Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Lars Lødemel Sandberg

Ablating a Graph Neural Network for Branching in Mixed-Integer Linear Programming

Master's thesis in Engineering Cybernetics
Supervisor: Lars Struen Imsland
Co-supervisor: Bjarne Grimstad
May 2021

NTNU
Norwegian University of
Science and Technology

Lars Lødemel Sandberg

Ablating a Graph Neural Network for Branching in Mixed-Integer Linear Programming

Master's thesis in Engineering Cybernetics
Supervisor: Lars Struen Imsland
Co-supervisor: Bjarne Grimstad
May 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Abstract

This thesis evaluates ablations of a graph convolutional neural network for machine learning aided branching proposed by Gasse et al. (2019) for faster solving of mixed-integer linear programming (MILP) problems. Efficient MILP solution algorithms are important for real-time optimization in various industries, including industrial production, logistics, transportation, and energy production. Reduced computation time via merging machine learning with the branch-and-bound solution algorithm can improve these algorithms without sacrificing the strong benefits of global optimization. In 2019, reliable results of improvement over top branching policies in open-source solvers were shown, and in 2020 these methods have been expanded to purely CPU-based solutions. Different network topologies and feature sets on both GPUs and CPUs have been presented, however, the trade-off between accuracy and efficiency with these models on varying hardware is still mostly unexplored. In order to address this, two variants of graph convolutional neural networks and three multi-layer perceptron configurations are trained via imitation learning on the strong branching algorithm with generated MILP problems using the new framework *Ecole*. The models are then incorporated into the SCIP optimization solver and evaluated on test problems. All models show near state-of-the-art efficiency when run on the GPU. The models containing graph convolutions show a considerably larger reduction in efficiency than the MLPs when run on the CPU.

The source code for this project is available at
`https://github.com/Sandbergo/branch2learn`

Sammendrag

Denne oppgaven evaluerer ablasjoner av et grafkonvolusjonelt nevralt nettverk for maskinlæringsassistert forgrening foreslått av Gasse et al. (2019) for mer effektiv løsning av blandede heltallsproblemer (MILP). Effektive MILP-løsningsalgoritmer er viktige for optimering i sanntid i mange industrier, blant annet produksjon, logistikk, transport og energiproduksjon. Reduksjon i beregningstid ved å kombinere maskinlæring og branch-and-bound-løsningsalgoritmen kan forbedre disse algoritmene uten å ofre de sterke fordelene til global optimering. I 2019 ble pålitelige resultater med forbedring over de beste forgreningsstrategiene i åpen-kildekodeløsere vist, og i 2020 ble disse metodene utvidet til rent CPU-baserte modeller. Forskjellige nettverkstopologier og datasett for både GPU og CPU har blit foreslått, men kompromisset mellom nøyaktighet og effektivitet for modellene på forskjellig maskinvare er for det meste uutforsket. For å adressere dette blir to grafkonvolusjonale nevralnett og tre flerlagsperceptroner trent gjennom imitasjonslæring av strong branching-algoritmen på genererte MILP-problemer i det nye rammeverket *Ecole*. Modellene blir deretter inkorporert i optimaliseringsløseren SCIP og evaluert på testproblemer. Alle modeller viser nær høyeste effektivitet mot sammenlignbare algoritmer på GPU. Modellene med grafkonvolusjoner viser et mye mer betydelig effektivitetstap enn modellene uten når beregningene gjennomføres på CPU.

Kildekoden for denne oppgaven er tilgjengelig på
`https://github.com/Sandbergo/learn2branch`

Preface

This work constitutes my master thesis on how well different machine learning models can improve on a mathematical programming algorithm. The work is a part of a master's degree in *Engineering Cybernetics*, with the specialization *Autonomous System Control*. The work equates to 30 ECTS-credits, equal to one semester.

I have completed the experiments using only free and open-source software, and all experiments were run on computer hardware I have paid for myself. The hardware and software will be specified in Chapter 3.

The thesis is based on the work from my specialization project, *Multi-Layer Perceptrons for Branching in Mixed-Integer Linear Programming* (2020), and chapters, sections, formulations, and figures are taken or adapted from that report. These sections will be indicated at the beginning of each chapter, but some minor adaptions will not be mentioned in order to keep the thesis unencumbered by excessive comments.

The idea for the thesis as well as the research questions were formulated by myself. While not officially affiliated with the DS4DM group at the Polytechnique Montréal, their continuous work on the Ecole framework has been instrumental to the thesis. My questions and suggestions were answered immediately, and their cooperation was

very important for the success of the project. I would like to specifically mention Maxime Gasse, Didier Chélat, and Antoine Prouvost for their help, as well as other researchers and programmers who make their work publicly available. I hope to make a contribution in that regard by also making the code for this thesis openly available on GitHub.

Lastly, thank you to my supervisor professor Lars Struen Imsland and my co-supervisor Bjarne Grimstad, with whom I have had discussions regarding the thesis.

Contents

List of Tables

List of Figures

Glossary

$\mathcal{NP}$ Non-deterministic Polynomial-time. 35, 36

AI Artificial Intelligence. 3, 4

B&B Branch and Bound. 2, 5–10, 14, 17–19, 22–24, 29, 34, 47, 48, 64, 66, 70–73, 75–77

BLP Binary Linear Programming. 13, 14

CO Combinatorial Optimization. 14, 30, 31, 49, 50

COIN-OR Computational Infrastructure for Operations Research. 3

CPU Central Processing Unit. i, iii, xi, 5–9, 41, 42, 47, 58–60, 68–70, 72, 73, 75, 76

DL Deep Learning. 4

DS4DM Canada Excellence Research Chair in Data Science for Decision Making. v, 6

Ecole Extensible Combinatorial Optimization Learning Environments. i, iii, v, 6, 8, 9, 23, 33, 34, 48, 50, 51, 64, 70, 71, 73, 75, 76

FiLM Feature-wise Linear Modulation. 6, 21

FSB Full Strong Branching. 20, 47, 48, 58, 68

GCNN Graph Convolutional Neural Networks. 5, 7–9, 21, 25, 28, 29, 40–42, 70, 72, 75, 76

GNN Graph (Convolutional) Neural Networks. 28

GPU Graphics Processing Unit. i, iii, xi, 5–9, 41, 47, 50, 51, 58–60, 68, 69, 72, 73, 75

IA Iterative Ablations. 8

ILP Integer Linear Programming. xii, 13, 14, 18

LP Linear Programming. xii, 11, 13, 49

MDP Markov Decision Process. 22, 23

MIB Most Infeasible Branching. 20

MILP Mixed Integer Linear Programming. i, iii, 2, 3, 5, 11, 13–15, 23, 25, 26, 30, 34, 49, 75

MINLP Mixed Integer Non-Linear Programming. 49

MIP Mixed Integer Programming. 49

ML Machine Learning. 3, 4, 6, 7, 10, 26, 30, 31, 34, 38, 41, 47, 50, 54, 68, 70, 71, 75–77

MLP Multi-Layer Perceptron. i, 4, 5, 7, 9, 27, 41, 42, 44, 46, 64, 68, 75

PB Pseudo-cost Branching. 20, 21, 48, 58, 68

PO-MDP Partially-observable Markov Decision Process. 23, 24, 50, 71

RL Reinforcement Learning. 30, 31, 50

RPB Reliability Pseudo-cost Branching. 20, 21, 48, 58, 68

SB Strong Branching. 20, 21, 37, 48, 65

SCIP Solving Constraint Integer Programs. i, iii, 3, 6, 7, 21, 47–51, 75, 76

SVM Support Vector Machine. 5, 21

1

Introduction

This chapter presents a short history, motivation, and background in the fields of mathematical programming and machine learning, summarizes the previous work on the topic of *learning-to-branch*, poses the research questions for the project, and explains the structure of the project. Sections 1.1 and 1.2 and 1.5 are adapted from the project report *Multi-Layer Perceptrons for Branching in Mixed-Integer Linear Programming* (2020).

1.1 Background

In this section, a background in the relevant fields of mathematical programming and machine learning is presented, as well as an overview of the previous work in combining these fields. A familiarity of the reader with the central concepts and terms in this field is assumed.

1.1.1 Mathematical Programming

The invention of efficient solution algorithms for linear mathematical programming problems is considered one of the great post-war inventions [1]. The simplex algorithm and its derivatives have since become ubiquitous in a number of disciplines including finance, engineering, transportation, and energy [2]. These algorithms allow for the efficient solution of the global optimum of linear functions with an objective function, a number of variables and a number of constraints on these variables. However, the simplex algorithm is limited to problems where the *feasible set* of possible solutions is convex.

To further increase the expressiveness of the linear programming *language*, the inclusion of non-convex constraints such as limiting variables to only take integer values, is a reoccurring limitation in modeling real-world problems in a mathematical programming language. The set of linear optimization problems with these integer constraints is known as *mixed-integer linear programs* (MILP). The inclusion of integer constraints to linear problems has proved to be a very challenging problem class to develop efficient solution algorithms for, and it is considered unlikely that a polynomial-time solution algorithm exists [3].

For problems including integer constraints, the naïve approach of comparing every possible combination of feasible solutions, known as *explicit enumeration*, will result in a solution algorithm that is of exponential complexity, which makes larger optimization problems intractable [4]. An alternative to explicit enumeration is *implicit enumeration*, where a large number of possible solutions do not have to be evaluated explicitly. *Branch-and-bound* (B&B), conceived in 1960 [5], is an algorithm for solving mathematical programming problems via implicit enumeration that has become the standard solution algorithm [4]. It has since received numerous improvements and extensions [4].

For time-constrained applications of these non-convex optimization problems, a decrease in time to calculate the optimal solution can result in significant improvements and has been a very active field of research for many decades [4]. For the interested reader, the modern advances of *branch-and-cut*, *column generation*, and *Bender's decomposition* algorithms are recommended reading in *Integer Programming* by Lawrence Wolsey [4].

Currently, the most efficient solution algorithms using Branch and Bound are proprietary algorithms, notably IBM CPLEX and Gurobi [6]. Open-source solvers also exist and are under continuous development, e.g. COIN-OR and SCIP Optimization Suites [6], [7]. The algorithms are highly complex and employ a large variety of methods to quickly solve the complex problems [4].

There is a large interest in both theoretical and practical mathematical programming for methods that can improve the efficiency of MILP optimization algorithms [4]. This can have a lasting impact on the nature of mathematical programming and is therefore an important area to explore further. Researchers in mathematical optimization have also noted the potential for expert-constructed branching strategies based on knowledge obtained from *statistical learning*, also known as *machine learning* (ML) [8].

1.1.2 Machine Learning

The current dominating paradigm of *artificial intelligence* (AI) is *machine learning*, where computers (algorithms) learn from experience (data) [9]. The capabilities of these models have had exponential success in recent years, much due to advances in computer hardware [9]. A variety of fields have seen breakthroughs by using ML methods, including medical diagnostics, industrial optimization, autonomous vehicles, and board games [9].

There are a number of sub-fields within machine learning. In this project, the class known as *supervised classification* is explored. Supervised classification is the general problem of dividing instances into classes based on past instances and their respective classes. These models learn through observing examples of past data and their classifications. The basic assumption is that the ML model will learn from its experience, and be accurate in classifying previously unseen data.

An important field within machine learning in the last decade is *deep learning* (DL), where models are built up of series of nonlinear functions [9]. The *multi-layer perceptron* (MLP), or feedforward deep neural network, is the most common model in Deep Learning [9]. Nonlinear representations are iteratively performed, giving the model a large *capacity* for representing complex relations between input and output.

For many tasks, e.g. tasks with a real-time component, the strength of ML models lies in the fast evaluation of the generated, nonlinear functions, also known as the *inference* [10]. This strength makes it possible for machines to take over tasks previously thought to require a human operator or increase efficiency greatly via automation.

Further, the results of the iterative optimization process of generating an ML model can find patterns in data that are difficult to discover with traditional statistics [9]. Much like chess grandmasters learn from observing the best AI algorithms, so too might experts gain knowledge in their respective fields by analyzing the inner workings of an algorithm built on statistical learning.

Within the field of machine learning, the concept of *ablations* and *ablation studies* are gaining interest [11]. In an ablation study, the performance of a learned model is examined after the removal of sections of the original model. This is performed in an attempt to understand the role of a section within a complex system.

1.2 Previous Work

There has been a recent surge of interest in leveraging machine learning methods in solving non-convex optimization problems, notably in a recent literature review conducted by Yoshua Bengio [3]. The aim is for statistical learning to aid in the efficient solving of complex problems without sacrificing the strong guarantees inherent in mathematical optimization solvers. These hybrid methods now show the potential to be competitive with the state-of-the-art solvers for these difficult problems [12].

An overview of the history of learning in B&B is given a survey paper by Lodi and Zarpellon [8]. To summarize, interest in using more advanced statistics to unravel the relations of a MILP problem and the optimal branching variable was first presented in 2009. Various approaches to learning have been attempted, with recent efforts to directly incorporate the learned algorithms into the solution algorithm from 2016 and onwards [8].

A thorough look into the possibilities of machine learning in B&B was conducted by Elias Khalil [13], in which he chose the term *data-driven algorithm design* for this approach. In *Learning to Branch* (2016) [14] imitation learning of an expert branching policy. The algorithm was competitive with a selection of modern solvers [14]. Recent advances using *graph convolutional neural networks* (GCNNs) have proved to consistently improve on the solution time of the best available open-source solvers by Gasse et al. (2019) [12].

The promising results found by Gasse et al. (2019) [12] were, however, criticized by Gupta et al. (2020) [15] for reliance on modern GPU processing power. They showed that the efficiency of the GCNN-aided algorithm did not improve on the native branching strategies when run on a CPU. Gupta et al. presented novel methods running only on the CPU that were able to improve on the native strategies. These methods include *support vector machines* (SVM), *multi-layer perceptrons* (MLP), and *feature-wise*

linear modulation models (FiLM) [15].

Though GPU aided algorithms are interesting in their own right, a fair comparison of algorithmic efficiency cannot be made when the machine learning-based models are aided by expensive, advanced and specialized GPU processing power, as in Gasse et al. (2019) [12]. However, the efficiency when run on a GPU will be interesting for applications where this is available. For this reason, all analysis of computational efficiency in this project will be performed on both GPU and CPU. Discrete optimization performed on GPUs is a very interesting topic, for a discussion on Branch and Bound algorithms on GPUs the reader is referred to Schultz et al. [16]. It is also assumed by the author that advances in ML aided optimization can combine nicely with advances in parallel computing and specialized hardware.

The methods of Gasse et al. (2019) [12] and the improvements made by Gupta et al. (2020) [15] have shown that data-driven methods can improve upon existing state-of-the-art solvers, and is therefore an avenue worth examining, exploring, and expanding further. Machine learning is a rapidly evolving field, and recent advances have shown to outclass the early proof-of-concept attempts, sometimes by considerable margins [17]. There is little reason to believe that ML-leveraged algorithms do not have this latent potential.

Recent attempts to facilitate the development of ML-aided B&B includes a notable project named *Extensible Combinatorial Optimization Learning Environments* (*Ecole*) [18], [19]. It was developed by the group *Data Science For Decision Making* (DS4DM), connected to the Polytechnique Montréal university. Ecole is an open-source framework for a controllable and extensible python interface to B&B algorithms and is built on SCIP and PySCIPOpt [18]. The framework is based on previous work from the group, notably Gasse et al. (2019) [12] and Gupta et al. (2020) [15], and aims to standardize research within the field of B&B algorithm improvement. A recent article by Cappart et al. (2021) [20] presents the current status of and the role of the Ecole framework in further developing this field. As of May 2021, no articles have

been published with results using this framework, bar the aforementioned introducing articles.

1.3 Motivation

As stated, the ubiquity of B&B algorithms in industry implies that increased efficiency in computation time will be very beneficial. The benefits could be in the form of reduced resource expenditure on computations, increased time resolution in real-time applications, or even new applications due to the increased efficiency.

Machine learning has been proved by many researchers to be a good candidate for improving B&B [12]–[15], [21]. Of these, the graph convolutional neural network presented in Gasse et al. (2019) [12] has received the most attention and provides a very interesting approach to the variable selection problem in B&B. The model showed satisfactory improvements over the SCIP native brancher, however, the model was shown to be non-competitive when running on a CPU in Gupta et al. (2020).

While the alternative models presented in Gupta et al. (2020) [15] are interesting, a thorough exploration of the more standard models in ML (GCNNs and MLPs) is assumed to be more fruitful if they can achieve similar performance. The GCNN developed and presented in Gasse et al. (2019) [12] (from now on referred to as the *Gasse GCNN*) might not be competitive on the CPU in the current configuration, however, this does not necessarily mean that the GCNN architecture is unsuited for the application on a CPU.

A comparison, with the solution efficiency reported on both GPU and CPU, of a selection of simplified variants of the Gasse GCNN can give further insight into the model accuracy versus solution efficiency trade-off. When future researchers chose ML models for learning in B&B, this work will serve as a resource to better understand

this trade-off on the two different computational resources.

The term *iterative ablations* (IA) will be used to describe the process of removing parts of a network sequentially. The term is, to the knowledge of the author, only used once before, in the context of ablative liver surgery in Seror (2015) [22].

In addition, the choice to implement the source code in the new Ecole framework is a conscious decision to aid in the standardization of how research is done within the field of improving B&B algorithms.

1.4 Research Questions

For this project, three research questions are considered. This is done to aid the reader in following the thesis. The questions will be answered explicitly in Chapter 5. In addition, the topics of accuracy-efficiency-implementation will be reoccurring throughout the chapters of the thesis.

The research questions are as follows:

(i) *What is the impact of iterative ablations on the accuracy of the Gasse GCNN?*

The GCNN successfully implemented in Gasse et al. (2019) [12] was criticized in Gupta et al. (2020) for not being competitive with the classical strategies when run on a CPU. Hybrid models were implemented in Gupta et al. (2019) [15], where GCNN features were combined with the general variable features. This is done to mitigate the loss in efficiency by running GCNNs on the CPU. To investigate whether the original model can be competitive on both the GPU and CPU, 5 models are constructed by iteratively reducing the model size. These models will be referred to as GNN2, GNN1, MLP3,

MLP2, and MLP1. The last models will consist of multi-layer perceptrons with only the variable features, and will therefore have dissimilar names. The MLP models are devised based on promising results found in the project *Multi-Layer Perceptrons for Learning to Branch* (2020).

(ii) *What is the impact of iterative ablations on the efficiency of the Gasse GCNN when run on the CPU and GPU as a part of the B&B algorithm*?

Machine learning model choice based on reduced complexity is cited as a motivation for design choices in Gupta et al. [15], however the impact of varying sizes of the same model has not been performed before, and the magnitude of the impact is unknown. Increased model capacity is known to facilitate higher accuracy [9], however, whether this has a detrimental effect on the Branch and Bound algorithm's overall performance is unknown. Results from methods that are more comparable will indicate the importance of accuracy versus computational complexity. It is also not clear whether the graph convolution as developed in the Gasse GCNN is the component resulting in a detrimental loss of CPU performance. By conducting all experiments on both types of hardware, results should be sufficient to conclude with a degree of certainty what the accuracy and efficiency trade-offs are and what differences there are between the CPU and GPU methods.

(iii) *What are the most promising research opportunities for learning in Branch and Bound?*

The general field has gained traction recently [3], and the author assumes more research and interest will come in the next few years. In the implementation of this project, the new framework Ecole is used. This is the first paper where Ecole is used as the basis for the experiments, and an independent review of this framework is due. The road towards a conform and standardized framework and practice to develop methods in

the field of ML aided B&B can increase productivity and usher in breakthroughs in this field. On a broader note, as there is so much interest in the field, providing ideas for further research appears highly relevant.

1.5 Thesis Structure

Chapter 1 contains an introduction to the thesis with a short background in the relevant field, an overview of previous work, a section on the motivation for the conducted experiments, a formulation of the three research questions, and an overview of the thesis structure. In the following chapter, Chapter 2, the necessary theoretical background in optimization and machine learning is presented, as well as a review of earlier work in the field. In Chapter 3, the data set, chosen training and testing methods, and experiments are presented, along with the architectural choices. Chapter 4 provides the results of the aforementioned experiments. Then, Chapter 5 contains discussions of the results, a critique of the experiments, and ideas for further work. Finally, Chapter 6 summarizes the project with a conclusion on the implications of the results. Additional material is provided in Appendix A and Appendix B.

2

Background

The following chapter lays the theoretical groundwork for the project. An understanding of linear algebra, numerical optimization, algorithms, and statistics is assumed. Sections 2.1, 2.2 and 2.4.1 are adapted from the project report *Multi-Layer Perceptrons for Branching in Mixed-Integer Linear Programming* (2020).

2.1 Mathematical Programming

This section presents the field of *mathematical programming* at the level relevant for understanding the thesis. In this work, the terms mathematical programming, numerical optimization, and optimization are used interchangeably. The differences between these stem largely from the different communities who use them. The section will first cover the topic of linear programming (LP), then the topic of mixed-integer linear programming (MILP), and lastly a section on computational complexity.

2.1.1 Linear Programming

In mathematical programming, the general linear problem can be stated as [12]:

$$\arg\min_{\mathbf{x}} \left\{ \mathbf{c}^\top \mathbf{x} \mid \mathbf{A}\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \in \mathbb{R}_+^n \right\}, \tag{2.1}$$

where $\mathbf{x} \in \mathbb{R}_+^n$ is the variable vector with the objective coefficient vector $\mathbf{c} \in \mathbb{R}^n$, the constraint coefficient matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and the constraint right-hand-side vector $\mathbf{b} \in \mathbb{R}^m$.

The size of the problem will be measured by the dimensions of the constraint coefficient matrix $\mathbf{A}$, where the number of rows and columns corresponds to the number of variables and constraints, respectively.

These problems are convex [4], and can be solved by several efficient algorithms. The simplex algorithm can solve problems on this form efficiently, and the same for interior-point methods [23]. These algorithms are good average performance but do not have guaranteed polynomial running time in the worst case. Guaranteed polynomial solution algorithms do exist, for instance *Karamkar's algorithm* [24].

2.1.2 Mixed Integer Linear Programming

Mixed integer linear programming is a superset of linear programming, where one or more of the variables can be restricted to discrete values. The general problem can in this case be stated as [12]:

$$\arg\min_{\mathbf{x}} \left\{ \mathbf{c}^\top \mathbf{x} \mid \mathbf{A}\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \in \mathbb{Z}_+^p \times \mathbb{R}_+^{n-p} \right\}, \tag{2.2}$$

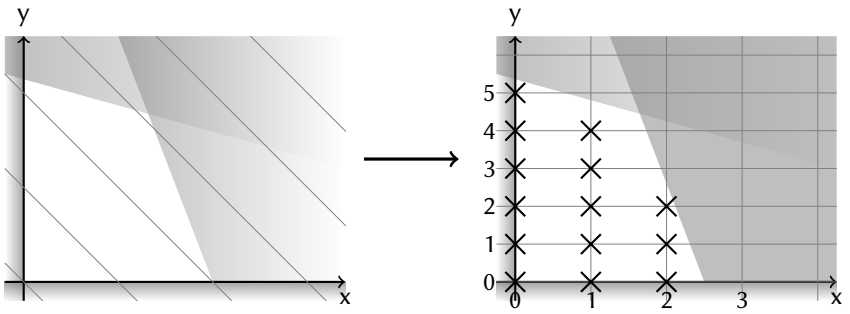

Figure 2.1: Illustration of an LP with its corresponding ILP, i.e. the LP with integrality constraints.

where p is the number of integer variables, otherwise the variables are the same as Equation (2.1).

In Figure 2.1, an LP problem and the problem with *integrality constraints* is shown. For the LP problem, the shaded areas represent the inequality constraints, where the diagonal lines represent the level curves of the objective function. In the ILP problem, the crosses represent the feasible solutions. The LP is also called a *relaxation* of the original ILP, which is fundamental to efficient solving algorithms of MILP problems.

A problem that includes integrality constraints cannot be convex [4]. The non-convexity of the feasible set of the problem constitutes a significant challenge, and it is considered unlikely that polynomial-time solutions exist [25]. MILP problems belong to the category of $\mathcal{NP}$-hard problems [25] (this class of problems will be discussed in Section 2.1.3).

A subset of MILP problems can be integer linear programming (ILP), where all variables are restricted to integer values, or binary linear programming (BLP), where all variables are restricted to binary values.

ILP and BLP problems belong to the category of *combinatorial optimization* (CO) problems, which has been the main focus of the efforts to solve entirely or partially with machine learning methods [3].

2.1.3 Computational Complexity

A basic understanding of computational complexity is required to justify the nature of the algorithms used to solve MILP problems.

Problems can be divided into *classes* by the nature of the algorithms that can solve these problems. Problems for which there exist algorithms that can solve the problem in a time that is *polynomial* of the problem size belong to class $\mathcal{P}$. Problems to which a correct solution can be verified in polynomial time belong to the class $\mathcal{NP}$ (non-deterministic polynomial time) [26].

Two other central complexity classes in this context are the $\mathcal{NP}$-complete and $\mathcal{NP}$-hard classes. The $\mathcal{NP}$-complete class contains problems that can be *reduced* to any other problem in the $\mathcal{NP}$-complete class in polynomial time. The $\mathcal{NP}$-hard class contains problems that are at least as hard as the problems in the $\mathcal{NP}$-complete group but has not been proved to be reducible to a $\mathcal{NP}$-complete problem. An illustration showing this is given in Figure 2.2. The general MILP belongs to the $\mathcal{NP}$-hard class, and some MILPs have been shown to belong to the $\mathcal{NP}$-complete class [26].

As stated, it is considered unlikely that MILP problems can be solved in polynomial time. Therefore, it is more fruitful to improve upon the best existing solution algorithms and evaluating the improvements on practical problems. As the improvements attempted by substituting variable selection algorithms do not affect the running time complexity of the B&B algorithm, this will not be discussed.

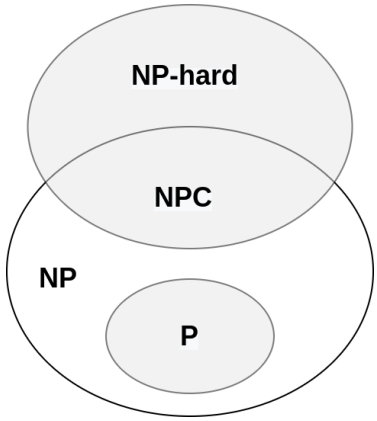

Figure 2.2: Illustration of the most commonly held view of the P, NP, NPC, and NP-hard relationship. Adapted from Cormen et al. (2009) [26].

2.2 Branch and Bound

A *relaxation* of a MILP problem is achieved by relaxing the integrality constraint, as shown in Figure 2.1. Obtaining the solution to the relaxed problem gives a lower bound on the optimal solution (for a minimization problem). Naturally, any feasible solution to the integrality-constrained problem gives an upper bound to the solution. Furthermore, if the solution to the relaxed problem adheres to the integrality constraints, it is also the solution to the MILP problem [4].

The most prevalent solution algorithm for MILP problems exploits these results, by sequentially dividing the solution space until the optimum with the integrality constraint is found. This is done by branching in a binary tree structure according to [12]:

$$x_i \leq \left\lfloor x_i^\star \right\rfloor \vee x_i \geq \left\lceil x_i^\star \right\rceil, \quad \exists\, i \leq p \mid x_i^\star \notin \mathbb{Z} \tag{2.3}$$

Further creating sub-problems with this binary decomposition. A general algorithm for this process is presented in Algorithm 1, and an illustration of this process is shown in Figure 2.3.

Algorithm 1: A generic branch-and-bound algorithm [27].

Result: Optimal point and solution value of given problem.
Set $L = \{X\}$ and initialize $\hat{x}$;
while $L \neq \emptyset$ **do**
    Select a subproblem S from L to explore;
    if *a solution* $\hat{x}_* \in \{x \in S \mid f(x) < f(\hat{x})\}$ *can be found* **then**
        Set $\hat{x} = \hat{x}_*$;
    end
    if S *cannot be pruned* **then**
        Partition S into $\{S_1, S_2..., S_r\}$;
        Insert $\{S_1, S_2..., S_r\}$ into L;
    end
    Remove S from L;
end
Return $\hat{x}$;

For each generated solution, represented by nodes in Figure 2.3, a relaxation of the problem is solved in order to obtain an upper and lower bound on the solution of the sub-problem. These values are shown on the top and bottom right, respectively. Generating upper and lower bounds for solutions allows for discarding a large number of solutions [4]. Branches can be *pruned* (meaning no further partitioning from that branch) if they meet at least one of the following three criteria [4]:

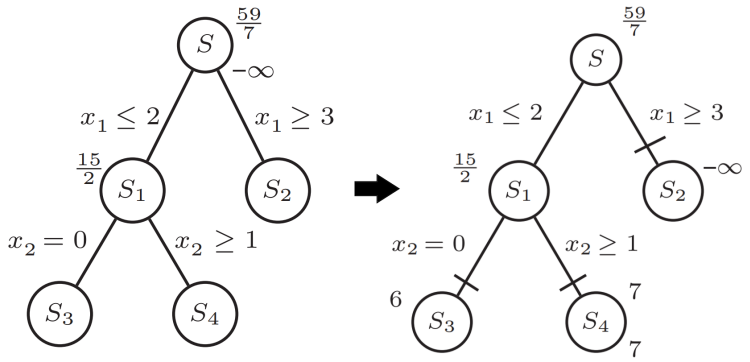

Figure 2.3: Illustration of the branch-and-bound algorithm adapted from a maximization problem in *Integer Programming* (2020) [4].

(i) Pruning by optimality: $Z^t = \{\max \boldsymbol{c}^\top \boldsymbol{x} : \boldsymbol{x} \in S_t\}$ has been solved.

(ii) Pruning by bound: $\overline{Z}^t \leq \underline{Z}^t$.

(iii) Pruning by infeasiblity: $S_t = \emptyset$.

For Figure 2.3, the graph on the right represents the tree after solving the relaxation, resulting in S_2 being pruned by infeasibility, S_3 pruned by bound, and S_4 pruned by optimality.

The choice of node and variable to branch on to find the optimum in the fewest number of branching processes is central to an efficient implementation of B&B. Partitioning the feasible set such that the node with the optimal value is found in the fewest possible branching iterations is the optimal policy.

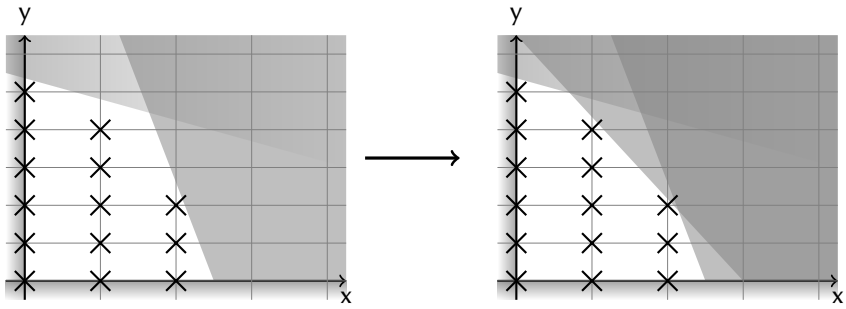

Figure 2.4: Figure of an ILP before and after an added valid inequality.

2.2.1 Valid Inequalities

Another important method used in B&B algorithms is the concept of valid inequalities. A valid inequality is an inequality that does not remove feasible solutions of the non-convex solution set but can remove potential solutions to the relaxed problems. A valid inequality can be expressed as:

$$\boldsymbol{\pi}^\top \mathbf{x} \leq \pi_0 \quad \forall\, \mathbf{x} \in \boldsymbol{X} \tag{2.4}$$

where $\boldsymbol{X}$ is the feasible set as described in Equation (2.2). These inequalities reduce the size of the feasible set for the relaxations of the problem without removing feasible solutions of the original problem. An illustration of an ILP with an added valid inequality is shown in Figure 2.4. Here the feasible set of the relaxation is reduced in size by the added constraint, while the feasible points of the ILP remain feasible after the application of the inequality, as is given in Equation (2.4).

Algorithms that find these inequalities during the B&B algorithm are called *branch-and-cut*. The nomenclature comes from calling the application of these inequalities *cuts* or *cutting planes*. When an application of inequalities are only employed on the root node (before dividing the solution space in the enumeration), the algorithm is

sometimes referred to as *cut-and-branch* rather than *branch-and-cut* [4].

2.2.2 Primal and Dual Heuristics

The modern implementations of B&B solvers base their efficiency on the implementation of *heuristics* [13], which are divided into the classes *primal* and *dual*. Heuristic is synonymous with "human-designed rule" in this context.

Primal heuristics are methods for finding feasible solutions at a given B&B node, where the quality, i.e. the distance to the optimal bound, is the determining factor to whether the feasible solution is useful or not [13]. These heuristics are as costly as they are useful, and modern solvers periodically run different heuristics at different times during the solution process [13].

Dual heuristics are the methods that find the lower bound of the optimization problem. This includes solution of relaxations of the problem as well as the addition of valid inequalities.

The relationship is summarized in this quote from Khalil (2020) [13]:

> [...] the primal side refers to the quest for good feasible solutions, whereas the dual side refers to the search for a proof of optimality.

2.2.3 Branching Variable Selection Policy

As mentioned, an important decision in the B&B algorithm is the choice of the variable that should be branched on. There exists many heuristics for solving this, who vary in computational complexity and accuracy. A good branching algorithm should choose

to branch on variables that lead to small solution trees (fewer nodes evaluated) and find these variables in a computationally efficient manner.

All popular variable selection policies depend on scoring the candidate branching variables, expressed as $s_i \in \mathbb{R}^1 \; \forall \; i \in C$, and then selecting the variable with the most optimal score [28]. The branching operation generates two child nodes, Q_i^- and Q_i^+. The branching candidate comparison is done by comparing the two objective function changes of each candidate, denoted as $\Delta_i^- := \bar{c}_{Q_i^-} - \bar{c}_Q$ and $\Delta_i^+ := \bar{c}_{Q_i^+} - \bar{c}_Q$ [28]. The final score is then typically calculated by a function similar to [28]:

$$score(q^-, q^+) = (1 - \mu) \cdot \min\{q^-, q^+\} + \mu \cdot \max\{q^-, q^+\}, \quad \mu \in [0, 1] \tag{2.5}$$

The current branching policy resulting in the smallest solution trees is known as *strong branching* (SB) [29], and the application of this branching policy at every node is known as *full strong branching* (FSB) [28]. This branching policy is based on determining the best variable to branch on by solving the relaxation for every candidate variable, and is therefore very computationally expensive compared to other methods [28].

Another branching policy is *most infeasible branching* (MIB), where the variable with the fractional part of the relaxation optimum closest to 0.5 is selected. This policy, though computationally inexpensive, has proved to be very poor [28].

An effective and popular policy is *pseudo-cost branching* (PB), which relies on the expected change in objective value based on previous branching on the variable in question [28]. In short, the objective gain per unit change in a variable is averaged over all nodes where it has been branched upon. This value is termed the *pseudo-cost* of the variable. As is evident, these values depend on a history of branching, and will therefore be inaccurate for the first decisions [28].

The policy known as *reliability pseudo-cost branching* (RPB) aims to mitigate the

inaccuracy of the PB algorithm by combining SB and PB [6]. In RPB, SB is employed for variables that are either uninitialized (never branched on before) or have *unreliable* pseudo-costs (pseudo-costs that stem from little data) [28]. This policy is the standard of the SCIP optimization suite [7].

In the literature, the branching policy is referred to as a policy, strategy, or rule. In this thesis *policy* is used.

2.2.4 Learned Branching Policy

Recently, attempts have been made to find a branching policy based on statistical learning.

Using machine learning, specifically imitation learning, to find good candidate variables for branching in a less computationally demanding manner was proposed by Elias Khalil [14]. Various methods for learning in branching include *support vector machine ranking* (SVM) [14], *graph convolutional neural networks* (GCNN) [12] and *feature-wise linear modulation* (FiLM) [15].

The fundamental assumption to this approach is that a computationally efficient approximation to the most computationally demanding but most accurate branching policy can be learned. The algorithm will use imitation learning on the branching expert to find a computationally less expensive non-linear function approximation to the expert algorithm's variable scoring. Then, the algorithm branches on the variable with the highest score.

2.3 Markov Decision Processes

This section presents the *Markov decision process* formulation of the variable selection problem.

2.3.1 Markov Decision Processes Formulation

Central to the advancement of learned policies in B&B is the interpretation of the solution algorithm as an *agent* in a *Markov decision process* (MDP) [12]. This interpretation relates the problem to a large collection of literature on the topic [30].

In an MDP, the agent is at time t in a state $\mathcal{S}_t$, from which it performs an action $\mathcal{A}_t$ that transforms the agent to the state $\mathcal{S}_{t+1}$ and receives the *reward* $\mathcal{R}_{t+1}$ [19]. The probability of an agent performing action a in state s is given as $\pi(a|s)$. The probability distribution for the agent to transition to a new state s' is given as $\mathbb{P}(s', r|a, s)$ [19].

A sequence of actions generates a sequence of trajectories τ, and is described as an *episode*. The probability of a trajectory is given in Prouvost et al. (2021) [18] as:

$$\mathbb{P}(\tau) \sim \underbrace{\mathbb{P}(\mathcal{S}_0)}_{\text{initial state}} \prod_{t=0}^{\infty} \underbrace{\pi(\mathcal{A}_t|\mathcal{S}_t)}_{\text{next action}} \underbrace{\mathbb{P}(\mathcal{S}_{t+1}, \mathcal{R}_{t+1}|\mathcal{A}_t, \mathcal{S}_t)}_{\text{next state}} \tag{2.6}$$

These definitions now allow a formulation of the MDP control problem, which is the problem of interest in this thesis. The control problem consists of finding the action policy that maximizes the reward.

2.3.2 Partially-observable Markov Decision Processes

A subset or generalization of an MDP is the *partially-observable Markov decision process* (PO-MDP) [31]. Processes of this class allow for uncertainty of the states as well as additional acquisition of state information [31]. The agent will therefore decide actions based on the observation of the state, given as O [19]. All past observations of the observations, rewards and actions are given in the history $\mathcal{H}_t$, given as [19]:

$$\mathcal{H}_t = \{O(\mathcal{S}_0), \mathcal{R}(S_0), \mathcal{A}_0, ..., O(\mathcal{S}_{t-1}), \mathcal{R}(S_{t-1}), \mathcal{A}_{t-1}, O(\mathcal{S}_t)\} \tag{2.7}$$

The generalization from MDP to PO-MDP concedes the Markovian nature of the trajectories [18].

In addition, the initial state is given by the distribution of the problem instance I, giving the relation $\mathbb{P}(\mathcal{S}_0) = \mathbb{P}(I)\mathbb{P}(\mathcal{S}_0|I)$ [19].

This results in the final formulation [19]:

$$\mathbb{P}(\tau) \sim \underbrace{\mathbb{P}(I)\mathbb{P}(\mathcal{S}_0|I)}_{\text{initial state}} \prod_{t=0}^{\infty} \underbrace{\pi(\mathcal{A}_t|\mathcal{H}_t)}_{\text{next action}} \underbrace{\mathbb{P}(\mathcal{S}_{t+1}, \mathcal{R}_{t+1}|\mathcal{A}_t, \mathcal{S}_t)}_{\text{next state}} \tag{2.8}$$

An illustration of the Markov decision process control loop from the documentation of Ecole is shown in Figure 2.5.

2.3.3 Branch & Bound as a PO-MDP

Interpreted in the language of MDPs, the B&B algorithm is the *environment* and a concrete MILP problem instance is an *episode* in this environment. The *agent* is the

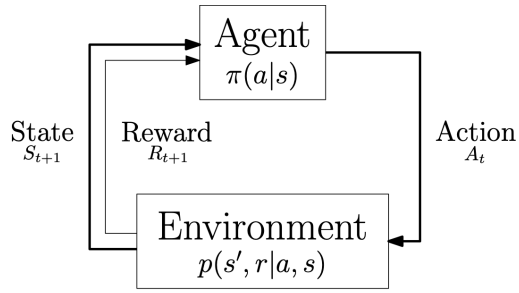

Figure 2.5: Illustration of the Markov decision process control loop. Figure from Prouvost et al. (2021) [19].

brancher, where in this thesis the variable selection policy is the component of interest, ignoring the node selection policy. The state of the solver consists of the B&B tree at that instance, as well as the observations at each node (the *history* of the PO-MDP).

This formulation is the basis for the *Ecole* framework, which is discussed in Section 3.5.3 The PO-MDP formulation allows for the agent in the B&B environment to be learned through reinforcement learning, discussed in Section 2.4.4.

2.3.4 Branch & Bound Observation

A prerequisite for learning in B&B is the observation of the state of the episode, i.e. the state of the solver of an instance at a specific node in the solution tree.

The features of a B&B node are divided into three classes: variable features, constraint features and edge features.

Variable Features

For a candidate branching variable, relevant features include the type of the variable

(binary, integer, etc.), whether the variable has a defined lower and/or upper bound, and whether the solution is at at either of these bounds. If not, the variable has a fractionality that represents the solution of the relaxed problem. At the solution node, the incumbent has a value that can be compared to incumbents at other nodes, as well as the relative impact of the variable on the objective value in the incumbent. The variable also has a state with respect to the solution of the relaxation with a simplex solution algorithm — if the variable is a basic or non-basic variable or other information relating to this solution. Presented in Khalil et al. (2016) [14] are also a number of other features that will not be utilized in this work.

Constraint Features

The cosine similarity represents a coefficient of the angle between the variable and the constraint. The bias of the constraint is also included. An additional feature is whether the variable is at the constraint in the relaxation Each constraint also has a value from the solution of the dual problem.

Edge Features

The edge features consist of the constraint coefficient, meaning the coefficient that is multiplied with the candidate variable. This will also give the relations between constraints and variables.

2.3.5 Bipartite Graph Representation

The application of GCNNs on MILP and sub-MILP problems rely on the bipartite representation of constraints and variables as presented in Gasse et al. (2019) [12].

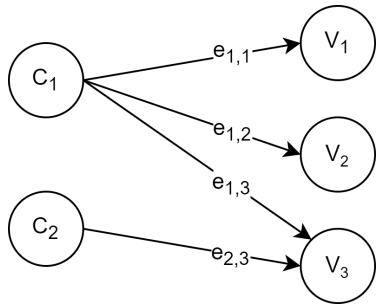

Figure 2.6: Example of a bipartite constraint-variable graph.

This concept will be introduced with an example MILP given as:

$$\begin{aligned} \min \quad & v_1 + v_2 + v_3 && (2.9)\\ s.t. \quad & v_1 + v_2 - v_3 \geq 1 && (c_1)\\ & v_3 \geq \frac{1}{2} && (c_2)\\ & \mathbf{v} \in \mathbb{B}^3 \end{aligned}$$

For Equation (2.9), the corresponding constraint-variable graph representation can be illustrated as in Figure 2.6.

Constraints and variables are the numbered nodes of the graphs, while the edges represent the relation between the nodes.

2.4 Machine Learning Models

The ML-models used in the thesis are presented in this section. First the multi-layer perceptron and graph convolutional neural network models, then the concepts of

ablation studies and *reinforcement learning*.

2.4.1 Multi-layer Perceptrons

Multi-layer perceptrons (MLPs), more commonly known as deep feed-forward neural networks, are recommended by Gupta et al. (2020) [15] as a less computationally expensive alternative to the approaches by Khalil et al. (2016) [14] and Gasse et al. (2019) [12].

MLPs are networks that generate a nonlinear function $y = f(\mathbf{x}; \boldsymbol{\theta})$, where x is the input, y is the output, and $\boldsymbol{\theta}$ represents the parameters of the function. The parameters are learned during repeated optimization, and will under ideal circumstances converge to approach the optimal function $y = f^*(\mathbf{x})$. The function is realized as a series of compositions of functions. The composed functions are represented as an acyclical, directed graph [32], and can be expressed as:

$$y = f_L \circ f_{L-1} \circ \ldots \circ f_1 \circ f_0(\mathbf{x}) \tag{2.10}$$

The functions are denoted as *layers* of the perceptron, and are implemented as affine functions of every input parameter at every node, $\mathbf{z}_l = \mathbf{x}_{l-1}^T \mathbf{w}_l + \mathbf{b}_l$. Applying non-linear function, known as an *activation function*, allows the MLP to represent arbitrary nonlinear functions [9]. This is expressed as $\mathbf{x}_l = \mathbf{a}(\mathbf{z}_l)$.

The computation of the output of the function given its input is known as a *forward pass* through the network. The required computations for a single input vector into a network with n hidden layers will include $n + 1$ matrix multiplications and $n + 1$ applications of the non-linear activation function, given that there is an activation function on the output.

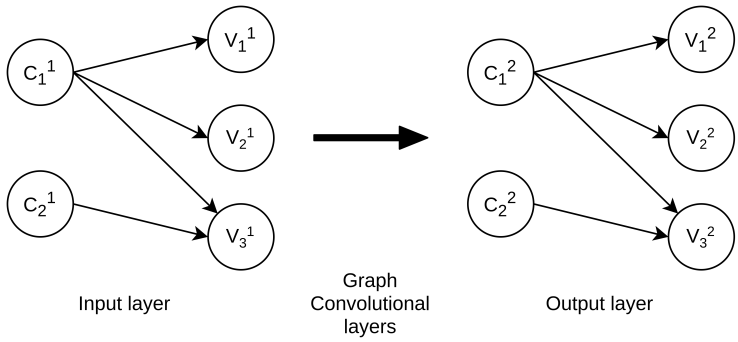

Figure 2.7: Example of a graph convolution on a bipartite constraint-variable graph.

2.4.2 Graph Convolutional Neural Networks

Graph convolutional neural network (GCNN) is a term for neural networks that have input data represented in a graph-structure that is processed by a convolution operation [33]. In this thesis, the terms GCNN and GNN will be used interchangeably.

The fundamental property of the graph convolution is its ability to create representations of irregular data without altering the structure of the data. This means that, for instance, nodes that share vertices can pass information to each other, so the feature representation of a node can utilize the features of neighboring nodes. An illustration of this using the bipartite graph from Figure 2.6 is given in Figure 2.7. The features of the nodes are transformed while maintaining the structure of the graph.

A graph convolution can be expressed as a matrix/tensor multiplication followed by a nonlinear activation function, as in Section 2.3. This will be explained in the case of a undirected graph for the sake of simplicity. A prerequisite for this is the representation of the graph with the adjacency matrix, denoted as $\tilde{\mathbf{A}}$. The adjacency matrix is a square $|\mathbf{V}| \times |\mathbf{V}|$ matrix containing 0 or 1 depending on whether the pair of vertices are connected or not [33]. In addition the adjacency matrix, the degree matrix $\tilde{\mathbf{D}} = \sum_j \tilde{\mathbf{A}}_{ij}$

is necessary in order to normalize the operation [33].

With the given definitions, the graph convolution operation can be expressed by the propagation rule [33]:

$$\mathbf{H}^{(l+1)} = a\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right) \tag{2.11}$$

where $\mathbf{H}^{(l)}$ is the matrix of activations in layer l, with the first layer $\mathbf{H}^0 = \mathbf{X}$. $\mathbf{W}^{(l)}$ is a layer of learned weights. $a(\cdot)$ is a nonlinear activation function. This operation is inspired by the first-order approximations to spectral filters on graphs [33].

Models that leverage the graph nature of combinatorial optimization problems have been shown to have satisfactory performance, see e.g. Dai et al. (2018) [34]. GCNNs are proposed by Gasse et al. (2019) [12] as an alternative to the feature-rich approaches by Khalil et al. (2016) [14]. The application of GCNNs on B&B algorithms rely on the bipartite constraint-variable representation at each node of the B&B solution tree.

The term *embeddings* will be used in this thesis for continuous-variable representations derived from the input features, as it is used in Gasse et al. (2019) [12].

The state of the B&B graph at a node can be represented as $s_t = (\mathcal{G}, \mathbf{C}, \mathbf{E}, \mathbf{V})$, where $\mathcal{G}$ represents the bipartite B&B solution graph at that time instance, $\mathbf{C}$ represents the constraints, $\mathbf{E}$ represents the *edges* (connections) between the variables and constraints, and $\mathbf{V}$ represents candidate variables.

Gasse et al. (2019) [12] presents three motivating points for why graph convolutions would be a good architecture for learning to branch:

(i) They are well-defined no matter the input graph size.

(ii) Their computational complexity is directly related to the density of the graph,

which makes it an ideal choice for processing typically sparse MILP problems.

(iii) They are permutation-invariant, that is they will always produce the same output no matter the order in which the nodes are presented.

2.4.3 Ablation Studies

The concept of ablation studies in machine learning, as presented in Meyes et al. (2019) [11] is presented in this section. Ablation studies hail from the field of neuroscience, in which a complex system, e.g. the brain, is examined after removing different sections. The function of the removed sections can then be inferred by the change in the observed reaction to external stimuli [11].

In the context of ML, ablation studies are a formalization of observing changes in performance after the removal of components of artificial neural networks [11]. The concept, or at least the formalization, is not yet considered a standard method in ML research [35]. In this thesis, the concept of an ablation study will be interpreted more broadly than in Meyes et al. (2019) [11], as the networks in this thesis are retrained after each section is removed. This form of ablation study is coined as *model ablation* in Sheikholeslami (2019) [35].

2.4.4 Reinforcement Learning

The subset of machine learning described as *reinforcement learning* (RL) is highly relevant in the context of ML in CO. No results will be discussed in this thesis, however, a background in the topic is necessary to understand both related work and the long-term goals of the field.

RL encompasses the problem of an *agent* learning a *policy* for behaving in an *envi-*

ronment so as to achieve a global objective. Any sequential decision-making problem with a measure of optimality that relies on past experience can be formulated as a RL problem [36]. The approach has seen success in a number of fields in the past years with the integration of deep learning models, often termed deep RL [36]. Most notable of the advancements might be AlphaZero, Google's successful chess-AI [37]. RL has the important property of being independent of data, meaning a number of core ML challenges (quantity, quality, and bias of data) are rendered irrelevant [9].

Many attempts have been made at implementing RL in CO, see for example Etheve et al. (2020) [21] or Tang et al. (2020) [38]. Approaches for learning variable selection, such as reported in Scavuzzo (2020) [39], rely on efficient and accurate pre-trained models based on imitation learning, like the models presented in this thesis. More knowledge is likely needed for the pure RL approach to take over the mantle. Recently, Cappart et al. (2021) [20] also concluded that useful RL policies are not mature yet.

3

Methods

In this chapter, the selected methods for the experiments are presented and discussed. An understanding of the theoretical groundwork of the project from Chapter 2 is assumed. A substantial part of the methods are taken directly or indirectly from the source code of Gasse et al. (2019) [12], Gupta et al. (2020) [15], and the Ecole source code [18]. Some sentences and formulations are adapted from the project report *Multi-Layer Perceptrons for Branching in Mixed-Integer Linear Programming* (2020), as this thesis shares methods with the report.

3.1 Dataset

This section presents the selected data set and the process of generating trainable and testable data for the models. This will include a presentation of the problem instances, the generation of the expert solutions as well as the features that will be the input of

the ML models.

3.1.1 Problem Instances

In order to evaluate the methods presented in this project to the previous advances in the field, artificially generated MILP problems found in Gasse et al. (2019) [12] are used to train and evaluate the models. These problems are also the standard implemented in Ecole. The problems are expressed as pure binary programs. The results of the algorithm are, however, generalizable to general MILP problems, as it extends the general B&B algorithm [12].

In the mentioned articles, four problem classes are used: set covering, combinatorial auctions, capacitated facility location, and maximum independent set. In this thesis, only the first two will be used due to problems in the Ecole framework that have since been resolved.

Set Covering

The set covering problem is implemented in Ecole as described in Balas & Ho (1980) [40] [1]. The general problem includes a set of vertices V_j and a matrix of costs c_j for activating an edge between any two vertices. The activation of an edge is modeled with the binary variable x_j and the characteristic vector of subset V_j denoted $\mathbf{A}_j$. The binary linear problem of covering all vertices at minimum cost can be expressed as

[1] The available version of this paper contains nearly illegible equations, the article Minoux (1987) [41] has therefore been used to supplement.

[41]:

$$\min \sum_{i=1}^{N} c_i x_i \tag{3.1}$$

$$s.t. \sum_{i=1}^{N} \mathbf{A}_i x_i \leq \mathbf{1} \tag{3.2}$$

$$\mathbf{x} \in \mathbb{B}^N \tag{3.3}$$

The problem is therefore a pure binary linear problem with only inequality constraints. The set covering problem is a well-known class and is one of Karp's 21 $\mathcal{NP}$-complete problems [42].

Combinatorial Auction

The combinatorial auction problem is the problem with the shortest solution time in the mentioned articles [12], [15]. The problem is implemented by Gasse et al. (2019) [12], and is based on the formulation presented in Leyton-Brown et al. (2020) [43]. The version is specifically the *arbitrary* formulation from the original article [43]. In this formulation, N is the number of combinations of items, p_i is the price of an item, $\mathcal{P}$ is the set of all items, and $a_{i,j} \in \mathbb{B}$ represents whether item j belongs to combination i. With this, the problem can be stated as [44]:

$$\max \sum_{i=1}^{N} p_i x_i \tag{3.4}$$

$$s.t. \sum_{i=1}^{N} \mathbf{a}_{i,j} x_i \leq 1, \quad \forall\, j \in \mathcal{P} \tag{3.5}$$

$$\mathbf{x} \in \mathbb{B}^N \tag{3.6}$$

The combinatorial auctions problem is based on multi-object auctions where bidders place monetary bids on bundles (combinations) of goods, and the optimization problem

is to find the bids that maximize the profit of the auctioneer [43]. The problems are considered realistic and economically oriented, and are applicable to five broad domains in which optimization is important: proximity in space, paths in space, arbitrary relationships, temporal matching, and temporal scheduling [43]. The problems are $\mathcal{NP}$-hard [45].

3.1.2 Expert Solution Generation

The generated problems are solved using the full strong branching policy, as explained in Section 2.2.3. To not interfere with branching policies, the application of cutting planes to the problem is restricted to the root node, before any branching decisions are made. This means that the Branch-and-bound algorithm used in these experiments could be called a *cut-and-branch* algorithm, as is discussed in Section 2.2.1. However, without loss of generality and for simplicity, the algorithm will be referred to as

branch-and-bound in this thesis despite the initial application of cutting planes.

Algorithm 2: Data collection algorithm

```
Result: Variable selection samples
for episode ∈ EPISODES do
    while episode not solved do
        ρ ← rand();
        if ρ ≤ 0.05 then
            samples ← node_state;
            scores ← SB_scores;
            Save scores and state;
            Branch on SB score;
        else
            Branch on pseudo-cost scores;
        end
    end
end
```

The algorithm for data collection is shown in Algorithm 2 and is adapted from Gasse et al. (2019) [12]. During training, 5 % of the branching variable decisions are done by the strong branching expert, compared to 5 % in Gasse et al. (2019) [12] and 100 % in Gupta et al. (2020) [15]. This method of generating expert samples has been criticized by Sun et al. (2021) [46], among other reasons because strengths of SB cannot be learned through variable selection only. The reader is referred to this article for further reading, in addition to the thesis Ross (2013) [47] for more on expert imitation learning data.

For every node, the SB policy assigns a score to every possible branching variable. The best variable to branch on according to SB is saved explicitly and then branched on. The best candidate variable is used for training, while the SB scores are used for

evaluating the learned policy. The placement of the selected variable compared to the candidate variables will be used to generate an accuracy score, to evaluate whether the algorithm is able to select among the top variables to branch on.

3.1.3 Branching Variable Features

At nodes where the strong branching policy is used, available features for every possible branching variable is saved. The features, as explained in Section 2.3.4, are divided into the groups variable features, constraint features and edge features. These features are presented in Table 3.1. For a further explanation on the relevant features, see Section 2.3.4.

The underlying assumption is that these features are sufficiently correlated with the optimal variable to branch on, as it is given by the Strong Branching algorithm.

3.2 Models

This section presents the ML models used in the experiments. These models are *iterative ablations* of the model presented in Gasse et al. (2019) [12], meaning each model is an increasingly simplified version of the original network.

3.2.1 Original Graph Convolutional Neural Network

First, the nature of the Gasse GCNN has to be formulated. The central component is the graph convolution, which is modelled as two half-convolutions. This means that the convolution is divided into to subsequent passes — one from variable to constraints

Tensor	Feature	Description
V (19)	objective	Objective coefficient, normalized.
	type (4)	Type (binary, integer, impl. integer, continuous) as a one-hot encoding.
	has_lb	Lower bound indicator.
	has_ub	Upper bound indicator
	reduced_cost	Reduced cost, normalized.
	sol_value	Solution value.
	sol_frac	Solution value fractionality.
	sol_is_at_lb	Solution value equals lower bound.
	sol_is_at_ub	Solution value equals upper bound.
	scaled_age	LP age, normalized.
	inc_val	Value in incumbent.
	avg_inc_val	Average value in incumbents.
	basis_status (4)	Simplex basis status (lower, basic, upper, zero) as a one-hot encoding.
C (5)	obj_cos_sim	Cosine similarity with objective.
	bias	Bias value, normalized with constraint coefficients.
	is_tight	Tightness indicator in LP solution.
	dualsol_val	Dual solution value, normalized.
	scaled_age	LP age, normalized with total number of LPs.
E (1)	coef	Constraint coefficient, normalized per constraint.

Table 3.1: Features of each variable in the data, adapted from Gasse et al. (2019) [12].

and one from constraints to variables [12].

The two half-convolutions can be expressed as [12]:

$$\mathbf{c}_i \leftarrow \mathbf{f}_C \left(\mathbf{c}_i \sum_j^{(i,j)\in\mathcal{E}} \mathbf{g}_C(\mathbf{c}_i, \mathbf{v}_j, \mathbf{e}_{i,j}) \right), \qquad \mathbf{v}_j \leftarrow \mathbf{f}_\mathcal{V} \left(\mathbf{c}_j \sum_j^{(i,j)\in\mathcal{E}} \mathbf{g}_\mathcal{V}(\mathbf{c}_i, \mathbf{v}_j, \mathbf{e}_{i,j}) \right) \tag{3.7}$$

for all $i \in C, j \in \mathcal{V}$. $\mathbf{f}_C, \mathbf{f}_\mathcal{V}, \mathbf{g}_C, \mathbf{g}_\mathcal{V}$ are 2-layer perceptrons.

The addition of the graph convolution operator results in the nodes of the bipartite variable-constraint graph containing information about its neighbors

Gasse et al. (2019) [12] also notes that it is common to normalize convolutional layers by the number of neighbors, however, they conclude that the loss of expressiveness by including this normalization is not beneficial. Therefore they include the affine normalization layer called a *prenorm* layer. The layer is applied after the convolution, and is expressed as $\mathbf{x} \leftarrow (\mathbf{x} - \boldsymbol{\beta})/\boldsymbol{\sigma}$.

In the original implementation, the prenorm layer's coefficients $(\boldsymbol{\beta}, \boldsymbol{\sigma})$ are approximated and fixed in a pre-training phase, while the implementation in this thesis uses the new PyTorch library-component `torch.nn.LayerNorm()`, where the coefficients are approximated continuously.

After the normalizations, the features are transformed to *embeddings* of a uniform size of 64 via linear neural network layers.

An illustration of the Gasse GCNN in given in Figure 3.1. n is the number of candidate variables, m is the number of constraints, $d = 19$ is the number of variable features, $e = 1$ is the number of edge features, $c = 5$ is the number of constraint features. The two convolutions are marked and the network layers are represented with a rectangle for each layer. The size of the hidden layers is consistently equal to 64. After

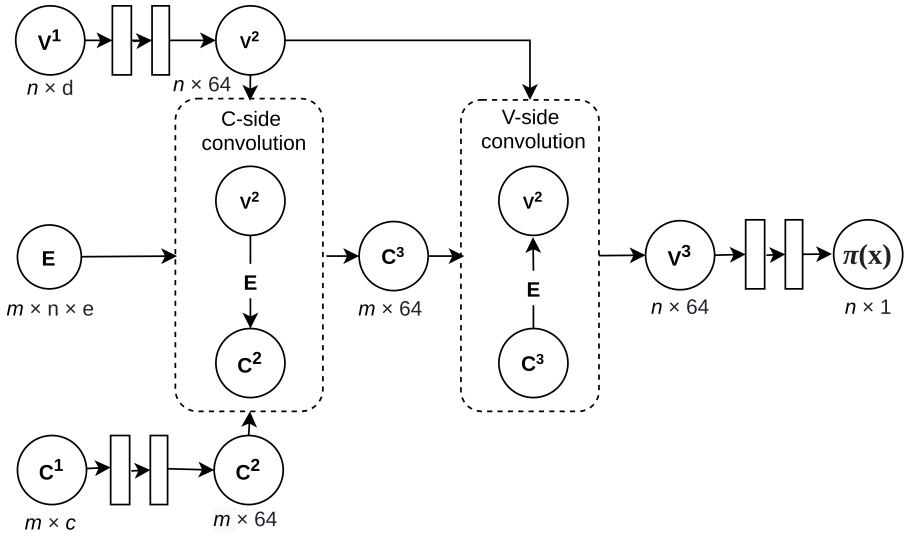

Figure 3.1: The graph convolutional neural network as specified in Gasse et al. (2019) [12].

the convolutions, a 2-layer perceptron transforms from the variable features (with information gained after the convolutions) to the single output. The dimensions for the tensors are marked in the figure.

3.2.2 Network Topologies

As stated, the ML-models used in the experiments are generated by iteratively ablating (removing components) from the original network. This results in models that are GCNNs and models that are pure MLPs, which will be compared for accuracy and efficiency on both GPU and CPU.

The ablated models are chosen based on three basic assumptions that are assumed to

be true based on the vast literature in deep learning[2] [9]. The assumptions are stated as:

(i) It is assumed that increasing the capacity of the network correlates to increasing the number of computations [9].

(ii) It is assumed that the variation in complexity will lead to differences in both accuracy and forward pass computation time [9].

(iii) It is assumed that the addition of the graph convolution will be a significant factor in the accuracy and efficiency of the models, particularly on the CPU, based on the results in the appendix of Gupta et al. (2020) [15].

Considering this, the experiments will be performed with a total of five models, meaning the original model and four models stemming from the iterative ablation of this model. Of these, two models will contain the graph convolution operation, and three will be multi-layer perceptrons that only use the variable data.

Results in the specialization project this thesis is based on showed consistent and significant changes in the accuracy/efficiency trade-off were found with a similar selection of MLP models, although these models relied on the extended variable feature set developed in Khalil et al. (2016) [14].

GNN2, illustrated in Figure 3.2, is identical (except for the minor, commented details), to the original Gasse GCNN.

GNN1, illustrated in Figure 3.2, is the model resulting from the first iteration of the ablation, which has reduced the number of layers in the embeddings and the output module. The graph convolutions are therefore unchanged.

[2]These assumptions are separate from the three research questions presented in Section 1.4, and will not be covered in the same detail.

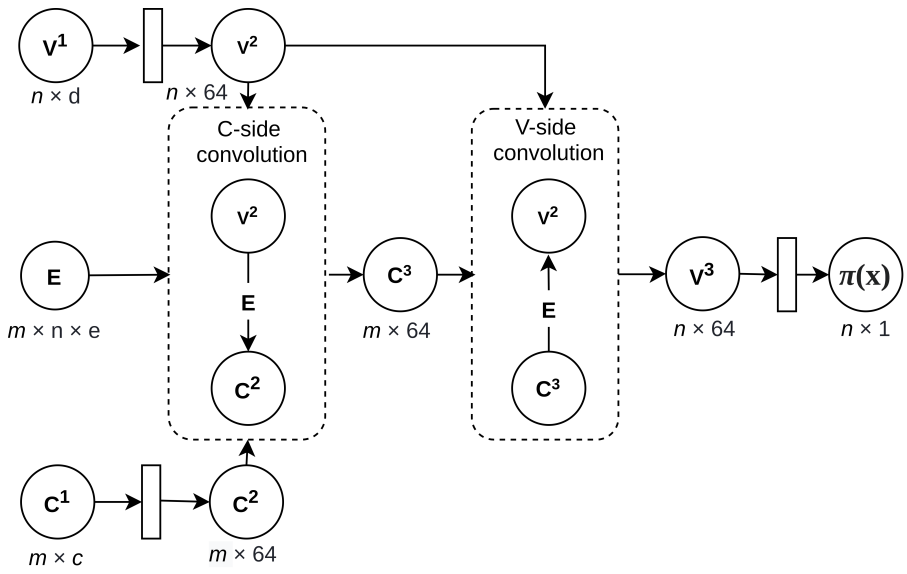

Figure 3.2: Topology of the ablated model GNN1.

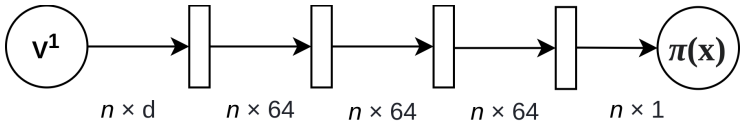

Figure 3.3: Topology of the larger sized MLP3 feed-forward network.

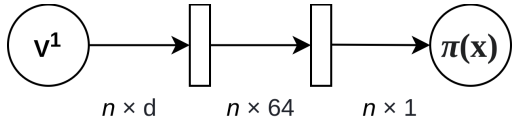

Figure 3.4: Topology of the medium sized MLP2 feed-forward network.

MLP3 is illustrated in Figure 3.3. The second ablation removes the convolutions in their entirety, resulting in an MLP model consisting of the prenorm layer and embedding of the variable features with the output module. This equates to a four-layer MLP

MLP2 is illustrated in Figure 3.5. The third ablation removes layers from the MLP3 model, yielding a network with lower capacity but fewer operations needed for evaluation. The two-layer MLP also results in a minimal network for representing non-linear relations between the features and the scores.

MLP1 is illustrated in Figure 3.4. The fourth ablation removes all hidden layers, resulting in an MLP model consisting of the prenorm layer and embedding of the variable features with the output module. This is equivalent to finding the optimal linear combination of input parameters[3]. A forward pass is equivalent to a single matrix multiplication.

[3]To be exact, MLP1 is not a Multi-Layer Perceptron, however, the name is chosen to be consistent with the other models. The central difference between the models is whether they contain graph convolutions or not (given by the acronym) and the number of parameters (given by the number).

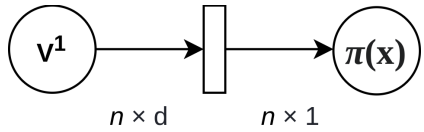

Figure 3.5: Topology of the smaller sized MLP1 feed-forward network.

3.2.3 Network hyperparameters

All hyperparameters except for the amount and size of the hidden layers is consistent for all models, in order to make a fairer judgment of the accuracy/efficiency trade-off. The activation function is the *recti-linear unit* (ReLU), expressed as $y = \max\{0, x\}$. This non-linear activation function is the least computationally expensive of the common activation functions. This is consistent with Gasse et al. (2019) [12] and Gupta et al. (2020) [15].

3.3 Training Protocol

In this section, the training protocol is presented, which explains how the parameters of the models are learned.

3.3.1 Loss function

The problem is expressed as a two-class classification problem, where the classes are *optimal variable* and *sub-optimal variable*, respectively. The first class contains only the top variable from the strong branching evaluation. The loss function in binary classification is typically chosen as the binary cross-entropy loss function [9],

calculated as:

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{|\mathcal{D}|} \sum_{(\boldsymbol{x},y)\in\mathcal{D}} \left(y_i \cdot \log(\hat{y}_i) + (1-y_i) \cdot \log(1-\hat{y}_i)\right), \tag{3.8}$$

where $\mathcal{D}$ is the data set, $\hat{y}_i$ is the predicted class based on the sample features x, and y_i is the classification by the Strong Branching expert. This means that for each branching decision, n candidate variables will be given a score analogous to the probability of that variable being the best variable according to the strong branching expert policy.

3.3.2 Training method

The parameters of the MLP are trained via mini-batch gradient descent [9], and is performed using the Adam optimizer [48]. The Adam optimizer performs gradient descent with adaptive moments in both the first and second order [48]. The batch size (number of problems processed at once) is set to 32. The learning rate is reduced upon plateaus[4] in validation loss by multiplying with 0.2. This is a well-known method to ensure convergence to a reasonably good local minimum [9].

Further, the training has implemented an *early stopping scheme*, to ensure that the models have converged, regardless of model complexity. If the validation loss does not decrease 16 epochs in a row, the model that resulted in the lowest validation loss is saved and the training process is stopped. This is because it is considered unlikely for the optimizer to improve the model after that amount of epochs without improvement.

All of these methods are implemented in order to give the fairest evaluation of the models, as individual hyperparameter optimization for each model would be too time-consuming to be performed.

[4] *Plateaus* are multiple epochs without reduction in the target loss.

3.3.3 Computer Hardware

Training of the models is performed on an NVIDIA RTX2070 SUPER GPU (8GB VRAM). The B&B solution efficiency evaluations are performed with an Intel i5-2500K CPU (4 cores, 6 threads) running at 3.31 GHz. This is less expensive hardware compared to Gasse et al. (2019) [12] and Gupta et al. (2020) [15], particularly the CPU. Variation in processing power is assumed to have an insignificant effect on the relative solution times of the methods, however, CPUs with internal acceleration methods for matrix operations might give varying results [49]. It is assumed that the results on the chosen hardware yield a more pessimistic performance overall for the ML based methods, and will therefore not be given too much attention in the thesis. Analysis of these differences in hardware and processing performance for both the classical and learned methods on both CPU and GPU is of interest, but far beyond the scope of this thesis.

3.4 Comparison Method

The method in which the learned models are compared to each other and the classical branching policies is presented in this section.

3.4.1 Classical Branching Policies

The accuracy and efficiency of the algorithm with the learned branching policy are evaluated against three branching policies native to the SCIP optimization suite, and explained in more detail in Section 2.2.3.

First is the full strong branching (FSB) policy [29], the slow but accurate expert policy that performs strong branching at every node for every variable. This algorithm is not

commonly used because of the computationally heavy variable decision process [28].

Pseudo-cost branching (PB) is the fast but inaccurate branching policy [50] that chooses the variable that maximizes the lower bound improvement according to the results of previous branching decisions [51].

Reliability pseudo-cost branching (RPB) [28] combines FSB and PB by performing SB on variables with low confidence in the pseudo-costs [50]. RPB is the default branching policy in the SCIP B&B solver.

The classical variable selection policies will be implemented by configuring the priority of the branching policies in SCIP.

3.4.2 Benchmarking

The learned branching policies are evaluated for both *accuracy* and *efficiency*.

The accuracy of the policies is evaluated against the expert decisions of the full strong branching algorithm. This is done by evaluating the average *top-k accuracy* over an unseen test set. Top-k accuracy measures whether the chosen branching variable was within the top k choices of the strong branching algorithm, based on the score given to each candidate variable by the algorithm. Only the learned models are measured in this regard.

The efficiency of the policies is evaluated by exchanging the branching policy of the optimization solver and testing on the data set, as described in Section 3.1. The counting of nodes and time is done with the built in capabilities of Ecole.

The number of nodes processed by the algorithm will also be reported. Fewer nodes

are not necessarily indicative of a better algorithm, however, this will be an important comparison, as it gives a proxy measurement of the accuracy/efficiency trade-off for the algorithms.

3.5 Software

A number of comprehensive software libraries are required to perform the experiments in this thesis, this section presents the three most essential.

3.5.1 SCIP

Solving Constraint Integer Programs (SCIP) [7], is the CO solver that lays the foundation for the data generation and model evaluation for the methods. It is considered the fastest non-commercial solver for MIP and MINLP problems [52]. It is written in C with wrappers for C++. The software is maintained by the *Zuse Institute Berlin* (ZIB) group.

The *SCIP Optimization Suite* contains, as of version 7, the simplex-based linear programming solver SoPlex [53], the automated decomposition solver GCG [54], the parallelization framework UG [55] and the MILP pre solving library PaPILO [52].

In the experiments in this thesis, the LP solver and node selection algorithms are the most relevant, as most of the more complex capabilities of SCIP are deactivated in order to make the results as fair and reproducible as possible.

3.5.2 PyTorch

PyTorch is the Python library that the machine learning models are written in. The library provides, most notably, dynamic eager execution, automatic differentiation, and GPU acceleration [56].

Necessary for constructing and training the graph convolutional neural networks, the *PyTorch Geometric* library was used [57]. The library facilitates deep learning on data with irregular structures, such as graphs. This includes methods for sparse data GPU acceleration and efficient mini-batch handling [57].

3.5.3 Ecole

To facilitate and standardize the development of ML-based improvements of CO algorithms, *Extensible Combinatorial Optimization Learning Environments* (Ecole) was developed [18], with the first version published in the last quarter of 2020. Ecole aims to mimic the *openAI Gym* framework [58], a popular framework for developing RL models. It is a python library written largely in efficient C++. The library is based on the open-source solver SCIP [7].

It also provides problem *generators* for the four problem classes presented in this thesis. As of version 0.5, these generators are not completely correct, and in the problem generators from Gupta et al. (2020) [15] were therefore used. These problems appear to be resolved in version 0.6.

The theoretical basis for Ecole is the PO-MDP formulation for Ecole, given in Section 2.3.

3.5.4 Development Environment

The project is run on only open-source software, in order to support and further develop the commonly available tools for machine learning and mathematical programming. For reproducibility, the libraries and their respective versions are listed here, as well as in the publicly available code repository.

The project code is written in `python 3.8.5` [59], and uses `pytorch 1.7.1` [56] for the machine learning models.

For GPU accelerated training, `CUDA 10.1` and `cudatoolkit 11.2` [60] is used.

The optimization problems are solved using the `SCIP 7.0.2` Optimization Suite [52], using the `SoPlex 4.0.1` [53] linear programming solver to solve the relaxed problems. The Python interface `PySCIPOpt 3.0.4` [61] is used in and alongside Ecole to facilitate communication with SCIP.

The computer is running the `Ubuntu 20.04.2` Linux distribution.

3.5.5 Code Repository

As stated, the code for reproducing the experiments is given in a publicly available code repository, found at `https://github.com/Sandbergo/branch2learn`. It is intended to be similar to the source code for Gasse et al. (2019) [12], and to be extensible in order to facilitate further work.

A presentation of the code is given in `README.md`, with installation instructions in `INSTALLATION.md`. Installation requires some manual installation but is mostly done using `Anaconda`. The eponymous subdirectory `branch2learn` contains python

scripts for running the respective processes of training and evaluating the models: `00_generate_instances.py`, `01_generate_data.py`, `02_train.py`, `03_test.py`, `04_evaluate.py`, and `05_evaluate_standard.py`. In addition, the directories for the models and utility functions are found there.

The directory `scripts` contains bash-scripts for reproducing all experiments used in this thesis.

4

Results

In this chapter, the results of the experiments are reported, as described in Chapter 3. Some sentences and formulations are adapted from the project report *Multi-Layer Perceptrons for Branching in Mixed-Integer Linear Programming* (2020), as this thesis shares methods with the report.

4.1 Data Set

Problem instances are generated as described in Section 3.1.1. In total, 2000 training instances, 500 validation instances, and 500 test instances are created. These are of size 100×500 for the combinatorial auction problems and 100×500 for the set covering problems, as stated in Section 3.1. This information is shown in Table 4.1.

The generated problem instances are then solved as specified in Section 3.1.2. This

results in a total of 50000 training samples, 10000 validation samples, and 10000 test samples for the combinatorial auctions problems, and the same for the set covering problems. This is also shown in Table 4.1.

The division into train, validation and test sets is done as is standard in ML development [9]. The train set will be used to optimize the model parameters, the validation set will be used to monitor the model's ability to generalize to unseen samples, and the test set will be used to evaluate the model.

A notable difference between the problem classes is the number of constraints. The combinatorial auction problems average about 200 constraints, while the set covering problems have 500 constraints.

250 problem instances are generated for evaluating the efficiency of the models.

		Train	Validate	Test
Auctions	Dimensions	100×500	100×500	100×500
	Instances	2000	500	250
	Samples	50000	10000	10000
Setcover	Dimensions	500×1000	500×1000	500×1000
	Instances	2000	500	500
	Samples	50000	10000	50000

Table 4.1: Data set, dimensions and number of problem instances for each problem class.

4.2 Training

The training was performed as described in Section 3.3. The training graph for MLP2 on the combinatorial auction data set is shown in Figure 4.1, and the training graph for GNN1 on the set covering data set is shown in Figure 4.2. The training and validation loss quickly converges before flattening out. There is no discernible difference between training and validation loss. Loss graphs for the other models are not presented, as they are very similar, and are therefore considered to be of little interest.

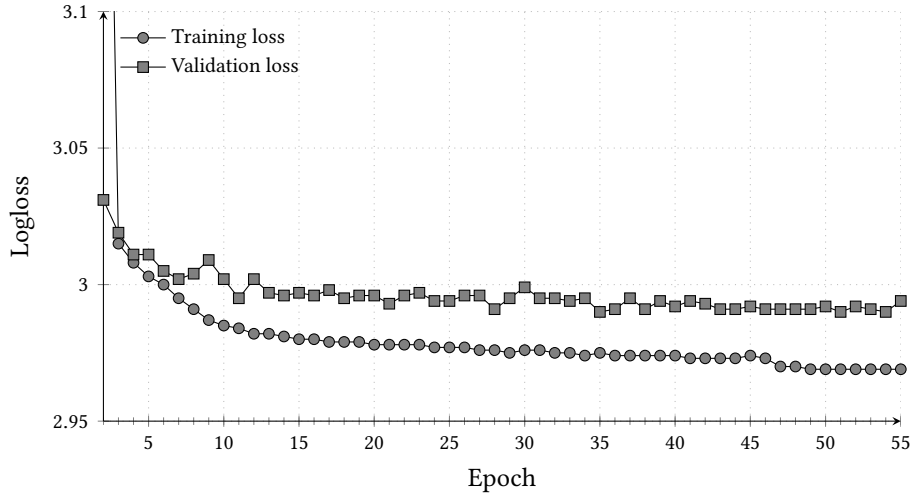

Figure 4.1: Training graph for MLP2 on the combinatorial auctions data set.

4.3 Accuracy

The accuracy of the models is measured by the top-k accuracy, as specified in Section 3.4.2. The top-k accuracy scores for k equal to 1, 5, and 10 for the models along with the benchmark accuracy for random variable selection are shown in Table 4.2.

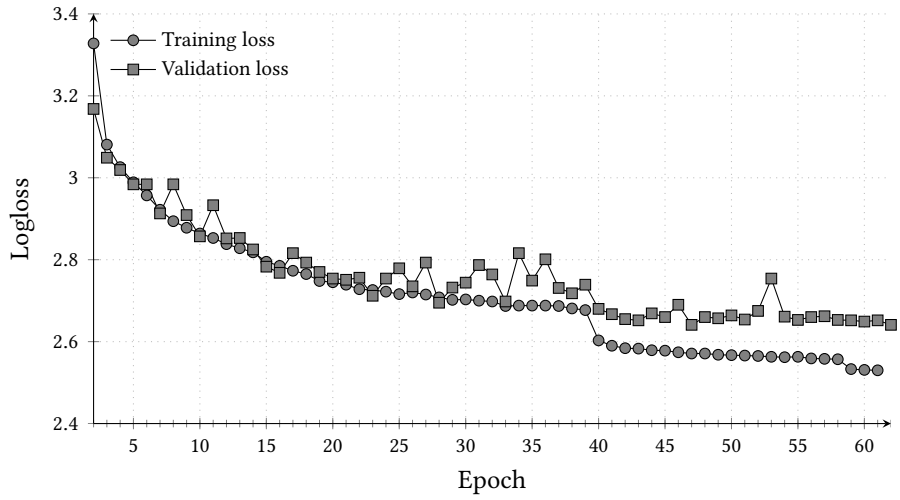

Figure 4.2: Training graph for GNN1 on the set covering data set.

Top-k accuracy for this application is defined as the percentage of selected branching variables within the top k variables as determined by the strong branching evaluation.

Comparison with the random variable selection policy shows considerable improvement in favor of the trained models. Accuracy decreases with the ablations, although there is little difference between GNN2 and GNN1, and MLP3 and MLP2, respectively. MLP1 performs considerably poorer than the other models, indicating that a non-linear relationship between the variable features and the variable score is beneficial. There is also a noteworthy decrease in accuracy after removing the graph convolutions, indicating that the module aids in prediction.

A comparison of the differences between models of the two problem classes shows that the decrease in accuracy from GNN1 to MLP3 is around three times larger for the set covering problems. This indicates that the addition of different problem classes in the evaluation of branching policies is necessary.

	Combinatorial Auction			Set Covering		
Model	acc@1	acc@5	acc@10	acc@1	acc@5	acc@10
random	15.8 %	34.5 %	44.6 %	14.6 %	32.6 %	41.2 %
MLP1	46.2 %	75.1 %	87.0 %	40.2 %	63.6 %	73.3 %
MLP2	50.1 %	78.9 %	89.9 %	49.1 %	75.5 %	88.1 %
MLP3	50.2 %	79.3 %	90.1 %	49.4 %	75.7 %	88.3 %
GNN1	53.0 %	87.5 %	95.9 %	57.9 %	89.4 %	97.0 %
GNN2	53.0 %	87.1 %	96.0 %	58.8 %	89.9 %	97.2 %

Table 4.2: Top-k accuracy scores for combinatorial auctions and set covering on the test set, representing the priority of the selected variable by the strong branching score evaluation.

Note that the GNN2 model should have similar accuracy to the corresponding model in Gasse et al. (2019) [12], however, the models have around 7% lower accuracy in this implementation. The cause for this is unknown, and the analysis of the models will not be affected by this.

4.4 Efficiency

Eight branching policies were compared on the problem data set. The branching policies are full strong branching, pseudo-cost branching, reliability pseudo-cost branching, and the five learned models. The results for the combinatorial auctions problems are shown in Table 4.3 and for the set covering problems in Table 4.4. *Time* is the mean solution time, *nodes* is the mean number of nodes in the solution graphs (calculated in accordance with the findings of Gamrath et al. (2018) [50]), *time/node* is the average time per node, calculated by dividing the two means, and *parameters* is the number of

(trainable) parameters of the model. The solution times and number of nodes are also provided with their respective standard deviations. For evaluating the time per node, it is worth noting that the average number of candidate variables may vary between methods. In other works relating to node and time measurements when comparing branching policies, a variant of the shifted geometric mean is used. This is not done in this thesis, see Appendix A for a short discussion on this. The branching policy with the shortest mean solution time is marked bold for the GPU and CPU times.

The solution time and number of nodes for the classical branching policies is as expected: FSB has the lowest number of nodes but the processing time per node is too high to be competitive, PB has a competitive solution time as the branching variable selection is performed very quickly, while RPB is a midpoint between the two algorithms.

For the learned models, the results show that the ablation from GNN2 to GNN1 does not particularly reduce the time per node, neither does the removal of hidden layers between the models MLP3 and MLP2. Time per node is considerably reduced after the graph convolutions are removed and reduced again when the model's hidden layers are removed. This is consistent over both the GPU and CPU variants.

Figure 4.3 and Figure 4.4 shows the results for the results from Table 4.3 and Table 4.4, respectively, including 95% confidence intervals calculated under the assumption that the solution times are normally distributed. See appendix A for a brief discussion on this assumption.

For the set covering problems, the learned models are less efficient than PB and RPB, which is inconsistent with the results from Gasse et al. (2020) [12]. This will not be given too much attention, as this is irrelevant to the comparative analysis of the ablations. Note also that the time per node is not equal between the two problem classes — the average number of candidate variables per node will vary between problems.

Model	Time [s]	Nodes	Time/node [ms]	parameters
FSB	4.08 ± 2.11	8 ± 8	462.6	—
PC	1.96 ± 0.84	355 ± 358	5.5	—
RPC	2.73 ± 1.18	16 ± 30	171.1	—
MLP1_g	1.58 ± 0.46	146 ± 149	10.8	19
MLP2_g	**1.56 ± 0.43**	119 ± 119	13.1	1344
MLP3_g	1.57 ± 0.44	118 ± 115	13.2	9702
GNN1_g	1.58 ± 0.42	92 ± 78	17.2	52083
GNN2_g	1.59 ± 0.43	91 ± 80	17.5	64562
MLP1_c	3.14 ± 1.53	147 ± 148	21.2	19
MLP2_c	3.07 ± 1.35	120 ± 118	25.4	1344
MLP3_c	3.08 ± 1.32	118 ± 112	26.0	9702
GNN1_c	5.34 ± 3.42	97 ± 86	54.8	52083
GNN2_c	5.61 ± 3.65	94 ± 84	59.3	64562

Table 4.3: Combinatorial auction solving time for classical and learned methods. Subscripts denote whether the model is run on the GPU or CPU.

Model	Time [s]	Nodes	Time/node [ms]	parameters
FSB	26.23 ± 33.79	31 ± 45	845.6	—
PC	7.78 ± 9.50	749 ± 1634	10.4	—
RPC	10.68 ± 8.01	295 ± 895	36.2	—
$MLP1_g$	16.28 ± 24.29	621 ± 1097	26.2	19
$MLP2_g$	15.75 ± 21.89	531 ± 917	29.6	1344
$MLP3_g$	15.83 ± 22.25	523 ± 899	30.3	9702
$GNN1_g$	11.95 ± 12.94	268 ± 384	44.6	52083
$GNN2_g$	**11.71 ± 12.92**	266 ± 382	44.0	64562
$MLP1_c$	17.57 ± 20.96	609 ± 1019	28.8	19
$MLP2_c$	17.39 ± 21.31	535 ± 929	32.5	1344
$MLP3_c$	17.38 ± 21.35	517 ± 889	33.6	9702
$GNN1_c$	60.06 ± 81.62	265 ± 382	226.1	52083
$GNN2_c$	63.11 ± 87.58	268 ± 392	235.1	64562

Table 4.4: Set covering solving time for classical and learned methods. Subscripts denote whether the model is run on the GPU or CPU.

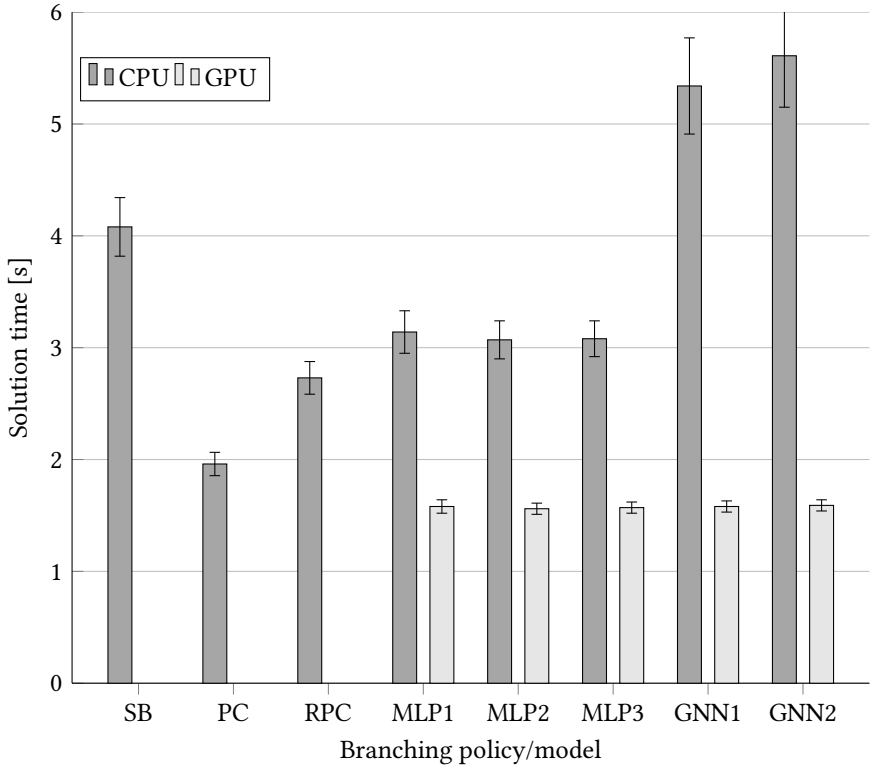

Figure 4.3: Combinatorial Auction test problem mean solution time when run on the GPU and CPU.

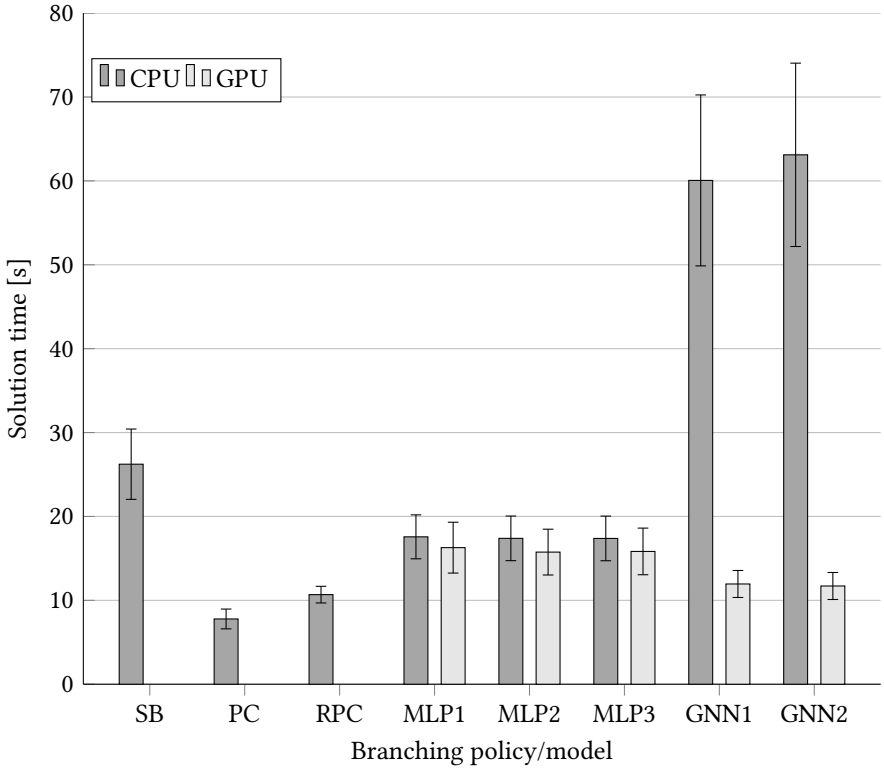

Figure 4.4: Set covering test problem mean solution time when run on the GPU and CPU.

5

Discussion

This chapter discusses the results obtained in Chapter 4, critiques the choice of experiments, presents ideas about further work in the field, and answers the research questions. Some sentences and formulations are adapted from the project report *Multi-Layer Perceptrons for Branching in Mixed-Integer Linear Programming* (2020), as this thesis shares methods with the report.

5.1 Data Set

The problem instances were only created from the combinatorial auction and set covering classes. As noted in Chapter 4, there are differences between performance on the two problem classes, indicating that problem classes are important for fair judgment on the branching policies. Especially as the largest loss of accuracy when removing the constraint features was found for the problem set with a larger number

of constraints. This need for various problem classes is consistent with previous work in evaluating B&B solvers [6]. Apart from this, a standardization regarding problem classes and sizes are necessary for fair evaluation of the B&B improvements.

Generating samples (branching problem samples with strong branching scores) was performed following Gasse et al. (2019) [12], using the data collection model presented in Ecole. The number of samples was reduced from 100000 to 50000.

On a practical note, several of the provided problem generator classes in Ecole were flawed during the creation of the experiments, which delayed the work and resulted in only two problem classes being present in the experiments. These problems were generated with the old generators from Gasse et al. (2019) [12], and solved (generating branching problem samples) using the Ecole framework. These issues have since been resolved. Other researchers are advised to implement a parallelization of the sample generation found in the source code of this thesis.

Increasing the problem size was chosen as the method for estimating the out-of-distribution efficiency of the MLP-aided B&B, as was done by Gasse et al. (2020) [12]. Other approaches to the problem generation and generalization efficiency estimation might look into problem distributions with a temporal component, i.e. where the test samples are from a provably different distribution, without dramatically increasing the difficulty. This might be more representative of problems that are time-constrained, and will give a different estimate of the out-of-distribution generalization error. In the articles of Gasse et al. (2019) [12] and Gupta et al. (2020) [15], generalization was measured by evaluating on problems of larger dimensions than was trained on. This was not done in this thesis, as the results were evaluated to be of little interest.

5.2 Training

The training is discussed based on general insights as presented in Goodfellow et al. (2016) [9], as it is difficult to come to decisive conclusions within the field of deep learning. On a general note, the graphs presented in Section 4.2 show a large reduction in the evaluation loss after the first epoch, indicating that the model optimization process quickly results in a reasonable model. This is consistent with a comparison of the random variable selection policy, as the untrained model is practically a random choice policy.

A sign of *overfitting* the model to the training data would be a validation loss that increases while the training loss decreases. This is not the case, meaning that there is no reason to believe that the model merely remembers the training data instead of learning an approximation of the relationship between the node features and the variable SB score.

All models stopped training within 100 epochs due to the early stopping scheme explained in Section 3.3, meaning that it is reasonable to assume that maintaining an identical optimizer and learning rate between the models was not detrimental to the accuracy of the trained models.

5.3 Accuracy

All models show a considerable improvement over the random choice heuristic, further verifying that the model training process was productive.

In addition, model accuracy strictly decreases as the model is ablated, indicating that the model is not over-parameterized to the point of being unable to reach a good

function approximation with the selected training method.

A clear result is the near-constant accuracy when ablating from GNN2 to GNN1 and MLP3 to MLP2. In these ablations, only hidden layers are removed. This can indicate an over-parameterization in terms of the number of hidden layers.

When model capacity decreases (via ablations) without a decrease in accuracy, two causes must be balanced when attempting to gain insight from the results:

(i) The training protocol is not able to discover the complex relations between the input features and the output scores of the Strong Branching algorithm.

(ii) The theoretical optimal branching function is not strongly dependent on complex, non-linear relations between input and output.

The latter argument seems more likely based on the results in this project and previous experiments [15] [12], however there are no convergence guarantees for the models, and option one can therefore not be ruled out.

A large loss of accuracy occurs when removing the graph convolutions (ablating from GNN1 to MLP3), meaning the graph convolutions are productive in aiding the scoring process. This is consistent for both model classes, though the loss of accuracy is double for the set covering problems. This might be explained through the nature of the problems, as the set covering problems has around 150 % more constraints than the combinatorial auctions problems. From this, it is possible to hypothesize that the value of the convolutions are problem-dependent. This is viewed as a problematic conclusion, as the B&B variable selection algorithm should ideally be a universally well-performing model, irrelevant of the particularities of the model. It must also be noted that further experiments with many more problem classes should be conducted to reach a definite conclusion in this regard.

Further, the ablation from MLP2 to MLP1, in which the non-linear capabilities of the model are removed, a large drop off in accuracy is found. This indicates that a non-linear transformation from input to output is useful, as a single-layer neural network is assumed to quickly converge to the optimal model [9]. This is true for the variable feature set, though a more expressive variable feature set such as the one found in Khalil et al. (2016) [14] might give better performance.

The lower accuracy scores compared to the results in Gasse et al. (2019) [12] can come from several sources. For the purposes of the ablation study, lower accuracy should not impact the results of comparison between the ablated models, however, it is not possible to rule out that the training process is sub-par compared to the original, which can mean that the models are under-trained compared to the models presented in Gasse et al. (2019) [12]. The reader is advised to keep this in mind.

Based on this, the following insights regarding the accuracies of the models seem probable based on the experiments:

(i) The graph convolutional operator is beneficial for the accuracy of the model, with the effect depending on the problem class.

(ii) The Gasse GCNN might be overparameterized, as removal of hidden layers did not show a considerable reduction in accuracy.

(iii) The change in accuracy after ablations varies in magnitude between the problem classes.

5.4 Efficiency

The central problem in *learning-to-branch* is the trade-off between computational efficiency and the number of nodes processed, where the goal is to solve problems in

the least amount of time. The results in Table 4.3 and Table 4.4 show that the time per branching decision varies greatly between the models. The time per node decreases strictly as the model is ablated, which is expected. This verifies the assumption that a reduction in model size leads to a reduction in computation time. The combination of the reduced accuracy and reduced computation time following the ablations allows for studying the aforementioned accuracy/efficiency trade-off.

Comparing the models with similar accuracy (GNN2 and GNN1, MLP3 and MLP2) shows that the decrease in the number of hidden layers does not particularly impact the time spent per node. The larger changes in time spent per node are found in the same ablation iterations as the reduction in model accuracy: after the removal of the graph convolutions and after all hidden layers are removed. This trend is clear for both problem classes and is consistent when run on both the GPU and CPU. This is particularly noticeable in the ablation MLP3 to MLP2, where the number of parameters is reduced by over 85 %, though the time per node is only reduced by less than 3 % for both problems.

A comparison between the classical branching policies (FSB, PB, RPB) shows that the methods are generally improved upon by the ML-models for the combinatorial auctions problems, but not for the set covering problems. This is not the case for the original Gasse GCNN, which can come from the reduced model accuracy in this implementation as well as lower capacity hardware. This will not be commented on further, as the comparisons of the models and hardware are the focus of the experiments. It is still not ideal that the models have seemingly not achieved the same improvements as the original implementation.

On the CPU, the results vary between the problem classes: Only the MLPs are near-competitive for the combinatorial auction problems, while the MLPs show little reduction in efficiency on the set covering problems. Either way, for both problem classes the results show that models with graph convolutions are far too inefficient when run on the CPU.

Lastly, inconsistencies between the differences in solution time when run on the GPU and CPU must be noted: For the set covering problem, the CPU restriction results in a considerably larger increase in solution time than for the combinatorial auction problems. This is assumed to be due to the larger number of constraints resulting in these operations being infeasible on the reduced capacity for large tensors on the CPU. Further, MLP3 and MLP2 have a larger decrease in performance when run on the CPU for the combinatorial auctions problems. This is surprising and should be examined by further experiments on more problem classes.

Based on this, the following insights regarding the efficiency of the models seem probable based on the experiments:

(i) All ablations show competitive efficiency when run on the GPU.

(ii) Removal of hidden layers does not particularly impact the computation time per node.

(iii) Graph convolutional models are far too inefficient to be used on a CPU only setup.

5.5 Critique of Experiments

The conducted experiments have given useful insights into the topic at hand, and a critique of those experiments is due. Three points of criticism are presented in this section:

Firstly, the variable performance when comparing the different models on the GPU and CPU between the two problem classes means that a larger number of classes should be evaluated. Conclusions based on the problem formulation differences are not reliable with so few problem classes. Due to the aforementioned issues regarding the problem

generators and the time-consuming process of training and evaluating all models for each problem.

Secondly, the accuracy of GNN2 is lower than the original Gasse GCNN implementation [12]. This can possibly mean that the model does not converge as successfully as the original implementation, or that other subtle differences result in a worse performance. This issue was not expected.

Thirdly, the implementation of the original GCNN in a new and different framework might reduce the validity of the ablation study, as a number of changes from the original have been made. This ties into the issues with the reduced performance, as changes were made in order to follow the implementations found in the Ecole source code. This was clear beforehand, however, the value of making the implementation in the Ecole was evaluated to outweigh this drawback.

5.6 Further Work

Further work in the field of ML based variable selection can gain insights from the results in this thesis. The experiments show that the viability for GCNN models are very limited when run on the CPU, showing that the model is not productive with this hardware restriction. In addition, results seem to indicate that both the accuracy and efficiency of the model on different hardware are reliant on the particularities of the problem formulation. This sets a precedent for a thorough evaluation of the models that are presented in future work within the field, as the applicability of the results is dependent on many factors. For applications of B&B run on specialized hardware this is less of an issue, but this limits the general usefulness of the models.

As ML approaches are highly dependent on the hardware, model choices and performance evaluation should be tied to the practical application of the algorithms. This is

the trend in Gupta et al. (2020) [15], and for applications of B&B on general, inexpensive hardware, this restriction is necessary. For this case, making promising features, known as observation functions in the PO-MDP literature, can be a method for improving the accuracy of the ML models without using the spatially and computationally expensive graph convolution model.

With this caveat noted, there are still many opportunities for ML aided B&B, especially as the Ecole framework is expanded upon. This includes both primal and dual heuristics of the solution algorithm. A recurring approach mentioned in the context of improving B&B is the reinforcement learning approach. This appears to be under development, with results in review by Etheve et al. (2020) [21]. On a larger scale, the work on learning-to-branch fits in the context of learned algorithms replacing expert-created algorithms. This has the potential to eventually remove the human component in algorithm construction. For now, closer attention to the features and correlations of the variable scoring might be the more fruitful endeavor, as the results in this project show that the deep learning approach does not yield significantly better results. Some results in this regard are given in appendix B.

The problem classes are a reoccurring point of interest in the results of the experiments, indicating that a varied selection of problem classes is necessary for a fair evaluation of the models' ability to be efficient in any application. Four problem generators currently exist in the Ecole framework, with the assumed benefit of more problem classes. Further, the most recent novel model presented by Zarpellon et al. (2020) [62] states the importance of trained branching models to generalize to unseen problem classes. The work in this field remains on artificial problems, however, there would be great interest in a practical implementation on real-world optimization problems, e.g. an optimal traffic routing algorithm running on an embedded system.

In addition, further work in the field should also strive to compare results with the top commercial solvers IBM CPLEX and Gurobi [6]. The automatic tuning of the highly parameterized commercial optimization solvers has also been shown to yield

significant improvements in solution time [63], and is therefore advised to include in further and more comprehensive comparisons.

5.7 Research Questions

In this section, the answers to the research questions from Section 1.4 are discussed based on the results.

The first question was stated as:

(i) *What is the impact of iterative ablations on the accuracy of the Gasse GCNN?*

The iterative ablations showed a consistent decrease in accuracy for both problem classes. The notable decreases in accuracy occurred after removing the graph convolutional modules (resulting in a pure multi-layer perceptron) and after all of the hidden layers of the model were removed, resulting in a linear model. A larger loss of accuracy was found for the set covering problems than the combinatorial auction problems after the removal of the graph convolutions. This is assumed to be because of the higher number of constraints in the set covering problems compared to the combinatorial auction problems.

The second question was formulated:

(ii) *What is the impact of iterative ablations on the efficiency of the Gasse GCNN when run on the CPU and GPU as a part of the B&B algorithm*?

All ablations resulted in decreased computation time per variable branching decision. All models were competitive with the classical branching policies (full strong branching,

pseudo-cost branching, reliability pseudo-cost branching) when run on the GPU. On the CPU, there was a significant decrease in performance for the models containing graph convolutions. This performance reduction was greater for the set covering problems, indicating that the number of constraints might exacerbate problems with running graph convolutional models on restricted hardware. Reducing the number of hidden layers did not impact the computation time per node in particular, and is therefore not considered a viable alternative for reducing the computation time.

Lastly, research question three:

(iii) *What are the most promising research opportunities for learning in Branch and Bound?*

This was covered in more detail in Section 5.6. In short, future work should include testing the models on varied problem classes, as well as considering what hardware the enhanced B&B algorithm is approved for. With this in mind, an increased focus on the observation of the B&B algorithm as well as analysis of the score computation process is important. The clearest opportunities lie in using reinforcement learning to further improve the models learned by imitation learning, in order to free the heuristics of the limits of the strong branching imitation. On the practical side, Ecole has proved to be a useful framework, and future researchers are advised to consider using it.

6

Conclusion

In this thesis, the graph convolutional neural network presented in Gasse et al. (2019) [12] was iteratively ablated. The ablations resulted in five models, among which two were graph convolutional neural networks and three were pure multi-layer perceptrons. The GCNN models used the bipartite graph nature of the constraint-variable relationship, while the MLPs only used the features of the candidate variables. The models were trained, tested, and evaluated on generated MILP problems from the problem classes combinatorial auctions and set covering. All resulting models were tested for accuracy on predicting the optimal branching variable according to the strong branching algorithm. The efficiency of the ML-enhanced solvers were evaluated by running the models on both a GPU and CPU. These experiments were chosen in order to gain insight into the model and help future researchers make informed choices on ML model selection.

The experiments were implemented in the new framework *Ecole* [18]. The framework provides an interface for the B&B solver SCIP, inspired by *OpenAI Gym* [58]. This

thesis is the first article to use Ecole except for the introductory papers by Provoust et al. (2020) [18] and Cappart et al. (2021) [20]. The framework was evaluated to be useful, especially as it is improved upon in the future.

The accuracy of the models consistently decreased as layers were removed from the original model. There was a significant loss of accuracy after removing the graph convolutional component as well as after removing all hidden layers of the model. Both problem sets showed the same tendency, though the degradation of accuracy was more dramatic for the set covering problems, where there is a larger number of constraints.

The computation time per variable decision also decreased with the ablations. Significant reductions in time per node were found after removing the graph convolutions and after removing all hidden layers. When the SCIP solver was run on test problems by performing the variable selection with the learned models, all models showed competitive efficiency with a selection of classical branching algorithms. When the models were run on the CPU, the more complex models suffered a large loss of efficiency. This particularly affected the models containing graph convolutions. Results were also indicative of the problem formulation being relevant for the viability of running GCNN models on the CPU.

The main implications of the results are the importance of considering the hardware the ML enhanced solver will be deployed on, as well as the problem types the models are tested on. The relation between the running times of the different models on both hardware implied non-trivial relations, which urges caution for future attempts at developing ML models for this purpose. The results in this thesis as well as the project *Multi-Layer Perceptrons for Branching in Mixed-Integer Linear Programming* (2020) and the article by Gupta et al. (2020) [15] stipulate a shift toward less computationally complex models with richer input features (observation functions). These models will be more universally applicable and predictable on the various hardware that will run B&B algorithms. In addition, the most recent attempt at learning to branch by

Zarpellon et al. (2020) [62] is consistent with this observation by training models that generalize across problem classes.

Future work should take into account the implications of this thesis in terms of both hardware and problem class. The analysis of how problem formulations can be detrimental to model performance is also highly relevant and should be taken into consideration when presenting data-driven methods with the implication of universally useful. Following the trend of the last few years, models pre-trained with imitation learning and improved with reinforcement learning can provide further advances in the field of ML enhanced B&B. These models will yield insights into the greater topic of machine-created algorithms, and may revolutionize how the hardest computational problems are solved in the future.

Appendix A

Time and Node Distributions

The distribution of solution times and number of nodes is central to the evaluation of the variable selection algorithms. In this thesis, the mean and standard deviation are calculated under the assumption of both of these variables having a distribution that can be approximated as a Gaussian (normal) distribution. This is in contrast with the main sources of this thesis (Gasse et al. (2019) [12], Gupta et al. (2020) [15]), which use shifted geometric means. The choice of distribution in this thesis is based on uncertainty in the distribution parameters and configurations in the previous works. An example of a time and node distribution with a superimposed normal distribution approximation is shown in Figure A.1 and Figure A.2. Some discrepancy is found for the time distribution, and more considerably for the number of nodes. The discrepancy is considered of little importance in the comparison of the models, particularly as the node number comparison is devoted little attention in this thesis. Further researchers are encourage to standardize the statistical side of branching comparison in larger detail.

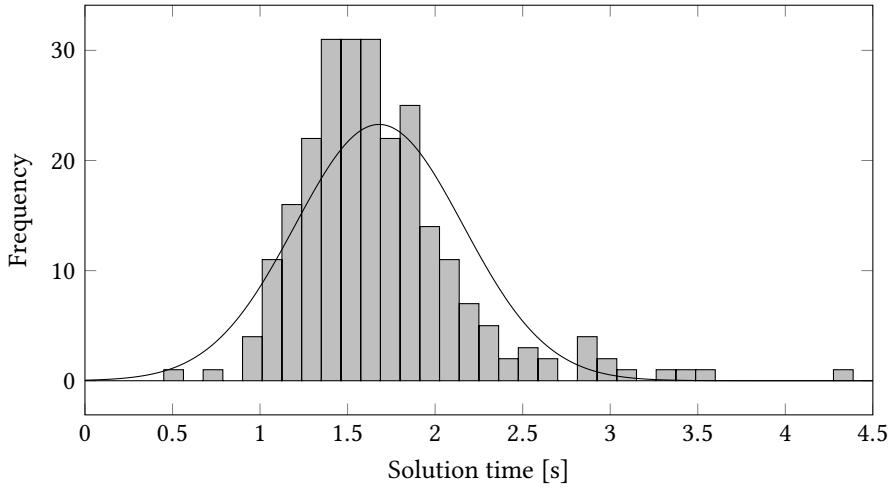

Figure A.1: Solution time for GNN1 on combinatorial auctions problems with a normal distribution approximation.

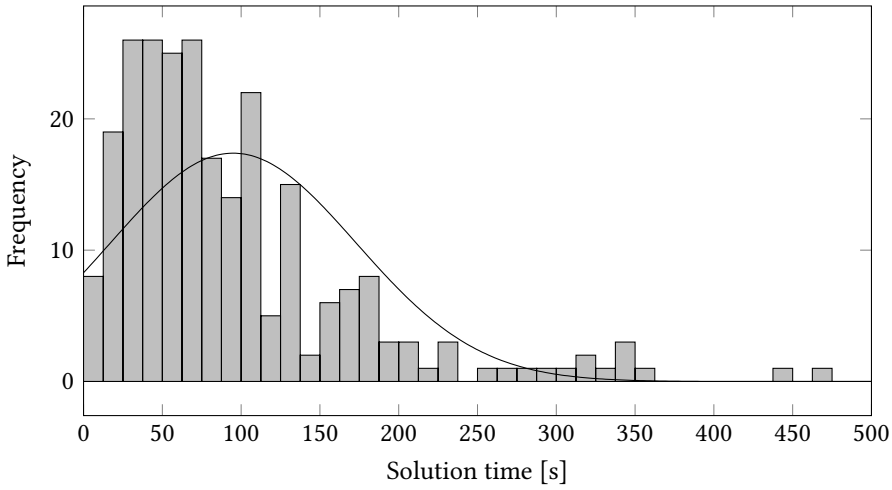

Figure A.2: Number of nodes after solving for GNN1 on combinatorial auctions problems with a normal distribution approximation.

Appendix B

Linear Model Coefficients

The coefficients of the MLP1 model constitute a linear classifier for predicting the top branching variable according to the Strong Branching algorithm. The input variables are normalized in the prenorm layer as explained in Section 3.2.1, and the coefficients are then min-max normalized between -1 and 1, calculated as:

$$\mathbf{x}_{minmax} = 2 \cdot \frac{\mathbf{x} - \mathbf{1} \cdot \min(\mathbf{x})}{\mathbf{1} \cdot (\max(\mathbf{x}) - \min(\mathbf{x}))} - \mathbf{1} \tag{B.1}$$

The result is presented in Table B.1. The variable type features are omitted, as these are equal for all samples and therefore do not contribute to the prediction.

A thorough analysis of the variable features and their correlation with the variable quality and/or Strong Branching score is interesting in its own right and highly relevant in an analysis of the quality of the feature set. This is outside of the scope and purpose of this thesis, but this little result is included in order to encourage future research.

Feature	Auctions	Setcover
objective	0.99	1.00
has_lb	0.37	0.58
has_ub	-0.79	-1.00
reduced_cost	-1.0	-0.89
sol_value	-0.64	-0.91
sol_frac	-0.97	-0.98
sol_is_at_lb	1.00	0.82
sol_is_at_ub	-0.76	-0.94
scaled_age	-0.69	-0.90
inc_val	-0.09	0.34
avg_inc_val	-0.38	0.20
basis_status_lower	0.97	0.90
basis_status_basic	-0.77	-0.93
basis_status_upper	-0.76	-0.94
basis_status_zero	-0.89	-0.98

Table B.1: Normalized coefficients for the MLP1 linear models. Irrelevant features are omitted.

References

[1] G. B. Dantzig, "Reminiscences about the origins of linear programming," in *Mathematical Programming The State of the Art*, Springer, 1983, pp. 78–86.

[2] M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, Eds., *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*. Springer, 2010.

[3] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: A methodological tour d'horizon," *European Journal of Operational Research*, 2020.

[4] L. Wolsey, *Integer Programming*, ser. Wiley Series in Discrete Mathematics and Optimization. Wiley, 2020.

[5] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," English, *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960.

[6] R. Anand, D. Aggarwal, and V. Chahar, "A comparative analysis of optimization solvers," Jul. 2017.

[7] T. Achterberg, "Scip: Solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 1–41, 2009.

[8] A. Lodi and G. Zarpellon, "On learning and branching: a survey," *TOP: An Official Journal of the Spanish Society of Statistics and Operations Research*, vol. 25, no. 2, pp. 207–236, Jul. 2017.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[10] D. Bertsimas and B. Stellato, "Online mixed-integer optimization in milliseconds," *arXiv preprint arXiv:1907.02206*, 2019.

[11] R. Meyes, M. Lu, C. W. de Puiseau, and T. Meisen, *Ablation studies in artificial neural networks*, 2019. arXiv: `1901.08644 [cs.NE]`.

[12] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi, "Exact combinatorial optimization with graph convolutional neural networks," in *Advances in Neural Information Processing Systems 32*, 2019.

[13] E. B. Khalil, "Towards tighter integration of machine learning and discrete optimization," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 2020.

[14] E. B. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina, "Learning to branch in mixed integer programming," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[15] P. Gupta, M. Gasse, E. B. Khalil, M. P. Kumar, A. Lodi, and Y. Bengio, "Hybrid models for learning to branch," in *Advances in Neural Information Processing Systems 33*, 2020.

[16] C. Schulz, G. Hasle, A. R. Brodtkorb, and T. R. Hagen, "Gpu computing in discrete optimization. part ii: Survey focused on routing problems," *EURO journal on transportation and logistics*, vol. 2, no. 1-2, pp. 159–186, 2013.

[17] A. Holzinger, P. Kieseberg, E. Weippl, and A. M. Tjoa, "Current advances, trends and challenges of machine learning and knowledge extraction: From machine learning to explainable ai," in *International Cross-Domain Conference for Machine Learning and Knowledge Extraction*, Springer, 2018, pp. 1–8.

[18] A. Prouvost, J. Dumouchelle, L. Scavuzzo, M. Gasse, D. Chételat, and A. Lodi, *Ecole: A gym-like library for machine learning in combinatorial optimization solvers*, 2020. arXiv: 2011.06069 [cs.LG].

[19] A. Prouvost, J. Dumouchelle, M. Gasse, D. Chételat, and A. Lodi, *Ecole: A library for learning inside milp solvers*, 2021. arXiv: 2104.02828 [cs.LG].

[20] Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Veličković, "Combinatorial optimization and reasoning with graph neural networks," *arXiv preprint arXiv:2102.09544*, 2021.

[21] M. Etheve, Z. Alès, C. Bissuel, O. Juan, and S. Kedad-Sidhoum, "Reinforcement learning for variable selection in a branch and bound algorithm," *Lecture Notes in Computer Science*, pp. 176–185, 2020.

[22] O. Seror, "Ablative therapies: Advantages and disadvantages of radiofrequency, cryotherapy, microwave and electroporation methods, or how to choose the right method for an individual patient?" *Diagnostic and Interventional Imaging*, vol. 96, no. 6, pp. 617–624, 2015, Diagnostic imaging of the abdomen.

[23] J. Nocedal and S. J. Wright, *Numerical Optimization*, second. New York, NY, USA: Springer, 2006.

[24] N. Karmarkar, "A new polynomial-time algorithm for linear programming," *Combinatorica*, vol. 4, no. 4, pp. 373–395, Dec. 1984.

[25] C. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Jan. 1982, vol. 32.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[27] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell, "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning," *Discrete Optimization*, vol. 19, pp. 79–102, 2016.

[28] T. Achterberg, T. Koch, and A. Martin, "Branching rules revisited," *Operations Research Letters*, vol. 33, no. 1, pp. 42–54, 2005.

[29] D. Applegate, R. Bixby, V. Chvátal, and B. Cook, "Finding cuts in the tsp (a preliminary report)," 1995.

[30] R. A. Howard, "Dynamic programming and markov processes.," 1960.

[31] G. E. Monahan, "State of the art—a survey of partially observable markov decision processes: Theory, models, and algorithms," *Management science*, vol. 28, no. 1, pp. 1–16, 1982.

[32] M. A. Nielsen, *Neural networks and deep learning*, misc, 2018.

[33] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016. arXiv: 1609.02907.

[34] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song, *Learning combinatorial optimization algorithms over graphs*, 2018. arXiv: 1704.01665 [cs.LG].

[35] S. Sheikholeslami, *Ablation programming for machine learning*, 2019.

[36] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *Foundations and Trends® in Machine Learning*, vol. 11, no. 3-4, pp. 219–354, 2018.

[37] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, *Mastering chess and shogi by self-play with a general reinforcement learning algorithm*, 2017. arXiv: 1712.01815 [cs.AI].

[38] Y. Tang, S. Agrawal, and Y. Faenza, *Reinforcement learning for integer programming: Learning to cut*, 2020. arXiv: 1906.04859 [cs.LG].

[39] L. Scavuzzo Montana, "Learning variable selection rules for the branch-and-bound algorithm using reinforcement learning," 2020.

[40] E. Balas and A. Ho, "Set covering algorithms using cutting planes, heuristics, and subgradient optimization: A computational study," in *Combinatorial Optimization*, M. W. Padberg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1980, pp. 37–60.

[41] M. Minoux, "A class of combinatorial problems with polynomially solvable large scale set covering/partitioning relaxations," en, *RAIRO - Operations Research - Recherche Opérationnelle*, vol. 21, no. 2, pp. 105–136, 1987.

[42] R. M. Karp, "Reducibility among combinatorial problems.," in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, Eds., ser. The IBM Research Symposia Series, Plenum Press, New York, 1972, pp. 85–103.

[43] K. Leyton-Brown, M. Pearson, and Y. Shoham, "Towards a universal test suite for combinatorial auction algorithms," in *Proceedings of the 2nd ACM Conference on Electronic Commerce*, ser. EC '00, Minneapolis, Minnesota, USA: Association for Computing Machinery, 2000, pp. 66–76.

[44] J. Abrache, T. G. Crainic, M. Gendreau, and M. Rekik, "Combinatorial auctions," *Annals of Operations Research*, vol. 153, no. 1, pp. 131–164, 2007.

[45] M. Dong, G. Sun, X. Wang, and Q. Zhang, "Combinatorial auction with time-frequency flexibility in cognitive radio networks," in *2012 Proceedings IEEE INFOCOM*, 2012, pp. 2282–2290.

[46] H. Sun, W. Chen, H. Li, A. Financial, and L. Song, "Improving learning to branch via reinforcement learning," 2021.

[47] S. Ross, "Interactive Learning for Sequential Decisions and Predictions," Jun. 2013.

[48] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

[49] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," 2011.

[50] G. Gamrath and C. Schubert, "Measuring the impact of branching rules for mixed-integer programming," in *Operations Research Proceedings 2017*, N. Kliewer, J. F. Ehmke, and R. Borndörfer, Eds., Cham: Springer International Publishing, 2018, pp. 165–170.

[51] M. Benichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and O. Vincent, "Experiments in mixed-integer linear programming," *Math. Program.*, vol. 1, no. 1, pp. 76–94, Dec. 1971.

[52] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, *et al.*, "The scip optimization suite 7.0," 2020.

[53] R. Wunderling, "Paralleler und objektorientierter simplex-algorithmus," 1996.

[54] G. Gamrath and M. Lübbecke, "Experiments with a generic dantzig-wolfe decomposition for integer programs," May 2010, pp. 239–252.

[55] Y. Shinano, "The ubiquity generator framework: 7 years of progress in parallelizing branch-and-bound," in *Operations Research Proceedings 2017*, 2018, pp. 143–149.

[56] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *arXiv preprint arXiv:1912.01703*, 2019.

[57] M. Fey and J. E. Lenssen, *Fast graph representation learning with pytorch geometric*, 2019. arXiv: 1903.02428 [cs.LG].

[58] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, 2016. arXiv: 1606.01540 [cs.LG].

[59] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[60] D. Kirk, B. S. Center, *et al.*, "Nvidia cuda software and gpu parallel computing architecture," 2008.

[61] S. Maher, M. Miltenberger, J. P. Pedroso, D. Rehfeldt, R. Schwarz, and F. Serrano, "Pyscipopt: Mathematical programming in python with the scip optimization suite," in *Mathematical Software – ICMS 2016*, G.-M. Greuel, T. Koch, P. Paule, and A. Sommese, Eds., Cham: Springer International Publishing, 2016, pp. 301–307.

[62] G. Zarpellon, J. Jo, A. Lodi, and Y. Bengio, *Parameterizing branch-and-bound search trees to learn branching policies*, 2020. arXiv: 2002.05120 [cs.LG].

[63] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Automated configuration of mixed integer programming solvers," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, A. Lodi, M. Milano, and P. Toth, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 186–202.

Lars Lødemel Sandberg

Ablating a Graph Neural Network for Branching in Mixed-Integer Linear Programming

NTNU
Norwegian University of
Science and Technology