ERASMUS UNIVERSITY ROTTERDAM
Erasmus School of Economics

A graph convolutional neural network for cutting plane selection in a branch-and-cut framework

Abstract

Optimization is an integral part of our modern-day society, and for many problems the algorithm of choice is branch and cut. This thesis aims to improve cutting plane selection in a branch-and-cut framework, by training a graph convolutional neural network to predict each cutting plane's bound improvement. In contrast to earlier research, this model allows us to explicitly capture the entire structure of the problem, possibly enhancing its ability to identify high-quality cutting planes. We train the model via imitation learning, using true bound improvement as an expert decision rule. To evaluate the performance of our proposed approach, we conduct an extensive analysis on both simulated and real-world benchmark instances. The results indicate that our model has learned to predict bound improvement to a certain extent, and that this enhances the branch-and-cut algorithm's efficiency. That is, our approach appears to reduce the number of nodes that the algorithm must visit to solve a problem to optimality, especially for difficult instances. However, due to its high computational burden, our proposed approach does not improve the algorithm's performance in terms of solution time. These findings emphasize the potential of machine learning methods for cutting plane selection, as a model that was trained via imitation learning appears to improve the efficiency of the branch-and-cut algorithm. Imitation learning can be readily applied to any other supervised machine learning method, which means that this thesis highlights a promising direction of future research.

Keywords: cut selection, machine learning, branch and cut, combinatorial optimization

Master's thesis
M.Sc. in Econometrics and Management Science

Name: Stefan van Berkum
Student ID: 467315

Supervisor: Olga Kuryatnikova, Ph.D.
Second assessor: Paul Bouman, Ph.D.

Date final version: July 14, 2022

Contents

Abbreviations

GCNN:	graph convolutional neural network.
GNN:	graph neural network.
IP:	integer programming.
LP:	linear programming.
MILP:	mixed-integer linear programming.
MIP:	mixed-integer programming.
MIQP:	mixed-integer quadratic programming.
MSE:	mean squared error.
ReLU:	rectified linear unit.
SVM:	support vector machine.

1 Introduction

Everyday, countless processes need to be optimized. From train scheduling to donor-patient pairing in kidney exchange programs, optimization is all around us (Wolsey, 2021). In many cases, we are faced with combinatorial optimization problems, in which the goal is to find an optimal solution among a finite set of possibilities. These problems are often NP-hard and their size usually prohibits brute-force search, which means that we need to resort to optimization techniques. Specialized algorithms exist for many well-known examples, such as the assignment problem (e.g., Kuhn, 1955) and shortest path problem (e.g., Dijkstra, 1959). In other cases, when a problem is non-standard or no efficient algorithm exists, it can often be formulated as a mixed-integer programming (MIP) problem and solved with a general-purpose MIP solver. Contemporary solvers generally employ a myriad of different algorithms and strategies, usually embedded in a branch-and-bound framework.

Branch and bound is a tree-based strategy for solving MIP problems, and is usually paired with a procedure that adds cutting planes (cuts, for short) to reduce the size of the problem. This joint strategy is often referred to as branch and cut, and is implemented in most modern solvers. Cutting plane methods can considerably improve the performance of branch and bound, often substantially reducing solving time (Mitchell, 2002).

Researchers are always looking to improve algorithms such as branch and cut, and machine learning has recently attracted much attention in this context. In general, it is difficult to solve a *constrained* problem with machine learning, since these methods are not designed to enforce hard constraints. However, machine learning excels at learning complex patterns and using this knowledge to make informed decisions. A solver that employs branch and cut requires many of such decisions to be made, both before and during the solving process. For instance, before solving one might want to decide whether the problem should be decomposed, in order to obtain a stronger formulation. During solving, one needs to select which nodes and variables to branch on, and which cuts to add to the linear programming (LP) relaxation. Often, these are local decisions embedded in the branch-and-cut framework, which is exactly the type of situation where machine learning can thrive, without the need to worry about solution feasibility.

In this thesis, our main goal is to improve cut selection in a branch-and-cut framework. In most solvers, cuts are generated by various cut generation algorithms, and then selected based on some measure of cut quality. An intuitive measure of cut quality is the bound improvement after adding it to the LP relaxation, but solving this relaxation for each cut candidate online (i.e., during solving) is usually too slow for this approach to be viable in practice. For this reason, cut quality is often estimated via simple heuristics, which sacrifice some accuracy to speed up inference. We propose to learn to estimate bound improvement, by extending a graph convolutional neural network (GCNN) previously used in branching variable selection. As model training can be taken offline (i.e., before solving), and inference is usually quite fast, this might yield a good compromise between speed and accuracy.

To examine the performance of our proposed approach, we consider three subgoals. The first subgoal is to construct and train a model that predicts bound improvement. For this purpose, we train a GCNN with simulated data by means of imitation learning, and test whether it is able to correctly rank cuts based on bound improvement. Our second subgoal is to investigate whether our proposed approach improves upon the current state of the art, and whether this generalizes to instances of arbitrary *size*. For this purpose, we incorporate our cut selection policy into a branch-and-cut solver, and evaluate its performance on simulated instances. In these first two subgoals we consider simulated data of four different problem types, and each model is trained, tested, and evaluated on instances of the same type. However, for use in a general-purpose MIP solver, it is imperative that our approach also generalizes to instances of arbitrary *type*. This is the third and final subgoal, which we evaluate using real-world benchmark instances.

Neural networks have been used before to improve cut selection (e.g., Tang, Agrawal, & Faenza, 2020; Huang et al., 2022), but to the best of our knowledge, we are the first to use a GCNN for this purpose. Moreover, earlier research on this topic has mainly focused on parts of the problem, such as the relation between the constraints and cuts (Tang et al., 2020) or features that are directly related to the cuts themselves (Huang et al., 2022). In contrast to this existing research, we use a GCNN that is capable of explicitly capturing the *entire* structure of the problem. We train our GCNN to directly predict the bound improvement that would be obtained by adding a single cut to the LP relaxation.

The remainder of this thesis is structured as follows. First, Section 2 discusses background information on optimization and machine learning, as well as recent advances in the integration of these fields. Section 3 then discusses the motivation behind our approach, the technical specifications of our model, and model training. Then, Section 4 outlines our experimental setup. Section 5 discusses the results, and finally Section 6 summarizes the results and ends by discussing possible avenues of future research.

2 Background

This section introduces some notation that we use throughout the thesis and provides a non-exhaustive discussion of optimization, machine learning, and the intersection of these fields. The literature on these topics is vast, so we only cover what is needed to understand the remainder of this thesis, and selected topics that we consider to be relevant to our work. This discussion is divided over three subsections, that each deal with one of the aforementioned subject areas. Specifically, Subsection 2.1 discusses the basics of optimization, with a focus on integer programming. Subsection 2.2 outlines the parts of machine learning that are relevant to our research, and Subsection 2.3 discusses recent advances in machine learning for optimization.

2.1 Optimization

Mathematical optimization is a technique that is often employed in the field of operations research, where the goal generally is to find the optimal solution to a constrained problem. In this thesis, we focus on *integer programming (IP)*, a type of optimization that deals with problems that consist of one or more integer variables. Problems of this type are typically harder to solve than their continuous counterparts, which is why they are of particular research interest. We will only consider linear problems, sometimes referred to as mixed-integer linear programming (MILP), although our approach is not necessarily limited to this area. IP is detailed in Subsection 2.1.1. As mentioned before, problems of this type are usually solved with the so-called branch-and-bound algorithm, which is often paired with a cutting plane strategy to form a technique called branch and cut. Subsection 2.1.2 discusses branch and bound, and Subsection 2.1.3 outlines the branch-and-cut framework. The discussion in Subsections 2.1.1–2.1.3 is based on the comprehensive overview of IP by Wolsey (2021). Subsection 2.1.4 gives an overview of different ways to measure cut quality. Finally, Subsection 2.1.5 outlines how cutting planes are implemented in SCIP, a non-commercial solver that is designed for academic use.

2.1.1 Integer programming

In mathematical optimization, problems typically consist of an *objective function* that we wish to either minimize or maximize with respect to the so-called *decision variables*, subject to a set of *constraints*. One of the most fundamental problems is the *linear program*, which can be defined in the standard form as

$$(LP) \qquad \max_{\boldsymbol{x}} \left\{ \boldsymbol{c}^T\boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \geq \boldsymbol{0} \right\}, \tag{1}$$

where $\boldsymbol{c}$ denotes the vector of objective function coefficients and $\boldsymbol{x}$ denotes the vector of decision variables, which together make up the objective function. The constraints are defined by a system of inequalities $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$. A convenient property of linear programs is that their feasible region is always *convex*, which generally makes them easier to solve (Nocedal & Wright, 2006).

In a linear program, the decision variables are allowed to take any value greater or equal to zero. In IP, however, one or more of these variables are restricted to be integer. In the case that only some variables are integer, we have a *mixed-integer program*, which can be defined in the standard form as

$$(MIP) \qquad \max_{\boldsymbol{x},\boldsymbol{y}} \left\{ \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{h}^T\boldsymbol{y} : \boldsymbol{A}\boldsymbol{x} + \boldsymbol{G}\boldsymbol{y} \leq \boldsymbol{b}, \boldsymbol{x} \in \mathbb{N}^n, \boldsymbol{y} \geq \boldsymbol{0} \right\}, \tag{2}$$

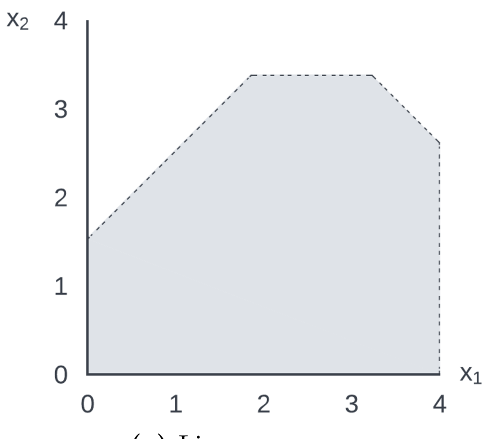
where $\boldsymbol{x}$ and $\boldsymbol{y}$ denote the integer and continuous decision variables, respectively. The set $\mathbb{N}^n$ denotes the n-dimensional set of natural numbers, with n being the dimension of $\boldsymbol{x}$. When all variables are restricted to be integer, we have an *integer program*, which can be defined in the standard form as

$$(IP) \qquad \max_{\boldsymbol{x}} \left\{ \boldsymbol{c}^T\boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \in \mathbb{N}^n \right\}. \tag{3}$$

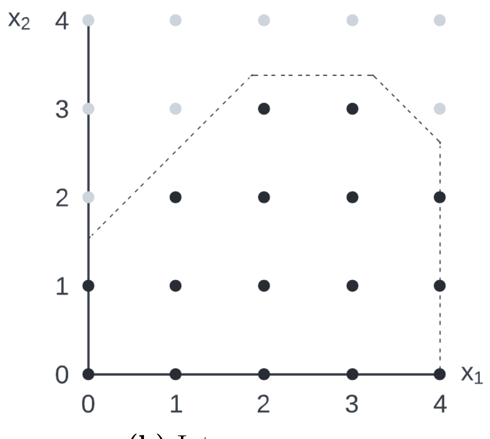
Many real-world applications of optimization are so-called combinatorial optimization problems. Here, the goal is to find an optimal subset $\mathcal{S} \subseteq \mathcal{U}$, where $\mathcal{U}$ is a set of elements, subject to certain constraints. This problem type can be broadly defined as

$$(COP) \qquad \max_{\mathcal{S}} \left\{ \sum_{i \in \mathcal{S}} c_i : \mathcal{S} \in \mathcal{F} \right\}. \tag{4}$$

where c_i is the objective function weight that corresponds to element $i \in \mathcal{U}$, and $\mathcal{F}$ denotes the set of feasible subsets of $\mathcal{U}$. These programs can often be formulated as a (mixed-)integer programming problem. For ease of notation, we will focus on integer programs in the remainder of this discussion.



(a) Linear program



(b) Integer program

Figure 1: An example of feasible regions of the same problem without (left) and with (right) integrality constraints on the decision variables x_1 and x_2. The dashed lines depict constraints, and the feasible region is depicted by a shaded area in the linear program and by black dots in the integer program.

A fundamental difference between the linear program described in Equation 1 and the integer program described in Equation 3, is that the integrality constraint causes the latter problem to be non-convex. This is illustrated in Figure 1, which depicts a simple example problem formulated both with and without integrality constraints. In the linear program, the feasible region encompasses the entire area within the bounds (axes and constraints). In the integer program, however, the feasible region comprises only the integer points that lie within those same bounds. A set $\mathcal{S} \in \mathbb{R}^n$ is a convex set if for any two points in $\mathcal{S}$, the line connecting those points lies entirely within $\mathcal{S}$ (Nocedal & Wright, 2006). Formally, this means that for any two points $\boldsymbol{x} \in \mathcal{S}$ and $\boldsymbol{y} \in \mathcal{S}$, we have that $\alpha\boldsymbol{x} + (1-\alpha)\boldsymbol{y} \in S$ for all $\alpha \in [0, 1]$. Clearly, integer programs do not satisfy this property, as any line between two points would be outside of the feasible region everywhere except in the two points themselves. Due to convexity, at least one of the vertices in the linear program must be an optimal solution. Algorithms for solving LP problems generally exploit this property, making them very efficient. In an integer program, however, this no longer holds, which means that we have to resort to other techniques.

2.1.2 Branch and bound

In the previous subsection, we concluded that we cannot use the usual methods to solve an integer program. So, how do we solve these integer programs? A naive approach would be to simply enumerate all possible solutions, but in real-world applications there are often so many possibilities that this is computationally intractable. Branch and bound is a technique based on this naive idea, but instead of *explicitly* enumerating all possible solutions, it does so *implicitly*. That is, it divides the problem into smaller subproblems, and decides for each subproblem whether it needs to be investigated further. This usually allows branch and bound to eliminate large parts of the search space, making it an effective algorithm for solving integer programs. In order to describe the branch-and-bound algorithm, we will first define two important components: the *LP relaxation* and *incumbent solution*.

The LP relaxation of an integer program is obtained by dropping the integrality constraint.

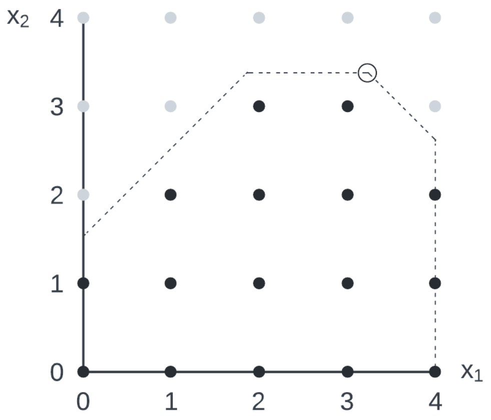


Figure 2: An example solution to the LP relaxation of an integer program. The dashed lines depict constraints, the feasible region is depicted by black dots, and the solution to the LP relaxation is circled.

This relaxation can now efficiently be solved using the usual methods, but the solution is often not feasible in the integer program. In fact, if it is feasible, then it must also an optimal solution to the integer program. To illustrate this, let us consider the case when we obtain a solution for the LP relaxation that is infeasible in the integer program. This situation is visualized in Figure 2, which depicts our running example with a possible solution to the LP relaxation. Clearly, this solution is not feasible in the integer program. As the feasible region of the LP relaxation includes both the integer points and all other solutions within the problem bounds, this solution will provide an upper bound on the optimal solution of our integer program.

Branch-and-bound algorithms can also employ so-called *primal heuristics*, which are used to generate primal solutions. Primal solutions are feasible solutions of the integer program, but they are not necessarily optimal, and can be used to provide a lower bound on the optimal solution. The best primal solution so far is called the *incumbent* solution. In general, these heuristics are not guaranteed to be successful, so in practice it might be worthwhile to try many different approaches and use the one that provides the best bound.

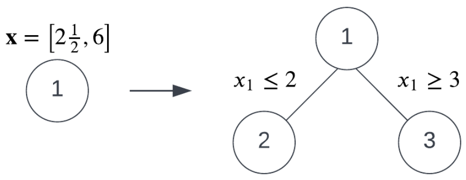


Figure 3: A visualization of branching in a branch-and-bound framework. The solution to the LP relaxation of the problem is shown above the root node.

Branch and bound is a tree-based approach, where the search can be visualized by means of a binary tree. The solution process starts with a *root node*, which is just the original integer program. For each node, the algorithm solves the LP relaxation for that particular subproblem. At every branch-and-bound iteration, the algorithm picks a leaf node for which the LP solution is non-integral, and branches on one of the fractional variables. This branching process is visualized in Figure 3, which depicts a very simple example with two variables x_1 and x_2. In this case, the solution to the LP relaxation at the root node is $\boldsymbol{x} = \left[2\frac{1}{2}, 6\right]$, and so the algorithm branches on the fractional variable x_1.

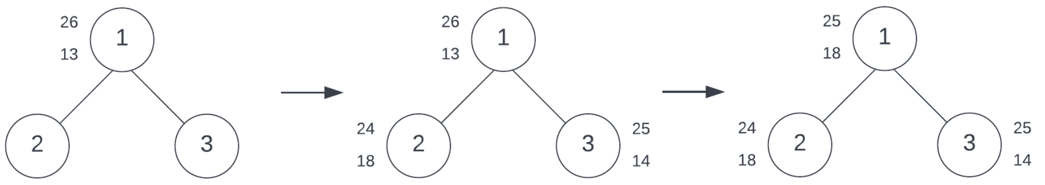
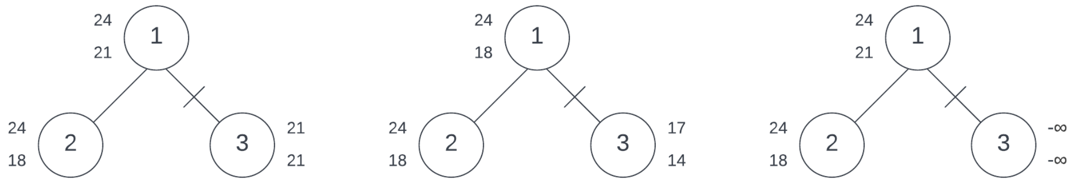


Figure 4: A visualization bound accounting in a branch-and-bound framework. The current upper and lower bound for each node is depicted beside it, with the upper bound being the top number.

During solving, the branch and bound algorithm keeps track of the best bounds by propagating the bounds that are obtained after branching up the tree. In a maximization setting, this means that the upper and lower bound of a parent node are the maxima of these bounds between its two child nodes. That is, the upper bound of the parent node $\bar{Z} = \max(\bar{Z}_1, \bar{Z}_2)$, where $\bar{Z}_1$ and $\bar{Z}_2$ denote the upper bounds at the two child nodes. Similarly, the lower bound of a parent node $\underline{Z} = \max(\underline{Z}_1, \underline{Z}_2)$, where $\underline{Z}_1$ and $\underline{Z}_2$ denote the lower bounds at the two child nodes. This bound accounting process is illustrated in Figure 4.



(a) Optimality **(b)** Bound **(c)** Infeasibility

Figure 5: Examples of pruning by optimality, bound, and infeasibility. The updated upper and lower bound for each node is depicted beside it, with the upper bound being the top number.

What makes branch and bound such a powerful technique, is that it cleverly uses these upper and lower bounds to decide whether a node needs to be explored further, a process called *pruning*. In branch and bound, there are three reasons for pruning a branch. The first reason is *pruning by optimality*, which happens when the solution to the LP relaxation is integral. As mentioned before, this implies that the solution is also optimal in the corresponding integer program, which means that we do not need to explore this branch any further. The second reason is *pruning by bound*, which occurs if the upper bound at a node is below the best lower bound. This implies that the solution that we will get from this branch will never be better than our incumbent solution, and hence it can be pruned. The third reason is *pruning by infeasibility*, which happens if a bound causes the subproblem at a node to be infeasible. For instance, if the solution to the LP relaxation yields a fractional variable $x_1 = 2\frac{1}{2}$, but the program contains a constraint $x_1 < 3$, then the node following the branching split $x_1 \geq 3$ will be infeasible. These three reasons for pruning a branch are illustrated in Figure 5.

This process of solving LP relaxations, branching, updating bounds, and pruning continues until there are no branches left to explore. When that happens, all possible solutions have been considered *implicitly* (i.e., without explicitly trying all of them), and therefore the incumbent solution must be optimal.

2.1.3 Branch and cut

Nowadays, branch and bound is often paired up with a strategy that adds cuts to reduce the size of the problem, resulting in a branch-and-cut algorithm. Initially, general-purpose cuts were regarded as mostly of theoretical interest. In the mid 1990s, however, excellent results on lift-and-project and Gomory mixed-integer cuts (Balas, Ceria, & Cornuéjols, 1993; Balas, Ceria, Cornuéjols, & Natraj, 1996) constituted a breakthrough, demonstrating the practical value of general-purpose cuts, when properly selected and employed (Dey & Molinaro, 2018).

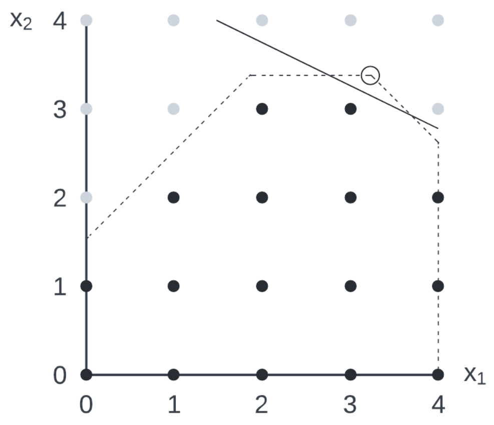


Figure 6: An example cut in an integer program. The dashed lines depict constraints, the feasible region is depicted by black dots, the solution to the LP relaxation is circled, and the cut is depicted as a solid line.

To understand what cuts are, we will first define the notion of a *valid inequality*. In an integer program with a set of feasible solutions $\mathcal{F}$, an inequality $\boldsymbol{x}^T\boldsymbol{\pi} \leq \pi_0$ is a valid inequality if it holds for any $\boldsymbol{x} \in \mathcal{F}$. A cutting plane is a valid inequality that, quite literally, cuts off a solution to the LP relaxation that is infeasible in the integer program. Figure 6 displays a possible cut in the context of our running example.

In an ideal situation, the region formed by the constraints in an integer program would be convex and tightly wrapped around the set of feasible solutions, which is called the problem's *convex hull*. With such a region, every solution to the LP relaxation would also be feasible in the integer program. Cutting planes are useful because they strengthen the problem formulation, approaching this ideal formulation. For instance, say we have a formulation

$$(IP) \qquad \max_{\boldsymbol{x}} \{\boldsymbol{c}^T\boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \in \mathbb{N}\}. \tag{5}$$

We can obtain an upper bound UB using the LP relaxation

$$(LP) \qquad \max_{\boldsymbol{x}} \{\boldsymbol{c}^T\boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \geq \boldsymbol{0}\}, \tag{6}$$

with feasible region

$$S = \{\boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \geq \boldsymbol{0}\}. \tag{7}$$

Say we find a non-integral solution to the LP relaxation $\hat{\boldsymbol{x}}$, with corresponding objective value $UB = \boldsymbol{c}^T\hat{\boldsymbol{x}}$. Now, let us introduce (one or more) cuts $\boldsymbol{D}\boldsymbol{x} \leq \boldsymbol{d}$ that cut off our infeasible solution

$\hat{\boldsymbol{x}}$, to obtain the new formulation

$$(LP') \qquad \max_{\boldsymbol{x}}\{\boldsymbol{c}^T\boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{D}\boldsymbol{x} \leq \boldsymbol{d}, \boldsymbol{x} \geq \boldsymbol{0}\}, \tag{8}$$

with feasible region

$$S' = \{\boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{D}\boldsymbol{x} \leq \boldsymbol{d}, \boldsymbol{x} \geq \boldsymbol{0}\}, \tag{9}$$

and corresponding solution $\hat{\boldsymbol{x}}'$ with upper bound $UB' = \boldsymbol{c}^T\hat{\boldsymbol{x}}'$. Clearly, $S' \subseteq S$, and hence $UB' \leq UB$. In other words, introducing cuts has led to an upper bound that is at least as strong as our original bound. Better bounds improve the performance of the branch-and-bound algorithm, illustrating how cutting plane algorithms can speed up computation.

Usually, these cuts are either added only at the root node or throughout the entire tree. How these cuts can be generated is beyond the scope of this thesis, but interested readers can refer to Wolsey (2021) for a comprehensive overview of cut generation techniques. In some situations, we can find many cutting planes, in which case it is common to sort them in some way and add a subset (Mitchell, 2002). Adding multiple cuts at every iteration appears to improve convergence, but the exact number of cuts to add is not clear (Dey & Molinaro, 2018).

2.1.4 Metrics of cut quality

As mentioned in the previous subsection, whenever cuts are applied it is common to sort them in some way, for which we need a measure of cut quality. Cuts can affect the performance of the algorithm in a variety of ways, and they are usually added in groups, which means that their performance is also not necessarily independent of other cuts. Therefore, different metrics exist that each focus on other aspects of cut quality. In this subsection, we will cover those metrics that we use in the remainder of this thesis. All metrics are based on the comprehensive overview of cut management and selection by Wesselmann and Suhl (2012), except for the directed cutoff distance, which was suggested by Gleixner et al. (2018). However, this is a non-exhaustive list, and interested readers can refer to Wesselmann and Suhl (2012) for more information, or Dey and Molinaro (2018) p. 244, for a very brief overview of some more involved metrics.

Support For a cut $\boldsymbol{\alpha}^T\boldsymbol{x} \leq \beta$, the support is the fraction of variables that have a non-zero position in $\boldsymbol{\alpha}$. That is, we define the support $s(\boldsymbol{\alpha})$ as

$$s(\boldsymbol{\alpha}) = \frac{n}{N}, \tag{10}$$

where n denotes the number of non-zero positions in $\boldsymbol{\alpha}$ and N denotes the total number of variables for a particular problem. Cuts that have large support are called dense, and are likely to slow down simplex computations and can possibly cause numerical instability.

Integral support The integral support of a cut $\boldsymbol{\alpha}^T\boldsymbol{x} \leq \beta$ is the fraction of its variables in $\boldsymbol{\alpha}$ that are constrained to be integer. The integral support $i(\boldsymbol{\alpha})$ can be defined as

$$i(\boldsymbol{\alpha}) = \frac{n_i}{n}, \tag{11}$$

where n_i denotes the number of integer-constrained variables in $\boldsymbol{\alpha}$ and n again denotes the total number of non-zero positions for this cut. We prefer cuts with high integral support (Wesselmann & Suhl, 2012).

Efficacy A cut's efficacy, or cutoff distance, measures the depth of a cut relative to the current LP solution. Formally, for a cut $\boldsymbol{\alpha}^T\boldsymbol{x} \leq \beta$, the efficacy measures the Euclidean distance between the current LP solution $\hat{\boldsymbol{x}}$ and the hyperplane $\boldsymbol{\alpha}^T\hat{\boldsymbol{x}} = \beta$. We can define the efficacy $e(\boldsymbol{\alpha}, \beta, \hat{\boldsymbol{x}})$ as

$$e(\boldsymbol{\alpha}, \beta, \hat{\boldsymbol{x}}) = \frac{\boldsymbol{\alpha}^T\hat{\boldsymbol{x}} - \beta}{||\boldsymbol{\alpha}||}. \tag{12}$$

Intuitively, deep cuts are of relatively high quality, as they cut off a comparatively large part of the search region, strengthening the problem formulation.

Directed cutoff distance The directed cutoff distance is similar to the cut's efficacy, but now we measure this distance in the direction of the feasible region. That is, for a cut $\boldsymbol{\alpha}^T\boldsymbol{x} \leq \beta$, instead of taking the shortest distance between the current LP solution $\hat{\boldsymbol{x}}$ and the hyperplane $\boldsymbol{\alpha}^T\hat{\boldsymbol{x}} = \beta$, we measure this distance directed towards the incumbent solution $\tilde{\boldsymbol{x}}$. The directed cutoff distance $d(\boldsymbol{\alpha}, \beta, \hat{\boldsymbol{x}}, \tilde{\boldsymbol{x}})$ is defined as

$$d(\boldsymbol{\alpha}, \beta, \hat{\boldsymbol{x}}, \tilde{\boldsymbol{x}}) = \frac{\boldsymbol{\alpha}^T\hat{\boldsymbol{x}} - \beta}{||\boldsymbol{\alpha}^T\boldsymbol{\delta}||}, \tag{13}$$

where $\boldsymbol{\delta} = \frac{\tilde{\boldsymbol{x}} - \hat{\boldsymbol{x}}}{||\tilde{\boldsymbol{x}} - \hat{\boldsymbol{x}}||}$, which represents the normalized direction from the LP solution to the incumbent solution. Again, similar to the efficacy, deeper cuts are preferable.

Objective parallelism The objective parallelism of a cut $\boldsymbol{\alpha}^T\boldsymbol{x} \leq \beta$ is measured by computing the absolute value of the cosine similarity between $\boldsymbol{\alpha}$ and the coefficient vector $\boldsymbol{c}$ of the objective function. Therefore, we can define the objective parallelism $o(\boldsymbol{\alpha})$ as

$$o(\boldsymbol{\alpha}) = \frac{|\boldsymbol{\alpha}^T\boldsymbol{c}|}{||\boldsymbol{\alpha}||\,||\boldsymbol{c}||}, \tag{14}$$

which equals zero if the cut and the objective function are orthogonal and one if they are parallel. We prefer cuts with high objective parallelism (Wesselmann & Suhl, 2012).

Parallelism Let us consider two cuts $(\boldsymbol{\alpha}^i)^T\boldsymbol{x} \leq \beta^i$ and $(\boldsymbol{\alpha}^j)^T\boldsymbol{x} \leq \beta^j$. We can again measure the parallelism between the coefficient vectors $\boldsymbol{\alpha}^i$ and $\boldsymbol{\alpha}^j$ by taking the absolute value of their cosine similarity. The parallelism $p(\boldsymbol{\alpha}^i, \boldsymbol{\alpha}^j)$ is then defined as

$$p(\boldsymbol{\alpha}^i, \boldsymbol{\alpha}^j) = \frac{|(\boldsymbol{\alpha}^i)^T\boldsymbol{\alpha}^j|}{||\boldsymbol{\alpha}^i||\,||\boldsymbol{\alpha}^j||}, \tag{15}$$

which equals zero if the cuts are orthogonal and one if they are parallel. High parallelism is an undesirable property for cuts, because we want to avoid adding unnecessary cuts. To illustrate this point, consider the case when two cuts are completely parallel (i.e., $p(\boldsymbol{\alpha}^i, \boldsymbol{\alpha}^j) = 1$). In this case, we would only like to choose the deepest one as the other would be redundant.

Bound improvement If cuts are especially useful through bound improvement, one might think, why not use it directly to measure cut quality? Indeed, this is one of the most intuitive metrics, and is often used to measure cut quality (e.g., Coniglio & Tieves, 2015). It captures multiple aspects of cut quality that affect the bound improvement after adding a cut to the LP relaxation, such as depth (e.g., efficacy) and orientation (e.g., objective parallelism). However, due to its high computational burden (solving an LP relaxation for each cut candidate), it is usually ineffective in practice. Therefore, it might be very useful to train a model to predict this bound improvement, which is what we aim to do in this thesis.

2.1.5 Cutting planes in SCIP

For all optimization-related tasks, we use the SCIP solver for constraint integer programming (Achterberg, 2009), which is one of the fastest non-commercial solvers available. SCIP is designed with academic use in mind, allowing the user to customize many parts of the solving process (e.g., cut and branching variable selection), through so-called plugins. Specifically, we use PySCIPOpt 4.2.0 (Maher et al., 2016), the Python implementation of SCIP, which uses the SCIP Optimization Suite 8.0 (Bestuzheva et al., 2021).

During solving, SCIP keeps track of a cut pool, which stores cuts that have been generated in the current *separation round* (i.e., a round of adding cuts) and cuts that were not selected in earlier rounds. Cuts can 'age' out of cut pool, if they are not selected for a given number of separation rounds. As of the SCIP Optimization Suite 5.0, SCIP uses a hashing approach to detect parallel cuts, and a method suggested by Andreello, Caprara, and Fischetti (2007) for filtering non-efficacious cuts out of the cut pool (Gleixner et al., 2017).

Since the first version of SCIP, it employs a hybrid cut selection strategy that ranks cuts based on a weighted sum of multiple metrics of cut quality. At first, this was based on efficacy, objective parallelism, and parallelism (Achterberg, 2009). As of the SCIP Optimization Suite 5.0, this hybrid cut selector does not use parallelism to score cuts anymore, but includes the integral support (Gleixner et al., 2017). In the SCIP Optimization Suite 6.0, directed cutoff distance was also added to the score (Gleixner et al., 2018), but this was dropped again in a later version. In its current implementation (SCIP Optimization Suite 8.0), the hybrid cut score $h(\boldsymbol{\alpha}, \beta, \hat{\boldsymbol{x}})$ for a cut $\boldsymbol{\alpha}^T \boldsymbol{x} \leq \beta$ is defined as

$$h(\boldsymbol{\alpha}, \beta, \hat{\boldsymbol{x}}, \tilde{\boldsymbol{x}}) = 1 \cdot e(\boldsymbol{\alpha}, \beta, \hat{\boldsymbol{x}}) + 0.1 \cdot i(\boldsymbol{\alpha}) + 0.1 \cdot o(\boldsymbol{\alpha}), \tag{16}$$

where each of the scores is defined as in Subsection 2.1.4.

While parallelism is not used anymore to compute the hybrid cut score, SCIP now adopts a strategy suggested by Wesselmann and Suhl (2012) to remove similar cuts from the pool, while keeping the ones with relatively high quality (Gleixner et al., 2017). In each separation round, this approach first marks all cuts that are parallel to a higher quality cut (i.e., one that has a higher hybrid cut score) for removal, with a maximum parallelism `p_max` $= 0.1$. For each cut that is marked for removal, it is removed if and only if the quality value is below `skip_factor` $= 0.9$ times that of the highest quality cut, and its parallelism does not exceed `p_max_ub` $= 0.5$. Finally, SCIP adds all cuts that survive this filtering process to the LP relaxation.

2.2 Machine learning

Machine learning is a field that encompasses many well-known subfields such as clustering and deep learning. The goal in machine learning can be broadly defined as trying to extract useful information from data, which is something that has become increasingly important in our data-driven world. Therefore, it comes as no surprise that machine learning has been the subject of much interest in both industry and academia.

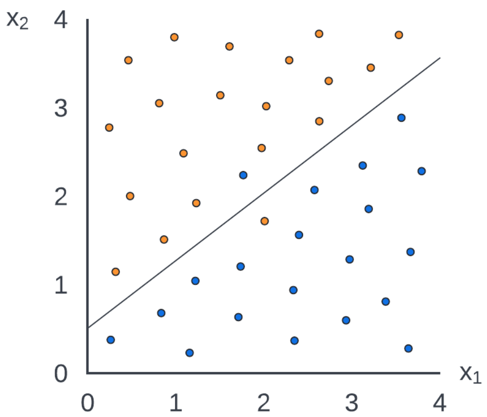
We can distinguish between three types of machine learning: *supervised*, *unsupervised*, and *reinforcement* learning. We can characterize supervised learning as learning by *demonstration*. In other words, the techniques that fall into this category require labeled data and learn to predict these labels. We can describe unsupervised learning as learning by *recognition*, meaning that it learns to recognize patterns in the data without requiring any labels. Finally, we can characterize reinforcement learning as learning by *experience*, which can be seen as a mix between supervised and unsupervised learning. That is, it does not require labeled data, but it does need to process training instances, learning along the way. Reinforcement learning does this by defining a reward function, which is used to train the model through trial and error.

In this thesis, we will focus on supervised learning, particularly in the context of imitation learning for GCNNs. For this purpose, Subsection 2.2.1 first discusses supervised learning in more detail, and Subsection 2.2.2 introduces imitation learning. Subsection 2.2.3 briefly outlines deep learning, and finally Subsection 2.2.4 introduces graph neural networks.

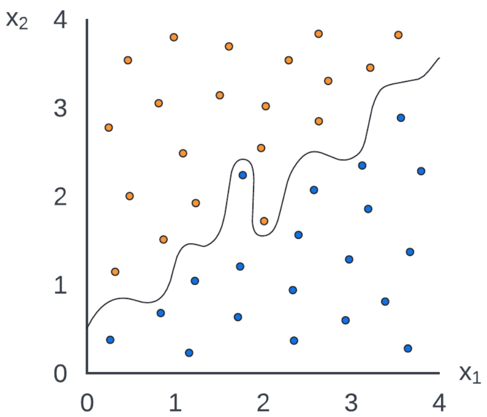
2.2.1 Supervised learning

As briefly mentioned before, supervised learning works with labeled data. Generally, the goal is to minimize some loss function, which depends on what we would like the model to learn. In a *regression* setting, we would like the model to approximate the data generating process. In this case, the labels are continuous and we would like the model to learn to approximate the relationship between these labels and a set of features. In regression, a common loss function is the mean squared error (MSE). In a *classification* setting, on the other hand, we want the model to classify observations as accurately as possible. In this case, the labels are categorical and we would like the model to accurately predict these labels based on a set of features. In classification, a common loss function is the cross-entropy loss.

A problem that is central to machine learning is *overfitting*, which means that the model learns to model the noise in the training set, yielding excellent performance on this particular set but poor generalization ability. This is often referred to as the bias-variance trade-off, since generalization performance often comes at the cost of flexibility. That is, very flexible models often have relatively low bias as they can closely follow the data, but high variance as they are very sensitive to the distribution of the training data. Conversely, very rigid models often have relatively large bias, but low variance. Figure 7 illustrates this trade-off, at the two extreme ends of the spectrum. A straight line is extremely rigid, and therefore it has high bias, but low variance. On the other hand, a flexible line that can perfectly fit the data will have low bias, but high variance. In the example, a straight line seems like a reasonable fit, but the required flexibility differs for each problem and should be examined on a case-by-case basis.



(a) High bias, low variance



(b) Low bias, high variance

Figure 7: An illustration of the bias-variance trade-off in a classification setting. The color of each point represents its label, and the predicted label is blue below the line and orange above it.

2.2.2 Imitation learning

Imitation learning is a technique that trains models to mimic some expert decision rule. It is a supervised alternative to reinforcement learning, which is especially useful when the latter would be computationally intractable or when a reward function is not readily available. The discussion in this subsection is largely based on the excellent survey on imitation learning by Hussein, Gaber, Elyan, and Jayne (2017).

Recall that in supervised learning, we train the model by demonstration using a set of known features and labels. In imitation learning, these features are *states* and the labels are *actions* taken by an expert. That is, the dataset consists of state-action pairs, which are usually represented as a *Markov decision process*, as is also common in reinforcement learning. Much like a Markov chain, a Markov decision process satisfies the Markov property, which means that the process is memoryless (Sutton & Barto, 2018). That is, the action taken at time t solely depends on the state at that time. Markov decision processes are a convenient choice for representing expert decisions in imitation learning, as their Markov property allows us to store state-action pairs without having to include earlier states.

Imitation learning allows us to model Markov decision processes, without the computational burden that comes with reinforcement learning. Reinforcement learning can be very slow due to its trial-and-error approach, especially in the early stages of training. Imitation learning overcomes this by simply asking an expert what to do at any given state, and using the resulting state-action pairs as features and labels for supervised learning.

In imitation learning, the goal is to blindly mimic the expert, which is sometimes preferable over reinforcement learning but also has its drawbacks. For instance, the performance of the learned policy is bounded by the expert, which is a limitation when the expert decision rule is suboptimal. Therefore, it should only be used when the learned policy is significantly faster than the expert (Bengio, Lodi, & Prouvost, 2021). To overcome this drawback, imitation learning is often followed up by reinforcement learning to fine-tune the model, increasing its generalization ability and possibly allowing it to outperform the expert (e.g., Silver et al., 2016).

2.2.3 Deep learning

Deep learning is a family of machine learning methods that are inspired by animal brains, which is why models within this category are often aptly called *artificial neural networks*. Neural networks have been omnipresent in academic research, due to their ability to model complex patterns directly from the raw input data. In other machine learning techniques, one often has to make assumptions about the data generating process, whereas a neural network can learn these relationships by itself. This makes it a very powerful tool for complex problems that are not easily modeled explicitly. In this thesis we use TensorFlow (Abadi et al., 2016), which is one of the most popular Python libraries for deep learning. TensorFlow makes it easy to build custom models, allowing for much flexibility in tailoring your model to the problem at hand.

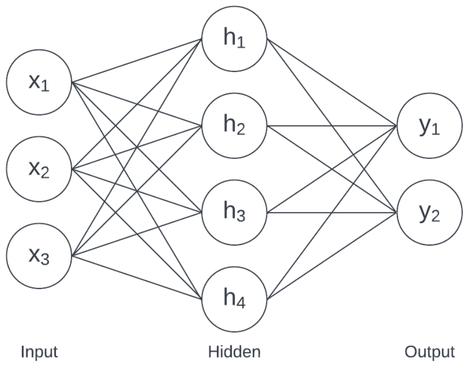


Figure 8: An example of a single-layer neural network with three features, four nodes in the hidden layer, and two outputs.

Neural networks are directed graphs that consist of one or more connected layers of nodes (sometimes called neurons or hidden units), the most basic being the single-layer feedforward neural network. These single-layer models consist of an *input layer*, a single *hidden layer*, and an *output layer*. This type of single-layer feedforward neural network is illustrated in Figure 8. The input layer simply has one node for each feature, and these connect to the nodes in the hidden layer. Usually, every node in a layer connects to every node in the following layer, in which case the layer is called *dense*. Each node in the hidden layer then computes a weighted sum of its inputs and possibly adds a bias (a constant). The result of this operation is often called the *activation*. This activation is then passed through an *activation function*, and passed on to the nodes in the output layer, which repeats the same procedure.

The activation function $h(a)$ can take many forms, but one of the most popular for hidden layers nowadays is the rectified linear unit (ReLU), which is defined as $h(a) = \max(0, a)$. The ReLU activation is the default recommendation for modern neural networks (Goodfellow, Bengio, & Courville, 2016), since it is computationally light and generally has good performance. Other choices that remain popular are the sigmoid function $h(a) = \frac{1}{1+e^{-a}}$ and the hyperbolic tangent $h(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$. These alternative activation functions are computationally heavier, but they add more non-linearity to the model, which might prove useful in some cases.

In the output layer, the choice for a particular activation function depends on the type of output that we expect the model to return. If we are in a regression setting, the activation in the output layer is usually simply the identity function $h(a) = a$. In a classification setting, the activation function in the output layer is often a sigmoid when there are just two classes (binary classification), and a softmax function $h(a_k) = \frac{e^{a_k}}{\sum_{j=1}^{K} e^{a_j}}$ in case there are more than two classes. Other activation functions might be chosen for the output layer if we know that the output has a specific structure. For instance, if a regression output is constrained to be on the interval $[0, 1]$, then one might opt for a sigmoid instead of a linear activation function.

It is often convenient to write operations in matrix form, so for this purpose we can denote the operation for a single node in a dense layer as

$$y(\boldsymbol{x}, \boldsymbol{w}, b) = h\left(\boldsymbol{x}^T \boldsymbol{w} + b\right), \tag{17}$$

where y is the scalar output that is returned by the node, $\boldsymbol{x}$ is the node's $m \times 1$ vector of inputs (i.e., the output of all nodes in the previous layer), $\boldsymbol{w}$ is node's $m \times 1$ weight vector, b is the bias that is added by this node, and $h(a)$ is its activation function. We can expand this notation by representing the entire dense layer of k nodes, as a single operation

$$\boldsymbol{y}\left(\boldsymbol{x}, \boldsymbol{W}, \boldsymbol{b}\right) = h\left(\boldsymbol{x}^T \boldsymbol{W} + \boldsymbol{b}^T\right), \tag{18}$$

where now $\boldsymbol{y}$ denotes a $1 \times k$ vector of outputs, $\boldsymbol{W}$ is an $m \times k$ matrix of weights, $\boldsymbol{b}$ is a vector of biases of size k, and $h(\boldsymbol{a})$ is the activation function for this layer applied element-wise. If we generalize this to allow for n inputs (e.g., observations), we get n vectors of outputs that can be stacked into a matrix, which we can compute as

$$\boldsymbol{Y}\left(\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{b}\right) = h\left(\boldsymbol{X}\boldsymbol{W} + \mathbf{1}_{n\times 1}\boldsymbol{b}^T\right), \tag{19}$$

where $\boldsymbol{Y}$ is an $n \times k$ matrix of outputs, $\boldsymbol{X}$ is now an $n \times m$ matrix of inputs, $\mathbf{1}_{n\times 1}$ denotes an $n \times 1$ vector of ones, and $h(\boldsymbol{A})$ is again the activation function for this layer applied element-wise.

The most conventional neural networks (sometimes referred to as multilayer perceptrons) build upon this single-layer model, by adding additional hidden layers in between the existing hidden layer and the output layer. These layers can vary in size, and need not all have the same activation function. Besides these standard neural networks, there are many other types of models that we will not be able to discuss in detail, but interested readers can refer to Goodfellow et al. (2016) for an extensive introduction to deep learning.

The standard way to train feedforward neural networks is through a method called *back-propagation*, which computes the gradient with respect to the weights in the network. This gradient can then be used to minimize loss over the training sample via *(stochastic) gradient descent*. The name backpropagation is derived from the fact that derivatives can be computed in a recursive manner using the chain rule, starting from the output moving backwards towards the input layer, propagating them back through the network. Usually, the training set is divided into *batches*. In stochastic gradient descent, the gradient is computed for each batch and the weights for each node are updated by subtracting the scaled gradient

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \gamma \nabla L(\boldsymbol{w_t}), \tag{20}$$

where $\boldsymbol{w}_t$ denotes the node's vector of weights at time t, γ denotes the *learning rate*, and $\nabla L(\boldsymbol{w_t})$

denotes the gradient of the training loss with respect to the weights at time t. Normally, it takes several passes through the entire training set to train the model, and these passes are commonly referred to as *epochs*. That is, for a training set of size n with batches of size b, a single epoch comprises approximately $\frac{n}{b}$ gradient updates. In this case, we assumed that the learning rate stays constant over time. However, it is usually desirable to dynamically adapt this rate during training, or use a dynamic optimizer such as Adam (Kingma & Ba, 2015).

As neural networks are very flexible, they are also especially prone to overfitting. Many techniques have been proposed to mitigate this issue, one of which is the use of a validation set. During training, the loss on the training set should generally decrease, as the model learns to model the training data. Eventually, it stops learning useful patterns, and starts to model noise. Ideally, we would like to stop training as soon as that happens. This is where the validation set comes into play, which is solely used to keep track of the generalization ability of the model during solving, and not for actual training. In its most basic implementation, we simply stop training as soon as the performance on the validation set starts to decrease.

2.2.4 Graph neural networks

Graph neural networks (GNNs) are deep-learning based methods that operate on graphs, which is useful as many real-life problems can be expressed as a graph. These methods have been around for a while, but recent advancements in deep learning have led to a resurgence of interest in GNNs (Zhou et al., 2020). We discuss these methods in detail because this is the type of neural network that we use in this thesis. Specifically, we will use a GCNN, which is the most common type of GNN. Our discussion on this topic is based on the comprehensive surveys by Zhou et al. (2020) and Wu et al. (2021).

Graphs can come in many different shapes and sizes, making it difficult to process them using our conventional neural networks, which work with a fixed set of input features. GNNs generally address this issue by keeping the graph structure intact, and updating the representation for each node separately (i.e., a vector for each node). These updated representations can then be processed further, to provide information on specific nodes or the graph as a whole. GCNNs are the most common type of GNN, which work with a so-called *convolution* operation to update the node representations. The most popular approaches are *spatial-based*, which means that they exploit the spatial relations that are naturally present in a graph. That is, these models usually update the representation of any given node by drawing information from its neighbors (adjacent nodes).

We can interpret spatial-based graph convolutions as a message passing process, where a node learns about its neighbors through information that is passed along its edges. With each iteration of the message passing process, a node's learning range is effectively expanded, by drawing information from its direct neighbors. That is, in the first iteration a node learns about its direct neighbors, but in the second iteration each neighbor has also learned about its own neighbors, so the node learns about those as well. In this way, a node's learning range is expanded by one connection on each iteration. This process is illustrated in Figure 9, in the context of a GCNN applied on the structural formula for aspirin. In the figure, double bonds are shown but in the actual graph this connection would likely be a single edge, with the fact

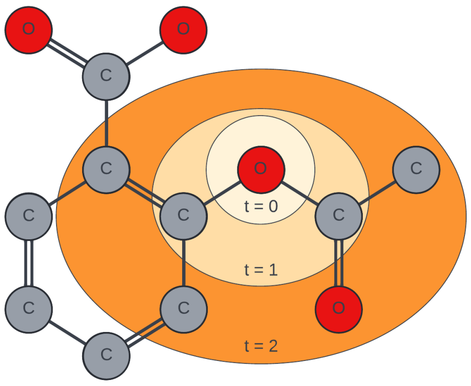


Figure 9: An illustration of how a node's learning range is expanded on each time step t, for a GCNN used on structural formula for acetylsalicylic acid (aspirin), excluding hydrogen atoms.

that this represents a double bond reflected in its features. In this particular example, one might want to finally combine all node representations into one score, for instance to determine whether or not this molecule is an illicit drug (graph classification).

Gilmer, Schoenholz, Riley, Vinyals, and Dahl (2017) define a general framework for these message passing neural networks, in terms of a two-phase procedure. The first phase is the *message passing phase*, where node representations (hidden states) are updated iteratively for T time steps. The hidden state $\boldsymbol{h}_v^t$ for any node v at time t is updated using a joint message representation $\boldsymbol{m}_v^{t+1}$ defined as

$$\boldsymbol{m}_v^{t+1} = \sum_{(i,j)\in\mathcal{E}_v} M_t(\boldsymbol{h}_v^t, \boldsymbol{h}_w^t, \boldsymbol{e}_{ij}), \tag{21}$$

where $\mathcal{E}_v$ denotes the set of edges that are incident to node v, M_t is a message function at time step t, and $\boldsymbol{e}_{ij}$ denotes a vector of edge features for the edge between w and v. For simplicity, we assume that the graph is undirected, so that $\boldsymbol{e}_{ij} = \boldsymbol{e}_{ji}$. This joint message representation is then used to update the hidden state for node v as follows

$$\boldsymbol{h}_v^{t+1} = U_t(\boldsymbol{h}_v^{t+1}, \boldsymbol{m}_v^{t+1}), \tag{22}$$

where U_t denotes a node update function at time step t. The second phase, called the *readout phase*, computes some output $\boldsymbol{y}$ based on all updated node representations

$$\boldsymbol{y} = R(\{\boldsymbol{h}_v^T : v \in \mathcal{N}\}), \tag{23}$$

where R denotes some readout function and $\mathcal{N}$ is the set of node indices in the graph. Note that this specification can accommodate a wide variety of methods, as Gilmer et al. (2017) aimed to define a single unifying framework.

2.3 Machine learning for optimization

In recent years, machine learning has often been proposed for use in the context of optimization. Bengio et al. (2021) identify two main motivations for integrating these techniques —

approximation and *discovery of new policies.* In the former, one tries to replace an expensive computation by a fast approximation, in order to speed up the solving process. In this setting, imitation learning is often used to train a model to mimic this slow policy. In the latter, when our goal is to discover new policies, one often turns to reinforcement learning, which allows the model to discover its own policies through experience.

There are different ways of integrating machine learning and optimization, from enhancing small parts of the algorithm to replacing the optimization routine altogether. As suggested by Bengio et al. (2021), we can divide these approaches into three categories — *end-to-end learning*, *learning to configure algorithms*, and *machine learning alongside optimization algorithms.* In this categorization, our proposed method falls within the latter category. For the purpose of this thesis, we will go through each of these three approaches separately, briefly outlining recent advances in each area. Interested readers can refer to Bengio et al. (2021) for an extensive overview. In our discussion, Subsection 2.3.1 first discusses end-to-end learning. Then, Subsection 2.3.2 outlines different ways of learning to configure algorithms. Finally, Subsection 2.3.3 discusses different ways of using machine learning alongside existing optimization routines.

2.3.1 End-to-end learning

In end-to-end learning, one attempts to solve the entire optimization problem with a machine learning model. A big problem with this type of approach is that machine learning methods cannot ensure the feasibility of the solution. Machine learning methods are designed to uncover patterns in data, but are not necessarily suited to constrained optimization (out of the box). In some cases, however, this issue can be circumvented.

For instance, a solution can sometimes be constructed greedily following some heuristic, such as in the traveling salesman problem. Dai, Khalil, Zhang, Dilkina, and Song (2017) propose to learn such a greedy heuristic using a GCNN. At each node, a graph representation of the problem is fed to the neural network, which returns an action value (i.e., which node to visit next). This type of heuristic approach ensures feasibility of all solutions, making machine learning a suitable candidate for learning such a policy.

When feasibility cannot be guaranteed, one could simply check whether the machine learning solution is feasible, and discard it if this is not the case. For instance, Selsam et al. (2019) train a GCNN to predict satisfiability in the context of a satisfisiability problem. In a satisiability problem, the goal is to find a solution such that a certain boolean formula evaluates to one. The authors note that this problem can be represented as a *bipartite graph*, which can then be analyzed by means of a GCNN that is trained to predict satisfiability. The authors show these predictions can be decoded into a feasible solution in over 70% of the cases.

2.3.2 Learning to configure algorithms

As mentioned before, a big drawback of end-to-end learning is that machine learning can provide no guarantee as to the feasibility of its solution. As there are numerous excellent algorithms and solvers available that inherently take constraints into account, it makes sense to consider a combination of the two methods. That is, find a way to harness the strength of both machine learning and classical optimization methods. This category of machine learning methods for optimization — learning to configure algorithms — attempts to form such a hybrid method. Most optimization methods have hyperparameters that need to be determined beforehand, which can be done with machine learning without interfering with solution feasibility.

For instance, Kruber, Lübbecke, and Parmentier (2017) use machine learning to determine whether a Dantzig-Wolfe decomposition should be applied to the MIP problem before solving. A Dantzig-Wolfe decomposition exploits the structure of a problem (if any), which can substantially strengthen the MIP formulation.

Similarly, Bonami, Lodi, and Zarpellon (2018) use machine learning to decide if linearizing a mixed-integer quadratic programming (MIQP) problem beforehand will reduce solving time. As the name suggests, MIQP problems have a quadratic part, which can often be linearized. As this is not always successful and linearization requires some computational effort, it is not necessarily beneficial. Therefore, machine learning lends itself well to such a situation.

2.3.3 Machine learning alongside optimization algorithms

In this type of method — machine learning alongside optimization algorithms — machine learning is incorporated into classical optimization methods. As discussed before, optimization routines such as branch-and-cut require many decisions to be made during solving. Many of these decisions have no impact on solution feasibility, which makes them ideal candidates for the use of machine learning. In the remainder of this discussion, we will go through some types of decisions that have been approached with machine learning.

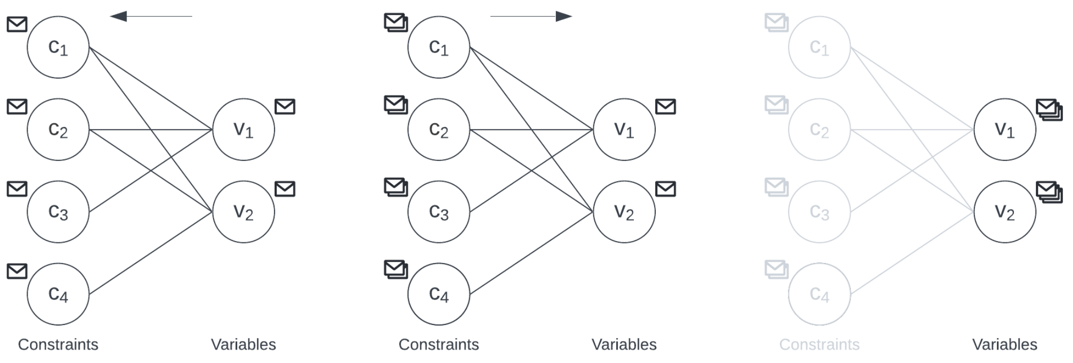
Primal heuristics A modern solver often employs multiple primal heuristics, which are run periodically to check for feasible solutions. The strategy that orchestrates when each heuristic should be run is often determined through trial-and-error. Khalil, Dilkina, Nemhauser, Ahmed, and Shao (2017) propose to learn such a policy, using machine learning. For this purpose, the authors train a logistic regression model via imitation learning, to predict whether or not a heuristic will find a new incumbent.

Node selection During the solving process, a branch and bound algorithm needs to repeatedly decide on which node to branch. In an attempt to tackle this issue, He, Daumé III, and Eisner (2014) propose to learn a policy that decides which node to branch on next. For this purpose, they formulate node selection as a Markov decision process, and use imitation learning to train a support vector machine (SVM) that predicts which node to branch on.

Variable selection When a node has been selected by the branch-and-bound algorithm, it still needs to decide which fractional variable to branch on. This decision lends itself particularly well to machine learning, as a good branching policy can be highly effective, and there exists a clear expert — strong branching. In strong branching, multiple variables are tentatively explored, and the best one is chosen for actual branching. Strong branching is highly effective, but computationally intensive, which makes it an ideal candidate for imitation learning.

One of the first of such works was by Khalil, Le Bodic, Song, Nemhauser, and Dilkina (2016), who propose a linear model for variable selection that is trained via imitation learning during solving. That is, in the early stages of solving strong branching is applied, which is then used as an expert decision rule for training the model. When it is sufficiently trained, the linear model replaces strong branching, speeding up computations.

Alvarez, Louveaux, and Wehenkel (2017) treat variable selection as a regression problem, directly predicting strong branching scores. For this purpose, the authors train a tree-based model using imitation learning. Contrary to Khalil et al. (2016), model training is done offline, and therefore their model can be used throughout the entire solving process.



(a) $t = 0$ **(b)** $t = 1$ **(c)** $t = 2$

Figure 10: An illustration of the message passing GCNN for branching variable selection, for a problem with four constraints and two variables. The time steps of the message passing process are denoted by t.

Gasse, Chételat, Ferroni, Charlin, and Lodi (2019) train a model to mimic strong branching via imitation learning, and formulate branching variables selection as a Markov decision process. This paper lays the foundation for our work, as we adapt their method for the use in cut selection. Similar to Selsam et al. (2019), the authors note that MIP problems can be represented as a bipartite graph, with constraints on one side and variables on the other. There is an edge between a variable and constraint if and only if this variable appears in the corresponding constraint, and the edge's features contain the exact coefficient. Moreover, a variable's coefficient in the objective function is included in its features. The authors construct a message passing GCNN, which is illustrated in Figure 10, for an example with four constraints and two variables. This graph could, for instance, belong to an optimization problem

$$\max_{v_1, v_2} \{3v_1 + 8v_2 : v_1 + v_2 \leq 5, 2v_1 + 5v_2 \leq 7, v_1 \geq 0, v_2 \geq 0\}. \tag{24}$$

At first, every node is only aware of itself, and it has a message about its features. In the first stage of the message passing phase ($t = 0$), the variables send their message to the connected constraints. These constraints now have an updated representation, as they have learned about adjacent variables and their connection to these nodes. In the second stage of message passing ($t = 1$), this packet of messages (the updated representation), is sent back to the variables. Now, the variables have learned about adjacent constraints, their relation to these nodes, and the relation of other variables to those same nodes. In the readout phase ($t = 2$), the constraints are disregarded and the packet of messages at each variable is processed to decide on which we should branch. The authors compare their approach to earlier research and the default branching rule in SCIP, and demonstrate that their proposed method attains superior results.

Cut selection Several machine learning approaches have also been proposed for cut selection, as this decision is often made heuristically in contemporary solvers. For instance, Baltean-Lugojan, Bonami, Misener, and Tramontani (2019) train a neural network to predict the objective value of a semidefinite programming relaxation in the context of quadratic optimization. With these predicted objective values one can quickly approximate the estimated bound improvement for any cut, which can then be employed to rank and select cuts. The authors train their model via imitation learning, using the true objective value as an expert decision rule.

Tang et al. (2020) focus on Gomory cuts, and train a neural network to select cuts using reinforcement learning. For this purpose, they formulate cut selection as a Markov decision process, and reward the model for adding cuts that tighten the LP formulation. That is, the reward function is the bound improvement that is attained after adding a cut. The experimental results indicate that this method reduces the number of nodes that are required to reach a certain degree of solution accuracy, compared to simple heuristics (e.g., random selection).

Huang et al. (2022) also employ a neural network for cut selection, but instead of individual cuts, the authors consider bags that consist of multiple cuts. For this purpose, the model is trained using supervised learning on a set of features and labels at the bag level. These features are a combination of structural information of the cuts and different measurements suggested by Wesselmann and Suhl (2012), and are aggregated at the bag level by taking the average over all cuts. The labels are based on the reduction ratio of solution time for a set of cuts, which can be computed as the authors only add cuts at the root node. The authors find that their proposed approach outperforms heuristics that are based on a single metric of cut quality.

3 Methodology

In this thesis, we propose to use a GCNN for cut selection. This section discusses our approach, both the motivation behind it and the technical details. For this purpose, Subsection 3.1 will first outline the motivation behind our proposed method. Then, Subsection 3.2 discusses how cut selection can be formulated as a Markov decision problem, which will be useful for training a model via imitation learning. Subsection 3.3 describes the exact model specification, and finally Subsection 3.4 outlines how we train this model using imitation learning.

3.1 Motivation

In contemporary solvers, cut candidates are usually sorted based on some metric of cut quality, and the best ones are added to the LP relaxation. For this purpose, most solvers employ simple heuristics to determine the cut quality. For instance, SCIP uses a hybrid cut selection rule that takes a weighted average of efficacy, integral support, and objective parallelism. We propose to learn such a policy via machine learning, which has proven successful in earlier research (e.g., Tang et al., 2020; Huang et al., 2022). This earlier research has mainly focused on parts of the problem. For instance, Tang et al. (2020) considers the coefficients of the constraints and cut candidates, which ignores the relation between cuts and the objective function. Conversely, Huang et al. (2022) only considers metrics that directly relate to the cuts themselves, taking the objective function into account but ignoring the problem's constraints. In this thesis, we therefore develop a method that can select cuts in light of the entire problem structure, which may enhance its ability to identify high-quality cuts.

The excellent results that were found by Gasse et al. (2019) imply that GCNNs are able to capture the problem structure quite well, as their model was able to make an informed decision as to which variable to branch on. In order to apply a GCNN to MIP problems, the authors noted that this type of problem can be represented as a bipartite graph, with constraints on the one side and variables on the other. As cuts have the same structure as constraints, we can extend this graph by connecting the cut candidates to the variables as well. We can then adapt the GCNN for the use in cut selection, by adding another partial graph convolution to the process. Moreover, instead of focusing on the variables in the readout phase, we are now interested in the final representation of the cut candidates, which we can use to predict their corresponding bound improvement. As discussed before, bound improvement is one of the most intuitive ways to measure cut quality, and is often used in the context of cut selection (e.g., Baltean-Lugojan et al., 2019; Tang et al., 2020).

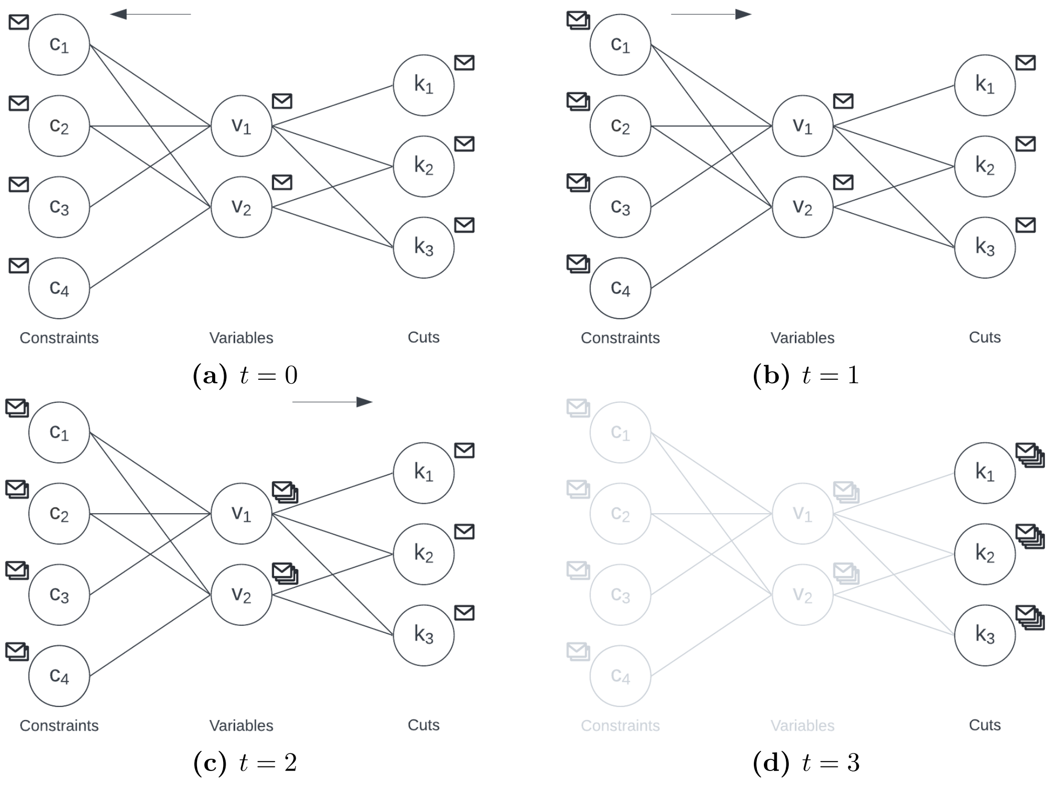


Figure 11: An illustration of the message passing GCNN for cutting plane selection, for a problem with four constraints, two variables, and three cut candidates. The time steps of the message passing process are denoted by t.

As we did for the original GCNN in the context of variable selection (see Subsection 2.3.3), we can represent the entire procedure as a message passing process. This is illustrated in Figure 11 for a problem with four constraints, two variables, and three cut candidates. Note that this graph could belong to the same optimization problem as described in Equation 24, with the addition of three cut candidates. Again, each node is initially only aware of itself, and has a message about its features. In the first stage of the message passing phase ($t = 0$), the variables send their message to the connected constraints. Now, the constraints have an updated representation, as they have learned about adjacent variables and their relation to these nodes. In the second stage of message passing ($t = 1$), this packet of messages (the updated representation), is sent back to the variables. Now, the variables have learned about adjacent constraints, their relation to these nodes, and the relation of other variables to those same nodes. Note that these first two stages are exactly as we described them in our earlier discussion regarding the model by Gasse et al. (2019), depicted in Figure 10. In the additional stage of message passing ($t = 2$), the updated representation for each variable is sent to its connected cuts. These cuts are now aware of themselves, the problem structure defined by the constraints and variables, and their relation to this structure. In the readout phase ($t = 3$), the constraints and variables are disregarded and the final representation for each cut is processed to predict its corresponding bound improvement. This can then be used to rank cuts and filter them with parallelism, as is also done for the hybrid cut score in SCIP (see Subsection 2.1.5).

In Subsection 2.1.4, we discussed that bound improvement is an intuitive metric of cut quality, but that explicitly calculating it for each cut is computationally prohibitive. This makes it an ideal candidate for imitation learning, which we will use to train the model, teaching it to mimic an expert that computes the attained bound improvement after adding a cut to the LP relaxation. By training the model before solving we take most of the computation offline, which allows us to quickly approximate bound improvement during solving. While such a model is slower than a simple heuristic in terms of inference, it might be able to more accurately predict bound improvement, possibly yielding a good compromise between speed and accuracy.

3.2 Cut selection as a Markov decision process

The starting point of any machine learning model is to define which features should be fed into the model and what should be the output. As is common in imitation learning, we shall define this by formulating cut selection as a Markov decision process. In the context of cut selection, the state at any point during solving can be defined as a set of features for the constraints, variables, cut candidates, and edges between these nodes. We define the action value to be the predicted bound improvement after adding a cut to the LP relaxation, relative to the current bound. These predictions can then be used to select cuts.

For the constraints and variables, we use a slightly trimmed version of the features that were used by Gasse et al. (2019), as these have proven to be effective. We drop the LP age of constraints and variables as this is no longer accessible in PySCIPOpt. Moreover, the simplex basis status of a variable is redundant if we also include indicators for whether the variable is at its lower or upper bound, and hence we drop these features as well. For the cuts, we include the right-hand side value β for a cut candidate $\boldsymbol{\alpha}^T \boldsymbol{x} \leq \beta$ as is done for the constraints, complemented by a comprehensive subset of the most common cut quality measures, which we described in Subsection 2.1.4. An overview of the features for each component is given in Table 1. For the constraint features, `rhs` is normalized by the constraint's norm, `is_tight` is equal to one if and only if the constraint is satisfied at equality, and `obj_cosine` takes on

Table 1: The features that are included in the feature tensor of each graph component.

Feature	Description
	Constraint features ($\boldsymbol{C}$)
`rhs`	The normalized right-hand side value β of a constraint $\boldsymbol{\alpha}^T\boldsymbol{x} \leq \beta$.
`is_tight`	An indicator function equal to one if the constraint is tight.
`obj_cosine`	The cosine similarity between the constraint and objective function.
`dual`	The constraint's normalized dual solution value.
	Variable features ($\boldsymbol{V}$)
`has_lb`	An indicator function equal to one if the variable has a lower bound.
`has_ub`	An indicator function equal to one if the variable has an upper bound.
`at_lb`	An indicator function equal to one if the variable is at its lower bound.
`at_ub`	An indicator function equal to one if the variable is at its upper bound.
`frac`	The variable's fractionality in the current LP solution.
`reduced_cost`	The variable's normalized reduced cost.
`lp_val`	The variable's value in the current LP solution.
`primal_val`	The variable's value in the best known primal (incumbent) solution.
`avg_primal`	The variable's average value over all known primal solutions.
	Cut features ($\boldsymbol{K}$)
`rhs`	The normalized right-hand side value β of a cut candidate $\boldsymbol{\alpha}^T\boldsymbol{x} \leq \beta$.
`support`	The cut's support.
`int_support`	The cut's integral support.
`efficacy`	The cut's efficacy.
`cutoff`	The cut's directed cutoff distance.
`obj_parallelism`	The cut's objective parallelism.
	Edge features ($\boldsymbol{E}$)
`coef`	The coefficient of a variable in a constraint or cut candidate.

Note. A detailed description of the cut quality metrics that are included in the cut features can be found in Subsection 2.1.4.

values on the interval $[-1, 1]$. An objective cosine of 0 means that the coefficient vectors are orthogonal, whereas a value of -1 or 1 means that the vectors are parallel, with a 180° or 0° angle between them, respectively. Finally, feature `dual` is normalized by the product of the constraint's norm and the objective function's norm. For the variable features, `at_lb` and `at_ub` are zero if the variable has no lower and upper bound, respectively. Feature `frac` is defined as `frac` $= 0.5 - |\hat{x} - \lfloor \hat{x} \rfloor - 0.5|$, where $\hat{x}$ denotes the value of variable x in the current LP solution, following Achterberg, Koch, and Martin (2005). Note that this expression simply computes how far the value is from the nearest integer. Finally, feature `reduced_cost` is normalized by the objective function's norm, and features `primal_val` and `avg_primal` are zero if no primal solution is available. For the cut candidate features, `rhs` is normalized by the cut's norm and `cutoff` is zero if no primal solution is available.

The action for each state-action pair is the predicted bound improvement, relative to the current bound. Defining the bound improvement relative to the current bound does not affect the ranking, but is needed to ensure scale invariance of the model's predictions. Absolute bound improvement generally decreases during solving (i.e., it gets harder to find good cuts), since the search region gets tighter with each iteration. Therefore, this scale invariance must be enforced if we want to predict cut quality at different stages of the solving process. Moreover, absolute bound improvement might differ radically across instances, so normalization enhances the model's ability to generalize to other instances. For this purpose, we define the relative bound improvement $b(\hat{\boldsymbol{x}}, \hat{\boldsymbol{x}}')$ as

$$b(\hat{\boldsymbol{x}}, \hat{\boldsymbol{x}}') = \frac{\left|\boldsymbol{c}^T\hat{\boldsymbol{x}} - \boldsymbol{c}^T\hat{\boldsymbol{x}}'\right|}{\left|\boldsymbol{c}^T\hat{\boldsymbol{x}}\right|}, \tag{25}$$

where $\hat{\boldsymbol{x}}$ denotes the current LP solution, $\hat{\boldsymbol{x}}'$ denotes the LP solution after adding a cut, and $\boldsymbol{c}$ denotes the vector of objective function coefficients. Since this metric simply measures the change relative to the current LP solution and we can assume that any change yields an improvement, this metric can be used for both maximization and minimization problems. To see why we can assume an improvement, recall that cuts strengthen the problem formulation, which implies that the bound can only improve or stay the same.

3.3 Model specification

In this subsection, we will detail our proposed GCNN model for cut selection. As discussed before, the model is similar to the GCNN that is used by Gasse et al. (2019) for branching variable selection, but it is adapted for use in the context of cut selection. Since their model proved successful in capturing the problem structure, we only changed what was necessary to adapt it for cut selection, leaving most of their composition intact. We will split the discussion into two parts. First, we introduce the partial graph convolution, which can be interpreted as passing a message from a set of sending nodes to a set of receiving nodes (i.e., one of the first three steps depicted in Figure 11). Then, using this partial convolution, we define the full GCNN for cut selection. Throughout our discussion we sometimes refer to pre-training, which we will elaborate on in the next subsection.

Algorithm 1 Partial graph convolution operation.

```
Input: An $s \times 64$ matrix $\boldsymbol{S}$, an $r \times 64$ matrix $\boldsymbol{R}$, an $s \times r \times e$ tensor $\boldsymbol{E}$, and a set of edges $\mathcal{E}$.
 1: $\boldsymbol{F} \leftarrow \boldsymbol{O}_{r\times 64}$
 2: for $(i, j) \in \mathcal{E}$ do
 3:     $\boldsymbol{u} \leftarrow \boldsymbol{S}_{i,*}\boldsymbol{W}^{1,1} + (\boldsymbol{E}_{i,j,*})^T \boldsymbol{W}^{1,2} + \boldsymbol{R}_{j,*}\boldsymbol{W}^{1,3} + \left(\boldsymbol{b}^1\right)^T$
 4:     $\boldsymbol{u} \leftarrow \texttt{ReLU}\left(\boldsymbol{u}/\sigma_1\right)$
 5:     $\boldsymbol{u} \leftarrow \boldsymbol{u}^T\boldsymbol{W}^2 + \left(\boldsymbol{b}^2\right)^T$
 6:     $\boldsymbol{F}_{j,*} \leftarrow \boldsymbol{F}_{j,*} + \boldsymbol{u}$
 7: end for
 8: $\boldsymbol{F} \leftarrow \boldsymbol{F}/\sigma_2$
 9: $\boldsymbol{R} \leftarrow [\boldsymbol{F}\ \boldsymbol{R}]$
10: $\boldsymbol{R} \leftarrow \texttt{ReLU}\left(\boldsymbol{R}\boldsymbol{W}^3 + \boldsymbol{1}_{r\times 1}\left(\boldsymbol{b}^3\right)^T\right)$
11: $\boldsymbol{R} \leftarrow \texttt{ReLU}\left(\boldsymbol{R}\boldsymbol{W}^4 + \boldsymbol{1}_{r\times 1}\left(\boldsymbol{b}^4\right)^T\right)$
```

Algorithm 1 describes the partial graph convolution operation, which is left exactly as implemented by Gasse et al. (2019), since no modification was required. The operation takes four input arguments: a sender feature matrix $\boldsymbol{S}$ with s nodes, a receiver feature matrix $\boldsymbol{R}$ with r nodes, an edge feature tensor $\boldsymbol{E}$ with e features, and a set of edges $\mathcal{E}$. Each edge is of the form (`sending_node`, `receiving_node`). In addition to these matrices, $\boldsymbol{O}_{a\times b}$ denotes an $a \times b$ matrix of zeros, $\boldsymbol{W}^i$ denotes a convolution-specific $a \times 64$ weight matrix of a 64-node dense layer i (row dimension a should match the input size), $\boldsymbol{b}^i$ denotes the corresponding bias vector of size 64, σ_j denotes a convolution-specific scaling parameter j that is determined during pre-training, and $\mathbf{1}_{a\times b}$ denotes an $a \times b$ matrix of ones. Different indices for $\boldsymbol{W}^i$, $\boldsymbol{b}^i$, and σ_j are meant to emphasize that these are different parameters, corresponding to different layers of the convolution. Note that these parameters also differ between convolutions. The partial graph convolution begins by computing an $r \times 64$ joint feature representation $\boldsymbol{F}$ in Lines 2-7. Each row in this matrix represents the joint feature representation of a receiving node, and it is updated for every sending node that it is connected to. At Line 3, for a given edge we pass each of the three feature vectors — sending node, edge, and receiving node — through a separate 64-node layer with a linear activation function and without bias, sum the result, and finally add a bias. Then, in Line 4, we scale the result and apply a ReLU activation function. In Lines 5 and 6, we pass the result through a 64-node layer with a linear activation function, and add the result to the row of the joint feature matrix $\boldsymbol{F}$ that corresponds to the receiving node. When all edges have been processed, the final result is scaled once more in Line 8, and is then concatenated with the initial feature matrix in Line 9 to form an $r \times 128$ matrix. This is then passed through two final 64-node layers with a ReLU activation functions in Lines 10 and 11, to obtain the updated representation for the receiving nodes $\boldsymbol{R}$. In terms of the previously defined message passing process, the sending nodes have now sent a message to the receiving nodes. With this message, the receiving nodes have learned about their adjacent sending nodes, and the edges between them.

Algorithm 2 now defines our GCNN for cut selection, using the partial graph convolution operation (`convolution`) that we described in Algorithm 1. The network takes seven input arguments: a constraint feature matrix $\boldsymbol{C}$ with m constraints and c features, a constraint edge feature tensor $\boldsymbol{E}^c$ with e features, a constraint edge set $\mathcal{E}_c$, a variable feature matrix $\boldsymbol{V}$ with n variables and v features, a cut feature matrix $\boldsymbol{K}$ with p cuts and k features, a cut edge feature tensor $\boldsymbol{E}^k$ with e features, and a cut edge set $\mathcal{E}_k$. Each edge is of the form (`variable_node`, `row_node`), where a row may be either a constraint or cut. In addition to these matrices, $\boldsymbol{W}^i$ denotes an $a \times 64$ weight matrix of a 64-node dense layer i (row dimension a should match the input size), $\boldsymbol{b}^i$ denotes the corresponding bias vector of size 64, $\boldsymbol{O}_{a\times b\times c}$ denotes an $a \times b \times c$ tensor of zeros, $\boldsymbol{w}^j$ denotes a weight vector of size 64 for a single-node dense layer, and b^j denotes the corresponding bias term. Again, different indices for $\boldsymbol{W}^i$, $\boldsymbol{b}^i$, $\boldsymbol{w}^j$, and b^j are meant to emphasize that these are different parameters, corresponding to different layers of the network. The network begins by normalizing all features and embedding the constraint, cut, and variable feature matrices in Lines 1-11. For this purpose, each column vector $\boldsymbol{x}$ is normalized by applying $(\boldsymbol{x} - \beta)/\sigma$, where β and σ are determined for each feature during pre-training. The normalized constraint, variable, and cut feature matrices are then embedded by passing them through two 64-node layers with ReLU activation functions. Note that after embedding, each feature matrix has the same number of rows (nodes) as the input matrix and 64 columns. In Lines 12-17, we reverse the order of the node indices in the constraint feature matrix and the constraint edge set, and save the resulting matrix and set for use in the constraint-to-variable convolution. Now, we can use our partial convolution operation that we defined in Algorithm 1 to execute three back-to-back convolutions. In Line 18, we perform the first stage of the message

Algorithm 2 GCNN for cut selection.

```
Input: An $m \times c$ matrix $\boldsymbol{C}$, an $n \times m \times e$ tensor $\boldsymbol{E}^c$, a set of edges $\mathcal{E}_c$, an $n \times v$ matrix $\boldsymbol{V}$, a
    $p \times k$ matrix $\boldsymbol{K}$, an $n \times p \times e$ tensor $\boldsymbol{E}^k$, and a set of edges $\mathcal{E}_k$.
 1: normalize($\boldsymbol{C}$)
 2: $\boldsymbol{C} \leftarrow$ ReLU $\left(\boldsymbol{C}\boldsymbol{W}^{c,1} + \mathbf{1}_{m\times 1}\left(\boldsymbol{b}^{c,1}\right)^T\right)$
 3: $\boldsymbol{C} \leftarrow$ ReLU $\left(\boldsymbol{C}\boldsymbol{W}^{c,2} + \mathbf{1}_{m\times 1}\left(\boldsymbol{b}^{c,2}\right)^T\right)$
 4: normalize($\boldsymbol{V}$)
 5: $\boldsymbol{V} \leftarrow$ ReLU $\left(\boldsymbol{V}\boldsymbol{W}^{v,1} + \mathbf{1}_{n\times 1}\left(\boldsymbol{b}^{v,1}\right)^T\right)$
 6: $\boldsymbol{V} \leftarrow$ ReLU $\left(\boldsymbol{V}\boldsymbol{W}^{v,2} + \mathbf{1}_{n\times 1}\left(\boldsymbol{b}^{v,2}\right)^T\right)$
 7: normalize($\boldsymbol{K}$)
 8: $\boldsymbol{K} \leftarrow$ ReLU $\left(\boldsymbol{K}\boldsymbol{W}^{k,1} + \mathbf{1}_{p\times 1}\left(\boldsymbol{b}^{k,1}\right)^T\right)$
 9: $\boldsymbol{K} \leftarrow$ ReLU $\left(\boldsymbol{K}\boldsymbol{W}^{k,2} + \mathbf{1}_{p\times 1}\left(\boldsymbol{b}^{k,2}\right)^T\right)$
10: normalize($\boldsymbol{E}^{\boldsymbol{c}}$)
11: normalize($\boldsymbol{E}^{\boldsymbol{k}}$)
12: $\boldsymbol{E}^v = \boldsymbol{O}_{m\times n\times e}$
13: $\mathcal{E}_v \leftarrow \varnothing$
14: for $(i, j) \in \mathcal{E}_c$ do
15:     $\boldsymbol{E}^v_{j,i,*} = \boldsymbol{E}^c_{i,j,*}$
16:     $\mathcal{E}_v \leftarrow \mathcal{E}_v \cup (j, i)$
17: end for
18: convolution($\boldsymbol{S} = \boldsymbol{V}, \boldsymbol{R} = \boldsymbol{C}, \boldsymbol{E} = \boldsymbol{E}^{\boldsymbol{c}}, \mathcal{E} = \mathcal{E}_c$)
19: convolution($\boldsymbol{S} = \boldsymbol{C}, \boldsymbol{R} = \boldsymbol{V}, \boldsymbol{E} = \boldsymbol{E}^{\boldsymbol{v}}, \mathcal{E} = \mathcal{E}_v$)
20: convolution($\boldsymbol{S} = \boldsymbol{V}, \boldsymbol{R} = \boldsymbol{K}, \boldsymbol{E} = \boldsymbol{E}^{\boldsymbol{k}}, \mathcal{E} = \mathcal{E}_k$)
21: $\boldsymbol{K} \leftarrow$ ReLU $\left(\boldsymbol{K}\boldsymbol{W}^{o} + \mathbf{1}_{p\times 1}\left(\boldsymbol{b}^{o}\right)^T\right)$
22: return $\boldsymbol{K}\boldsymbol{w} + b \cdot \mathbf{1}_{p\times 1}$
```

passing phase (Figure 11a), passing a message from the variables to the constraints. Then, in Line 19, we execute the second stage of message passing (Figure 11b), passing the updated representation back from the constraints to the variables. In Line 20, we perform the third stage of message passing (Figure 11c), sending the updated variable representation to the cuts. Finally, in Lines 21 and 22, we perform the readout phase (Figure 11d), passing the resulting problem-aware cut representation through a 64-node layer with a ReLU activation function and a single-node layer with a linear activation function to obtain an output vector of size p. This output vector now contains the predicted relative bound improvement for each cut candidate.

Note that the final layer differs from the implementation by Gasse et al. (2019), as they use a softmax layer without bias. The reason for this difference is that branching variable selection is a classification problem, while we want to estimate the relative bound improvement (a regression problem). For this reason, a softmax is not appropriate. We considered using a ReLU activation function in the output layer, to reflect the fact that the relative bound improvement is always non-negative. However, as bound improvements are usually close to zero, this would encourage the model to output zeros, causing the corresponding gradients to drop to zero as well. If this were to happen for all outputs, this could effectively halt training, a phenomenon sometimes referred to as the dying ReLU problem. To avoid this, we have opted to use the identity (linear) activation function for the output layer, as is common in a regression setting. Moreover, now that we use an identity activation function instead of a softmax, the bias term might be useful and hence we include it in the model. Note that for a softmax function, this bias would not do anything as it shifts every output by the same value.

3.4 Training

As we argued in Subsection 3.1, a model that predicts bound improvement naturally lends itself to imitation learning, as it is an intuitive measure of cut quality but computationally expensive. For this purpose, we train the model on the state-action pairs (samples) that we defined in Subsection 3.2 in a supervised fashion. We use a two-step training procedure, which was devised by Gasse et al. (2019) to train their GCNN for branching variable selection. As this procedure proved successful, and our model is very similar to their GCNN, we will use that same procedure with settings tuned to our specific application. This procedure requires two sets of samples, one for training and one for validation.

The first step in the training procedure is pre-training, which we mentioned before without detailing how this is done. Pre-training is used to determine shifting and scaling parameters, which are used in the so-called pre-norm layers of the GCNN. These pre-norm layers were defined by Gasse et al. (2019), and they simply shift and scale input by their shifting and scaling parameters, respectively. In the partial graph convolution that is described in Algorithm 1, these layers simply scale each element by an estimate of its standard deviation, denoted by σ_1 and σ_2 in Algorithm1. In the GCNN for cut selection that is described in Algorithm 2, the pre-norm layers are used to normalize each feature using an estimate of its mean and standard deviation (`normalize` in Algorithm 2). These estimates are obtained before actual training, using an online mean and variance estimation algorithm suggested by Chan, Golub, and LeVeque (1982). That is, the mean and variance are estimated iteratively by looping over batched data. For a batch with m elements, the mean μ_n that was computed over n elements is updated as

$$\mu_{n+m} = \mu_n + \frac{m}{n+m}\left(\tilde{\mu} - \mu_n\right), \tag{26}$$

where $\tilde{\mu}$ denotes the mean over the elements in the batch. Instead of updating the variance directly, this approach updates the sum of squared differences from the current mean $S_n = \sum_{i=1}^{n}\left(x_i - \mu_n\right)^2$ and then computes the estimated variance by taking $\sigma^2 = \frac{S_n}{n}$. If we again take a batch with m elements, the the current sum of squared differences is updated as

$$S_{n+m} = S_n + \tilde{S} + \frac{n \cdot m}{n+m}\left(\tilde{\mu} - \mu_n\right)^2, \tag{27}$$

where $\tilde{S}$ denotes the sum of squared deviations from the batch mean computed over the elements in the batch. Ultimately, an estimate of the standard deviation can of course be obtained by taking the square root of our estimate for the variance. This entire procedure is executed for each pre-norm layer using 10% of the training set. This subset is drawn by taking every tenth sample, and the same set is used for each pre-norm layer.

After pre-training, the procedure moves to training the weights and biases of the GCNN.

To avoid overfitting, this procedure employs a validation set, which is often used to avoid overfitting, as discussed before in Subsection 2.2.3. Moreover, it randomly selects 10% of the training samples with replacement on each epoch, and uses these samples to train the model via the Adam algorithm. That is, on every epoch, the model does not see the entire training set, but a randomly drawn subset. This might enhance the model's generalization ability, as the model sees different training data on every epoch. Usually, this is a luxury that we cannot afford due to limited training data. In this setting, however, this sampling approach is feasible as we can generate as much labeled data as we want. At each epoch, the procedure uses the MSE between actual and predicted relative bound improvements to compute the loss. This is different from the cross-entropy loss that is employed by Gasse et al. (2019) for the purpose of training their model for classification, since we are in a regression setting. The training scheme that was proposed by Gasse et al. (2019) also includes a dynamic learning rate for the Adam optimizer. At every 10 consecutive epochs without improvement on the validation set, the learning rate is divided by five, and at 20 consecutive epochs without improvement training terminates.

In their procedure, Gasse et al. (2019) use a batch size of 128 for pre-training and 32 for training the network's weights and biases, with an initial learning rate of 0.001. In our implementation and with our hardware, these batches did not fit into GPU memory. Therefore, we changed it to the largest value that still fits, which yields a batch size of 2 for pre-training and 4 for training the network's weights and biases. During preliminary experiments, we noticed that with this batch size, the model often gets stuck in poor local minima. We suspect that this issue is related to the dying ReLU problem we discussed before, as this is often observed when the learning rate is too high for the chosen batch size. In general, small batches should be used in combination with a relatively small learning rate, as each batch carries a comparatively unrepresentative sample of the population (i.e., the full training set) and could therefore provide a relatively noisy gradient. If one does not do this, a poor batch can cause the model to change drastically in the wrong direction, possibly killing off ReLUs in the process. As mentioned before, there is generally no way for a dead ReLU to recover as its gradient is zero, which might cause the model to get stuck in a poor local minimum. For this reason, we decreased the learning rate to 0.0001, which has greatly improved the stability and performance of our training procedure.

4 Experimental setup

As mentioned before, this the main goal of this thesis is to improve cut selection in a branch-and-cut framework. To evaluate the performance of our proposed cut selection strategy, we consider three subgoals — construct and train a model that predicts bound improvement, investigate whether it improves upon the current state of the art, and examine whether this generalizes to real-world instances of arbitrary type. The first two subgoals are addressed using simulated data, following the extensive analysis that was performed by Gasse et al. (2019) for their GCNN. For this purpose, we generate instances for four different problem types — set covering, combinatorial auction, capacitated facility location, and maximum independent set problems. Subsection 4.1 discusses these simulated instances. To train the model, we must extract samples (state-action pairs) from these instances, which is outlined in Subsection 4.2. To examine whether our first subgoal was achieved, we train different models and test their ability to rank cuts based on bound improvement on a set of testing samples, which is outlined in Subsection 4.3. For the second and third subgoal, we evaluate the performance of our approach in practice by incorporating the trained models into a branch-and-cut algorithm. To evaluate the second subgoal, we use simulated instances of the same problem type that was used for training each model, and we evaluate the policy's performance on instances of varying size. We address the third subgoal by evaluating the performance of our proposed approach on generic-type benchmarking instances. Subsection 4.4 describes these experiments in detail.

In all experiments except performance evaluation using benchmark instances, we use a vanilla solver to minimize noise from elements other than cut selection. Using such a controlled environment is common in evaluating machine learning methods for optimization (e.g., Gasse et al., 2019; Tang et al., 2020; Huang et al., 2022). For this purpose, we turn off presolver restarts as suggested by Gasse et al. (2019), and use a simple most infeasible branching rule as suggested by Tang et al. (2020). Most infeasible branching is a policy that simply chooses the most fractional variable (i.e., the most infeasible). Besides this, all SCIP parameters are kept at their default value. The code for reproducing all experiments is available at https://github.com/stefanvanberkum/gcnn-cut-selector. All sampling and performance evaluation is run on a single thin node of the Dutch national supercomputer Snellius, which consists of two AMD Rome 7H12 CPUs with 64 cores each at 2.6GHz. Model training and testing is done on a machine with an Intel i7-11800H CPU with 8 cores and 16 threads at 2.3GHz and an NVIDIA GeForce RTX 3050 Ti Laptop GPU with 4GB of memory.

4.1 Data generation

For the largest part of our experiments, we use simulated data. The experimental setup that pertains to these simulated instances is based on the methodology of Gasse et al. (2019), as their data is representative of real-world instances and follows current recommendations for evaluating the performance this type of method (Bengio et al., 2021). As mentioned before, the model is trained on four different problem types — set covering, combinatorial auction, capacitated facility location, and maximum independent set problems — to examine its performance on a wide variety of instances. For this purpose, we generate 10,000 instances per problem type for sampling the training data, 2,000 for validation data, and another 2,000 for testing data. Moreover, for each problem we generate 20 evaluation instances for three different sizes (i.e., 60 instances per problem type). The smallest (easy), are the same size as the training instances. The larger instances (medium and hard), are used to evaluate the generalization ability of our approach to instances of arbitrary size.

In the remainder of this discussion, we will outline the four different problem types and how they are simulated. All simulation algorithms are implemented following Gasse et al. (2019), with evaluation instance sizes scaled to our specific implementation. That is, the sizes are adjusted such that the hardest problems are usually solvable within one hour using our vanilla solver. First, Subsection 4.1.1 discusses the set covering problem instances. Subsection 4.1.2

details the combinatorial auction problem instances, and Subsection 4.1.3 introduces the capacitated facility location problem instances. Finally, Subsection 4.1.4 discusses the maximum independent set problem instances.

4.1.1 Set covering problem

In a basic set covering problem, we are given a set of elements $\mathcal{U}$, and a collection of sets $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m\}$. The goal is to find a minimum-cardinality subset of sets from $\mathcal{S}$, whose union equals $\mathcal{U}$. In our experiments, we consider a weighted set covering problem, where we are also given the cost that is associated with each set. In this case, the goal is to find a minimum-cost subset, whose union again equals $\mathcal{U}$. This problem can therefore be formulated as a minimization problem with objective function $\boldsymbol{c}^T\boldsymbol{x}$, where $\boldsymbol{c}$ denotes a vector of costs, and $\boldsymbol{x}$ is a binary vector which is one at index i if and only if the corresponding set is chosen. The constraints stipulate that for every element, we have to choose at least one of the sets that contains that particular element.

These instances are simulated based on a scheme that was proposed by Balas and Ho (1980). For this purpose, we first randomly generate a binary coefficient matrix of 5% density (i.e., 5% non-zero). The rows in this matrix represent the elements, whereas the columns represent the sets. An element at position (i, j) of the matrix then equals one if element i is in set j, and zero otherwise. The matrix is generated subject to the requirement that every row has at least one non-zero entry, and each column has at least two. That is, every element appears in at least one set and every set contains at least two elements. Moreover, costs are randomly drawn from $\{1, 2, \ldots, 100\}$. We train and test the model using instances with 500 rows and 1000 columns $(500 \times 1,000)$, and we evaluate our approach on instances with dimensions $500 \times 1,000$ (easy), $700 \times 1,000$ (medium), and $900 \times 1,000$ (hard).

4.1.2 Combinatorial auction problem

A combinatorial auction is a multi-object auction, where bidders can bid on bundles of goods. In some cases this type of auction is attractive, because it allows the bidders to adjust bids based on cross-price elasticity relationships between individual products. For instance, a bidder might be willing to pay amount x for a table and amount y for a set of chairs, but an amount $z > x + y$ for the complete set. The combinatorial optimization problem that naturally arises from this type of auction is how a seller can maximize his revenue, given a set of bids for different bundles of goods. This type of problem is aptly called a combinatorial auction problem. The problem can be formulated as a maximization problem with objective function $\boldsymbol{c}^T\boldsymbol{x}$, where $\boldsymbol{c}$ is a vector of bid values and $\boldsymbol{x}$ is a binary vector that equals one at index i if and only if the corresponding bid is chosen. The constraints state that for every item, we can only choose one of the bids that contain it.

The instances of this type are simulated based on the scheme that was suggested by Leyton-Brown, Pearson, and Shoham (2000). First, we randomly draw common resale values for each good from $U(1, 100)$, where $U(a, b)$ denotes the uniform distribution over the interval $[a, b)$. Then, we generate compatibilities between goods (the prior probability that they appear together in a bundle). For each bidder, we randomly draw a per-item interest from $U(0, 1)$, which we then multiply by 50 to obtain a bidder's deviation from the common resale value. Then, an initial bundle is constructed by taking a random item with probabilities proportional to the bidder's interests. Now, we randomly add more items to the bundle, with probabilities proportional to the item compatibilities with items already present in the bundle, and the bidder's private valuation of the items. At every iteration, we add another item with probability 0.7 and we stop at the latest when there are no more items left to choose from. The bid for this bundle is then computed by adding an additivity term $n^{1.2}$ to the bidder's private valuation of the items, where n denotes the size of the bundle. Finally, we generate a maximum of five bids that are substitutable for the original bid, subject to the constraint that each of them requests at least one good from the original bid. Bundles are substitutable if they fall within the bidder's budget ($1.5 \cdot \texttt{price}$) and above the minimum resale value ($0.5 \cdot \texttt{price}$), where `price` denotes the bidder's initial bid. Only one of the substitutable bids (including the initial bid) can be chosen by a bidder, and this is incorporated into the constraints. We train and test the model using instances with 100 items and 500 bids (100×500), and we evaluate our proposed approach on instances with dimensions 100×500 (easy), 150×750 (medium), and $200 \times 1,000$ (hard).

4.1.3 Capacitated facility location problem

In a capacitated facility problem, we are given a set of customers and a set of facilities. Each customer has a certain demand and cost of being served from each of the facilities (transportation costs), and each facility has a certain capacity and cost of opening (fixed costs). The goal is to serve all customers at minimum cost, subject to the facilities' capacity constraints. This problem can be formulated as a minimization problem with an objective function equal to the sum of transportation costs for each customer scaled by the fraction of demand that is served by a given facility, and the sum of the fixed costs for all open facilities. The constraints stipulate that every customer must be served and that a facility can only serve customers if it is open. The formulation can be tightened by adding an optional constraint that the capacity of all open facilities should be larger than or equal to the total demand. Moreover, we can add optional constraints that stipulate that any single customer can only be served by an open facility, which yields one constraint per customer and facility.

These instances are generated based on the scheme that was proposed by Cornuejols, Sridharan, and Thizy (1991). First, we randomly place customers and facilities on a 1x1 surface, to simulate a real-world map. Secondly, customer demands are randomly drawn from $\{5, 6, \ldots, 35\}$, facility capacities s_j are randomly drawn from $\{10, 11, \ldots, 160\}$, and the fixed costs are generated using the formula $u_1 + u_2 \cdot \sqrt{s_j}$, where u_1 and u_2 are randomly drawn from $\{0, 1, \ldots, 90\}$ and $\{100, 101, \ldots, 110\}$, respectively. Here, the formula for generating fixed costs includes the square root of supply to reflect economies of scale. Then, capacities are scaled to obtain a capacity to demand ratio of five. Finally, we multiply the Euclidean distance between each customer and facility by 10 to obtain the unit cost of serving a customer from a particular facility, which is then multiplied by the demand of that customer to obtain the total transportation costs. We train and test the model using instances with 100 customers and 100 facilities (100×100), and we evaluate our proposed approach on instances with dimensions 100×100 (easy), 150×150 (medium), and 200×200 (hard).

4.1.4 Maximum independent set problem

An independent set is a subset of nodes in a graph, for which there is no edge connecting any of the nodes in the set. The goal in a maximum independent set problem is to find the largest of such independent sets. The problem can therefore be formulated as a maximization problem with objective function $\boldsymbol{e}^T\boldsymbol{x}$, where $\boldsymbol{e}$ denotes a vector of ones and $\boldsymbol{x}$ is a binary vector that equals one at index i if and only if the corresponding node is selected. For the constraints, we start by noting that one can only select a single node from any two nodes that are connected by an edge. While this is a valid formulation for our problem, we can strengthen it by noting that if we have a clique $\mathcal{S}$ (a fully-connected set), any independent set can only pick at most one node from $\mathcal{S}$. For each clique, we can therefore add the constraint that we may only pick a single node from it, and remove all previously added constraints for edges in the clique. For this purpose, we greedily divide the graph into cliques. At each iteration, we create a new clique by picking a node that has not been assigned to a clique yet. Then, we get all its neighbors that are also not assigned and we pick the neighbor with the largest degree. For this node, we go through each of its own neighbors, adding a node to the clique if and only if all nodes that are already in the clique are connected to that neighboring node as well. This continues until all nodes are assigned to a clique.

To generate independent set instances, we in principle only need to generate a random graph. For this purpose, following Gasse et al. (2019), we generate a random graph using the Barabási-Albert algorithm that was introduced by Barabási and Albert (1999). This algorithm dynamically models a scale-free network, based on two principles:

- *Growth*: a graph starts with a small number of nodes (m_0) and at every time-step, a new node is added that connects to a certain number of nodes already present in the system (m).

- *Preferential attachment*: new nodes prefer to link to nodes that already have many connections. That is, the probability that a new node i will be connected to node j is proportional to the connectivity k_j of that particular node, such that

$$P(\boldsymbol{E}_{ij} = 1 \mid k_j) = \frac{k_j}{\sum_l k_l}, \tag{28}$$

  where $P(\boldsymbol{E}_{ij} = 1 \mid k_j)$ denotes the probability that node i links to node j, given the connectivity of node j.

In our implementation, following Gasse et al. (2019), we set $m_0 = m = 4$. We train and test the model using graphs of 500 nodes, and we evaluate our proposed approach on instances with 500 (easy), 800 (medium), and 1,100 (hard) nodes.

4.2 Sampling

To train and test the model, we sample expert decisions from our simulated training, validation, and testing instances, following the sampling scheme of Gasse et al. (2019). The expert simply computes the true bound improvement by tentatively adding each cut and reoptimizing the

LP relaxation, similar to strong branching. This expert decision is combined with a set of features to form a state-action pair, as described in Subsection 3.2. To generate a diverse set of samples, the expert is only queried at 5% of the separation rounds. Whenever the expert is not queried, the sampling agent falls back to the usual hybrid cut score to speed up the process. Note that only expert decisions are recorded, so state-action pairs for which the action is the true bound improvement. A time limit of five minutes is used for solving instances, again to ensure a diverse set of samples. During sampling, we create three sets for each problem — for training, validation, and testing. Each of these sets is drawn from its corresponding set of instances. That is, the training, validation, and testing sets of each problem are drawn from that problem's simulated training, validation, and testing instances, respectively. We sample 100,000 state-action pairs for each training set, 20,000 for each validation set, and another 20,000 for each testing set.

4.3 Testing the model's ability to rank cutting planes

With the training and validation set for each problem, we can train the model as described in Subsection 3.4. As suggested by Gasse et al. (2019), for each problem type, we train five models using the same training set, by providing the training algorithm with a different seed value for each model. These seeds are used for any stochastic operation during training, such as drawing 10% of the training set on each epoch for updating the weights and biases. With our specific training procedure, a different seed greatly affects the way that the model sees the training data, resulting in an entirely different model (although not completely independent). By considering five seeds, we can evaluate the stability of our proposed approach, without the need to sample 20 different datasets.

When each model has been trained using the simulated training sets (i.e., five models for each of the four problem types), the first step is to investigate whether the model actually does what we want it to do — correctly rank cuts based on expected bound improvement. This is basically conventional model testing, as is common in machine learning. Note that this experiment corresponds to our first subgoal, which is to construct and train a model that predicts bound improvement. For this purpose, we test the model using our testing set, and we report the average percentage of cuts that it ranked *completely correct*. That is, for a sample of 10 cuts, if it ranks all of them correctly but erroneously swaps cuts 6 and 7, then the accuracy would be 50%. In our cut selection approach, it is especially important to get the first part of the ranking correct, as this determines which cuts make it into the LP relaxation. That is, the first (highest) scores are used to filter lower quality cuts with parallelism. Moreover, the bound improvement of the last cuts is often negligible, which implies that last part of the ranking does not say much about a model's ability to score cut quality. Therefore, it makes sense to focus on the high-quality cuts, making this proposed metric an interesting and interpretable way to evaluate the model's ability to rank cuts. Note that the models have not seen the testing samples during training, allowing us to get an unbiased estimate of their generalization ability.

To establish a baseline, we compare the model's performance to a random ranking and SCIP's default hybrid cut score. While the hybrid cut score is not designed to predict bound improvement, it *is* designed to rank cuts based on their quality. If we assume that true bound improvement is a reasonable metric of individual cut quality, then we can directly compare the rankings that are provided by our GCNN-based approach to those that are returned by the hybrid cut score.

4.4 Evaluating the model's ability to improve branch and cut

Besides testing the ability of our model to rank cuts, it is interesting to know how our method performs when it is incorporated into a branch-and-cut algorithm. For this purpose, we first use simulated evaluation instances of each problem type to investigate whether our proposed approach improves upon the current state of the art (our second subgoal). For each problem, we analyze the performance of our approach by using the trained models to solve all evaluation instances (easy, medium, and hard) of the same problem type. For each of the five training seeds, a different seed is passed to SCIP to minimize the dependence between observations, in order to get as much information from our results as possible. For each problem, this means that we get 100 solutions per difficulty level (20 instances for 5 seeds). In the same way, for each problem and seed, the evaluation instances are solved using the default hybrid cut score, which allows us to compare the performance of our proposed approach to SCIP's default hybrid cut selector.

We compare the performance of these two cut selectors by reporting standard metrics for MIP benchmarking, similar to Gasse et al. (2019). For each problem and difficulty level, we report the 1-shifted geometric mean of solving time, the number of times that each selector was the fastest in solving a particular instance (wins), and the 1-shifted geometric mean over the number of branch-and-bound nodes that were required to solve the problems to optimality (node counts). Here, we deviate slightly from Gasse et al. (2019), as they report the arithmetic mean for the number of nodes. The shifted geometric mean was suggested by Achterberg (2007), to focus on ratios instead of totals. As we are mostly interested in ratios in MIP benchmarking (i.e., what is the relative improvement or deterioration), and individual MIP problems can vary a lot in terms of solving times and node counts, the shifted geometric mean is commonly reported in this type of research. For this reason, we also report the shifted geometric mean over the node counts, as the same reasoning for solving times applies to node counts as well. The shifted geometric mean is defined as

$$\gamma_s(x_1, x_2, \ldots, x_k) = \prod_{i=1}^{k} \left(\max\left(x_i + s, 1\right)\right)^{\frac{1}{k}} - s, \tag{29}$$

where γ_s denotes the s-shifted geometric mean, x_i denotes the observed value (e.g., solving time) for instance i, and k denotes the total number of instances that we compute the shifted geometric mean over. The 1-shifted geometric mean of solving time and number wins are two robust metrics to measure the performance of our approach in terms of *speed*. The 1-shifted geometric mean over the node counts is of interest as it is hardware independent and measures the performance of our approach in terms of *efficiency*. For a branch-and-bound algorithm, the process is considered to be relatively efficient if it requires comparatively few nodes to solve the problem to optimality, since it has then apparently been able to prune a relatively large part of the search space. For fair measurement, we only compute the number of wins over the instances that were solved within the time limit by at least one of the approaches, as there is no winner if both methods took more than an hour. Moreover, we only consider node counts for instances that were solved within the time limit by both methods, as forced termination leads to lower node counts if the gap to optimality is large, skewing the observed efficiency. All instances are solved using a solving time limit of one hour, and solving time is measured in terms of CPU time.

In addition to the experiments using simulated data, which were done in a controlled environment (i.e., with a vanilla solver) and on instances of the same type of problem, we would like to know how our proposed approach would perform in practice for a general-purpose MIP solver (our third subgoal). For this purpose, we incorporate it into a full-fledged solver by turning presolving restarts back on and using the default SCIP branching rule. We again impose a time limit of one hour. For each problem type, we select the model that performs best on the testing set (i.e., that has the lowest MSE), and we use this to solve a set of benchmark instances, as one would in practice. The set that we consider in this thesis is the MIPLIB 2010 benchmark test set for MIP problems, which is designed to span a wide variety of problems, and consists of 87 problem instances that are all solvable with modern solvers (Koch et al., 2011). Again, we report the 1-shifted geometric mean of solving time, the number of wins for each approach, and the 1-shifted geometric mean over the node counts, which are widely used metrics for MIP benchmarking.

5 Results

As mentioned before, we have three subgoals that we each test via a different experiment. To assess whether our proposed model is able to predict bound improvement (our first subgoal), we test its performance on a set of testing samples. The results of this analysis are discussed in Subsection 5.1. Our second and third subgoal evaluate the performance of our approach in a branch-and-cut algorithm. The results of these experiments are discussed in Subsection 5.2.

5.1 Testing the model's ability to rank cutting planes

To evaluate our model's ability to rank cutting planes, we test each problem type separately on a set of 20,000 testing samples, and we compare it to a random ranking and SCIP's default hybrid cut score. For each problem, Table 2 reports the average percentage of cuts that each ranking strategy ranked correctly, averaged over the five seeds. In addition to the mean, we report the sample standard deviation over these five seeds to give an indication of the model's stability. The sample standard deviation is defined as $\sqrt{\frac{1}{N-1}\left(x-\bar{x}\right)^2}$, where N denotes the number of samples, x denotes the variable of interest, and $\bar{x}$ denotes the mean of x. Note that the hybrid cut score does not involve any stochasticity, and therefore provides the same ranking for every seed, yielding a sample standard deviation of zero. The main takeaway from this table is that our GCNN appears to substantially improve upon random rankings, and that it outperforms the hybrid cut selection rule for each problem type. As we argued in Subsection 4.3, we assume that bound improvement is a reasonable measure of cut quality, which implies that we can directly compare the rankings that are provided by our GCNN to those of the

Table 2: The average percentage of times that each ranking strategy ranked a given percentage of cut candidates correctly, tested on the same problem type that was used for training.

Selector	SC	CA	CFL	MIS
Random	20.86 ± 0.11	19.83 ± 0.15	35.70 ± 0.18	10.88 ± 0.14
Hybrid	23.32 ± 0.00	29.39 ± 0.00	45.21 ± 0.00	13.40 ± 0.00
GCNN	$\mathbf{28.73} \pm 0.66$	$\mathbf{29.66} \pm 0.19$	$\mathbf{46.90} \pm 0.50$	$\mathbf{16.33} \pm 0.37$

Note. SC: set covering; CA: combinatorial auction; CFL: capacitated facility location; MIS: maximum independent set. The highest accuracy is indicated in boldface.

hybrid cut score.

The results in Table 2 show that on average, the models get the order exactly right for the first 30% of cuts. As we mentioned in Subsection 4.3, the first part of the ranking is the most important in our specific implementation, if we assume that low-quality cuts are filtered out of the candidate pool. If we were to assume the converse (i.e., most cuts are added), then in fact the ranking does not matter much at all. These results imply that we have achieved our first subgoal, and that we have succeeded at constructing and training a model that predicts bound improvement, at least to a certain extent. While there seems to be no apparent reason for the differences in performance across problem types, it appears to be correlated with the performance of the random ranking. This implies that this difference is caused by something structural, such as the number of cuts in each sample, as opposed to something that influences how predictable the rankings are. For instance, the capacitated facility location instance samples contain comparatively little cuts (20–35% relative to the other problems), possibly making it easier to get a large fraction of the ranking correct.

5.2 Evaluating the model's ability to improve branch and cut

Now that we have asserted that our model improves upon SCIP's default hybrid cut score in terms of ranking cuts based on bound improvement, another important factor comes into play — inference speed. A GCNN has relatively slow inference speed compared to a simple heuristic, simply because it involves many matrix multiplications while the hybrid cut score only requires computing a weighted sum of three vectors. Moreover, we need to extract extra features compared to the hybrid cut score, including statistics on the problem's constraints and variables. These computations are all quite fast, but performing them at every separation round might add up. However, as our model appears to provide a better ranking than the hybrid cut score, it might yield a good compromise between inference speed and accuracy. To evaluate whether this is the case, we first investigate whether it improves upon the state-of-the art and whether this generalizes to larger instances (our second subgoal), by comparing it to SCIP's hybrid cut selector on simulated instances. The results of this analysis are discussed in Subsection 5.2.1. Finally, for our approach to be effective in a general-purpose MIP solver, it must also be able to generalize to instances of arbitrary type (our third subgoal). For this purpose, we evaluate its performance on a set of benchmark instances. The results of this analysis are outlined in Subsection 5.2.2.

5.2.1 Simulated data

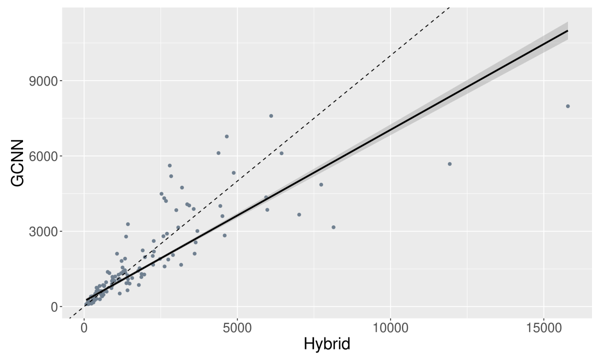
Table 3: The 1-shifted geometric mean of solving time, the number of wins, and the 1-shifted geometric mean of the node counts for each cut selector and training problem type, evaluated on simulated data of the same type.

Selector	Easy			Medium			Hard		
	Time	Wins	Nodes	Time	Wins	Nodes	Time	Wins	Nodes
				Set covering					
Hybrid	**14.78** ± 25%	**100**	647 ± 25%	**237.36** ± 21%	**90**	2212 ± 27%	**554.98** ± 16%	**42**	**4119** ± 9%
GCNN	336.84 ± 27%	0	**638** ± 23%	2416.51 ± 10%	0	**2064** ± 17%	3380.57 ± 3%	0	4192 ± 16%
				Combinatorial auction					
Hybrid	**4.61** ± 29%	**100**	412 ± 31%	**59.90** ± 25%	**99**	**3403** ± 27%	**557.60** ± 18%	**89**	8506 ± 21%
GCNN	39.27 ± 31%	0	**399** ± 25%	701.07 ± 30%	0	3420 ± 29%	3529.78 ± 2%	0	**6805** ± 32%
				Capacitated facility location					
Hybrid	**45.07** ± 11%	**100**	171 ± 16%	**253.62** ± 17%	**98**	646 ± 20%	**679.50** ± 12%	**72**	712 ± 12%
GCNN	200.04 ± 14%	0	**170** ± 17%	789.29 ± 19%	0	**640** ± 16%	1484.94 ± 13%	1	**684** ± 10%
				Maximum independent set					
Hybrid	**10.61** ± 22%	**98**	**149** ± 44%	**101.59** ± 22%	**99**	**1365** ± 28%	**1113.00** ± 20%	**81**	4086 ± 34%
GCNN	24.71 ± 29%	2	155 ± 35%	303.33 ± 28%	1	1377 ± 35%	2585.93 ± 13%	0	**4049** ± 31%

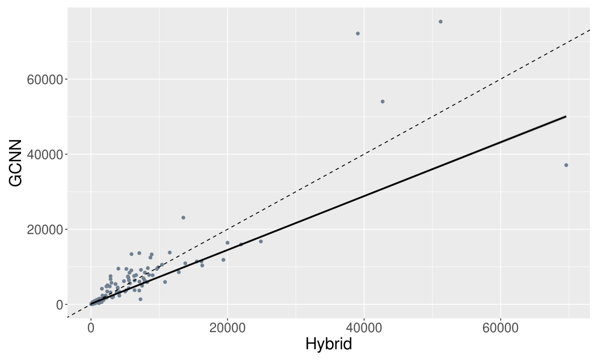
Note. The best score is indicated in boldface.

To evaluate how our proposed approach performs in a branch-and-cut algorithm, we compare its speed and efficiency to SCIP's default hybrid cut selector on our simulated evaluation instances. For this purpose, we evaluate each model on instances of the same problem type that was used for training. Table 3 reports the 1-shifted geometric mean of solving time (time), the number of times that each selector was the fastest in solving a particular instance (wins), and the 1-shifted geometric mean of the number of nodes that it took each cut selector to arrive at an optimal solution (nodes). Moreover, we report the average per-instance sample standard deviation of solving time and node counts. For instances that were solved only once, we set the sample standard deviation to zero. The results in this table show a clear winner in terms of solving time, which is the default hybrid cut selector. It is substantially faster and it wins in almost all cases. As we mentioned before, there is this trade-off between inference speed and accuracy. In this case, it appears that the investment that we make in terms of accuracy by using the GCNN was not worth the additional computational costs, tipping the scale in the hybrid cut selector's favor. In terms of node counts, the results do not indicate a clear winner, which might be due to the fact that our GCNN does not drastically improve the cut ranking. That is, in most cases we can expect the performance of these methods to be similar in terms of efficiency, especially for easy instances. The 1-shifted geometric mean is robust to very high node counts, by downplaying the importance of large values. However, we might expect an improved cut selection policy to do especially well on difficult instances, which are associated with high node counts. This implies that this single metric might not provide us with the full picture, and that this does not rule out the fact that our proposed approach improves upon the hybrid cut selector in terms of efficiency.

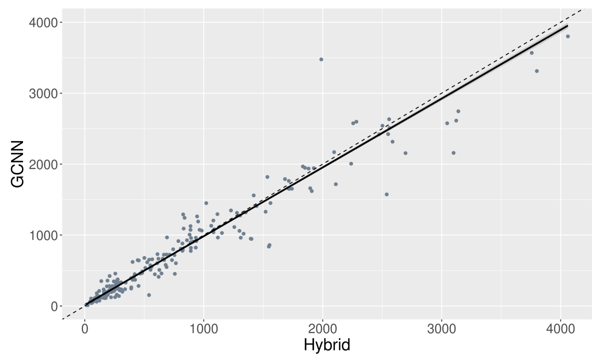
To further investigate the performance of our proposed approach in terms of efficiency, we need to find a good way to compare the node counts. For this purpose, Figure 12 plots the node counts of our proposed GCNN cut selector against those of the hybrid cut selector. These



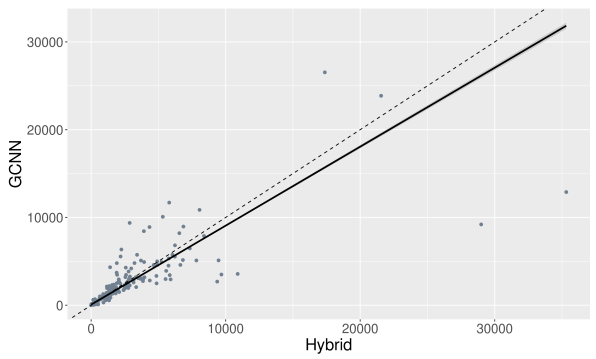
(a) Set covering



(b) Combinatorial auction



(c) Capacitated facility location



(d) Maximum independent set

Figure 12: The node counts of the GCNN selector plotted against those of the hybrid cut selector for each training problem type, evaluated on simulated data of the same type. The dashed lines depict 45° lines, the solid lines depict MM-estimators, and the shaded region surrounding each of those lines depicts the corresponding 95% confidence interval.

plots pool all solving attempts for each problem type, so that we can investigate the effect of our proposed approach for instances of varying difficulty. If the selectors are similar in terms of efficiency, we expect the node counts to be similar as well, which implies that these counts should approximately lie on the 45° line (depicted in the figure as a dashed line). If our proposed approach is more efficient on average, we expect its corresponding node counts to be lower on average than those of the hybrid cut selector (i.e., below the 45° line). To investigate whether this is the case, we can fit a regression line on the data points, and examine where it lies relative to the 45° line. However, as node count can be quite noisy (i.e., there might be outliers) and standard linear regression is prone to outliers, we opt for a robust regression method. For this purpose, we fit an MM-estimator, which is a robust regression method proposed by Yohai (1987). These estimators combine a high breakdown point with high efficiency, and have been shown to be among the most effective robust regression methods (Yu & Yao, 2017). We compute these estimators using the well-known MASS package for R (Venables & Ripley, 2002), which we plot using the popular ggplot2 package (Wickham, 2016). In each of the plots (some more than others), we now see the pattern that we would expect. For very easy instances, our proposed approach is approximately as efficient as the hybrid cut selector. As the instances get harder, our proposed GCNN approach appears to improve upon the hybrid cut selector in terms of efficiency. Therefore, although our proposed approach does not yield any speedups, it appears to improve the efficiency of the branch-and-cut algorithm. There is no obvious explanation for

Table 4: The 1-shifted geometric mean of solving time, the number of wins, and the 1-shifted geometric mean of the node counts for each cut selector and training problem type, evaluated on MIPLIB 2010 benchmark data of arbitrary type.

Selector	SC			CA			CFL			MIS		
	Time	Wins	Nodes	Time	Wins	Nodes	Time	Wins	Nodes	Time	Wins	Nodes
Hybrid	**366.39**	**59**	1432	**278.65**	**54**	**1237**	**325.73**	**56**	**1221**	**296.35**	**61**	**1084**
GCNN	887.02	5	**1197**	811.38	3	1720	881.76	1	1417	1005.15	0	1124

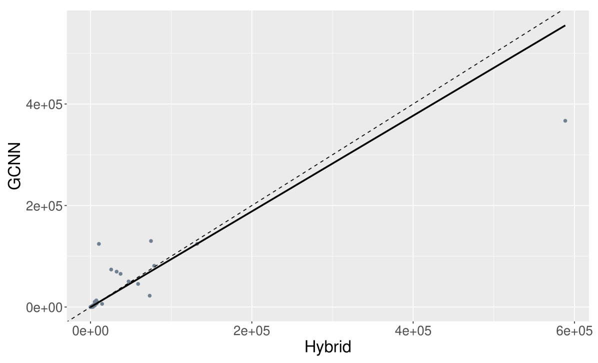
Note. SC: set covering; CA: combinatorial auction; CFL: capacitated facility location; MIS: maximum independent set. The best score is indicated in boldface.

the difference in relative performance between problem types, as it seems neither correlated with the ranking performance (absolute or relative to the hybrid cut score) nor the average number of cuts per sample.

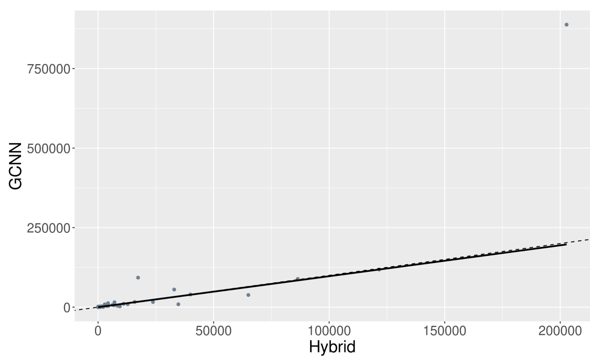
5.2.2 Benchmark data

As a final experiment, we analyze the performance of our approach when it is implemented in a full-fledged solver for instances of arbitrary type. To this end, we evaluate the models that were trained for each problem type separately on generic-type MIPLIB 2010 benchmark instances. Table 4 again reports the 1-shifted geometric mean of time, the number of wins, and the 1-shifted geometric mean of node counts. Note that we do not report the per-instance sample standard deviation in this case, as only the best model is selected to solve the instances (i.e., they are only solved once per problem type). Again, our proposed method is substantially slower than the hybrid cut selector, which is unsurprising since this was the case for our evaluation instances too. The node counts appear to be slightly more in favor of the hybrid cut selector, although they do not indicate a clear winner. Again, the 1-shifted geometric mean of the node counts might not capture the full picture, and some extra investigation might prove useful.

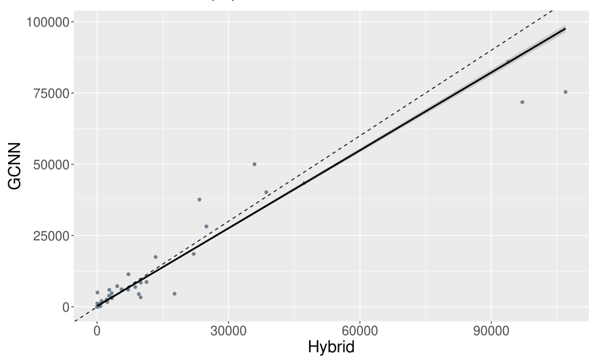
To take a closer look at our method's performance in terms of efficiency, Figure 13 again plots the node counts of our GCNN-based approach against the those of the hybrid cut selector. As before, we depict the MM-estimator and analyze its position relative to the 45° line. These plots show that in three out of four cases, the performance of our proposed approach in terms of efficiency generalizes to instances of arbitrary type, to a certain extent. That is, our proposed approach appears to improve upon the hybrid cut selector in terms of efficiency for all models but the one that was trained on maximum independent set problem instances. This is interesting as each model was only trained on instances of a single problem type, which implies that even better generalization ability might be attained by training a model on a diverse set of instances. Again, the difference in relative performance between problem types used for training has no obvious explanation, as it seems neither correlated with the ranking performance (absolute or relative to the hybrid cut score), the average number of cuts per sample, nor the performance on simulated instances. This could be by chance, or it may suggest that some simulated problem types (e.g., capacitated facility location) are more representative of the cuts in generic-type problems than others (e.g., combinatorial auction). One could take this into consideration when training a model for use in a general-purpose MIP solver, or one could consider training on a diverse set of instances.



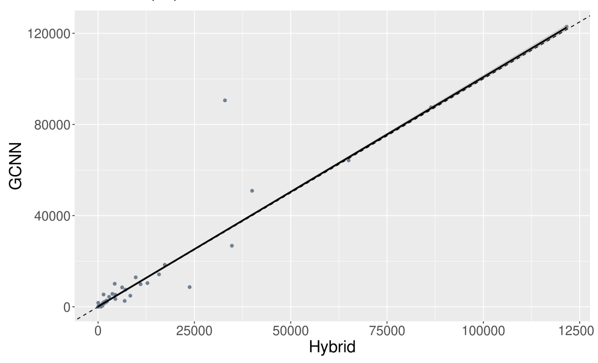
(a) Set covering



(b) Combinatorial auction



(c) Capacitated facility location



(d) Maximum independent set

Figure 13: The node counts of the GCNN selector plotted against those of the hybrid cut selector for each training problem type, evaluated on MIPLIB 2010 benchmark data of arbitrary type. The dashed lines depict 45° lines, the solid lines depict MM-estimators, and the shaded region surrounding each of those lines depicts the corresponding 95% confidence interval.

6 Conclusions

In this thesis, we explored the possibility of improving cut selection in a branch-and-cut framework by learning to score cuts based on their bound improvement. For this purpose, we formulated three subgoals — construct and train a model that predicts bound improvement, investigate whether it improves upon the current state of the art, and evaluate whether this performance generalizes to instances of arbitrary type. In this final part of the thesis, Subsection 6.1 first briefly summarizes our findings by considering each of these subgoals separately. Subsection 6.2 then outlines some possible limitation of our proposed approach. Finally, Subsection 6.3 outlines possible avenues of future research.

6.1 Summary

The most fundamental subgoal of this thesis was to construct and train a model that predicts bound improvement. For this purpose, we implemented a GCNN that was previously used in the context of variable selection. We then trained this model via imitation learning, using true bound improvement as an expert decision rule. We trained and tested the model on four different problem types for five different seeds, using simulated data. The testing results indicate that our model is able to predict bound improvement to a certain extent, as it improves upon both random selection and SCIP's hybrid cut score in ranking cuts based on this metric. Although the hybrid cut score is not designed to predict bound improvement, we argued that it can be used for comparing the rankings, as it is designed to rank cuts based on their predicted quality.

Our second subgoal was to investigate whether our GCNN-based cut selection policy improves upon the current state of the art. To this end, we compared its performance to SCIP's default hybrid cut selector on simulated evaluation instances. For each problem, we solved the evaluation instances with each of the five models (i.e., one for each seed). Although our proposed approach does not yield any improvements in terms of solving time, the results indicate that it improves the efficiency of the branch-and-cut algorithm. That is, for difficult instances it appears to decrease the number of nodes that the algorithm requires to reach an optimal solution. This finding is interesting, as it shows that a model that is trained to predict bound improvement via imitation learning has the potential of enhancing the cut selection policy of a branch-and-cut algorithm.

Our third and final subgoal was to examine whether the performance of our approach generalizes to instances of arbitrary type. For this purpose, we evaluated its performance on a set of real-world benchmark instances. For each problem, we solved this benchmark set by using the model that performed best in model testing, as one would in practice. Moreover, while the earlier experiments were all done using a vanilla solver, we made these experiments as realistic as possible by incorporating our cut selection strategy in a full-fledged solver. Again, our proposed approach does not yield any improvements in terms of solving time, but the results indicate that its performance in terms of efficiency generalizes to instances of arbitrary type, to a certain extent. This finding again highlights the potential of a model that is trained to predict bound improvement via imitation learning, as its performance appears to generalize to instances of other types, even though it was only trained on a single type of problem.

6.2 Limitations

In designing a method for cut selection, we face a trade-off between inference speed and prediction accuracy. For instance, SCIP's default hybrid cut selector uses a simple heuristic to score cuts, which is very fast but it might not be the best measure of cut quality. Our approach appears to improve upon this hybrid cut selector in terms of accuracy, but it seems that this gain in accuracy is not worth the extra computational burden.

A fundamental assumption of our research is that bound improvement is a good measure of cut quality, as our model mimics this scoring strategy and we use it to evaluate its ranking performance. As Bengio et al. (2021) noted, the performance of a model that is trained via imitation learning is limited by the performance of the expert. Therefore, it is important that bound improvement actually captures cut quality, in order for our model to perform well. We argue that bound improvement might not be perfect, but that it is the best we have for evaluating the quality of a single cut. Our first argument in favor of this proposition is that bound improvement captures multiple aspects of cut quality. Deep cuts that are well-oriented can be expected to attain a large bound improvement, which implies that it is inherently related to efficacy and objective parallelism. SCIP's default hybrid cut score includes these two metrics, supplemented by integral support. As shown by Wesselmann and Suhl (2012), high integral support is preferable, and can be very useful in a cut selection policy. A cut with high integral support places a constraint on many (possibly fractional) integer-constrained variables. One can argue that high integral support is preferable, because this constrains the LP relaxation's freedom to choose these variables, possibly leading to a large bound improvement. Therefore, bound improvement captures all metrics that are included in the hybrid cut score in some way, although it might weigh them differently. One thing that neither bound improvement nor SCIP's hybrid cuts score take into account, is the fact that dense cuts (i.e., with high support), are likely to slow down computations. A more direct approach to measuring cut quality is to directly measure its associated reduction in solving time. This can be computed on the bag level (e.g., Huang et al., 2022), but for a single cut this difference in solving time is likely to be negligible. To the best of our knowledge, there is no superior alternative to bound improvement for measuring individual cut quality (yet). Therefore, while bound improvement is not perfect, it is our best option.

6.3 Future research

With the limitations of our suggested approach in mind, we can identify possible ways of moving forward. The first possibility is to improve our proposed approach, by increasing the model's inference speed and its predictive performance. In tuning the model's hyperparameters, we have implemented what Yang and Shami (2020) refer to as “grad student descent”. That is, we have taken the hyperparameters that worked well for Gasse et al. (2019), and slightly adapted them to obtain satisfactory results within our limited time frame. Therefore, there might still be substantial room for improvement, both in terms of inference speed and prediction accuracy. For instance, decreasing the initial learning rate from 0.001 to 0.0001 has tremendously improved the model's performance, so careful tuning might result in even better results. Moreover, we have mostly left the composition of the network untouched, as it proved effective in the work of Gasse et al. (2019). However, the actual implementation has changed quite a bit (e.g., we added another convolution), and the performance of our approach might therefore benefit from tuning the model architecture as well. For instance, we might consider reducing the number of nodes for each layer to speed up inference or even remove some layers altogether. Moreover, as our output is constrained to be non-negative, it might be interesting to consider other types of output activation functions. Now we use an identity activation function, which can take values below zero. As mentioned in Subsection 3.3, a ReLU activation was problematic due to the dying ReLU problem. However, alternatives exist such as a Leaky ReLU, which prevents the gradient from dropping to zero by employing a small slope for negative values. It might also be worthwhile to investigate the effect of using these alternative activation functions throughout the entire network, as this could allow for a larger learning rate. To speed up inference, one might also consider creating a lightweight implementation of the model, possibly in C.

A second possibility is to investigate possible adaptations of our approach. For instance, one might consider only applying the GCNN-based cut selector only when there are many cuts to select. Moreover, it could be interesting to investigate the use of reinforcement learning for fine-tuning the model's parameters after imitation learning, which might further improve the model's predictive performance. As a reward function one could again use the actual bound improvement that is attained after adding a cut to the LP relaxation, but now in the context of a reinforcement learning policy such as the one that was employed by Tang et al. (2020). Moreover, it might be interesting to examine whether training a model on a diverse set of instances might improve its generalization ability. While the results indicate that models that are trained on a single type of instance could already generalize to instances of arbitrary type, the performance appears to drop when a model is evaluated on generic instances. To mitigate this issue, one might consider training a model on a diverse set of instances. In other fields, especially natural language processing, there have been so-called domain-adversarial approaches that aim to improve a model's generalization ability, by explicitly training a model to be incapable of distinguishing between different domains. A similar approach might work in this case, for instance by training it on all four types of simulated instances, rewarding the model's predictive performance and punishing its ability to distinguish between different problem types. By using this twofold approach, the model is forced to identify patterns that are common across problem types. Such an approach could possibly allow the model to generalize even better, which is crucial if a policy is to perform well in a general-purpose MIP solver. However, it could also be the case that distinguishing between problem types is useful for identifying high-quality cuts. Therefore, further research is required before we can establish whether domain adversarial learning is actually helpful in this context.

Finally, it might also be worthwhile to investigate other (smaller) models that have faster inference speed, such as simple linear regression. The results in this thesis indicate that a model that is trained to predict bound improvement via imitation learning might improve the efficiency of a branch-and-cut algorithm. As our imitation learning approach is not limited to a GCNN model, it can be readily applied to other machine learning methods as well. This could be an intriguing direction of future research, as smaller models could perhaps attain similar performance in terms of efficiency, without sacrificing too much inference speed.

In conclusion, our work has shown that a model that is trained to predict bound improvement might improve the efficiency of a branch-and-cut algorithm. In particular, our imitation learning approach has proven to be a viable alternative to other methods of training models for cut selection. Imitation learning can be readily applied to any other supervised machine learning method, which means that our findings emphasize the potential of machine learning for cut selection. Therefore, while our approach in its current implementation might not be viable in practice, it highlight a promising direction of future research.

Acknowledgments

This work was carried out on the Dutch national e-infrastructure with the support of SURF Cooperative.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., . . . Zheng, X. (2016). TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv [cs.DC]*.

Achterberg, T. (2007). *Constraint integer programming* (Doctoral thesis). Technische Universität Berlin. Berlin.

Achterberg, T. (2009). SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, *1*(1), 1–41.

Achterberg, T., Koch, T., & Martin, A. (2005). Branching rules revisited. *Operations Research Letters*, *33*(1), 42–54.

Alvarez, A. M., Louveaux, Q., & Wehenkel, L. (2017). A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, *29*(1), 185–195.

Andreello, G., Caprara, A., & Fischetti, M. (2007). Embedding $\{0, \frac{1}{2}\}$-cuts in a branch-and-cut framework: A computational study. *INFORMS Journal on Computing*, *19*(2), 229–238.

Balas, E., Ceria, S., Cornuéjols, G., & Natraj, N. (1996). Gomory cuts revisited. *Operations Research Letters*, *19*(1), 1–9.

Balas, E., Ceria, S., & Cornuéjols, G. (1993). A lift-and-project cutting plane algorithm for mixed 0–1 programs. *Mathematical Programming*, *58*(1), 295–324.

Balas, E., & Ho, A. (1980). Set covering algorithms using cutting planes, heuristics, and subgradient optimization: A computational study. In M. W. Padberg (Ed.), *Combinatorial optimization* (pp. 37–60). Springer.

Baltean-Lugojan, R., Bonami, P., Misener, R., & Tramontani, A. (2019). *Selecting cutting planes for quadratic semidefinite outer-approximation via trained neural networks* (tech. rep.). Optimization Online.

Barabási, A.-L., & Albert, R. (1999). Emergence of scaling in random networks. *Science*, *286*(5439), 509–512.

Bengio, Y., Lodi, A., & Prouvost, A. (2021). Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, *290*(2), 405–421.

Bestuzheva, K., Besançon, M., Chen, W.-K., Chmiela, A., Donkiewicz, T., Van Doornmalen, J., Eifler, L., Gaul, O., Gamrath, G., Gleixner, A., Gottwald, L., Graczyk, C., Halbig, K., Hoen, A., Hojny, C., Van der Hulst, R., Koch, T., Lübbecke, M., Maher, S. J., . . . Witzig, J. (2021). *The SCIP Optimization Suite 8.0* (tech. rep.). ZIB.

Bonami, P., Lodi, A., & Zarpellon, G. (2018). Learning a classification of mixed-integer quadratic programming problems. *Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2018)*, 595–604.

Chan, T. F., Golub, G. H., & LeVeque, R. J. (1982). Updating formulae and a pairwise algorithm for computing sample variances. *International Conference on Computational Statistics (COMPSTAT 1982)*, 30–41.

Coniglio, S., & Tieves, M. (2015). On the generation of cutting planes which maximize the bound improvement. *International Symposium on Experimental Algorithms (SEA 2015)*, 97–109.

Cornuejols, G., Sridharan, R., & Thizy, J. M. (1991). A comparison of heuristics and relaxations for the capacitated plant location problem. *European Journal of Operational Research*, *50*(3), 280–297.

Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., & Song, L. (2017). Learning combinatorial optimization algorithms over graphs. *Neural Information Processing Systems (NIPS 2017)*, 6351–6361.

Dey, S. S., & Molinaro, M. (2018). Theoretical challenges towards cutting-plane selection. *Mathematical Programming*, *170*(1), 237–266.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271.

Gasse, M., Chételat, D., Ferroni, N., Charlin, L., & Lodi, A. (2019). Exact combinatorial optimization with graph convolutional neural networks. *Neural Information Processing Systems (NeurIPS 2019)*, 15580–15592.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural message passing for quantum chemistry. *International Conference on Machine Learning (PMLR 70)*, 1263–1272.

Gleixner, A., Bastubbe, M., Eifler, L., Gally, T., Gamrath, G., Gottwald, R. L., Hendel, G., Hojny, C., Koch, T., Lübbecke, M. E., Maher, S. J., Miltenberger, M., Müller, B., Pfetsch, M. E., Puchert, C., Rehfeldt, D., Schlösser, F., Schubert, C., Serrano, F., ... Witzig, J. (2018). *The SCIP Optimization Suite 6.0* (tech. rep.). ZIB.

Gleixner, A., Eifler, L., Gally, T., Gamrath, G., Gemander, P., Gottwald, R. L., Hendel, G., Hojny, C., Koch, T., Miltenberger, M., Müller, B., Pfetsch, M., Puchert, C., Rehfeldt, D., Schlösser, F., Serrano, F., Shinano, Y., Viernickel, J. M., Vigerske, S., ... Witzig, J. (2017). *The SCIP Optimization Suite 5.0* (tech. rep.). ZIB.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

He, H., Daumé III, H., & Eisner, J. M. (2014). Learning to search in branch and bound algorithms. *Neural Information Processing Systems (NIPS 2014)*.

Huang, Z., Wang, K., Liu, F., Zhen, H.-L., Zhang, W., Yuan, M., Hao, J., Yu, Y., & Wang, J. (2022). Learning to select cuts for efficient mixed-integer programming. *Pattern Recognition*, *123*, 108353.

Hussein, A., Gaber, M. M., Elyan, E., & Jayne, C. (2017). Imitation learning: a survey of learning methods. *ACM Computing Surveys*, *50*(2).

Khalil, E., Le Bodic, P., Song, L., Nemhauser, G., & Dilkina, B. (2016). Learning to branch in mixed integer programming. *AAAI Conference on Artificial Intelligence*, 724–731.

Khalil, E. B., Dilkina, B., Nemhauser, G. L., Ahmed, S., & Shao, Y. (2017). Learning to run heuristics in tree search. *International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 659–666.

Kingma, D. P., & Ba, J. L. (2015). Adam: A method for stochastic optimization. *International Conference on Learning Representations (ICLR 2015)*.

Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R. E., Danna, E., Gamrath, G., Gleixner, A. M., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D. E., & Wolter, K. (2011). MIPLIB 2010: Mixed integer programming library version 5. *Mathematical Programming Computation*, *3*(2), 103–163.

Kruber, M., Lübbecke, M. E., & Parmentier, A. (2017). Learning when to use a decomposition. *Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2017)*, 202–210.

Kuhn, H. W. (1955). The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, *2*(1-2), 83–97.

Leyton-Brown, K., Pearson, M., & Shoham, Y. (2000). Towards a universal test suite for combinatorial auction algorithms. *Electronic Commerce (EC 2000)*, 66–76.

Maher, S., Miltenberger, M., Pedroso, J. P., Rehfeldt, D., Schwarz, R., & Serrano, F. (2016). PySCIPOpt: Mathematical programming in Python with the SCIP Optimization Suite. *International Congress on Mathematical Software (ICMS 2016)*, 301–307.

Mitchell, J. E. (2002). Branch-and-cut algorithms for combinatorial optimization problems. In P. M. Pardalos & M. G. C. Resende (Eds.), *Handbook of applied optimization* (pp. 65–77). Oxford University Press.

Nocedal, J., & Wright, S. J. (2006). *Numerical optimization* (2nd ed.). Springer.

Selsam, D., Lamm, M., Bünz, B., Liang, P., De Moura, L., & Dill, D. L. (2019). Learning a SAT solver from single-bit supervision. *International Conference on Learning Representations (ICLR 2019)*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd ed.). MIT Press.

Tang, Y., Agrawal, S., & Faenza, Y. (2020). Reinforcement learning for integer programming: learning to cut. *International Conference on Machine Learning (PMLR 119)*, 9367–9376.

Venables, W. N., & Ripley, B. D. (2002). *Modern applied statistics with S* (4th ed.). Springer.

Wesselmann, F., & Suhl, U. H. (2012). *Implementing cutting plane management and selection techniques* (tech. rep.). University of Paderborn.

Wickham, H. (2016). *ggplot2: Elegant graphics for data analysis* (1st ed.). Springer.

Wolsey, L. A. (2021). *Integer programming* (2nd ed.). Wiley.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2021). A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, *32*(1), 4–24.

Yang, L., & Shami, A. (2020). On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, *415*, 295–316.

Yohai, V. J. (1987). High breakdown-point and high efficiency robust estimates for regression. *The Annals of Statistics*, *15*(2), 642–656.

Yu, C., & Yao, W. (2017). Robust linear regression: A review and comparison. *Communications in Statistics - Simulation and Computation*, *46*(8), 6261–6282.

Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., & Sun, M. (2020). Graph neural networks: A review of methods and applications. *AI Open*, *1*, 57–81.

Appendices

The following appendices briefly outline some additional information that is added just for completeness. First, Appendix A discusses the sampling statistics, which provide some additional information on the composition of our datasets. Then, Appendix B states the running times, which may be interesting for those who wish to build upon our work. Finally, Appendix C gives a brief description of all files in our GitHub repository.

A Sampling statistics

First, we report the sampling statistics for the training, validation, and testing sets.

Table A1: The sampling statistics for each problem type.

Instances	Training	Validation	Testing
	Set covering		
Total	3184	653	606
Unique	2725	556	518
	Combinatorial auction		
Total	15225	3269	2839
Unique	7813	1584	1543
	Capacitated facility location		
Total	49756	10306	10320
Unique	9933	1985	1991
	Maximum independent set		
Total	30000	5819	5710
Unique	9480	1887	1895

Table A1 presents the total and unique number of instances that were required to sample the given set. Recall that we generated 10,000 instances for training and each training set consists of 100,000 samples, and that we generated 2,000 instances for validation and testing and the validation and testing sets both contain 20,000 samples.

B Running times

Another thing that might be interesting, especially for reproducing the experiments, is the running time for each component. Table B1 depicts the estimated running time per job in CPU hours (i.e., on a single CPU core) for all jobs except training and testing, where we display regular wall-clock time for one model. Note that these are rough estimates, but they can be used as an indication of how long each job will take. As mentioned before, all sampling and evaluation is run on a single thin node of the Dutch national supercomputer Snellius, which consists of two AMD Rome 7H12 CPUs with 64 cores each at 2.6GHz. Model training and

Table B1: Estimated number of hours for each job.

Job	Time (h)							
	SC		CA		CFL		MIS	
	Train	Other	Train	Other	Train	Other	Train	Other
Instance generation	2.37							
Sampling	92.37	24.66	46.22	11.16	1021.19	204.92	132.48	31.20
Training	11.90*		4.01*		13.35*		6.13*	
Testing	0.03*		0.01*		0.07*		0.01*	
Evaluating	1571.34							
Benchmarking	758.19							
Summarizing	0.00							

Note. SC: set covering; CA: combinatorial auction; CFL: capacitated facility location; MIS: maximum independent set.
* Wall-clock time per model.

testing is done on a machine with an Intel i7-11800H CPU with 8 cores and 16 threads at 2.3GHz and an NVIDIA GeForce RTX 3050 Ti Laptop GPU with 4GB of memory.

C Code description

While an extensive README is available at our GitHub repository, for completeness we very briefly outline the code that we used to conduct the experiments. All code is available at https://github.com/stefanvanberkum/gcnn-cut-selector. The following files are included in the repository:

- `data_collector.py`: This module is used for sampling expert state-action pairs.
- `instance_generator.py`: This module is used for randomly generating set covering, combinatorial auction, capacitated facility location, and maximum independent set problem instances.
- `main.py`: This is the main execution environment, from which all experiments can be run.
- `model.py`: This module provides the GCNN model functionality.
- `model_benchmarker.py`: This module is used for benchmarking our proposed approach.
- `model_evaluator.py`: This module is used for evaluating our proposed approach.
- `model_tester.py`: This module is used for testing trained models.
- `model_trainer.py`: This module is used for training models.
- `plotter.R`: This module is used for plotting node counts.
- `summarizer.py`: This module is used to summarize all obtained results.
- `utils.py`: This module provides some general utility methods.