8	3000	0	0.0013	14
OCT(1)					2997	0	2997	3	0.0002	
OPT(1)	markshare_4_0	4	30	4	2908	92	3000	0	0.2300	5
OCT(1)					2932	68	3000	0	6.1100	
OPT(1)	beasleyC3	3000	1250	1250	3000	0	3000	0	0	44
OCT(1)					3000	0	3000	0	0.0001	
OPT(1)	neos-827175	25341	21350	11154	3000	0	3000	0	0	15
OCT(1)					3000	0	3000	0	0	

constraint matrix and the composition of variables. We use $|\mathcal{C}|$, $|\mathcal{I}|$, N_{con} to denote the number of continuous variables, the number of integer variables and the number of constraints, respectively. The number of constraints is the sum of the numbers of equality constraints, inequality constraints, upper and lower bound constraints. The rest of the notations are identical to Section 5.4.

5.5.1 Comparison of OPT(1) and OCT(1)

Table 5.11 presents the comparison between OPT(1) and OCT(1) on selected MIPLIB problems. For the problem ns1830653, $\boldsymbol{b}_{ineq577:676}$ is used as $\bar{\boldsymbol{\theta}}$, and the parameters are drawn from $\mathcal{B}(\bar{\boldsymbol{\theta}}, 0.01)$. For mas76, $\boldsymbol{b}_{ineq1:12}$ is used as $\bar{\boldsymbol{\theta}}$ and the parameters are drawn from $\mathcal{B}(\bar{\boldsymbol{\theta}}, 1)$. For binkar10_1, the entire $\boldsymbol{b}_{ineq}$ is used as $\bar{\boldsymbol{\theta}}$ and the parameters are drawn from $\mathcal{B}(\bar{\boldsymbol{\theta}}, 0.5)$. For markshare_4_0, $\boldsymbol{c}_{1:4}$ is used as $\bar{\boldsymbol{\theta}}$ and the parameters are drawn from $\mathcal{B}(\bar{\boldsymbol{\theta}}, 1)$. For beasleyC3, $\boldsymbol{c}_{1251:1300}$ is used as $\bar{\boldsymbol{\theta}}$ and the parameters are drawn from $\mathcal{B}(\bar{\boldsymbol{\theta}}, 0.5)$. For neos-827175, $\boldsymbol{b}_{ineq1:15}$ is used as $\bar{\boldsymbol{\theta}}$ and the parameters are drawn from $\mathcal{B}(\bar{\boldsymbol{\theta}}, 0.25)$. We assign $M = 1000000$ to an infeasible solution for all problems.

We draw the following conclusions:

- OCT(1) outputs slightly more infeasible solutions compared to OPT(1).
- The maximum suboptimalities of OCT(1) and OPT(1) are generally similar in four problems. In binkar10_1, OCT(1) has an edge and in markshare_4_0, OPT(1) has a significant edge.
- On the number of suboptimal solutions, OCT(1) generally has a slight edge over OPT(1).
- Consistent with our conclusions on synthetic data, OPT has an edge on finding feasible solutions while OCT has a slight edge on the number of suboptimal solutions.

5.5.2 Comparison of OPT(k), OPT(k, Q) and OCT(k)

In this section, we compare OPT(k), OPT(k, Q) and OCT(k) under varying k and r. We test on two MIPLIB problems, binkar10_1 and mas76. Sizes of these problems are included in Table 5.11. We fix Q to $\lfloor k/2 \rfloor$.

Table 5.12 and Table 5.13 contain the experiment results on binkar10_1 and mas76, respectively. For binkar10_1, the entire right-hand-side vector $\boldsymbol{b}$ is used as $\bar{\boldsymbol{\theta}}$ and the parameters are drawn from $\mathcal{B}(\bar{\boldsymbol{\theta}}, 1)$, $\mathcal{B}(\bar{\boldsymbol{\theta}}, 3)$, $\mathcal{B}(\bar{\boldsymbol{\theta}}, 4)$, respectively. For mas76, $\boldsymbol{b}_{2:\text{end}}$ is used as $\bar{\boldsymbol{\theta}}$ and the parameters are drawn from $\mathcal{B}(\bar{\boldsymbol{\theta}}, 2)$, $\mathcal{B}(\bar{\boldsymbol{\theta}}, 4)$, $\mathcal{B}(\bar{\boldsymbol{\theta}}, 15)$, respectively. We assign $M = 1000000$ to an infeasible solution for both problems.

We draw the following conclusions:

- OPT-based algorithms have a significant edge on finding feasible solutions compared with OCT(k). This edge widens as $|\mathcal{S}|$ increases. As k increases, this edge decreases.
- OCT(k) has an edge on the number of suboptimal solutions compared to OPT(k). However, OPT($k, \lfloor k/2 \rfloor$) decreases this gap.
- OCT(k) and OPT-based algorithms are similar in terms of the maximum suboptimality.

5.6 Conclusion

We introduced a prescriptive machine learning approach to accelerate the solve process of MICO problems using OPT(k). Unlike classification algorithms, OPT(k) considers the quality of all available strategies for each instance to learn a policy. Additionally, we can compute the counterfactuals directly, which distinguishes our task from the usual prescriptive tasks. This allows OPT(k) to distinguish between strategies with varying degrees of suboptimality and infeasibility, resulting in improved solutions. We demonstrated that OPT(k) outperforms OCT(k) especially in the sense that it is less likely to output infeasible solutions. We believe this characteristic makes OPT(k) a safer algorithm to be deployed to real-world applications. We also introduced OPT(k, Q), a generalization of OPT(k), which improves the quality of the feasible solutions the algorithm outputs.

For future research directions, a theoretical explanation on why prescriptive methods could be better than predictive methods at avoiding infeasible (or highly undesirable) outputs will be insightful. Furthermore, in our approach, we assume that the problem size as well as the number of parameters are fixed throughout the training and deployment phases. It would be a fruitful research direction to extend our approach to the more general case, where we have to solve problems with varying sizes in the deployment phase. Finally, comparison of prescriptive and predictive methods in other learning-augmented optimization approaches will be interesting.

Table 5.12: binkar10_1.

Learner	k	Accurate	Suboptimal	Feasible	Infeasible	sub_{max}	r	$\|\mathcal{S}\|$
OPT(k)		2271	729	3000	0	0.0020		
OCT(k)	1	2869	104	2973	27	0.0010	1	50
OPT(k,$\lfloor k/2 \rfloor$)		2271	729	3000	0	0.0020		
OPT(k)		2665	335	3000	0	0.0020		
OCT(k)	5	2896	101	2997	3	0.0010	1	50
OPT(k,$\lfloor k/2 \rfloor$)		2665	335	3000	0	0.0020		
OPT(k)		2723	277	3000	0	0.0020		
OCT(k)	10	2898	101	2999	1	0.0010	1	50
OPT(k,$\lfloor k/2 \rfloor$)		2893	107	3000	0	0.0010		
OPT(k)		2890	110	3000	0	0.0010		
OCT(k)	30	2899	100	2999	1	0.0003	1	50
OPT(k,$\lfloor k/2 \rfloor$)		2899	101	3000	0	0.0010		
OPT(k)		1089	1901	2990	10	0.0090		
OCT(k)	1	2535	28	2563	436	0.0080	3	423
OPT(k,$\lfloor k/2 \rfloor$)		1089	1901	2990	10	0.0090		
OPT(k)		1140	1855	2995	5	0.0090		
OCT(k)	5	2899	20	2919	81	0.0080	3	423
OPT(k,$\lfloor k/2 \rfloor$)		1788	1207	2995	5	0.0080		
OPT(k)		1215	1780	2995	5	0.0090		
OCT(k)	10	2940	22	2962	38	0.0080	3	423
OPT(k,$\lfloor k/2 \rfloor$)		2578	417	2995	5	0.0080		
OPT(k)		2479	520	2999	1	0.0080		
OCT(k)	30	2975	9	2984	16	0.0080	3	423
OPT(k,$\lfloor k/2 \rfloor$)		2921	77	2998	2	0.0080		
OPT(k)		2605	395	3000	0	0.0080		
OCT(k)	60	2984	9	2993	7	0.0080	3	423
OPT(k,$\lfloor k/2 \rfloor$)		2954	46	3000	0	0.0080		
OPT(k)		276	2618	2894	106	0.0100		
OCT(k)	1	2179	169	2348	652	0.0080	4	1023
OPT(k,$\lfloor k/2 \rfloor$)		276	2618	2894	106	0.0100		
OPT(k)		1006	1985	2991	9	0.0090		
OCT(k)	5	2603	133	2736	264	0.0080	4	1023
OPT(k,$\lfloor k/2 \rfloor$)		1101	1844	2945	55	0.0100		
OPT(k)		1011	1983	2994	6	0.0090		
OCT(k)	10	2746	98	2844	156	0.0080	4	1023
OPT(k,$\lfloor k/2 \rfloor$)		2233	760	2993	7	0.0080		
OPT(k)		2070	926	2996	4	0.0080		
OCT(k)	30	2832	87	2919	81	0.0080	4	1023
OPT(k,$\lfloor k/2 \rfloor$)		2502	494	2996	4	0.0080		
OPT(k)		2232	765	2997	3	0.0080		
OCT(k)	60	2846	87	2933	67	0.0080	4	1023
OPT(k,$\lfloor k/2 \rfloor$)		2711	286	2997	3	0.0080		

Table 5.13: mas76.

Learner	k	Accurate	Suboptimal	Feasible	Infeasible	sub_{max}	r	$\|\mathcal{S}\|$
OPT(k)		2911	88	2999	1	0.19		
OCT(k)	1	2965	0	2965	35	0	2	22
OPT(k,$\lfloor k/2 \rfloor$)		2911	88	2999	1	0.19		
OPT(k)		2912	88	3000	0	0.19		
OCT(k)	5	3000	0	3000	0	0	2	22
OPT(k,$\lfloor k/2 \rfloor$)		2912	88	3000	0	0.19		
OPT(k)		2912	88	3000	0	0.19		
OCT(k)	10	3000	0	3000	0	0	2	22
OPT(k,$\lfloor k/2 \rfloor$)		2991	9	3000	0	0.19		
OPT(k)		2893	106	2999	1	0.55		
OCT(k)	1	2924	6	2930	70	0.27	4	36
OPT(k,$\lfloor k/2 \rfloor$)		2893	106	2999	1	0.55		
OPT(k)		2896	104	3000	0	0.55		
OCT(k)	5	2993	7	3000	0	0.37	4	36
OPT(k,$\lfloor k/2 \rfloor$)		2896	104	3000	0	0.55		
OPT(k)		2896	104	3000	0	0.55		
OCT(k)	10	2996	4	3000	0	0.27	4	36
OPT(k,$\lfloor k/2 \rfloor$)		2986	14	3000	0	0.55		
OPT(k)		2969	31	3000	0	0.55		
OCT(k)	30	3000	0	3000	0	0	4	36
OPT(k,$\lfloor k/2 \rfloor$)		2996	4	3000	0	0.27		
OPT(k)		2383	613	2996	4	0.38		
OCT(k)	1	2904	14	2918	82	0.27	15	71
OPT(k,$\lfloor k/2 \rfloor$)		2383	613	2996	4	0.38		
OPT(k)		2435	563	2998	2	0.38		
OCT(k)	5	2994	4	2998	2	0.27	15	71
OPT(k,$\lfloor k/2 \rfloor$)		2395	602	2997	3	0.38		
OPT(k)		2437	563	3000	0	0.38		
OCT(k)	10	2994	4	2998	2	0.27	15	71
OPT(k,$\lfloor k/2 \rfloor$)		2437	563	3000	0	0.38		
OPT(k)		2437	563	3000	0	0.38		
OCT(k)	30	2998	2	3000	0	0.002	15	71
OPT(k,$\lfloor k/2 \rfloor$)		3000	0	3000	0	0		
OPT(k)		3000	0	3000	0	0		
OCT(k)	60	3000	0	3000	0	0	15	71
OPT(k,$\lfloor k/2 \rfloor$)		3000	0	3000	0	0		

Chapter 6

Conclusions

In this thesis, we have developed and demonstrated methods to expedite the solution of a wide array of optimization and control problems using decision tree algorithms. Each part of this thesis has provided novel insights and methodologies that speeds up the process of solving complex problems.

The first part of this thesis introduced a machine learning approach to the optimal control of MFQNETs. We proved that OCT-H can learn an optimal policy for MFQNET control problems. Computational experiments demonstrated the effectiveness of the proposed method in learning the optimal policy.

In the second part, we focused on the control of FRMAB problems. We derived fundamental properties of FRMAB problems and designed an efficient numerical algorithm. By introducing a feature augmentation technique and applying OCT-H to the augmented features, we incorporated nonlinearities into the model. Computational experiments demonstrated that the proposed method effectively learns high-quality feedback policies.

The third part studied two-stage linear adaptive robust optimization problems with binary here-and-now variables and polyhedral uncertainty sets. We developed a method to encode the optimal here-and-now decisions, the associated worst-case scenarios, and the optimal wait-and-see decisions into optimal strategies. We trained machine learning models to predict high-quality strategies for unseen instances. Additionally, we introduced novel methods to expedite training data generation and reduce the number of different target classes required for training. Using the proposed method, large-scale ARO problems can be solved more than 10 millions times faster compared to the state-of-the-art numerical algorithm.

Finally, the fourth part introduced a prescriptive machine learning approach for MICO problems. We utilized OPT as a prescriptive algorithm and demonstrated that OPT-based methods outperform classification algorithms like OCT in finding feasible solutions for various MICO problems.

Looking ahead, the integration of learning components into real-world decision systems presents ongoing challenges, particularly in ensuring robustness. Machine learning models are typically designed to provide predictions that are accurate on average. However, a critical question arises when confronted with predictions of extremely low quality. In such scenarios, we risk encountering slow convergence or potentially even infeasible solutions, although the outcome may vary based on the problem class or the specific prediction target. Addressing these challenges requires further advancements in methodologies that ensure both speed and

feasibility, even under low-quality predictions.

Appendix A

Appendix to Chapter 3

We prove that for the problems considered in Section 3.5, the state constraints $0 < x_i(t) < H_i, \forall i \in [n]$, are automatically satisfied. Similar to the logic in the proof of Proposition 4, we prove that in each interval $[t_s, t_{s+1})$ with a constant control $\boldsymbol{u}_s$, if $x_i(t_s) \in (0, H_i)$, then $x_i(t) \in (0, H_i), \forall t \in [t_s, t_{s+1})$. This implies that if $\boldsymbol{x}(0) \in (0, H_i)$, then $\boldsymbol{x}(t) \in (0, H_i), \forall t \in [0, T]$.

Proposition 6. *For the machine maintenance problem described in Section 3.5, the state trajectory stays within $(0,1)^n$ regardless of the control trajectory.*

Proof. We directly apply the results in (3.7). For the machine maintenance problem, $\beta_i(u_{s,i}) = -\alpha_i(u_{s,i})$. If $\beta_i(u_{s,i}) = 0$, then $x_i(t) = x_i(t_s) \in (0,1), \forall t \in [t_s, t_{s+1})$. If $\beta_i(u_{s,i}) \neq 0$, then

$$\begin{aligned} x_i(t) &= x_i(t_s) + (1 - x_i(t_s))(1 - e^{-h_i(t-t_s)}) \\ &= 1 - e^{-h_i(t-t_s)}(1 - x(t_s)). \end{aligned}$$

The first equality proves that $x_i(t) > 0$, and the second equality proves that $x_i(t) < 1, \forall t \in [t_s, t_{s+1})$. □

Proposition 7. *For the epidemic control problem described in Section 3.5, the state trajectory stays within $(0,1)^n$ regardless of the control trajectory.*

Proof. We directly apply the results in (3.9) that

$$x_i(t) = \frac{K\alpha_i(u_{s,i})e^{\alpha_i(u_{s,i})t}}{1 - K\beta_i(u_{s,i})e^{\alpha_i(u_{s,i})t}}.$$

From the above expression, $x_i(t)$ is a monotone function of t in this interval. If $\alpha_i(u_{s,i}) > 0$, then

$$\begin{aligned} \lim_{t\to\infty} x_i(t) &= -\frac{\alpha_i(u_{s,i})}{\beta_i(u_{s,i})} \\ &= 1 - \frac{u_{s,i}(\mu_0 - \mu_1) - \mu_0}{u_{s,i}(\lambda_0 - \lambda_1) - \lambda_0} \end{aligned}$$

Due to the assumption that $\alpha_i(u_{s,i}) > 0$,

$$u_{s,i}(\mu_0 - \mu_1) - \mu_0 > u_{s,i}(\lambda_0 - \lambda_1) - \lambda_0,$$

where $u_{s,i}(\mu_0 - \mu_1) - \mu_0 = \mu_0(u_{s,i} - 1) - u_{s,i}\mu_1 < 0$. Hence, $-\frac{\alpha_i(u_{s,i})}{\beta_i(u_{s,i})} \in (0, 1)$.

If $\alpha_i(u_{s,i}) < 0$, $\lim_{t\to\infty} x_i(t) = 0$. This guarantees that $x_i(t) \in (0, 1)$ in this interval. □

Proposition 8. *For the fisheries control problem described in Section 3.5, the state trajectory stays within* $(0, H_i)^n$ *regardless of the control trajectory.*

Proof. The proof is almost identical to the case of epidemic control. Again, we directly apply the results in (3.9) that

$$x_i(t) = \frac{K\alpha_i(u_{s,i})e^{\alpha_i(u_{s,i})t}}{1 - K\beta_i(u_{s,i})e^{\alpha_i(u_{s,i})t}}.$$

From the above expression, $x_i(t)$ is a monotone function of t in this interval. If $\alpha_i(u_{s,i}) > 0$, then

$$\begin{aligned}\lim_{t\to\infty} x_i(t) &= -\frac{\alpha_i(u_{s,i})}{\beta_i(u_{s,i})} \\ &= \frac{H_i(r_i - q_i u_{s,i})}{r_i}\end{aligned}$$

Due to the assumption that $\alpha_i(u_{s,i}) = r_i - q_i u_{s,i} > 0$, $\frac{H_i(r_i - q_i u_{s,i})}{r_i} \in (0, H_i)$.

If $\alpha_i(u_{s,i}) < 0$, $\lim_{t\to\infty} x_i(t) = 0$. This guarantees that $x_i(t) \in (0, H_i)$ in this interval. □

Appendix B

Appendix to Chapter 4

A1 Analysis on Algorithm 5 and 6

In this section, we analyze the impact of varying tolerance parameters within the CCG algorithm and the choice of different initial points utilized in its subroutine.

A1.1 Analysis on the Tolerance Parameters

We analyze the impact of different tolerance parameters ϵ_1 and ϵ_2 on the runtime of Algorithm 5. A smaller value for ϵ_1 results in higher-quality here-and-now decisions, while a smaller ϵ_2 results in more precise worst-case scenarios for a given here-and-now decision. By assessing various combinations of these parameters, we analyze the trade-off between solution quality and computational cost.

For this experiment, we solve 100 random instances of the unit commitment problem with $n = 100, m = 24$. In Table A1, we report the mean runtime of Algorithm 5 for each choice of ϵ_1 and ϵ_2. Naturally, smaller ϵ_1 and ϵ_2 leads to longer runtime.

ϵ_1	ϵ_2	Runtime (in seconds)
0.050		315.14
0.025	0.01	537.08
0.010		599.38
0.050		495.16
0.025	0.001	603.77
0.010		787.58

Table A1: Runtime of Algorithm 5 with different tolerance parameters.

$\boldsymbol{x}_i$	$\tilde{\mathcal{Q}}(\boldsymbol{x}_i)$	$\boldsymbol{d}_i$	Runtime
Near-optimal	0.002	1	1.000
Suboptimal	0.001	1.4	0.980
Ones	0.000	1	1.052

Table A2: Variability of Algorithm 6 under different initial points.

A1.2 Initial Point in Algorithm 6

We investigate the performance variability of Algorithm 6 under different initial points $\boldsymbol{d}_0$. We use three distinct here-and-now decisions as inputs for Algorithm 6: near-optimal here-and-now decision, suboptimal here-and-now decision, and the matrix of all ones. Initially, we solve 100 random instances of the unit commitment problem with $n = 100, m = 24$. From each instance, we extract one suboptimal and one near-optimal here-and-now decision. Then, for each instance and each here-and-now decision, we randomly generate 10 different initial points and evaluate the runtime and the outputs of Algorithm 6 with $\epsilon_2 = 0.01$. We measure the difference between the maximum runtime and the minimum runtime, divided by the mean runtime for scaling. Similarly, we measure the difference between the maximum and the minimum of the objective values ($\tilde{\mathcal{Q}}(\boldsymbol{x}_i)$) divided by the mean. Additionally, we record the number of unique $\boldsymbol{d}_i$ obtained using the ten different initial points. The means of these values across 100 instances are reported for analysis.

Table A2 presents the results of this experiment. In first column, we report the type of here-and-now decision used. In the second and the third column, we report the variability of the objective value and the number of unique scenarios, respectively. In the fourth column, we report the variability of runtime. Our observations suggest that Algorithm 6 demonstrates relatively low variability, as various initial points tend to produce consistent results overall. The only source of variability appears to be in runtime, indicating that depending on the initial points, it may require more iterations to converge. Nevertheless, the termination point remains similar across different initializations.

A2 Additional Computational Experiments

A2.1 Testing Under Distributional Shift

In this section, we assess the robustness of our approach by evaluating its performance when the test set is generated from a distribution different from the training set. This experiment is conducted using the inventory control problem with $n = 25$.

Similar to the previous experiments, we sample $\boldsymbol{c}^2$ uniformly from $B(\bar{\boldsymbol{c}}^2, r_1)$, where $\bar{c}_i^2$ is sampled from $U(40, 60)$ and remains fixed throughout the sample generation process. Likewise, we sample $\boldsymbol{c}^3$ uniformly from $B(\bar{\boldsymbol{c}}^3, r_2)$, where $\bar{c}_i^3$ is sampled from $U(60, 80)$ and fixed. We first generate a training set with $r_1 = r_2 = 5$ and then test the model on test sets generated with varying r_1 and r_2. Each test set consists of 3000 instances. The analysis focuses on understanding how the approach's performance degrades as the testing distribution'

radius increases. Results are provided in Table A3, with r_1 and r_2 in the table indicating the radii used to generate the test set. For this training set, the number of distinct strategies for the here-and-now decisions, the worst-case scenarios, and the wait-and-see decisions are 8, 6, and 28, respectively.

Target	k	r_1	r_2	Accuracy	Infeasibility	sub_{max}
$s_{\boldsymbol{x}}$	1	7.5	7.5	0.93	0	0.0012
	5			0.98	0	0.0009
$s_{\boldsymbol{d}}$	1	7.5	7.5	0.72	0	0.0010
	5			0.94	0	0.0010
	10			0.95	0	0.0010
$s_{\boldsymbol{y}}$	1	7.5	7.5	1.00	0	0.0012
	5			1.00	0	0.0009
	10			1.00	0	0.0009
$s_{\boldsymbol{x}}$	1	10	10	0.76	0	0.0025
	5			0.88	0	0.0015
$s_{\boldsymbol{d}}$	1	10	10	0.54	0	0.0021
	5			0.78	0	0.0021
	10			0.80	0	0.0021
$s_{\boldsymbol{y}}$	1	10	10	1.00	0	0.0008
	5			1.00	0	0.0008
	10			1.00	0	0.0008
$s_{\boldsymbol{x}}$	1	15	15	0.38	0	0.0053
	5			0.51	0	0.0047
$s_{\boldsymbol{d}}$	1	15	15	0.21	0	0.0051
	5			0.38	0	0.0051
	10			0.39	0	0.0051
$s_{\boldsymbol{y}}$	1	15	15	1.00	0	0.0008
	5			1.00	0	0.0008
	10			1.00	0	0.0008

Table A3: Numerical results under distributional shift.

Results

- Increasing the radius of the testing distribution results in a deterioration of overall performance.
- Even under distributional shift, the predictions are consistently feasible.

- When the testing distribution has double the radius of the training distribution, maximum suboptimalities still remain below 0.0053, despite a significant decrease in accuracy.

A2.2 Effect of Training Data Size

In this section, we explore the impact of varying the size of the training set. We use a fixed test set consisting of 3000 instances. These experiments are conducted on the inventory control problem with $n = 25$. Both the training and the test sets are generated with $r_1 = r_2 = 5$. Results are presented in Table A4.

Results

- Despite the significantly smaller training size compared to previous experiments, prediction accuracies consistently reach 1.00 or 0.99, with zero infeasibility.
- Increasing the training size leads to improved overall performance, characterized by slightly lower accuracy and maximum suboptimality.

A2.3 Offline Computation Time

In this section, we analyze the computational burden associated with the offline phase (Phase 1 and 2) of our approach. We select two problems from previous experiments and report the time required for generating a training set, computing a reward matrix and training a decision tree using the reward matrix. For this experiment, we do not use Algorithm 9 and 10.

It is worth noting that training set generation and reward matrix computation can be performed in parallel, which may result in varying total computation times depending on available computational resources. Therefore, rather than providing the total time required, we report the time needed to generate a single training instance and a single entry in the reward matrix. In addition, note that the solve times for strategies $s_{\boldsymbol{x}}$ and $s_{\boldsymbol{d}}$ are identical since they can be computed simultaneously using Algorithm 5. However, calculating $s_{\boldsymbol{y}}$ involves solving an additional linear programming problem.

We have chosen two problems for this analysis: the inventory control problem with $n = 1000, \Gamma = 45$, and the unit commitment problem with $n = 100, m = 24, \Gamma = 2$. These problems were selected for their distinct characteristics. The inventory control problem exhibits the largest dimensionality in the feature vector $\boldsymbol{\theta}$, while the unit commitment problem has the longest computation time to solve a single ARO instance. For the unit commitment problem, we use the lenient tolerance as in Section 4.6.6.

Results are presented in Tables A5 and A6. For the inventory control problem, data generation and reward matrix computation are fast, while the training process can take more than two days. Conversely, for the unit commitment problem, data generation and reward matrix computation may take days, but the training phase is relatively fast. This observation underscores the fact that depending on the problem size and the choice of key parameters, data generation and training times can range from hours to days in practical applications.

Target	k	Training Size	Accuracy	Infeasibility	sub_{max}
$s_{\boldsymbol{x}}$	1	3000	1.00	0	0.0005
	5		1.00	0	0.0005
$s_{\boldsymbol{d}}$	1	3000	0.99	0	0.0005
	5		0.99	0	0.0005
	10		1.00	0	0.0005
$s_{\boldsymbol{y}}$	1	3000	1.00	0	0.0005
	5		1.00	0	0.0002
	10		1.00	0	0.0002
$s_{\boldsymbol{x}}$	1	5000	1.00	0	0.0005
	5		1.00	0	0
$s_{\boldsymbol{d}}$	1	5000	0.99	0	0.0005
	5		1.00	0	0.0005
	10		1.00	0	0.0005
$s_{\boldsymbol{y}}$	1	5000	1.00	0	0.0005
	5		1.00	0	0.0002
	10		1.00	0	0.0002
$s_{\boldsymbol{x}}$	1	10000	1.00	0	0.0005
	5		1.00	0	0
$s_{\boldsymbol{d}}$	1	10000	0.99	0	0.0005
	5		1.00	0	0.0005
	10		1.00	0	0.0005
$s_{\boldsymbol{y}}$	1	10000	1.00	0	0.0005
	5		1.00	0	0
	10		1.00	0	0
$s_{\boldsymbol{x}}$	1	15000	0.99	0	0.0005
	5		1.00	0	0
$s_{\boldsymbol{d}}$	1	15000	1.00	0	0.0005
	5		1.00	0	0.0005
	10		1.00	0	0.0005
$s_{\boldsymbol{y}}$	1	15000	1.00	0	0.0005
	5		1.00	0	0
	10		1.00	0	0

Table A4: Numerical results under varying size of training set.

Target	Task	Time (hr:min:s)
$s_{\boldsymbol{x}}$	Solve	00:00:0.15
$s_{\boldsymbol{d}}$		00:00:0.15
$s_{\boldsymbol{y}}$		00:00:0.20
$s_{\boldsymbol{x}}$	Reward Matrix	00:00:0.04
$s_{\boldsymbol{d}}$		00:00:0.08
$s_{\boldsymbol{y}}$		00:00:0.05
$s_{\boldsymbol{x}}$	Training	04:35:23
$s_{\boldsymbol{d}}$		48:52:25
$s_{\boldsymbol{y}}$		00:34:32

Table A5: Inventory control problem.

Target	Task	Time (hr:min:s)
$s_{\boldsymbol{x}}$	Solve	00:05:15
$s_{\boldsymbol{d}}$		00:05:15
$s_{\boldsymbol{y}}$		00:05:45
$s_{\boldsymbol{x}}$	Reward Matrix	00:00:48
$s_{\boldsymbol{d}}$		00:01:12
$s_{\boldsymbol{y}}$		00:01:02
$s_{\boldsymbol{x}}$	Training	00:00:21
$s_{\boldsymbol{d}}$		00:00:10
$s_{\boldsymbol{y}}$		00:00:15

Table A6: Unit commitment problem.

A2.4 Effect of the Size of the Uncertainty Sets

In this section, we analyze the performance of our approach under different sizes of uncertainty sets. While previous experiments have already explored uncertainty sets of varying sizes, we fix a single problem and vary the size of the uncertainty sets more extensively in this experiment. The problem we choose is the inventory control problem with $n = 25$. Throughout this section, we use Algorithm 11 with $K = 1$ to reduce the number of strategies for the wait-and-see variables and the number of training data is fixed to 15000. The experiment settings remain identical to Section A2.2.

We report the results in Table A7. We consistently observe very high-quality solutions across all variations. The primary distinction arises in the number of distinct strategies for the wait-and-see variables (denoted by the column $|\boldsymbol{\tau}|$). As expected, larger uncertainty sets lead to higher number of strategies. However, leveraging Algorithm 11 mitigates any adverse effects on the quality of our approach, similar to the findings in Section 4.6.4. This result demonstrates the effectiveness of our approach regardless of the size of the uncertainty sets.

A3 Description of the Unit Commitment Problem

In this section, we describe the deterministic version of the unit commitment problem. The original data from [37] is for 10 generators and 24 periods ($n = 10$ and $m = 24$). For larger problems we adequately extended the original data, following [37].

Constants

- A_j : Coefficient of the piecewise linear production cost function of unit j
- a_j, b_j, c_j : Coefficient of the quadratic production cost function of unit j
- cc_j, hc_j, t_j^{cold} : Coefficients of the startup cost function of unit
- C_j : Shutdown cost of unit j.
- $D(k)$: Load demand in period k
- DT_j : Minimum downtime of unit j
- F_{lj} : Slope of block l of the piecewise linear production cost function of unit j

Target	k	Γ	Accuracy	Infeasibility	sub_{max}	$\lvert\mathcal{S}\rvert$	$\lvert\boldsymbol{\tau}\rvert$
$s_{\boldsymbol{x}}$	1	5	1.00	0	0.0003	8	*NA*
	5		1.00	0	0.0001		
$s_{\boldsymbol{d}}$	1	5	1.00	0	0.0003	27	*NA*
	5		1.00	0	0.0003		
$s_{\boldsymbol{y}}$	1	5	0.99	0	0.0003	8	26
	5		1.00	0	0.0001		
$s_{\boldsymbol{x}}$	1	10	1.00	0	0.0006	11	*NA*
	5		1.00	0	0.0000		
$s_{\boldsymbol{d}}$	1	10	0.99	0	0.0005	33	*NA*
	5		1.00	0	0.0003		
	10		1.00	0	0.0003		
$s_{\boldsymbol{y}}$	1	10	1.00	0	0.0003	11	104
	5		1.00	0	0.0000		
$s_{\boldsymbol{x}}$	1	15	1.00	0	0.0005	9	*NA*
	5		1.00	0	0.0000		
$s_{\boldsymbol{d}}$	1	15	0.99	0	0.0005	27	*NA*
	5		1.00	0	0.0005		
	10		1.00	0	0.0005		
$s_{\boldsymbol{y}}$	1	15	1.00	0	0.0005	9	124
	5		1.00	0	0.0000		
$s_{\boldsymbol{x}}$	1	20	0.99	0	0.0003	8	*NA*
	5		1.00	0	0.0002		
$s_{\boldsymbol{d}}$	1	20	0.99	0	0.0009	26	*NA*
	5		0.99	0	0.0003		
	10		1.00	0	0.0003		
$s_{\boldsymbol{y}}$	1	20	0.99	0	0.0002	8	162
	5		1.00	0	0.0205		
$s_{\boldsymbol{x}}$	1	25	1.00	0	0.0005	9	*NA*
	5		1.00	0	0.0000		
$s_{\boldsymbol{d}}$	1	25	0.99	0	0.0005	28	*NA*
	5		0.99	0	0.0005		
	10		1.00	0	0.0005		
$s_{\boldsymbol{y}}$	1	25	1.00	0	0.0005	9	211
	5		1.00	0	0.0000		

Table A7: Numerical results under varying size of uncertainty sets.

- G_j : Number of periods unit j must be initially online due to its minimum up time constraint.
- K_j^t : Cost of the interval t of the stairwise startup cost function of unit j.
- L_j : Number of periods unit j must be initially offline due to its minimum down time constraint
- LD_j : Number of intervals of the stairwise startup cost function of unit j
- NL_j : Number of segments of the piecewise linear production cost function of unit j
- $\bar{P}_j$: Capacity of unit j
- $\underline{P}_j$: Minimum power output of unit j
- $R(k)$: Spinning reserve requirement in period k
- RD_j : Ramp down limit of unit j
- RU_j : Ramp up limit of unit j
- $S_j(0)$: Number of periods unit j has been offline prior to the first period of the time span (end of period 0)
- SD_j : Shut down ramp limit of unit j
- SU_j : Start-up ramp limit of unit j
- m : Number of periods of the time span
- T_{lj} : Upper limit of block l of the piecewise linear production cost function of unit j.
- U_j^0 : Number of periods unit j has been online prior to the first period of the time span (end of period 0)
- UT_j : Minimum up time of unit j
- $V_j(0)$: Initial commitment state of unit j (1 if it is online, 0 otherwise)

Variables
- $c_j^d(k)$: Shutdown cost of unit j in period k
- $c_j^p(k)$: Production cost of unit j in period k
- $c_j^u(k)$: Startup cost of unit j in period k
- $p_j(k)$: Power output of unit j in period k
- $\bar{p}_j(k)$: Maximum available power output of unit j in period k
- $t_j^{off}(k)$: Number of periods unit j has been offline prior to the startup in period k.
- $v_j(k)$: Binary variable that is equal to 1 if unit j is online in period k and 0 otherwise.
- $\delta_{lj}(k)$: Power produced in block l of the piecewise linear production cost function of unit j in period k.

Sets
- J : Set of indices of the generating units.
- K : Set of indices of time periods.

Model All constraints are defined $\forall j \in J$ or $\forall k \in K$, unless otherwise noted.

$$\begin{aligned} \min \quad & \sum_{k \in K} \sum_{j \in J} c_j^p(k) + c_j^u(k) + c_j^d(k) \\ \text{s.t.} \quad & \sum_{j \in J} p_j(k) \geq D(k) \end{aligned} \tag{B.1}$$

$$\sum_{j\in J} \bar{p}_j(k) \geq D(k) + R(k) \tag{B.2}$$

$$c_j^p(k) = A_j v_j(k) + \sum_{l=1}^{NL_j} F_{lj}\delta_l(j,k) \tag{B.3}$$

$$p_j(k) = \sum_{l=1}^{NL_j} \delta_l(j,k) + \underline{\mathrm{P}}_j v_j(k) \tag{B.4}$$

$$\delta_1(j,k) \leq T_{1j} - \underline{\mathrm{P}}_j \tag{B.5}$$

$$\delta_l(j,k) \leq T_{lj} - T_{l-1j} \tag{B.6}$$

$$\delta_N L_j(j,k) \leq \bar{P}_j - T_{NL_j-1j} \tag{B.7}$$

$$\delta_l(j,k) \geq 0 \quad \forall l = 1 \ldots NL_j \tag{B.8}$$

$$A_j = a_j + b_j\underline{\mathrm{P}}_j + c_j\underline{\mathrm{P}}_j^2 \tag{B.9}$$

$$c_j^u(k) \geq K_j^t[v_j(k) - \sum_{n=1}^{t} v_j(k-n)] \quad \forall t = 1 \ldots ND_j \tag{B.10}$$

$$c_j^u(k) \geq 0 \tag{B.11}$$

$$c_j^d(k) \geq C_j[v_j(k-1) - v_j(k)] \tag{B.12}$$

$$c_j^d(k) \geq 0 \tag{B.13}$$

$$\underline{\mathrm{P}}_j v_j(k) \leq p_j(k) \leq \bar{p}_j(k) \tag{B.14}$$

$$0 \leq \bar{p}_j(k) \leq \bar{P}_j v_j(k) \tag{B.15}$$

$$\bar{p}_j(k) \leq p_j(k-1) + RU_j v_j(k-1) + SU_j[v_j(k) - v_j(k-1)] + \bar{P}_j[1 - v_j(k)] \tag{B.16}$$

$$\bar{p}_j(k) \leq \bar{P}_j v_j(k+1) + SD_j[v_j(k) - v_j(k+1)] \tag{B.17}$$

$$p_j(k-1) - p_j(k) \leq RD_j v_j(k) + SD_j[v_j(k-1) - v_j(k)] + \bar{P}_j[1 - v_j(k-1)] \tag{B.18}$$

$$\forall k = 1 \ldots m-1$$

$$\bar{p}_j(k) \leq \bar{P}_j v_j(k+1) + SD_j[v_j(k) - v_j(k+1)] \tag{B.19}$$

$$p_j(k-1) - p_j(k) \leq RD_j v_j(k) + SD_j[v_j(k-1) - v_j(k)] + \bar{P}_j[1 - v_j(k-1)] \tag{B.20}$$

$$\sum_{k=1}^{G_j}[1 - v_j(k)] = 0 \tag{B.21}$$

$$\sum_{n=k}^{k+UT_j-1} v_j(n) \geq UT_j[v_j(k) - v_j(k-1)] \quad \forall k = G_j + 1 \ldots m - UT_j + 1 \tag{B.22}$$

$$\sum_{n=k}^{T}\{v_j(n) - [v_j(k) - v_j(k-1)]\} \geq 0 \quad \forall k = m - UT_j + 2 \ldots m \tag{B.23}$$

$$\sum_{k=1}^{L_j} v_j(k) = 0 \tag{B.24}$$

$$\sum_{n=k}^{k+DT_j-1} [1 - v_j(n)] \geq DT_j[v_j(k-1) - v_j(k)] \quad \forall k = L_j + 1 \ldots m - DT_j + 1 \tag{B.25}$$

$$\sum_{n=k}^{m}\{1 - v_j(n) - [v_j(k-1) - v_j(k)]\} \geq 0 \quad \forall k = m - DT_j + 2 \ldots m \tag{B.26}$$

Constraint (B.1) and (B.2) represent the power balance constraint and the spinning reserve margins, respectively. Constraints (B.3) - (B.9) represent the stepwise approximation of production cost function, which is originally a quadratic function. (B.10)-(B.11) and (B.12) - (B.13) represent the stepwise startup cost and the stepwise shutdown cost, respectively (every time a generator is turned on or turned down, it incurs a cost). (B.14) - (B.15) represent the power generation limits. (B.16) - (B.20) are ramp-up, startup ramp limits, shutdown ramp limits and ramp-down limits, respectively. These constraints dictate that a generator can only change its production level within a certain bound. (B.21) - (B.23) are minimum up time constraints and (B.24) - (B.26) are minimum down time constraints. These constraints represent the physical limit that if a generator is turned on at some point, it must remain on for certain period. Likewise, if a generator is turned off, it must remain that way for certain period.

References

[1] A. Abbou and V. Makis, "Group maintenance: A restless bandits approach," *INFORMS J. Comput.*, vol. 31, pp. 719–731, 2019.

[2] A. M. Alvarez, Q. Louveaux, and L. Wehenkel, "A machine learning-based approximation of strong branching," *INFORMS Journal on Computing*, vol. 29, no. 1, pp. 185–195, 2017. DOI: 10.1287/ijoc.2016.0723. eprint: https://doi.org/10.1287/ijoc.2016.0723. URL: https://doi.org/10.1287/ijoc.2016.0723.

[3] M. Amram, J. Dunn, and Y. D. Zhuo, "Optimal policy trees," *Machine Learning*, Mar. 2022, ISSN: 1573-0565. DOI: 10.1007/s10994-022-06128-5. URL: https://doi.org/10.1007/s10994-022-06128-5.

[4] E. J. Anderson, P. Nash, and A. F. Perold, "Some properties of a class of continuous linear programs," *SIAM Journal on Control and Optimization*, vol. 21, no. 5, pp. 758–765, 1983.

[5] M. Ang, Y. Lim, and M. Sim, "Robust storage assignment in unit-load warehouses," *Management Science*, vol. 58, no. 11, pp. 2114–2130, 2012.

[6] F. Avram, "Optimal control of fluid limits of queueing networks and stochasticity corrections," *Lecture in Applied Mathematics-American Mathematical Society*, vol. 33, pp. 1–36, 1997.

[7] F. Avram, D. Bertsimas, and M. Ricard, "An optimal control approach to optimization of multiclas queueing network," *IMA volumes in Mathematics and its Applications*, vol. 71, pp. 199–234, 1995.

[8] N. Bacaër, "Verhulst and the logistic equation (1838)," in *A Short History of Mathematical Population Dynamics*. London: Springer London, 2011, pp. 35–39, ISBN: 978-0-85729-115-8. DOI: 10.1007/978-0-85729-115-8_6. URL: https://doi.org/10.1007/978-0-85729-115-8_6.

[9] M.-F. Balcan, D. DeBlasio, T. Dick, C. Kingsford, T. Sandholm, and E. Vitercik, "How much data is sufficient to learn high-performing algorithms? Generalization guarantees for data-driven algorithm design," in *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2021, Virtual, Italy: Association for Computing Machinery, 2021, pp. 919–932, ISBN: 9781450380539. DOI: 10.1145/3406325.3451036. URL: https://doi.org/10.1145/3406325.3451036.

[10] M.-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik, "Learning to branch," in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, Jul. 2018, pp. 344–353. URL: https://proceedings.mlr.press/v80/balcan18a.html.

[11] M.-F. Balcan, S. Prasad, T. Sandholm, and E. Vitercik, "Structural analysis of branch-and-cut and the learnability of gomory mixed integer cuts," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22, New Orleans, LA, USA: Curran Associates Inc., 2024, ISBN: 9781713871088.

[12] M.-F. Balcan, T. Sandholm, and E. Vitercik, "Learning to optimize computational resources: Frugal training with generalization guarantees," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, pp. 3227–3234, Apr. 2020.

[13] M.-F. Balcan, T. Sandholm, and E. Vitercik, "Refined bounds for algorithm configuration: The knife-edge of dual class approximability," in *Proceedings of the 37th International Conference on Machine Learning*, H. D. III and A. Singh, Eds., ser. Proceedings of Machine Learning Research, vol. 119, PMLR, Jul. 2020, pp. 580–590. URL: https://proceedings.mlr.press/v119/balcan20a.html.

[14] D. Bampou and D. Kuhn, "Polynomial approximations for continuous linear programs," *SIAM Journal on Optimization*, vol. 22, no. 2, pp. 628–648, 2012.

[15] N. Bäuerle, "Optimal control of queueing networks: An approach via fluid models," *Advances in Applied Probability*, vol. 34, no. 2, pp. 313–328, 2002. DOI: 10.1239/aap/1025131220.

[16] A. Ben-Tal, A. Goryashko, E. Guslitzer, and A. Nemirovski, "Adjustable robust solutions of uncertain linear programs," *Mathematical Programming*, vol. 99, pp. 351–376, 2004.

[17] A. Ben-Tal, A. Nemirovski, and G. Laurent El, *Robust Optimization*. CRC Press, 2015.

[18] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: A methodological tour d'horizon," *European Journal of Operational Research*, vol. 290, no. 2, pp. 405–421, 2021, ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2020.07.063. URL: https://www.sciencedirect.com/science/article/pii/S0377221720306895.

[19] L. Bertacco, M. Fischetti, and A. Lodi, "A feasibility pump heuristic for general mixed-integer problems," *Discrete Optimization*, vol. 4, no. 1, pp. 63–76, 2007.

[20] D. Bertsimas and J. Dunn, "Optimal classification trees," *Machine Learning*, vol. 106, no. 1, pp. 1039–1082, 2017.

[21] D. Bertsimas and J. Dunn, *Machine learning under a modern optimization lens*. Dynamic Ideas, 2019.

[22] D. Bertsimas and D. den Hertog, *Robust and Adaptive Optimization*. Dynamic Ideas, 2022.

[23] D. Bertsimas, E. Litvinov, A. Sun, J. Zhao, and T. Zheng, "Adaptive robust optimization for the security constrained unit commitment problem," *IEEE Transactions on Power Systems*, vol. 28, no. 1, pp. 52–63, 2013.

[24] D. Bertsimas and B. Stellato, "The voice of optimization," *Machine Learning*, vol. 110, pp. 249–277, 2 2021.

[25] D. Bertsimas and B. Stellato, "Online mixed-integer optimization in milliseconds," *INFORMS Journal on Computing*, 2022, appeared online.

[26] D. Bertsimas and R. Weismantel, *Optimization over Integers*. Dynamic Ideas, 2005.

[27] D. Bertsimas and D. Gamarnik, *Queueing Theory: Classical and Modern Methods*. Dynamic Ideas, 2022.

[28] D. Bertsimas and C. W. Kim, "A prescriptive machine learning approach to mixed-integer convex optimization," *INFORMS Journal on Computing*, vol. 35, no. 6, pp. 1225–1241, 2023.

[29] D. Bertsimas and C. W. Kim, *Optimal control of multiclass fluid queueing networks: A machine learning approach*, 2023. arXiv: 2307.12405 [cs.LG]. URL: https://arxiv.org/abs/2307.12405.

[30] D. Bertsimas and C. W. Kim, "A machine learning approach to two-stage adaptive robust optimization," *European Journal of Operational Research*, 2024.

[31] D. Bertsimas, E. Nasrabadi, and I. C. Paschalidis, "Robust fluid processing networks," *IEEE Transactions on Automatic Control*, vol. 60, no. 3, pp. 715–728, 2015.

[32] D. Bertsimas and J. Sethuraman, "From fluid relaxations to practical algorithms for job shop scheduling: The makespan objective," *Mathematical Programming*, vol. 92, no. 1, pp. 61–102, 2002.

[33] M. Biggs, W. Sun, and M. Ettl, "Model distillation for revenue optimization: Interpretable personalized pricing," in *Proceedings of the 38th International Conference on Machine Learning*, M. Meila and T. Zhang, Eds., ser. Proceedings of Machine Learning Research, vol. 139, PMLR, Jul. 2021, pp. 946–956. URL: https://proceedings.mlr.press/v139/biggs21a.html.

[34] L. Breiman, J. Friedman, C. J. Stone, and R. Olshen, *Classification and Regression Trees*. Chapman and Hall/CRC, 1984.

[35] C. G. Broyden, "A class of methods for solving nonlinear simultaneous equations," *Math. Comp.*, vol. 19, pp. 577–593, 1965.

[36] A. Caprara, M. Fischetti, and P. Toth, "A heuristic method for the set covering problem," *Operations Research*, vol. 47, no. 5, pp. 730–743, 1999.

[37] M. Carrion and J. Arroyo, "A computationally efficient mixed-integer linear formulation for the thermal unit commitment problem," *IEEE Transactions on Power Systems*, vol. 21, pp. 1371–1378, 2006.

[38] E. Carrizosa, C. Molero-Río, and D. Romero Morales, "Mathematical optimization in classification and regression trees," *TOP*, vol. 29, no. 1, pp. 5–33, Apr. 2021, ISSN: 1863-8279. DOI: 10.1007/s11750-021-00594-1. URL: https://doi.org/10.1007/s11750-021-00594-1.

[39] A. Cauligi, P. Culbertson, E. Schmerling, M. Schwager, B. Stellato, and M. Pavone, "CoCo: Online mixed-integer control via supervised learning," *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 1447–1454, 2022.

[40] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, ACM, 2016, pp. 785–794, ISBN: 978-1-4503-4232-2.

[41] J. K. Cochran and K. T. Roche, "A multi-class queuing network analysis methodology for improving hospital emergency department performance," *Computers & Operations Research*, vol. 36, no. 5, pp. 1497–1512, 2009.

[42] I. Cohen, K. Postek, and S. Shtern, "An adaptive robust optimization model for parallel machine scheduling," *European Journal of Operational Research*, vol. 306, no. 1, pp. 83–104, 2023.

[43] J. Dai, "On positive harris recurrence of multiclass queueing networks: A unified approach via fluid limit models," *The Annals of Applied Probability*, vol. 5, no. 1, pp. 49–77, 1995.

[44] J. Dai and M. Gluzman, "Queueing network controls via deep reinforcement learning," *Stochastic Systems*, vol. 12, no. 1, pp. 30–67, 2022.

[45] J. Dai and G. Weiss, "A fluid heuristic for minimizing makespan in job-shops," *Operations Research*, vol. 50, no. 4, pp. 692–707, 2002.

[46] H. J. Diekhoff, P. Lory, H. J. Oberle, H. J. Pesch, P. Rentrop, and R. Seydel, "Comparing routines for the numerical solution of initial value problems of ordinary differential equations in multiple shooting," *Numerische Mathematik*, vol. 27, no. 4, pp. 449–469, 1976.

[47] V. Dumas, "Diverging paths in fifo fluid networks," *IEEE Transactions on Automatic Control*, vol. 44, no. 1, pp. 191–194, 1999.

[48] M. Fischetti, M. Monaci, and D. Salvagnin, "Mixed-integer linear programming heuristics for the prepack optimization problem," *Discrete Optimization*, vol. 22, pp. 195–205, 2016.

[49] L. Fleischer and J. Sethuraman, "Efficient algorithms for separated continuous linear programs: The multicommodity flow problem with holding costs and extensions," *Mathematics of Operations Research*, vol. 30, no. 4, pp. 916–938, 2005.

[50] T. Fliedner and J. Liesiö, "Adjustable robustness for multi-attribute project portfolio selection," *European Journal of Operational Research*, vol. 252, no. 3, pp. 931–946, 2016, ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2016.01.058. URL: https://www.sciencedirect.com/science/article/pii/S0377221716300017.

[51] D. Gamarnik and J. J. Hasenbein, "Instability in stochastic and fluid queueing networks," *The Annals of Applied Probability*, vol. 15, no. 3, pp. 1652–1690, 2005.

[52] D. M. Gay, "Some convergence properties of Broyden's method," *SIAM J. Numer. Analysis*, vol. 16, pp. 623–630, 1979.

[53] A. Gleixner, G. Hendel, G. Gamrath, *et al.*, "MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library," *Mathematical Programming Computation*, 2021. DOI: 10.1007/s12532-020-00194-3. URL: https://doi.org/10.1007/s12532-020-00194-3.

[54] M. Goerigk and M. Khosravi, "Optimal scenario reduction for one- and two-stage robust optimization with discrete uncertainty in the objective," *European Journal of Operational Research*, vol. 310, no. 2, pp. 529–551, 2023.

[55] D. Grass, J. P. Caulkins, G. Feichtinger, G. Tragler, and. D. A. Behrens, Eds., *Optimal Control of Nonlinear Processes: With Applications in Drugs, Corruption, and Terror*. Berlin: Springer, 2008.

[56] Gurobi Optimization, LLC, *Gurobi Optimizer Reference Manual*, 2023. URL: https://www.gurobi.com.

[57] J. M. Harrison and L. M. Wein, "Scheduling networks of queues: Heavy traffic analysis of a two-station closed network," *Operations Research*, vol. 38, no. 6, pp. 1052–1064, 1990. (visited on 01/19/2023).

[58] E. Hüllermeier, "Prescriptive machine learning for automated decision making: Challenges and opportunities," *CoRR*, vol. abs/2112.08268, 2021. arXiv: 2112.08268. URL: https://arxiv.org/abs/2112.08268.

[59] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, ser. LION'05, Rome, Italy: Springer-Verlag, 2011, pp. 507–523, ISBN: 9783642255656. DOI: 10.1007/978-3-642-25566-3_40. URL: https://doi.org/10.1007/978-3-642-25566-3_40.

[60] L. Interpretable AI, *Interpretable ai documentation*, 2023. URL: https://www.interpretable.ai.

[61] M. I. Kamien and N. L. Schwartz, "Optimal maintenance and sale age for a machine subject to failure," *Management Science*, vol. 17, no. 8, B-495-B–504, 1971.

[62] B. Khalaf and D. Hutchinson, "Parallel algorithms for initial value problems: Parallel shooting," *Parallel computing*, vol. 18, no. 6, pp. 661–673, 1992.

[63] E. B. Khalil, P. L. Bodic, L. Song, G. Nemhauser, and B. Dilkina, "Learning to branch in mixed integer programming," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16, Phoenix, Arizona: AAAI Press, 2016, pp. 724–731.

[64] J. Kim and I. Yang, "Hamilton-jacobi-bellman equations for q-learning in continuous time," in *Proceedings of the 2nd Conference on Learning for Dynamics and Control*, A. M. Bayen, A. Jadbabaie, G. Pappas, P. A. Parrilo, B. Recht, C. Tomlin, and M. Zeilinger, Eds., ser. Proceedings of Machine Learning Research, vol. 120, PMLR, Jun. 2020, pp. 739–748. URL: https://proceedings.mlr.press/v120/kim20b.html.

[65] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, San Diega, CA, USA, 2015.

[66] J. Kleinberg and E. Tardos, *Algorithm Design*. USA: Addison-Wesley Longman Publishing Co., Inc., 2005, ISBN: 0321295358.

[67] P. R. Kumar, "Re-entrant lines," *Queueing Systems*, vol. 13, no. 1, pp. 87–110, 1993.

[68] M. Larrnaaga, U. Ayesta, and I. M. Verloop, "Dynamic control of birth-and-death restless bandits: Application to resource-allocation problems," *IEEE/ACM Transactions on Networking*, vol. 24, no. 6, pp. 3812–3825, 2016. DOI: 10.1109/TNET.2016.2562564.

[69] J. Lee and R. S. Sutton, "Policy iterations for reinforcement learning problems in continuous time and space — fundamental theory and methods," *Automatica*, vol. 126, p. 109 421, 2021.

[70] H. Lefebvre, E. Malaguti, and M. Monaci, "Adjustable robust optimization with discrete uncertainty," *INFORMS Journal on Computing*, vol. 36, no. 1, pp. 78–96, 2024.

[71] D. Lehmann, R. Müller, and T. Sandholm, *Combinatorial Auctions*. MIT Press, 2006, ch. 12, pp. 297–318.

[72] B. Liu, Q. Xie, and E. Modiano, "Reinforcement learning for optimal control of queueing systems," in *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2019, pp. 663–670.

[73] A. Lodi and G. Zarpellon, "On learning and branching: A survey," *TOP*, vol. 25, pp. 207–236, 2017.

[74] X. Luo and D. Bertsimas, "A new algorithm for state-constrained separated continuous linear programs," *SIAM Journal on Control and Optimization*, vol. 37, no. 1, pp. 177–210, 1998.

[75] M. Lutter, B. Belousov, S. Mannor, D. Fox, A. Garg, and J. Peters, "Continuous-time fitted value iteration for robust policies," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 5, pp. 5534–5548, 2023. DOI: 10.1109/TPAMI.2022.3215769.

[76] C. Maglaras, "Dynamic scheduling in multiclass queueing networks: Stability under discrete-review policies," *Queueing Systems*, vol. 31, no. 3, pp. 171–206, 1999.

[77] C. Maglaras, "Discrete-review policies for scheduling stochastic networks: trajectory tracking and fluid-scale asymptotic optimality," *The Annals of Applied Probability*, vol. 10, no. 3, pp. 897–929, 2000.

[78] A. Mate, L. Madaan, A. Taneja, N. Madhiwalla, S. Verma, G. Singh, A. Hegde, P. Varakantham, and M. Tambe, "Field study in deploying restless multi-armed bandits: Assisting non-profits in improving maternal and child health," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, pp. 12 017–12 025, Jun. 2022. DOI: 10.1609/aaai.v36i11.21460.

[79] S. Meyn, *Control Techniques for Complex Networks*, 1st. USA: Cambridge University Press, 2007.

[80] S. P. Meyn, "Transience of multiclass queueing networks via fluid limit models," *The Annals of Applied Probability*, vol. 5, no. 4, pp. 946–957, 1995.

[81] D. Miroslav, J. Langford, and L. Li, "Doubly robust policy evaluation and learning," in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, International Conference on Machine Learning, 2011, pp. 1097–1104.

[82] N. Modi, P. Mary, and C. Moy, "Transfer restless multi-armed bandit policy for energy-efficient heterogeneous cellular network," *EURASIP Journal on Advances in Signal Processing*, vol. 2019, no. 1, p. 46, 2019.

[83] V. Nair, S. Bartunov, F. Gimeno, *et al.*, *Solving mixed integer programs using neural networks*, 2021. arXiv: 2012.13349 [math.OC]. URL: https://arxiv.org/abs/2012.13349.

[84] G. Nemhauser and L. Wolsey, *Integer and Combinatorial Optimization*. John Wiley and Sons, Ltd, 1988.

[85] J. Niño-Mora, "Markovian restless bandits and index policies: A review," *Mathematics*, vol. 11, p. 1639, 2023.

[86] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *CoRR*, vol. abs/1912.01703, 2019. arXiv: 1912.01703. URL: http://arxiv.org/abs/1912.01703.

[87] D. Pessach, G. Singer, D. Avrahami, H. Chalutz Ben-Gal, E. Shmueli, and I. Ben-Gal, "Employees recruitment: A prescriptive analytics approach via machine learning and mathematical programming," *Decision Support Systems*, vol. 134, p. 113 290, 2020, ISSN: 0167-9236. DOI: https://doi.org/10.1016/j.dss.2020.113290. URL: https://www.sciencedirect.com/science/article/pii/S0167923620300452.

[88] P. K. Pollett, "An sis epidemic model with individual variation," *Mathematical Biosciences and Engineering*, vol. 21, no. 4, pp. 5446–5455, 2024, ISSN: 1551-0018. DOI: 10.3934/mbe.2024240. URL: https://www.aimspress.com/article/doi/10.3934/mbe.2024240.

[89] L. S. Pontryagin, V. G. Boltyanskii, R. V. Gamkrelidze, and E. F. Mishechenko, "The mathematical theory of optimal processes. viii + 360 s. new york/london 1962. john wiley & sons. preis 90/–," *Zamm-zeitschrift Fur Angewandte Mathematik Und Mechanik*, vol. 43, pp. 514–515, 1963.

[90] M. C. Pullan, "Existence and duality theory for separated continuous linear programs," *Mathematical Modelling of Systems*, vol. 3, no. 3, pp. 219–245, 1997.

[91] M. C. Pullan, "An algorithm for a class of continuous linear programs," *SIAM Journal on Control and Optimization*, vol. 31, no. 6, pp. 1558–1577, 1993.

[92] M. C. Pullan, "Forms of optimal solutions for separated continuous linear programs," *SIAM Journal on Control and Optimization*, vol. 33, no. 6, pp. 1952–1977, 1995.

[93] M. C. Pullan, "A duality theory for separated continuous linear programs," *SIAM Journal on Control and Optimization*, vol. 34, no. 3, pp. 931–965, 1996.

[94] M. C. Pullan, "An extended algorithm for separated continuous linear programs," *Mathematical Programming*, vol. 93, no. 3, pp. 415–451, 2002.

[95] M. Raeis, A. Tizghadam, and A. Leon-Garcia, "Queue-learning: A reinforcement learning approach for providing quality of service," in *AAAI Conference on Artificial Intelligence*, 2021.

[96] A. Rybko and A. Stolyar, "On the ergodicity of stochastic processes describing functioning of open queueing networks," *Problemy Peredachi Informatsii*, vol. 28, no. 3, pp. 3–26, 1992.

[97] M. B. Schaefer, "Some considerations of population dynamics and economics in relation to the management of the commercial marine fisheries," *Journal of the Fisheries Research Board of Canada*, vol. 14, no. 5, pp. 669–681, 1957.

[98] H. Scott Gordon, "The economic theory of a common-property resource: The fishery," *Bulletin of Mathematical Biology*, vol. 53, no. 1, pp. 231–252, 1991.

[99] C. See and M. Sim, "Robust approximation to multiperiod inventory management," *Operational Research*, vol. 58, no. 3, pp. 583–594, 2009.

[100] S. P. Sethi, *Optimal Control Theory*. Springer, 2019.

[101] E. Shindin, M. Masin, G. Weiss, and A. Zadorojniy, "Revised sclp-simplex algorithm with application to large-scale fluid processing networks," in *2021 60th IEEE Conference on Decision and Control (CDC)*, 2021, pp. 3863–3868.

[102] R. Srikant and L. Ying, *Communication Networks: An Optimization, Control and Stochastic Networks Perspective*. USA: Cambridge University Press, 2014.

[103] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis* (Texts in applied mathematics), 3rd ed. New York, NY: Springer, 2013, vol. 12.

[104] A. L. Stolyar, "On the stability of multiclass queueing networks: A relaxed sufficient condition via limiting fluid processes," *Markov Processes And Related Fields*, vol. 1, pp. 491–512, 4 1995.

[105] A. Sun and A. Lorca, "Adaptive robust optimization for daily power system operation," in *2014 Power Systems Computation Conference (PSCC)*, IEEE, 2014, pp. 1–9.

[106] A. Sun and A. Lorca, "Adaptive robust optimization with dynamic uncertainty sets for multi-period economic dispatch under significant wind," *IEEE Transactions on Power Systems*, vol. 30, pp. 1702–1713, 4 2015.

[107] R. Takapouia, N. Moehleb, S. Boyd, and A. Bemporad, "A simple effective heuristic for embedded mixed-integer quadratic programming," *International Journal of Control*, pp. 1–11, 2017.

[108] K. Wang, M. Aydemir, and A. Jacquillat, "Scenario-based robust optimization for two-stage decision making under binary uncertainty," *INFORMS Journal on Optimization*, vol. 0, no. 0, null, 2023.

[109] G. Weiss, "A simplex based algorithm to solve separated continuous linear programs," *Mathematical Programming*, vol. 115, pp. 151–198, 2008.

[110] P. Whittle, "Restless bandits: Activity allocation in a changing world," *J. Appl. Probab.*, vol. 25A, pp. 287–298, 1988.

[111] İ. Yanıkoğlu, B. Gorissen, and D. den Hertog, "A survey of adjustable robust optimization," *European Journal of Operational Research*, vol. 277, pp. 799–813, Sep. 2019. DOI: 10.1016/j.ejor.2018.08.031.

[112] B. Zeng and L. Zhao, "Solving two-stage robust optimization problems using a column-and-constraint generation method," *Operations Research Letters*, vol. 41, no. 5, pp. 457–461, 2013.

[113] Y. Zhao, D. Zeng, A. J. Rush, and M. R. Kosorok, "Estimating individualized treatment rules using outcome weighted learning," *Journal of American Statistical Association*, vol. 107, pp. 1106–1118, 2012.

[114] J. Zhen, D. den Hertog, and M. Sim, "Adjustable robust optimization via fourier-motzkin elimination," *Operations Research*, vol. 66, no. 4, pp. 1086–1100, 2018.