Eötvös Loránd University

Faculty of Science

Institute of Mathematics

Machine learning driven branching in mixed integer programming

Applied mathematics MSc thesis

János Karl

Supervisors:

András Lukács, Péter Madarasi

Budapest

2023

Contents

1 Introduction

The goal of combinatorial optimization is to find an optimal configuration in discrete spaces, where exhaustive search is unmanageable. It is applied to a large scale of problems such as logistics, retail, biology, and even data science [36]. Furthermore, many of the NP-hard computer science problems can be tackled with combinatorial optimization. Consequently, the majority of exact algorithms which find the optimal solution with certainty, have an exponential running time in the worst case [41]. But they progressively approach the optimal solution, thus if they are interrupted before termination, they can provide a feasible solution, and an optimality bound, that tells us how far is the current solution from the optimal. This is an extremely powerful property, that enables effective usage in practice and makes them the core of modern solvers.

Most combinatorial optimization problems can be formulated as mixed integer linear programs. In practice, they are usually solved with the branch-and-bound algorithm [27] that builds a search tree by recursively partitioning the solution space. There are two important decisions to be made during the performance of algorithm, which affect the solving time. Namely, node selection: selecting the next node to evaluate, and variable selection: selecting the variable by which we partition the solution space [32]. There are a handful of different methods that determine how to tackle these selection problems, usually in the form of hard-coded heuristics [20]. In most cases, they minimize the overall solving time, but there are some rules that result in smaller trees in exchange for longer running times at the decisions, strong branching is such an approach. Gasse et al. [18] presented a solution that roughly makes the same decisions [23], but runs significantly faster. More precisely, they trained a neural network to imitate the heuristic decisions. This line of work raises several challenges. For example, finding a way to encode the linear programs and the state of the branch-and-bound tree [7], or finding an architecture that is capable of learning the problem, and generalizing for different problem sizes and for slightly altered versions.

They solved these challenges in [18] with a graph neural network [13, 21, 37, 39] that exploits a natural bipartite graph representation of mixed integer linear programs and handles inputs of different sizes well. They treat it as a classification problem, thus they only focus on the decision made by the strong branching rule.

There were other earlier approaches to clone the behavior of the strong branching

rule. Alvarez et al. [7] treat the task as a regression problem and train their network to estimate the strong branching scores for every variable. While Khalil et al. [25] and Hansknecht et al. [22] formulate a ranking problem and learn the ordering of the candidates given by strong branching. All of the previously listed approaches are much harder problems than classification, because of it, they have bigger potential with the right architecture. On the other hand, they rely on massive feature engineering, and they are not generalizing well.

In this thesis, our goal is to provide a comprehensive description of the approach presented in [18], addressing the technical details and their effect on the method. In Section 2, we introduce the basics of mixed integer linear programs and the branch-and-bound algorithm, covering the decision problems as well. In Section 3, we present the deep learning concepts needed throughout the process, focusing on graph convolution. In Section 4, we guide through the procedure as a whole, from the data generation to the output of the neural network. In Section 5, we present the results of our experiments with our conclusions. Finally in Section 6, we list our concluding remarks and our plans for the future.

2 Operation research tools

This section is written based on the corresponding sections of the dissertation of Gerald Gamrath [16].

2.1 Mixed integer linear programs

We can define *mixed integer linear programs* (MIP) as follows:

$$x_{MIP} = \min\{c^T x : Ax \leq b, l \leq x \leq u, x \in \mathbb{R}^n, x_i \in \mathbb{Z} \text{ for } i \in \mathrm{I}\}.$$

Where $c \in \mathbb{R}^n$ denotes the coefficient vector of the objective function, $A \in \mathbb{R}^{m \times n}$ the constraint matrix and $b \in \mathbb{R}^m$ the right hand side of the constraints. The variables x_i from the subset $\mathbf{I} \subseteq \{1, 2, \ldots, n\}$ are restricted to take on integer values, all the others are real-valued. Optionally, variables can have upper and lower bounds denoted by $u \in (\mathbb{R} \cup \{+\infty\})^n$ and $l \in (\mathbb{R} \cup \{-\infty\})^n$. We minimize the linear objective function and denote the optimal solution by z_{MIP}. Note that a MIP defined in other forms e.g. using both $\geq$ and $\leq$-inequalities or maximizing the objective function can be transformed into the aforementioned general form. We encode MIPs with the formula $\mathbf{P}(c, A, b, l, u, \mathbf{N}, \mathbf{I})$ inherited from the original notation.

The LP relaxation of the MIP is the original linear program without the integrality conditions, i.e. with $\mathbf{I}$ set to $\emptyset$:

$$x_{LP} = \min\{c^T x : Ax \leq b, l \leq x \leq u, x \in \mathbb{R}^n\}.$$

Solving mixed integer linear programs is NP-hard, but one can solve the LP relaxation in polynomial time [25, 24]. The most important algorithms used in practice are different variants of the dual simplex algorithm [28, 10], which has exponential runtime in the worst case, but runs fast for most problems. The relaxation is less restrictive compared to the MIP so its feasible area contains the feasible solutions of the original problem, thus its optimal solution provides a lower bound to the original problem and is called the dual bound of the problem. Figure 1 illustrates the solution set of a two-dimensional integer

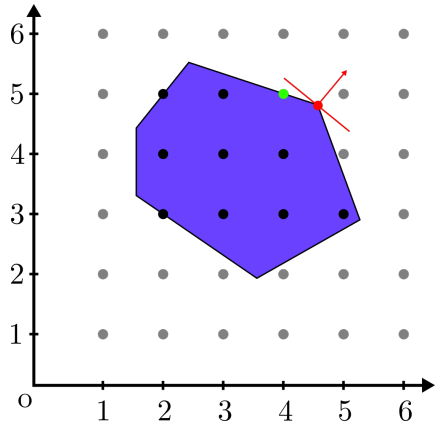

Figure 1: The set of feasible solutions of a MIP's LP relaxation.

linear program together with its LP relaxation. The red arrow shows the minimization direction, the red point is the optimal LP solution, and the blue area is the set of feasible solutions to the LP. Every black dot within this area is a feasible solution of the original MIP, the optimal integer solution is colored green.

2.2 Branch and bound algorithm

The most used algorithm for solving a MIP is the branch-and-bound method, which we will briefly introduce in the following. Let $P_0 = \mathbf{P}(c, A, b, l, u, \mathbf{N}, \mathbf{I})$ be the original problem. First we solve its LP relaxation. Let $\hat{\mathbf{x}}$ denote its optimal solution, if it satisfies the integrality conditions given in the original problem, then we found the optimal solution of the MIP. Otherwise, there is a variable x_i that does not satisfy the integrality conditions. In this case, we execute a branching step along this variable. There are several variations but we only present the most basic one. We create two sub-problems by splitting the domain of the chosen variable. We achieve this by adding $x_i \geq \lceil \hat{x}_i \rceil$ and $x_i \leq \lfloor \hat{x}_i \rfloor$ separately as a new constraint to the original problem, thus creating the up and down child respectively. After that we continue solving the sub-problems recursively until we meet any stopping criteria.

Figure 2 shows how the sub-problems are created from the example illustrated on Figure 1 by splitting the domain of x_1 and x_2 into two parts. The remaining blue area denotes the feasible set of the LP relaxations of the two child problems. During the al-

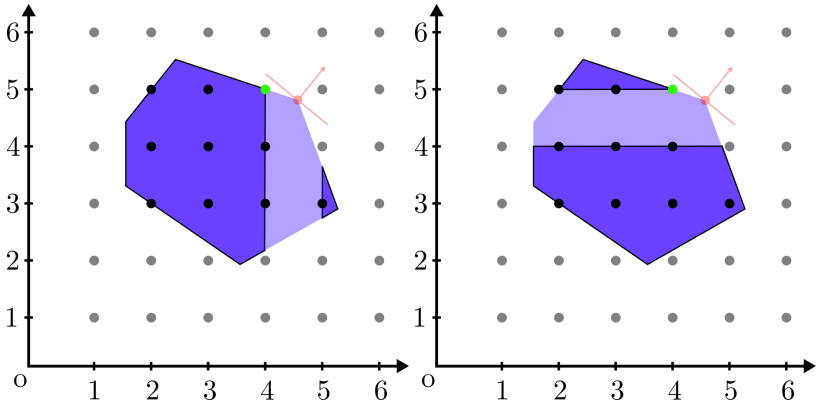

Figure 2: Example of a cut along x_1 and x_2.

gorithm, if we find a feasible solution according to the original problem, then its value serves as an upper bound to the optimal value of the original problem and is called a primal bound, that is denoted by z^*. When processing a sub-problem P' with optimal value $z' = c^T\hat{\mathbf{x}}$ there are three possible scenarios:

- P' is infeasible or $z' \geq z^*$. So P' does not contain a solution with objective value better than the primal bound, therefore P' can be disregarded.
- The optimal solution of P' is feasible to the original problem and $z' < z^*$. It means we found a new best-known solution called the incumbent. We store this value and update the primal bound, and discard P' since we solved it to optimality.
- The optimal solution of P' is not feasible to the original problem and $z' < z^*$. Then we apply the branching step described above.

This procedure can be represented as a tree. The root node corresponds to the original MIP, and every time we split a problem P' into sub-problems P'_1, P'_2, nodes corresponding to those become the children of the node corresponding to P'. So one can regard branch-and-bound as a search tree.

There are two major difficulties that are interesting from our point of view. The first one is the node selection which determines the order in which the tree is traversed and it aims at the primal bound side of the algorithm for finding an optimal solution. The other one is the variable selection at a given node and it focuses on the alteration of the

dual bound of a given problem. It was shown that in practice the latter one has a bigger impact on the size of the tree at the end of the algorithm [5] and consequently on the time spent solving a MIP. The method that determines how branching is performed, in our case variable selection is called *branching rule*. The set of variables we may branch on, i.e. fractional variables are called branching candidates. Choosing the optimal variable is NP-hard as well, even for satisfiability problems [30]. Most of the classical branching rules try to maximize the change in the child nodes, thus increasing the dual bounds.

2.3 Strong branching rule

We will focus on strong branching rule, which usually results in very small branch-and-bound trees but in practice, it is prohibitively computation heavy. The first variant was proposed by Gauthier and Ribière [19] as a method to use the difference of dual bounds to compute pseudocosts of branching candidates. But to use it as a branching rule independent of pseudocosts was only initiated later [31, 8]. This method computes the so-called strong branching score for every branching candidate at every node during the algorithm. In detail we solve the LP relaxations of the up and down child and then we combine the dual bounds into a score. After we have this score for every candidate, we simply choose the one with the highest score. There are several techniques to compute this score. Let Δ_i^- and Δ_i^+ be the dual bound improvement of the down and up child, respectively. One of the most straightforward methods is to simply add the two improvements together, this is called *sum score*

$$s_i := \Delta_i^- + \Delta_i^+.$$

Another natural solution is to take the smallest of the two to get the *minimum score* for

$$s_i := \min\{\Delta_i^-, \Delta_i^-\}.$$

A more sophisticated approach is the *weighted (convex) sum score* proposed by Eckstein [15]

$$s_i := \alpha \min\{\Delta_i^-, \Delta_i^-\} + (1-\alpha)\max\{\Delta_i^-, \Delta_i^-\}.$$

After several iterations [15, 31] it was shown [4] that $\alpha = \frac{5}{6}$ performed the best in a state-of-the-art MIP solver. In general, a best practice is to choose an α that is larger than 0.5 to lay greater emphasis on the smaller gain.
But the exact method used by many state-of-the-art solvers to compute branching score is called product score [3]

$$s_i := \max\{\Delta_i^-, \varepsilon\} \cdot \max\{\Delta_i^+, \varepsilon\},$$

where ε is a small positive constant. The goal of ε is to allow the comparison of candidates with zero improvements in one direction. It is shown empirically that the product score outperforms all of the previously mentioned scoring methods with $\varepsilon = 10^{-6}$ [3].

2.4 Node selection rules

For the sake of completeness, we present the basics of node selection [33] by showing three commonly used rules. The first one is depth-first search, a heuristic rule, which always selects a child node of the previously processed node, if possible. The advantage of this method is that it enables reusability and easy updating of most of the data structures used in the algorithm. Since it has few open nodes it requires less memory. On the other hand, it may create large subtrees that can be pruned later after a new incumbent is found and the global dual bound is only increased at later phases. The second is best-first search, which concentrates on dual bounds, thus it always selects the node with the smallest dual bound. Empirically, it results in small number of evaluated nodes [3], but it requires relatively large processing time per node. Eventually, we have best-estimate search that focuses on primal bounds. It computes a value from pesudocosts and the LP solution to estimate the best integer solution that can be reached from that node, then it selects the smallest. In state-of-the-art solvers, hybrid methods are often preferred, which combine some of the above-introduced rules. For example, we can start

with depth-first search and follow it until we reach a node without a child that can be processed. Then we select the node with the best-estimate value and continue depth-first search from there. There are plenty of sophisticated hybrid methods [19, 3, 40, 34]. The rule used by SCIP starts with best-estimate search, and every tenth time it selects the node with best-first search. This enables that the dual bound also increases during the whole process not just at the end.

3 Deep learning tools

This section follows the paper of Peter Valiĉkoviĉ [39] and an example provided by Nikolas Adaloglou on aisummer [6].

Before diving into graph convolutional neural networks, we briefly look into convolutional layers through an example of image recognition. When working with high-resolution pictures in real-life scenarios, using fully connected layers only has several disadvantages. To start with, it produces a large number of learnable parameters that would take too much time to teach the network. Moreover, due to the fact that our world is well structured, there are building blocks that define images no matter where they are located (the shape of ears, eyes, nose, and the arrangement of the face categorizes an animal regardless it is rear or front view). Convolutional layers are just meant to solve this. There is a smaller kernel matrix that is slid across the image matrix and captures local patterns while reducing the size of the original matrix. Formally, in every position of the kernel matrix, we compute the dot product with the underlying part of the image, thus compressing the information stored in the batch of adjacent pixels, and recognizing different shapes depending on the kernel.

3.1 Fundamental functions

Based on the previous example, we can introduce some basic principles that ensure similar properties when applying convolution to graphs. By a graph, we always mean a simple undirected graph that has no loops or parallel edges. If loops and parallel edges are allowed we use the term multigraph. The adjacency matrix A of a graph $G = (V, E)$

is an $n \times n$ square matrix, where $n = |V|$. For an undirected graph G, $A_{i,j} = 1$ if and only if there is an edge between $x_i \in V$ and $x_j \in V$, 0 else. One can easily see that A is symmetric. In the directed case $A_{i,j} = 1$ if and only if an edge goes from $x_i \in V$ to $x_j \in V$, 0 else. Additional information attached to the edges and vertices is called feature vectors. Now let us temporarily focus only on the nodes. We can stack the feature vectors of nodes into an $X \in \mathbb{R}^{n \times k}$ matrix where n is the number of nodes and k is the number of node features. We want to find such functions that the order of the nodes does not affect the outcome, i.e. for every permutation of nodes we get the same result. This property is called permutation invariance. Formally, we say a function f is permutation invariant if

$$f(PX) = f(X)$$

holds for every permutation matrix P. We can easily add edges to this formula. To define permutation invariance for functions on the adjacency matrix, we have to apply the same permutation both for rows and columns, that is, we require that

$$f(PAP^{\mathrm{T}}) = f(A),$$

holds for every permutation matrix P. It is easily achievable if we take an independent function and apply it to every row separately, and after we use a permutation invariant aggregation function. For example, we can take

$$f(X) = \max\Big(\sum_{i \in V(G)} a \times x_i \Big)$$

for some $a \in \mathbb{R}^k$, and x_i being the feature vector of the node i. But that is not exactly what we are trying to achieve, so to take this idea a step forward, we even want to identify which part of the input belongs to what slice of the output. This property is permutation equivariance, and it gives us slightly more information at the end. We say that f is permutation equivariant if

$$f(PX) = Pf(X)$$

$$f(PAP^{\mathrm{T}}) = Pf(A)$$

for every permutation matrix P. In a basic scenario, we compute a so-called h latent vector independently for every node feature vector x. For a more complex approach, we can combine more of these functions. For example

$$h_i = \psi(x_i) \quad \rightarrow \quad h'_i = \phi(h_i) \quad \rightarrow \quad f(X) = \begin{bmatrix} - & h'_1 & - \\ & \vdots & \\ - & h'_n & - \end{bmatrix}.$$

Now we can shift our attention to grasp locality. One can define locality very naturally on graphs. A path between two distinct nodes of a graph is a sequence of edges that joins a sequence of nodes, and all nodes and edges are distinct. The distance d between two vertices is the number of edges in the shortest path going between them. For a node x_i, its l-hop neighborhood contains vertices which are no further than l

$$N_i^l := \{x_j : d(x_i, x_j) \leq l\}.$$

For $l = 1$ we get the vertices that are directly connected to x_i and we denote it with N_i, we note that usually this version is used in practice. Now our goal is to find a permutation invariant local function g that combines the feature vectors of vertices in N_i^l and from that build a permutation equivariant f:

$$h_i = g(x_i, N_i^l) \quad \rightarrow \quad f(X) = \begin{bmatrix} - & h_1 & - \\ & \vdots & \\ - & h_n & - \end{bmatrix}.$$

For a real-life example and implementation see Section 3.3.

3.2 Graph neural networks

One can observe that finding a suitable g is crucial. We briefly introduce how these functions can be classified into three categories: convolutional, attentional, and message passing. The main difference is what kind of information they attach along the edges. They become more and more complex, consequently, they are able to learn more

compound patterns but require significantly more computation capacity. We will compare them with the help of a general formula, that computes latent vectors. Let ϕ and ψ be permutation equivariant functions and $\bigoplus$ a permutation invariant aggregation function.

For convolutional graph neural networks it is

$$h_i = \phi\Big(x_i, \bigoplus_{x_j \in N_i} c_{ij}\psi(x_j)\Big).$$

We attach a weight to every neighbor of x_i, which shows how much x_i values the features of x_j, we can think of them as coefficients in a weighted combination, usually, we obtain them directly from the adjacency matrix.

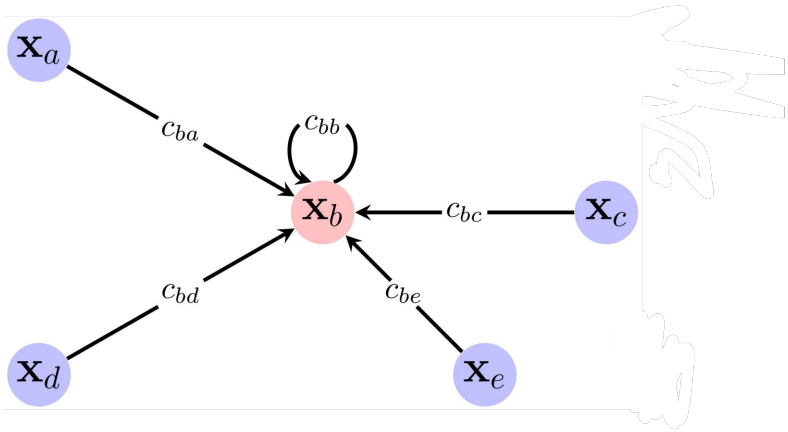

Figure 3: Representation of a convolutional step

In the case of attentional GNNs we get

$$h_i = \phi\Big(x_i, \bigoplus_{x_j \in N_i} a(x_i, x_j)\psi(x_j)\Big).$$

Where $a : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$ is an arbitrary function that takes two feature vectors as input and gives us a coefficient. With this function, we replaced our fixed coefficients with learnable weights. This approach is able to learn more complex weighted combinations with a relatively small amount of information.

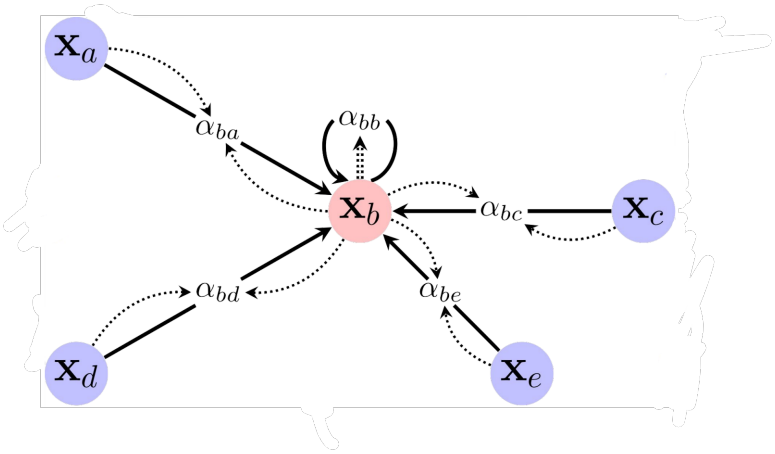

Figure 4: Representation of an attentional step

Ultimately we have message passing GNNs with

$$h_i = \phi\Big(x_i, \bigoplus_{x_j \in N_i} \psi(x_i, x_j)\Big).$$

For every pair of x_i, x_j an arbitrary message vector is computed that is calculated from their feature vectors.

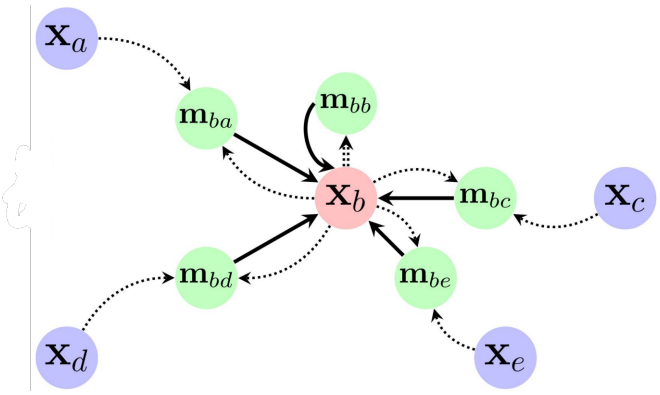

Figure 5: Representation of a message passing step

3.3 A simple version of a graph convolutional layer

During this project we decided to stick with convolutional neural networks, so now we provide a mathematical insight into how they work in practice. We introduce the degree matrix D of a graph G that contains the degree of every vertex in the main diagonal, i.e. $D_{ij} = 0$, $\forall i \neq j$ and $D_{ii} = d(x_i)$, $\forall x_i \in V(G)$. The Laplacian matrix of G is defined as

$$L = D - A.$$

If G is undirected and it has no self-loops and parallel edges then

$$L_{ii} = d(x_i) \text{ and if } i \neq j \text{ then } L_{ij} = \begin{cases} -1, & \text{iff } (x_i, x_j) \in E(G) \\ 0, & \text{otherwise} \end{cases}.$$

The Laplacian matrix contains several useful pieces of information about the underlying graph, but vertices can have varying connectivity to a large extent, therefore a big difference can emerge between the value of elements in L. We want to avoid this because it creates instabilities during the learning phase because of the use of gradient-based methods. To get around this problem, we simply apply normalization to the Laplacian

$$L'_{norm} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}} = I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}.$$

Observe that now every diagonal element will become one (if it has at least one neighbour), and the non-diagonal elements will have the value $L_{norm_{ij}} = \frac{1}{\left(d(x_i) \cdot d(x_j)\right)^{\frac{1}{2}}}$. Another modification we usually utilize is that we add self-loops to every vertex. By that, we make sure that at every node we take into account its own features, and its neighbours as well. So our final matrix is

$$L_{norm} = I - D^{-\frac{1}{2}} (A + I) D^{-\frac{1}{2}}.$$

We should point out one additional really important property of the Laplacian matrix. If we take A to the power of k, then the element A^k_{ij} gives us the number of walks of length k going from i to j. If we take L to the power of k, we get the same matrix as we would get from teh normalization process starting with A^k, thus we got a way to introduce locality

through the matrix formulation.

Now we can construct the basic formula of a graph convolutional layer

$$Y = (L_{norm}X)W,$$

where W is the matrix of trainable weights, and it can be considered as a fully connected layer. As the picture illustrates, there are two layers of nodes where every node from the

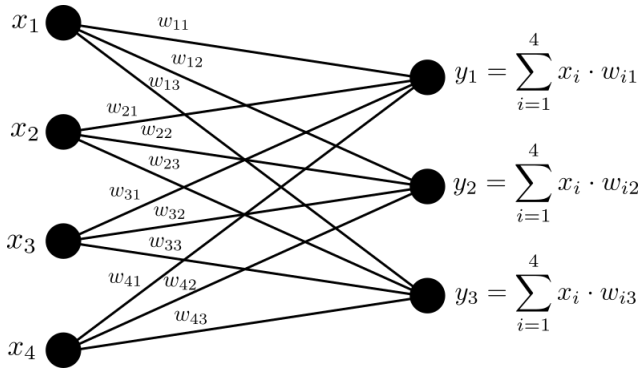

Figure 6: A two-layer perceptron (fully connected layer) without activation function and bias.

first layer is connected to each in the second, thus forming a fully connected layer. The value of a single perceptron in the second layer is computed by summing the weighted values of perceptrons in the first layer. The initial values are derived from an input vector. Therefore the output for a single vector is calculated by $y_i = x_i W$, and if we want to process several vectors at a time, we can easily do it by $Y = XW$. Now our only job is to show that this formula fits the general blueprint shown in Section 3.2. Taking one row l_i from L_{norm}, we obtain l_{ij} weights for the corresponding vertex x_i and its neighborhood N_i. By multiplying it with X, we obtain the weighted sum of the feature vectors of x_i and N_i. The multiplication with W grants a learnable permutation equivariant function. Formally, we get

$$h_i = \phi\Big(x_i, \bigoplus_{x_j \in N_i} c_{ij}\psi(x_j)\Big) = \Big[\sum_{x_j \in \{x_i\} \cup N_i} l_{ij} id(x_j)\Big] W.$$

As we will see, more compound architecture is needed to tackle complex problems. It is achievable by stacking more permutation equivariant functions and using node em-

bedding techniques.

4 Imitation learning for strong branching

The aim of this project was to perform better than the academic state-of-the-art solver SCIP [17]. The method we choose was imitation learning [23] with graph convolutional neural networks [39] based on the work of Gasse et al. [18]. Imitation learning is a subtype of supervised learning. We generate instances of MIPs, compute the branching scores with the solver and observe the best candidate. Then we teach a neural network to pick the variable the solver would have. Of course, we can not mimic SCIP with certainty but a good enough estimation could lead to lower solving time because the network's forward pass takes significantly less time than solving two LPs for every fractional variable.

4.1 Benchmark problems

We need data to teach our neural network. The first step is to generate mixed integer linear programs. We have chosen three NP-hard combinatorial optimization problems.

The first one is the set cover [9], where a set of n elements $U = \{1, 2, \ldots, n\}$ and a set of m subsets $S \subset U$ is given such that $\bigcup_{s \in S} s = U$. The problem is to find the smallest sub-collection of S whose union is equal to U. We can formulate the linear program as follows:

$$x_s \in \{0,1\} \text{ for all } s \in S$$

$$\min \sum_{s \in S} x_s$$

$$\text{subject to:} \quad \sum_{s : e \in s} x_s \geq 1, \ \forall e \in U$$

The second is the maximum independent set problem [12]. The aim is to find the largest set of pairwise non-adjacent vertices in a graph.

$$x_i \in \{0,1\} \text{ for all } i \in V(G),$$

$$\max \sum_{i \in V(G)} x_i$$

$$\text{subject to:} \quad x_i + x_j \leq 1, \ \forall \{i,j\} \in E(G).$$

The third problem is combinatorial auctions [29]. This is a type of smart market, where participants bid on heterogeneous batches of items and the goal is to sell these packages of items for a maximal profit. We can think of the items as the vertices of a hypergraph and the bids as edges. Now our job is to find a maximal weight, independent edge set. With $H = (V, E)$, $c : E \to \mathbb{R}$

$$x_i \in \{0,1\} \text{ for all } i \in E(H),$$

$$\max \sum_i c x_i$$

$$\text{subject to:} \quad x_i + x_j \leq 1, \quad \text{if} \quad i \cap j \neq \emptyset$$

There are three parameters that have a major impact on the solving time. Namely the number of rows and columns, and the density that determines how many variables are present in a constraint on average. We distinguished three different problem sizes based on the average solving time, with parameter tuning. The small dataset consisted of instances that took 30 seconds for SCIP to solve. Normal size required one minute, and large needed 180 seconds. It is important to note that in many cases these results were acquired with large variance in running times. In the case of set cover [9], we generated the instances by setting the number of rows, and the number of columns. For the independent set [12], we changed the number of nodes and the affinity. In the case of combinatorial auctions [29], we take into account the number of items, and the number of bids. There are other parameters as well, but we only changed these when we were testing the running times. The values of the parameters for every benchmark problem can be seen in Table 1.

	Set cover	Independent set	Combinatorial auctions
Small (~30s)	(500/1000)	(420/5)	(100/500)
Normal (~60s)	(700/1370)	(1000/8)	(220/800)
Large (~180)	(800/1670)	(1700/12)	(280/1340)

Table 1: Value of parameters in different benchmark problems.

4.2 Features

To create the data, we had to define feature variables and a target variable. The target variable is clearly the best branching candidate chosen by the solver. To obtain feature variables in the first place, we somehow have to encode a mixed integer linear program into a graph that we can feed to our network. A natural approach is to build a bipartite graph G with one node class being the constraints with $C \in \mathbb{R}^{m \times c}$ being their feature matrix, the other is the set of variables with feature matrix $V \in \mathbb{R}^{n \times v}$. An edge goes between two nodes if the variable corresponding to one end is involved in the constraint corresponding to the other end, the edges have feature matrix $E \in \mathbb{R}^{m \times\ n \times e}$. The question is now that what information should we attach to nodes and edges, that are meaningful enough to describe the target variable.

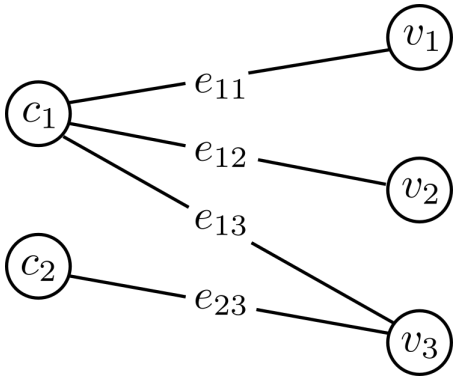

Figure 7: Graph encoding of a MIP.

When defining the features, we decided to follow the paper [18]. For constraints, we computed their cosine similarity with the objective, their normalized bias value, their tightness indicator in the LP solution, their normalized dual solution value, and the normalized age of the LP. Edges only had the normalized constraint coefficient as the only feature. Variables carry the longest feature vectors consisting of their type (i.e. binary, integer, impl. integer, or continuous), their normalized coefficient in the objective, whether they have a lower or upper bound, whether their solution is at these bounds, the integral and fractional part of their solution value, their simplex basis status (i.e. lower, basic, upper, zero) as a one-hot encoding, their normalized reduced cost, the normalized age of the LP, their value in the incumbent, and their average value in all incumbents. Khalil et al. [26] introduced 72 atomic features that are based on the node

LP and the candidate variables. They are mainly statistics about the structural role of a variable within the node LP, additionally some historical data on the variable. They can be divided into the following two classes. The first one is called static, it includes all features that are computed at the root node and do not depend on the current LP. Every other feature falls into the dynamic category.

4.3 Training data

For each benchmark, we generated 100000 random instances then we started to solve them with SCIP with random sampling. During the branch and bound algorithm we stop at random nodes and construct the bipartite graph representation of the current MIP. We save the graph, the branching candidates, and the best variable with some additional data like the index of the instance, the seed of the random generator, or the depth of the current node. We continue this sampling with repetition, every time with a new seed, until we reach the desired number of data files, which amounts to 100000 training and 20000-20000 validation and test files in our case. With this method, we do not necessarily use all of the instances, i.e. we may use several nodes from one instance and none from another. As we can see in Table 2 this phenomenon really emerges, ignoring almost half of the generated samples. This ratio can be reproduced using only 2000 instances to make 10000 data files.

	Set cover		Ind. set		Comb. auctions	
	Training	Test	Training	Test	Training	Test
Total	100000	20000	100000	20000	100000	20000
Unique	57392	11621	63450	10923	59464	11376

Table 2: Value of parameters in different benchmarks

4.4 Architecture of our GCNN

At this point, we can feed this data to our graph convolutional neural network. The input of our model is the bipartite graph representation (G, E, V, C) described in Sec-

tion 4.2, and it performs a graph convolution. As a first step, it embeds our feature vectors into higher dimension with trainable two-layer perceptrons ψ. Then we exploit the structure of bipartite graphs to break down the convolution into two half steps. Since a node only has neighbors from the other vertex class, we can make two successive passes. One from variables to constraints and one from constraints to variables. Formally

$$c_i' = \gamma_c\Big(v_i, \phi_c\big(\sum_{v_j \in N_i} a_{ij}\psi_c(v_j)\big)\Big) \quad \rightarrow \quad v_i' = \gamma_v\Big(c_i, \phi_v\big(\sum_{c_j \in N_i} a_{ij}\psi_v(c_j)\big)\Big),$$

where v_i and c_i denotes the original feature vectors, v_i' and c_i' is the feature vector of variables and constraint after the half convolution. Functions ϕ and γ are 3-layer per-

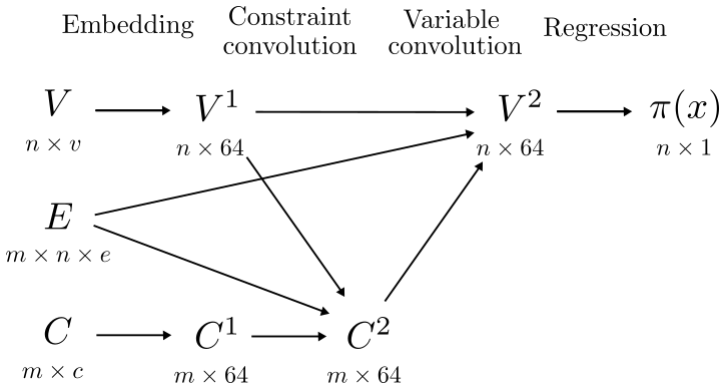

Figure 8: Network architecture

ceptrons of the same size at a given half step with relu activation functions and dropout at the middle layer, coefficients a_{ij} are directly derived from the adjacency matrix. After the two half convolutions, we acquire a graph with the same topology as the input but with different node features. Intuitively, for every constraint we assign the combination of its feature vector with the feature vector of the variables that are involved in it. After that, we switch the role of variables and constraints and repeat this step with the new constraint features. This way each variable receives information about all constraints it is included in, and about the variables that are involved in any of these constraints. At the end of the process, we implement a regressional module that computes a probability distribution over all variables. Then we compute cross-entropy loss and improve our network by minimizing it.

5 Results

The original implementation [2] used TensorFlow as a deep learning framework, later it was transcribed to Pytorch [1]. In both cases, they achieved a faster running time compared to SCIP with default settings, their model could solve instances around 40% faster. These results were achieved on relatively small instances. For the set cover, they considered a 500×500 training data size with density 0.05, and the largest size they tested on had 2000 columns with the same number of rows and density as the smaller one. As a starting point, we implemented our own Pytorch version of the graph neural network with slight changes in the model. We could reproduce approximately the same results, thus we started to experiment with other modifications both in the data and in the model. Our new datasets are described in Section 4.1.

5.1 Initial tests

We established a new baseline model that was used mostly for testing. The major changes we applied to the original were the following: we changed the layer norm to dropout, simplified the message passing module by using a convolutional one instead, and reduced the sizes of the 2-layer perceptrons. Furthermore, in an epoch during training, we iterate through every training sample in random order instead of random sampling, and for consistency, we applied some adjustments which ensure that given a random seed, the program is deterministic. Our first tests were promising because we reached the same results as before on our smaller datasets. Table 3 shows the solving times and the number of created B&B nodes, the number of LPs solved of the two models produced on 20 test instances. The last row contains the number of instances solved within the one-hour time limit. These results were obtained in a set cover version of size

	Original	New baseline	SCIP
Solving time (s)	22871	25402	31541
Number of nodes	2265	1030	1482
Number of LPs solved	1792	2832	2397
Solved instances	19	16	16

Table 3: Results of the two initial models in 20 instances

500×1000 with density 0.05, and the coefficients in the objective function were all one. As we can see, the new model had similar solving times but used almost two times more nodes on average to solve the instances. During the training phase, we save the value of the loss function, and in every fifth epoch, we compute the accuracy of the model, e.g. for a given k, we inspect if the chosen variable by our network is among the best k variables according to SCIP. The value of k was chosen from the set $\{1,3,5,10\}$.

5.2 Hyperparameter optimization

As the next step, we tried to get the best performance out of the baseline model by hyperparameter optimization. We turned our attention to four specific parameters: the batch size, the learning rate decay, the random seed, and the batch size together with the learning rate.

We started with the random seed because we experienced earlier that by using different seeds we get significantly different results. This seed is responsible for initializing every parameter that requires some randomness, for example, the starting weights of the model, or the order of training samples in an epoch. The random seed is an integer between 0 and $2^{32}-1$. We picked random integers i uniformly from $[0,32]$ and set the seed to 2^i (or 2^i-1 if $i=32$). These experiments resulted in roughly the same curves, they learned at the same rate and reached the same accuracies. Table 4 shows the best and worst accuracies and loss values. While Figure 9 illustrates the curves of accuracies for $k=1$, i.e. for the best variable according to strong branching score. The best-performing model was initiated with seed 2^8 and the worst is the one with 2^{25}. We can see minor differences and we can observe that the higher model quality is visible at tighter accuracies.

Seed	0	1	2	2^8	2^{12}	2^{24}	2^{25}	$2^{32}-1$
Loss	2.622	2.589	2.572	2.571	2.578	2.575	2.637	2.631
Top 1 accuracy	62.7%	64.4%	64.4%	64.5%	64.1%	64.3%	62.5%	62.7%
Top 3 accuracy	83.1%	84.4%	84.5%	84.6%	84.3%	84.3%	83%	83%
Top 5 accuracy	90.8%	91.7%	91.8%	91.8%	91.8%	91.7%	90.7%	90.6%
Top 10 accuracy	97.5%	97.8%	97.9%	98%	97.9%	97.9%	97.4%	97.5%

Table 4: Best reached values during seed testing.

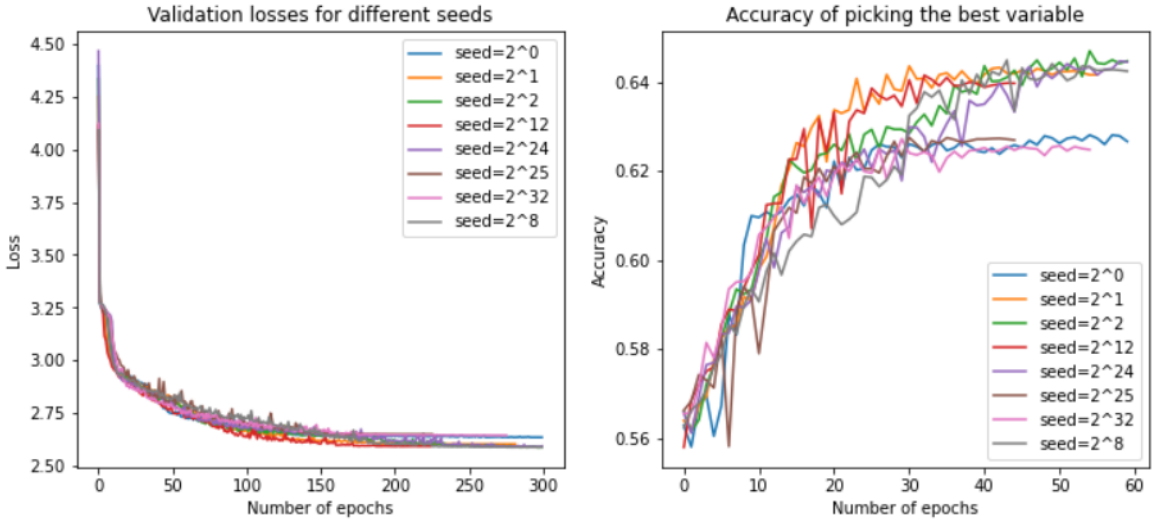

Figure 9: Loss and accuracy learning curves from seed testing.

The second step was to inspect learning rate decay. We decided to investigate this parameter because the learning curves flatten out really quickly after just a few epochs, see Figure 10. Furthermore, we use the optimizer Adam and it has a learning rate decay heuristic as a built-in functionality. We suspected that the optimizer in conjunction with our manual reduction causes the aforementioned problem. Figure 10 shows the loss function and the accuracies during the learning phase. In this case, we encountered even less contrast between the models, even in the top 1 accuracy they had less than one percent difference. We note that the model with the longest training time had the best accuracy and loss function value, while the one with the shortest training time had the worst. Another phenomenon can be seen in this example. All the models were started with the same settings, so they produced the same values until the point where they were modified by the different learning rate decays.

Learning rate decay	0.1	0.3	0.5	0.8	1	1.5
Loss	2.575	2.571	2.565	2.553	2.594	2.588
Top 1 accuracy	64.1%	64.1%	64.2%	64.7%	63.8%	64.1%
Top 3 accuracy	84.3%	84.3%	84.4%	84.7%	84%	84.2%
Top 5 accuracy	91.7%	91.7%	91.8%	92.1%	91.4%	91.6%
Top 10 accuracy	97.9%	97.9%	97.9%	98%	97.8%	97.8%

Table 5: Best reached values during learning rate decay testing.

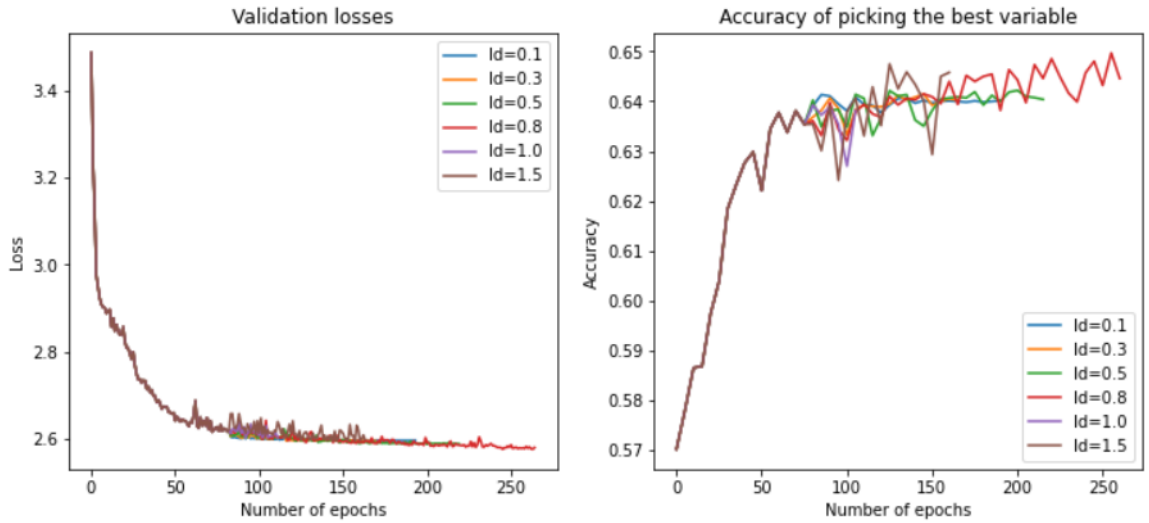

Figure 10: Loss and accuracy learning curves from learning rate decay testing.

With batch size we wanted to see if we can increase the speed of training. Usually, a larger batch size is preferable because it enables faster training times because of parallel computations run on the GPUs. On the other hand, too big of a batch leads to poor generalization. We tested the powers of two as common batch sizes from 32 to 1024. With a batch size of 1024, we nearly used the total memory of a GPU. As we lowered the size, the training phase required more and more computation capacity, e.g. with batch size of 32, we could only run one experiment at a time, thus at later experiments we discarded both. We got the best accuracies from the model with the lowest batch size, which picked the best branching candidate according to SCIP around 66.5% of the time. Figure 11 shows the usual learning curves of all models, with different batch sizes. The model with batch size of 1024 produced the worst results, lagging behind the best model by 4% in the top 1 accuracy. The learning rate controls the pace at which we update

Batch size	32	64	128	256	1024
Loss	2.461	2.525	2.55	2.592	2.622
Top 1 accuracy	66.5%	65.5%	64.7%	63.5%	62.7%
Top 3 accuracy	86.6%	85.4%	84.8%	83.8%	83.1%
Top 5 accuracy	93.4%	92.6%	92.1%	91.4%	90.8%
Top 10 accuracy	98.5%	98.2%	98.1%	97.8%	97.5%

Table 6: Best reached values during batch size testing.

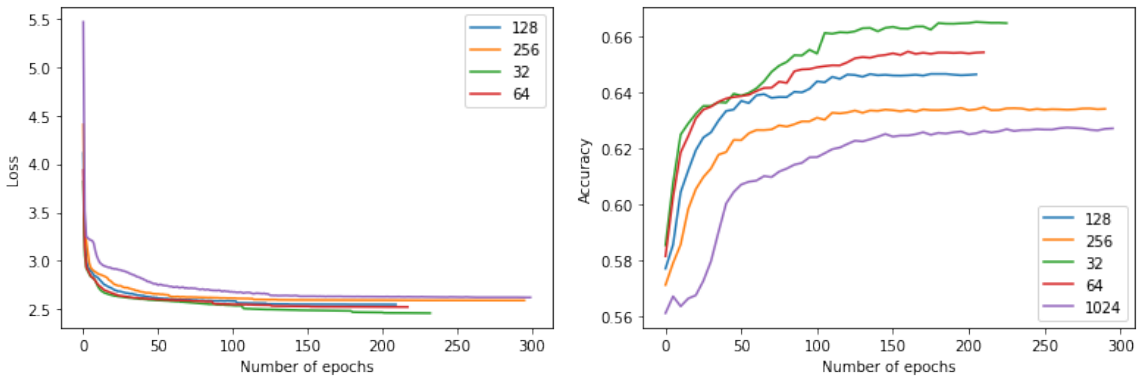

Figure 11: Loss and accuracy learning curves from batch size testing.

the model's weights in response to the estimated error from the loss function. Choosing too small learning rate causes long training times, which may stuck at some point, whereas a value too big could result in unstable learning or in a sub-optimal weight set at the end. Taking the batch size into account, we wanted to find a near-optimal setting, where the model learns the fastest. During these tests, we experimented with some set-ups which in practice should be avoided, and we switched to our newly generated bigger dataset described in Section 4.1. In Figure 12, we present the common graph of learning curves, the first part of the label is the learning rate as decimals, and the second part is the batch size. As expected, models with impractically high learning rates show early instabilities resulting in early stopping, because of poor performance. It is interesting that besides these functions, the best and worst results were produced by models sharing the same learning rate of 0.0016, and with extreme batch sizes, causing a two percent difference between them in the top 1 accuracy.

Learning rate	0.0009	0.0009	0.0016	0.0016	0.00072	00072	0.0023	0.0023
Batch size	512	64	512	64	512	64	512	64
Loss	3.458	3.413	3.473	3.384	3.465	3.385	3.405	3.435
Top 1	15.4%	16.1%	14.8%	16.8%	15.4%	16.8%	16.5%	15.5%
Top 3	31.8%	32.6%	31%	33.6%	31.6%	33.6%	33.2%	32%
Top 5	41.8%	42.5%	40.9%	43.5%	41.6%	43.5%	43.1%	42.1%
Top 10	56.5%	57.1%	56%	57.7%	56.3%	57.8%	57.5%	56.6%

Table 7: Best reached values during the joint tests of batch size and learning rate.

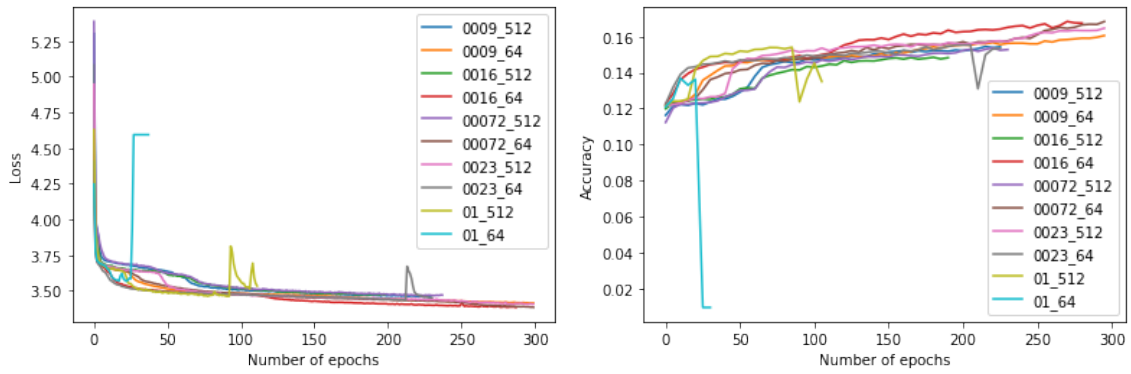

Figure 12: Loss and accuracy learning curves from the joint tests of batch size and learning rate.

5.3 Dataset sizes

One can observe that the values in these graphs are significantly lower than in the previous examples shown in Section 5.2. Considering that every hyperparameter was similar to the ones used in the previous experiments, these results could only be caused by the new dataset. So we shifted our attention to datasets generated from MIPs with different sizes. Compared to the previous samples, the new one had around one and a half times more columns and rows as well. We used a network of the same size as before, so we expected some kind of fall-off, but not at this scale. On the smaller problems, our network picked the best variable more than half the time, now it is reduced to just around every sixth choice. Even in the case of the top ten variables, it only picks one of them in half of all cases, while earlier almost every time selected one of them. To get a comprehensive picture of this phenomenon, we trained models on both sizes of problems, then cross-evaluated them on both test sets. We note that the solving time of an instance and the number of nodes used in the branch-and-bound tree are highly correlated, so we refrained from showing both of them for a given training example because they are very similar to each other.

Figure 13 shows the relationship between the models and the datasets. The left images contain the average solving times measured in the smaller and larger dataset, respectively. In one image from left to right we can see the performance of the model trained on the smaller dataset, the solving time of SCIP, and the performance of the model trained on the bigger dataset. In this section, we will refer to the model trained on

the bigger set as the *bigger model* and the other as the *smaller model* despite the same size. The first thing to note is that our network is not capable of properly imitating the strong branching score, not even in the smaller case. We will cover the possible problems and inaccuracies in our architecture in Section 6.1. Another interesting thing is that the bigger model solved instances from both sets faster than the smaller one. Intuitively, it is not clear why this happened, because, during the training phase, it produced significantly worse accuracies, but performed better in both instances. Eventually, we can observe that the bigger model generalizes well, and even performs better on the smaller set compared to SCIP, while the smaller model produces ten times worse values when transferring it to the bigger set.

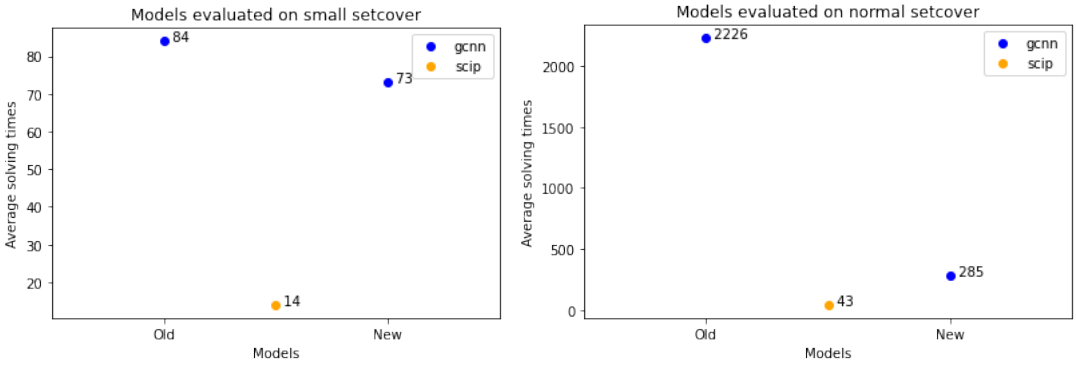

Figure 13: Solving time of cross-evaluated models on different problem sizes.

5.4 Evaluating on different benchmark problems

At this point, we investigated how the models generalize to different problems with about the same size. The tests run at different speeds, so we have varying numbers of instances evaluated for every benchmark problem. The results can be seen in Figure 14. The left picture contains the average number of B&B nodes used by our models to solve an instance. The type of the instance is denoted by colors, and the x-axis shows which problem the models were trained on. The picture on the right contains the baseline given by the solver. We can observe that the models evaluated on the same set they were trained on have around ten times more nodes than SCIP. The next thing we note is that we can determine the hardness of the problems by these results. For the solver, the

easiest problem seems to be the set cover and the hardest is the independent set. While for our network, this order is not that obvious. It is not surprising that every model has the best performance on the same set it was trained on. Furthermore, we can say that for our network the hardest problem is the combinatorial auctions and the easiest is the set cover.

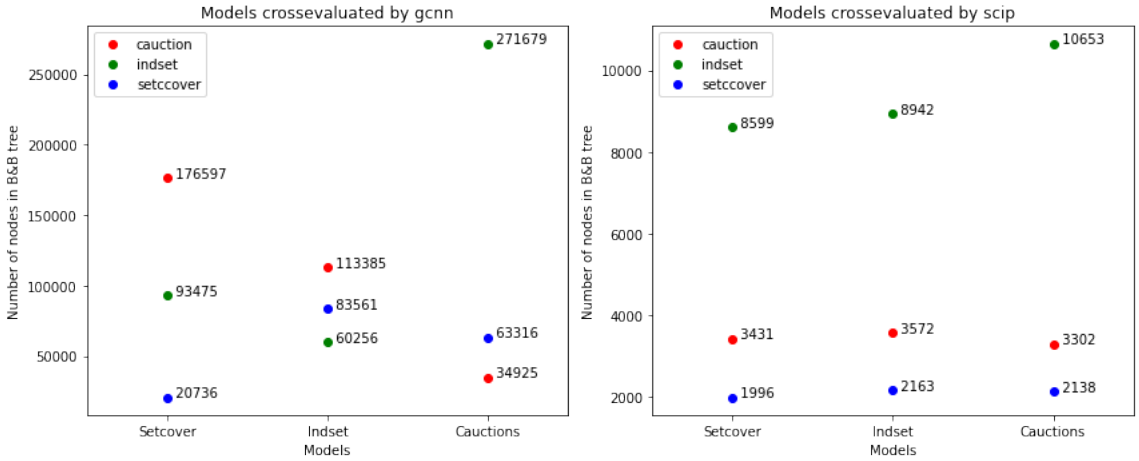

Figure 14: Crossevaluated models in different benchmark problems.

5.5 On running times

At the training (and test) data sizes, we could only reach those running times with high variance as mentioned in Section 4.1. In the case of the normal (60s) dataset, 80% of the instances can be solved in under one minute, and there are a few instances with extremely high solving time. Figure 15 contains an example on the left with 4462 samples, where a bar shows how many instances are solved within the time. On the right, we can see the distribution of all instances that can be solved faster than one minute. We note that this uncertainty depends a lot on the specific problem. We were experimenting with other benchmark problems, for example with strip packing, which aims to place rectangles of different sizes orthogonally without overlapping into a 2-dimensional bounded rectangular area with minimal height. We tried the same approach to set the parameters but every attempt eventuated instances with a few seconds of solving time and others with more than one hour.

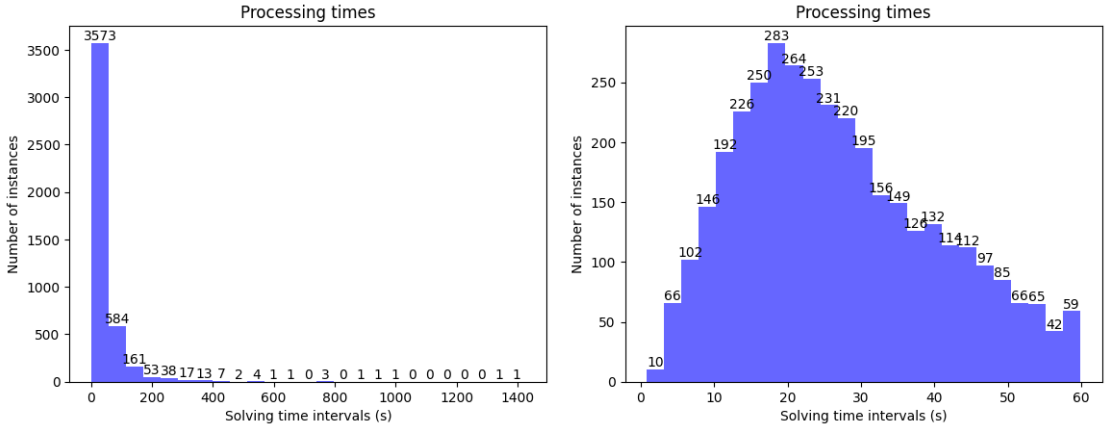

Figure 15: Histograms of running times in the normal dataset.

This phenomenon creates an inaccuracy during the evaluation we have to deal with. We use a time limit, that is usually set to one hour, which stops the solving process when we reach that point and saves the current state of the solver or network. Consequently, some cases emerged where the solver could complete its process on 19 out of 20 instances while the network only on 16 of them. So some results are a bit distorted, especially on larger problems. In the example shown in Figure 16, the left image shows the running times of our model, and the points at the top are all at the time limit. This means they would take much more time to solve in practice and we would get even worse performance, in respect of solving times, and tree size. We were working on

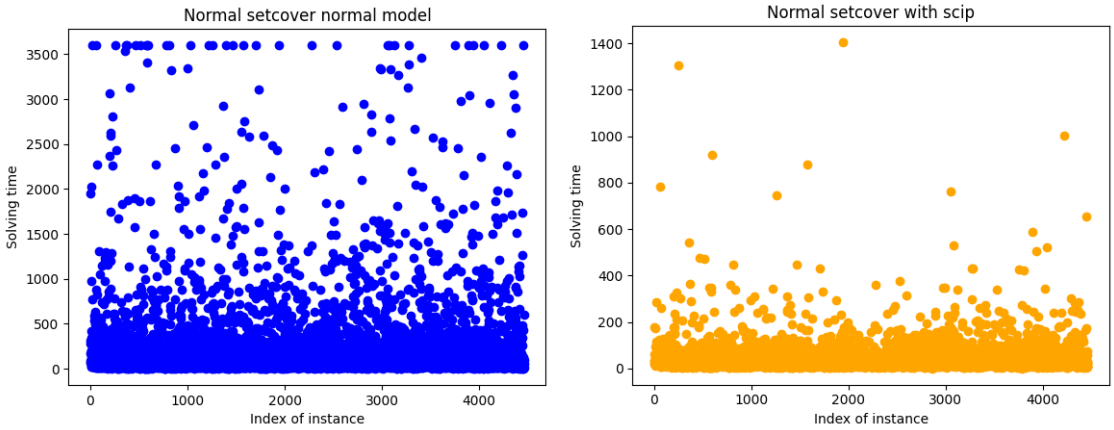

Figure 16: Running times on 4462 instances.

another approach to eliminate this problem. We considered several different parameter settings that roughly produced the same average solving times. Then we started to generate instances with all of them and we kept those that fell in a predetermined interval, for example, 45-75 seconds in case of the normal dataset. The effects of this technique are yet to be tested.

6 Concluding remarks and future work

All results presented in Section 5 can be reproduced during various test runs. But there are two major drawbacks we want to point out, namely the instability of the model when it comes to the evaluation, and the incapacity of learning more complex problems. We saw examples of the latter issue in the previous section, and we will cover the potential solutions in Section 6.3. In the next section, we describe the former through an example.

6.1 Model instability

On the left of Figure 17, we can see the loss function values during the training phase. We saved the model weights at every fifth epoch, then picked a few and evaluated all of them on the same instances, and their average solving times can be seen in the right plot, the numbers mean the corresponding epoch, and 'acc' means the weights with the best accuracy. While the loss function is decreasing with one or two stagnant parts, the solving times differ from this pattern significantly. According to our expectations, the model weights from the 191st epoch should perform one of the best, and yet it is the worst, and the order of the models from the different epochs seems random. Intuitively this means that the loss function we use is not sufficient enough to grasp the problem, i.e. we should not only inspect the best branching candidate, but we have to incorporate all possible variables and we should look for different methods as well.

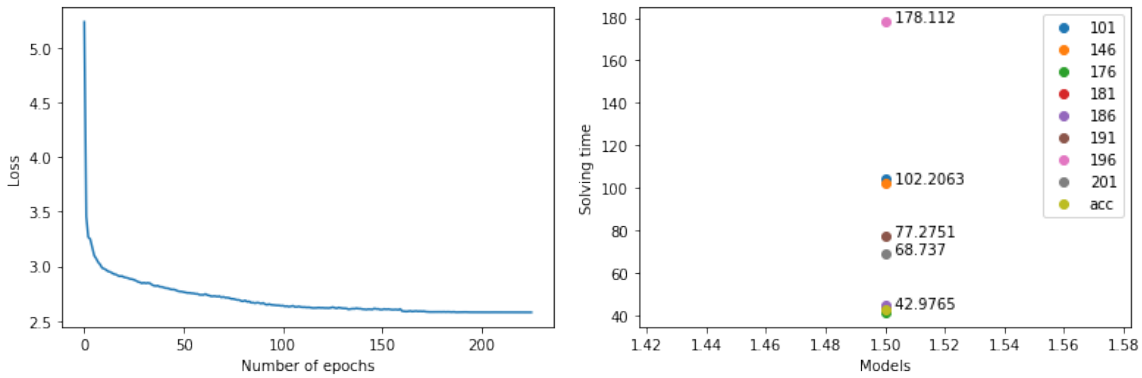

Figure 17: Loss function curve and solving times from different epochs.

6.2 Ordinal classification

There are two important parts in the architecture of our model we aim to investigate thoroughly. The first one is the loss function. As mentioned in the introduction, a middle ground would be to learn the order of candidates (between only learning the best variable, and learning all the branching scores themselves). This approach is called ordinal classification in the literature. Several problems belong to this field, i.e. age prediction, estimation of the severity of illnesses, or the assessment of products. We did not produce results yet using this type of classification, but we briefly describe the different methods used in ordinal classification.

We can formulate the problem as follows [35]: X denotes the input space, in our case the representation of a branching candidate and $Y = \{r_1, r_2 \dots r_k\}$ the output space with ordered ranks $r_k \succ r_{k-1} \succ \cdots \succ r_1$, with $\succ$ denoting the ordering between them, by the strong branching score. The training samples are in the form of $\{x, y\}$. Our goal is to find a $h : X \to Y$ function that assigns a rank to a given input while minimizing a risk function $R(h)$, which is derived from the *absolute cost matrix* $C \in \mathbb{R}^{k \times k}$, where $C_{y,r} = |y - r|$ is the cost of predicting (x, y) as rank r.

The first method [35] creates a binary classifier for each $r_i \in \{r_1, \dots, r_{k-1}\}$ that predicts whether y_i is larger than r_i. Formally, we transform the training data into the form $\{x, y^i, w^i\}_{i=1}^k$, where

$$y^i = \begin{cases} 1, \text{ if } (y > r_i) \\ 0, \text{ otherwise} \end{cases}$$

and $w^i = |C_{y,i} - C_{y,i+1}| = 1$. We attach $k-1$ distinct binary classifiers at the end of our network that consists of two perceptions with softmax, that store whether y_i is bigger than the corresponding rank. Then the rank is computed with $h(x) = 1 + \sum_{j=1}^{k-1} f_j(x)$, where $f_i(x) \in \{0, 1\}$ is the result of the i-th classifier. For a given input there are $k-1$ outputs. Let λ_i denote the *importance coefficient* of the i-th output. The loss function for N inputs is computed with

$$E = -\frac{1}{N} \sum_{j=1}^{N} \sum_{i=1}^{k-1} \lambda_i \mathbb{1}\{f_i(x_j) = y_j^i\} w_j^i \log(P(f_i(x_j)|x_j, W^i)),$$

where W^i denotes the parameters of the i-th classifier. The drawback of this approach is that we have to attach k classifiers for each branching candidate that is 1000^2 classifiers in total in case of the smallest problems we have constructed. Since these tasks can be run simultaneously and independently, the outputs will not be consistent, i.e. for $k = 4$ it is possible that $r_1 = r_4 = 1$ and $r_2 = r_3 = 0$.

We say that the output probabilities are *consistent* if $P(y, r_1) \geq P(y, r_2) \geq \cdots \geq P(y, r_{k-1})$. The solution is given by the Consistent Rank Logits framework [14]. This method uses the one-hot encoded version of y, where the i-th entry is 1 if $y > r_i$, 0 otherwise. We have $k-1$ binary classifiers like in the previous model, but each classifier has the same W weights, different b_i biases, and we apply a sigmoid function to every output perceptron. That way all of them will have a value between 0 and 1, so we can interpret these values as probabilities:

$$P(y > r_i) = P(y^i = 1) = \sigma(\sum_{l=1}^{m} w_l a_l + b) = \sigma(aW + b),$$

where $a = (a_1, \dots a_m)$ is the vector representation of the input before classification. This formula is similar to the one shown in Figure 6, with only one perceptron in the last layer and an additional bias value. During training, we minimize the weighted cross-entropy loss of the $k-1$ classifiers with

$$E = -\sum_{i=1}^{N} \sum_{j=1}^{k-1} \lambda_j \Big[\log(\sigma(a_i W + b_i)) y_i^j + \log(1 - \sigma(a_i W + b_i))(1 - y_i^j) \Big].$$

To get the predictions we have to count the output perceptrons where the computed probabilities are higher than 0.5, i.e. $h(x) = 1 + \sum_{j=1}^{k-1} \mathbb{1}\{P(y^j = 1) > 0.5\}$. It is proven in [14] that if W^*, b^* are obtained by minimizing the previously defined loss function, then $b_1 \geq b_2 \geq \cdots \geq b_k - 1$, consequently $P(y^1 = 1) \geq P(y^2 = 1) \geq \cdots \geq P(y^{k-1} = 1)$.

One year later, the same authors improved their approach [38] and named the new version CORN. In the new model, they wanted to keep the consistency, without shared weights in the classifiers. The main modification is that instead of computing probabilities $P(y > r_i)$, we calculate the conditional probabilities $P(y > r_i | y > r_{i-1})$, $i = 1, \dots, k-1$.

Observe that $\{y > r_i\} \subseteq \{y > r_i - 1\}$. We can obtain the sought probabilities with

$$P(y > r_i) = \prod_{j=1}^{i} P(y > r_j | y > r_{j-1}).$$

We have to distribute the data to be able to effectively learn the conditional probabilities.

- S_1 contains every (x_i, y_i) pair.
- S_2 contains only pairs (x_i, y_i), where $y_i > r_1$.
  $\vdots$
- S_k consists of pairs (x_i, y_i), where $y_i > r_{k-1}$.

We use the set S_i to train the classifier responsible for computing $P(y > r_i | y > r_{i-1})$, with loss function

$$E = -\frac{1}{\sum_{i=1}^{k-1} |S_i|} \sum_{j=1}^{k-1} \sum_{i \in |S_j|} \Big[\log(\sigma(a_i W + b_i)) y_i^j + \log(1 - \sigma(a_i W + b_i))(1 - y_i^j) \Big].$$

Without diving deep into theory, we describe another significantly different approach: computing a *unimodal distribution* on the rank classes, where the mode means the probability of the given class, and if we take steps in any direction from the mode we expect the probabilities of other classes to be smaller. In [11] this goal is achieved by having only one output neuron, that predicts the parameter of a discrete unimodal distribution. In this paper, two such distributions were examined, the binomial and the Poisson.

6.3 The graph neural network

The second is the convolutional part. The most obvious direction is to test the remaining two modules, e.g. the attentional and the message passing module, introduced in Section 3.2. The long-term goal is to find a way to create trainable messages for two given node features along an edge that is capable of learning the essence of bigger and more complex problems. Additionally, it is an interesting question how far can we get

with the simpler attentional graph neural network? Since in practice, these methods are achieved with multi-layer perceptrons, a lot depends on their sizes as well. During the tests on the normal set cover, we trained the same network three times with layer sizes 32, 64, and 128. We achieved the best results with the middle one. It shows, that it is worth laying a great emphasis on finding the best combination of placement, the number, and the size of the multi-layer perceptrons.

6.4 Another branching score

We created a "new" branching score. It takes a branching step for every candidate and solves the problem in both children, then returns the number of nodes of the two formed B&B trees. Then combines these two numbers into a score. It is an even stronger branching score because it always selects the variable that results in the smallest B&B tree. In exchange, it takes much more time to generate the training data, because we have to solve the same instance several times to get one training sample, but it can easily be parallelized. We did not make any experiments on this data, because at first, we wanted to imitate the strong branching score better, and then improve the given model.

6.5 Closing thoughts

Tackling mixed integer linear programs with machine learning is a hard task, and there are several valid approaches. In this thesis, we introduced the theoretical foundations and explored a specific approach. During our experiments, we shed light on several key aspects that play a crucial role in this technique, thus they require more attention in the future to reach better performance. As a first step, we will apply the changes mentioned in the previous sections and investigate them to achieve better results. After that, our goal is to combine this method, with a machine learning driven cut generation process, and with that create an effective solver, which makes its decisions based on deep learning techniques only.

References

[1] GitHub - ds4dm/ecole. `https://github.com/ds4dm/ecole/blob/master/examples/branching-imitation/example.ipynb`.

[2] GitHub - ds4dm/learn2branch: Exact Combinatorial Optimization with Graph Convolutional Neural Networks (NeurIPS 2019) — github.com. `https://github.com/ds4dm/learn2branch`.

[3] Tobias Achterberg. *Constraint integer programming*. PhD thesis, 2007.

[4] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2005.

[5] Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. *Facets of Combinatorial Optimization: Festschrift for Martin Grötschel*, pages 449–481, 2013.

[6] Nikolas Adaloglou. How Graph Neural Networks (GNN) work: introduction to graph convolutions from scratch | AI Summer — theaisummer.com. `https://theaisummer.com/graph-convolutional-networks/`.

[7] Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1):185–195, 2017.

[8] David Applegate, Robert Bixby, William Cook, and Vasek Chvátal. On the solution of traveling salesman problems. 1998.

[9] Egon Balas and Andrew Ho. *Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study*. Springer, 1980.

[10] Evelyn Martin Lansdowne Beale. An alternative method for linear programming. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 50, pages 513–523. Cambridge University Press, 1954.

[11] Christopher Beckham and Christopher Pal. Unimodal probability distributions for deep ordinal classification. In *International Conference on Machine Learning*, pages 411–419. PMLR, 2017.

[12] David Bergman, Andre A Cire, Willem-Jan Van Hoeve, and John N Hooker. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1):47–66, 2016.

[13] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

[14] Wenzhi Cao, Vahid Mirjalili, and Sebastian Raschka. Rank consistent ordinal regression for neural networks with application to age estimation. *Pattern Recognition Letters*, 140:325–331, 2020.

[15] Jonathan Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the cm-5. *SIAM journal on optimization*, 4(4):794–814, 1994.

[16] Gerald Gamrath. *Enhanced predictions and structure exploitation in branch-and-bound*. PhD thesis, 2020.

[17] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Pierre Le Bodic, Stephen J. Maher, Frederic Matter, Matthias Miltenberger, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 7.0. ZIB-Report 20-10, Zuse Institute Berlin, March 2020.

[18] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *Advances in neural information processing systems*, 32, 2019.

[19] Jean-Marie Gauthier and Gerard Ribiere. Experiments in mixed-integer linear programming using pseudo-costs. *Mathematical Programming*, 12:26–47, 1977.

[20] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, et al. Miplib 2017: data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 13(3):443–490, 2021.

[21] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 729–734. IEEE, 2005.

[22] Christoph Hansknecht, Imke Joormann, and Sebastian Stiller. Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem. *arXiv preprint arXiv:1805.01415*, 2018.

[23] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

[24] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311, 1984.

[25] Leonid Genrikhovich Khachiyan. A polynomial algorithm in linear programming. In *Doklady Akademii Nauk*, volume 244, pages 1093–1096. Russian Academy of Sciences, 1979.

[26] Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[27] Ailsa H Land and Alison G Doig. *An automatic method for solving discrete programming problems*. Springer, 2010.

[28] Carlton E Lemke. The dual method of solving the linear programming problem. *Naval Research Logistics Quarterly*, 1(1):36–47, 1954.

[29] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76, 2000.

[30] Paolo Liberatore. On the complexity of choosing the branching literal in dpll. *Artificial intelligence*, 116(1-2):315–326, 2000.

[31] Jeff T Linderoth and Martin WP Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11(2):173–187, 1999.

[32] Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *Top*, 25:207–236, 2017.

[33] David R Morrison, Sheldon H Jacobson, Jason J Sauppe, and Edward C Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79–102, 2016.

[34] David R Morrison, Jason J Sauppe, Wenda Zhang, Sheldon H Jacobson, and Edward C Sewell. Cyclic best first search: Using contours to guide branch-and-bound algorithms. *Naval Research Logistics (NRL)*, 64(1):64–82, 2017.

[35] Zhenxing Niu, Mo Zhou, Le Wang, Xinbo Gao, and Gang Hua. Ordinal regression with multiple output cnn for age estimation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4920–4928, 2016.

[36] Vangelis Th Paschos. *Applications of combinatorial optimization*, volume 3. John Wiley & Sons, 2014.

[37] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[38] Xintong Shi, Wenzhi Cao, and Sebastian Raschka. Deep neural networks for rank-consistent ordinal regression based on conditional probabilities. *arXiv preprint arXiv:2111.08851*, 2021.

[39] Petar Velickovic, William Fedus, William L Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. Deep graph infomax. *ICLR (Poster)*, 2(3):4, 2019.

[40] Daniel T Wojtaszek and John W Chinneck. Faster mip solutions via new node selection rules. *Computers & operations research*, 37(9):1544–1556, 2010.

[41] Laurence A Wolsey. *Integer programming*. John Wiley & Sons, 2020.