Reinforcement Learning for Integer Programming Variable Selection

Logan Grout Connor Lawless
School of Operations Research and Information Engineering
Cornell University

May 17, 2021

Abstract

Integer programming is a popular tool used to model a wide range of combinatorial optimization problems. Modern integer programming solvers, such as CPLEX and Gurobi, are built around a simple enumeration algorithm called branch and bound. Within the branch and bound algorithm, there are a number of heuristics that guide how the algorithm constructs and traverses a search tree. In this project, we focus on one such heuristic - selecting which decision variable to branch on in a branch and bound tree. We formulate the branching variable selection problem as a Markov Decision Process, and learn a branching policy via reinforcement learning. We leverage a deep graph convolutional architecture to parameterize a branching policy, and learn the policy through the use of evolutionary strategies. Our algorithm is able to outperform benchmark branch selection algorithms in terms of overall computation time on randomly generated set covering problems. A python implementation of our approach can be found here. [1]

1 Introduction

Integer programming (IP) is a popular tool to model a range of combinatorial optimization problems such as scheduling, vehicle routing, and production planning. However despite it's popularity, IP remains a computationally challenging problem both in theory, due to the NP hard nature of the problem, and in practice. Modern IP solvers are built around a popular algorithm called branch and bound, which solves a sequence of linear relaxations of the IP formulation by recursively partitioning the feasible region into a search tree. Within the branch and bound framework, a number of heuristics are used to solve sequential decision making problems such as node selection (i.e. choosing which node in the search tree to evaluate), and variable selection (i.e. choosing which variable to partition the feasible region on). Currently, solvers use a set of hard-coded heuristics to solve these problems, and carefully tune parameters in the heuristics based on a representative sample of IP problems.

[1] `https://github.coecis.cornell.edu/cal379/Learning-To-Branch-via-Evolutionary-Strategies`

However in many domains, similar types of IP problems are solved repeatedly (i.e. day to day production planning, vehicle routing), which might differ greatly from the sample of IP problems used to tune the heuristics. Reinforcement learning (RL) presents a promising approach to learn heuristics for these sequential decision making tasks that are specialized for a specific class of problems. In this project, we investigate the use of RL to find specialized variable selection heuristics.

1.1 Outline of report

This report begins by outlining the fundamentals of branch and bound algorithms for linear programming, and discuss our formulation of the problem as an MDP in section 2. In section 3 we discuss our deep learning approach to the problem, including the graph convolutional neural network architecture [2] and the evolutionary strategy training procedure. Finally we include an empirical analysis of our approach in section 4.

2 Problem Formulation

2.1 Integer Programming: Branch and Bound

A mixed-integer linear program is an optimization problem of the following form:

$$\underset{x}{\operatorname{argmin}}\{c^T x : Ax \leq b, x \in \mathbb{Z}^p \times \mathbb{R}^{n-p}\} \tag{1}$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $p \leq n$ represents the number of integer variables. The linear relaxation of problem 1 is obtained by removing the integrality requirement on the p variables to obtain a continuous linear program (LP). During the branch and bound algorithm [1] the LP relaxation of the IP is solved. If the resulting solution is integral (i.e. all the variables originally constrained to be integral are integral), then the solution is optimal for the original problem. Otherwise we split the feasible region of the LP relaxation according to one variable x_i that does not respect integrality in the current LP solution x^*:

$$x_i \leq \lfloor x_i^* \rfloor, x_i \geq \lceil x_i^* \rceil, \exists i \leq p : x_i^* \notin \mathbb{Z}$$

Each sub-problem is then solved, and the binary decomposition procedure is repeated giving rise to a search tree. All non-integral solutions provide a lower bound on the objective of the IP, and all integral solutions provide an upper bound. The procedure is completed when the upper and lower bounds are equal or none of the sub-problems can be decomposed further. An important step in this branch and bound procedure is selecting which fractional variable to branch on, as the choice of variable can have a large impact on the total number of nodes needed to solve the problem.

2.2 MDP Formulation

We formulate the branch selection problem as an MDP as follows. The environment is the IP solver (i.e. SCIP, Gurobi) and the brancher (i.e. the algorithm choosing which variable

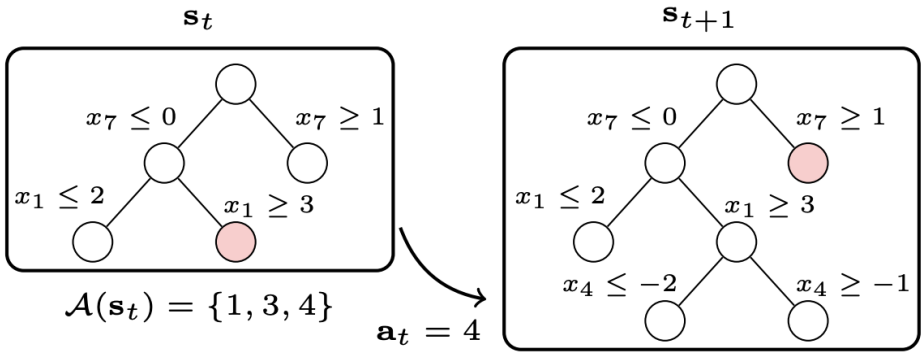

Figure 1: Sample MDP stage of a branching problem. The node in red represents the current node of focus. [2]

to branch on) is the agent. At the t-th time step, the current **state** is the branch and bound tree with all past branching decisions, the best integer solution so far, the LP solution of each node, the current focused node (i.e. which LP relaxation we want to perform a branching operation at), and other solver statistics. The brancher then selects an **action** $a_t \in \{1, ..., p\}$ which represents which fractional variable to branch on according to a policy $\pi(a_t|s_t)$. The solver then performs the branching operation - computing the two new LP relaxations, doing any pruning and processing as needed, and finally selecting the next node to focus on (this is called the node selection problem and is out of the scope of this MDP formulation). An episode is completed when there are no leaf nodes left to branch on, or the problem has been solved. Note that since our goal is solve the problem with as few nodes as possible, our **reward** is -1 for each time step with a discount factor of $\gamma = 1$. This process is illustrated in Figure 1. Note that the transition probabilities are deterministic for a given integer programming problem instance and solver.

Following [2], we encode the state s_t of a branch and bound process to be a bipartite graph with both node and edge features $(\mathcal{G}, C, E, V)$ as shown in Figure 2. On one side of the bipartite graph we have nodes corresponding to constraints in the IP with $C \in \mathcal{R}^{m \times c}$ being the feature matrix for the constraints. On the other side of the graph we have nodes corresponding to variables in the IP, and their associated feature matrix $V \in \mathcal{R}^{n \times d}$. Finally, we have an edge in the bipartite graph if the variable is involved in the constraint (i.e. $A_{i,j} \neq 0$ for constraint i and variable j). The edges also have an associated feature tensor $E \in \mathcal{R}^{m \times n \times e}$. The features used for each element of the graph are listed in Figure 3.

One of the biggest challenges to working with IP solvers is that popular solvers like CPLEX and Gurobi are proprietary and don't make their internals available for changes. Luckily, Ecole, a new package built on top of an open-source solve SCIP, makes branching decisions accessible in a gym-style environment [6]. The implementation also supports the state representations described above.

The one remaining aspect to specify is the set of integer programming problems (each of which will constitute an episode). While Gasse et al. [2] look at 4 different canonical sets of integer programming problems, and generate random instances of each, we focus on the set covering problem. We look specifically at small problem instances (1000 variables, 500

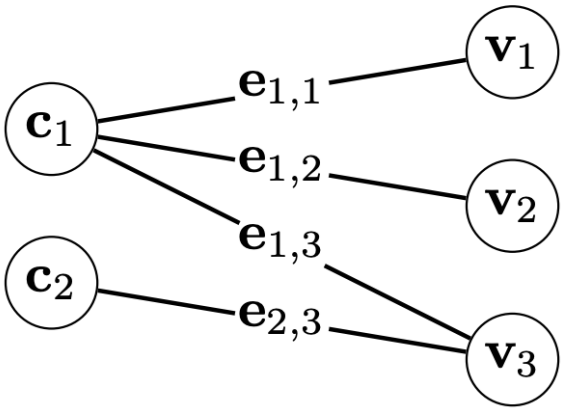

Figure 2: Bipartite Graph Representation of a problem with $n = 3$ variables and $c = 2$ constraints [2]

constriants) so we can run sufficient experiments to train our algorithm.

2.3 Benchmark Policies

There are two general approaches to the branching problem in the existing literature: 1) heuristics from the integer programming community, and 2) imitation learning approaches to replicating slow but effective heuristics. In the former, there are two popular approaches: full strong branching, and pseudocost branching. Full strong branching (FSB) [1] is a heuristic that works by computing the bound improvement by branching on every possible variable and returning the variable with the largest bound improvement. While FSB has been shown empirically to lead to smaller trees, it's extremely slow to run (it requires solving two LPs for every possible branching variable). A quicker heuristic is pseudocost branching, which computes an estimate of the expected bound improvement for each variable (see [1] for details). Pseudocost branching is a quicker heuristic than FSB, but empirically leads to much worse performance in terms of nodes searched.

In recent years, researchers have begun leveraging supervised machine learning as a heuristic to select variables to branch on [4, 2, 3]. In all of the existing approaches, researchers attempt to learn the decisions of full strong branching [1] - aiming to replicate it's performance at a fraction of the computation time. The current state of the art approach is the use of a graph convolutional architecture (GCNN) trained via imitation learning using samples collected from running full strong branching [2].

3 Deep Reinforcement Learning Approach

One of the fundamental shortcomings of existing approaches to the branching problem is that they simply aim to replicate full strong branching. However, there's no theoretical or practical guarantee that full strong branching is the optimal branching policy. Our aim

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

Figure 3: List of features for each component of bipartite graph [2]

is to show that using reinforcement learning to design a policy can outperform heuristics trying to replicate full strong branching, both in terms of computation speed and branch and bound tree size. We extend the approach of Gasse et al. [2] and use a GCNN to parametrize a branching policy. However, rather than use imitation learning we train the network using reinforcement learning. We leverage evolutionary strategies [7] a blackbox optimization tool that's seen recent empirical success for reinforcement learning problems in integer programming [8].

The remainder of this section is organized as follows. Section 3.1 outlines the GCNN architecture for policy parameterization, and section 3.2 outlines the evolutionary strategies algorithm. Finally, we discuss some implementation details that help reduce the computation time to learn an effective policy in section 3.3

3.1 Policy Network Architecture

Our policy, $\pi_\theta(a|s)$ is a graph convolutional neural network, identical to the one used in [2]. Let θ represent the parameters of the GCNN. The input is the state, encoded as a bipartite graph with features on the nodes and edges as described above. The output is an n-dimensional vector, where n is the number of variablesin the IP, such that the i-th entry is $\pi_\theta(x_i|s)$. See Figure 4 for a visual overview of our architecture.

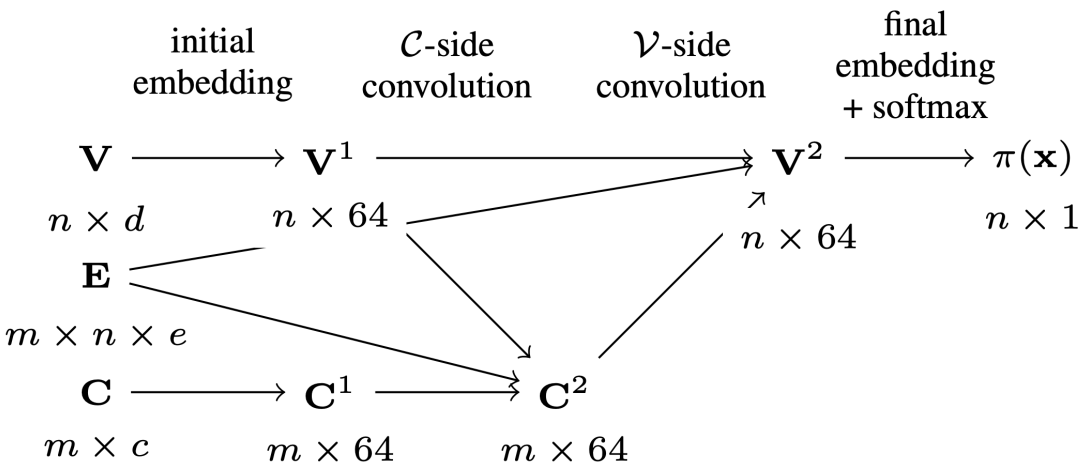

Figure 4: Overview of the graph convolutional neural network architecture for a bipartite graph [2]

The GCNN works in two passes as follows. In the first pass, we consider each constraint. For every edge incident to that constraint in the bipartite representation, that is for each variable that participates in that constraint, the features of the edge, variable, and constraint are combined via a 2-layer perceptron with a ReLU activation function, $g_{\mathcal{C}}$ and summed together. Then this, and the feature vector of the constraint being considered are run through another 2-layer perceptron with a ReLU activation function, $f_{\mathcal{C}}$. The output of this becomes then new feature vector for this constraint. The second pass does the same for each variable, except with two different 2-layer perceptrons with ReLU activation functions, $g_{\mathcal{V}}$ and $f_{\mathcal{V}}$. Mathematically, these two passes take the form:

$$\mathbf{c}_i \leftarrow \mathbf{f}_{\mathcal{C}}\left(\mathbf{c}_i, \sum_{j}^{(i,j)\in\mathcal{E}} \mathbf{g}_{\mathcal{C}}(\mathbf{c}_i, \mathbf{v}_j, \mathbf{e}_{i,j})\right), \quad \mathbf{v}_j \leftarrow \mathbf{f}_{\mathcal{V}}\left(\mathbf{v}_j, \sum_{i}^{(i,j)\in\mathcal{E}} \mathbf{g}_{\mathcal{V}}(\mathbf{c}_i, \mathbf{v}_j, \mathbf{e}_{i,j})\right)$$

A convenient property of this architecture, is that it is agnostic the exact number of constraints and variables included in the formulation. This allows the same policy network to be used on integer programming problems of different sizes, a necessary property for a policy that aims to generalize to large instances. While this policy architecture will output a probability for all decision variables, for inference we only select valid actions (i.e. decision variables that aren't fixed) to help filter infeasible actions. A future direction could involve removing fixed variables from the bipartite graph representation, ensuring that the network only outputs feasible actions without additional intervention.

3.2 Training: Evolutionary Strategies

It has been noted that while the decisions made in the branch and bound algorithm can be formulated as an MDP, trying to find optimal policies for these MDPs using traditional RL algorithms runs into issues [2]. Rather than train a policy for node selection using a standard MDP-based RL algorithm, we trained our policy using an Evolution Strategy

(ES) algorithm. These algorithms have been studied recently to train neural networks, and specifically as an alternative to traditional RL algorithms [7]. This strategy, of using ES to train a policy for a RL model has already been applied to another problem in IP solving: the selection of cutting planes [8].

The main idea of ES is quite simple. Rather than improve your policy parameter by moving in the direction of the gradient, compute a function that approximates the gradient, and move in that direction. To estimate the gradient, we produce a small perturbation vector $\epsilon \sim N(0, \mathbb{1})$. We then define $\theta' = \theta + \sigma\epsilon$ for some fixed constant $\sigma > 0$. To generate an estimate of the policy's return, denoted $J(\theta')$, we run the policy on M instances and average the results. Our gradient estimate $\hat{g}$ is thus:

$$\hat{g}_\theta = \frac{1}{N} \sum_{i=1}^{N} J\big(\pi_{\theta_i'}\big) \frac{\epsilon_i}{\sigma}$$

where N is the batch size (i.e. number of noisy parameter evaluations). We use ADAM [5] to then perform gradient ascent using our gradient estimate.

ES has some attractive qualities that make it well suited for use in this setting. For one, it has lower variance than traditional policy gradient methods for settings with large episode lengths, and where individual actions can have long-term effects [7]. The execution of one instance of an integer program can require a large number of time steps/nodes (i.e. large instances of the set cover problem require 50K + nodes with a state of the art solver). Moreover, individual actions near the root of a branch and bound tree can have long-term impacts on the overall size of the tree [1]. Finally, since our approach doesn't require maintaining a value estimate, like proximal policy optimization or other actor critic methods, it allows us to warm-start or solution procedure (discussed in the following section) using the imitation learning scheme from Gasse et al. [2].

3.3 Implementation Details

One of the challenges of doing reinforcement learning for branching variable selection from scratch (i.e. randomized initial policy) is that it can take an onerous amount of time to find a good policy. Poor performing policies lead to larger episode lengths (i.e. larger trees) which can take orders of magnitude longer than baseline policies. To speed up our training process, we warm-started our policy following the imitation learning methodology of Gasse et al. [2]. We start by generating expert demonstrations (i.e. states and the action selected by full strong branching) from randomnly generated set cover instances. Since strong branching leads to small search trees and thus would only generate training samples near the root of the branch and bound tree, we generate samples by randomly alternating between the expert policy and random branching to ensure we explore large potential states. However, we only retain samples using the expert policy for pre-training. We then train a GCNN using supervised learning with the following imitation learning loss function:

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{(\mathbf{s},\mathbf{a}^*)\in\mathcal{D}} \log \pi_\theta(\mathbf{a}^* \mid \mathbf{s})$$

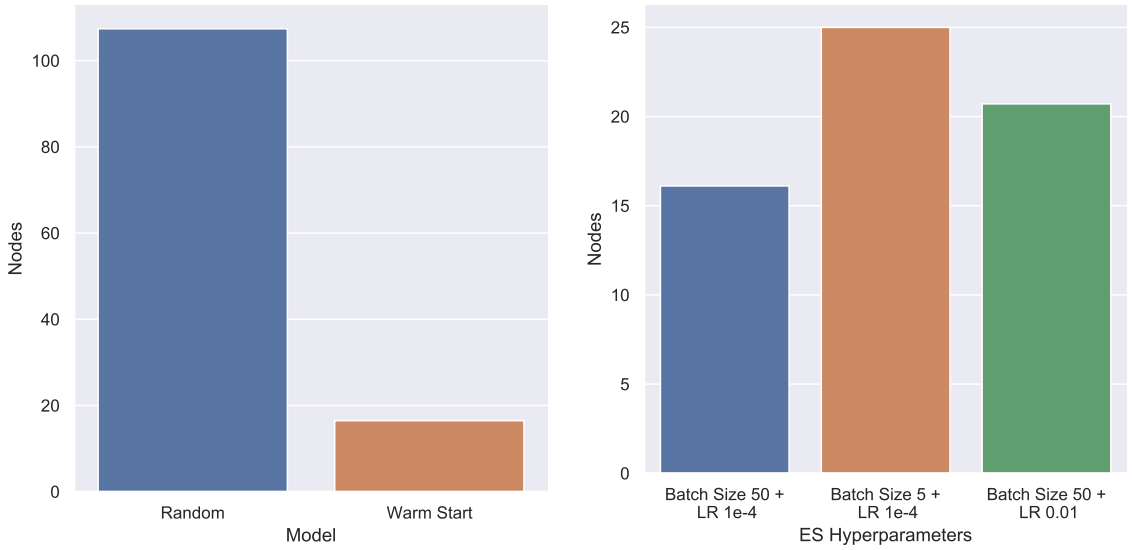

Figure 5: (Left) Effect of Warm Starting on Policy Performance. (Right) Ablation test on the impact of ES hyper-parameters. Nodes indicates the 1-shifted geometric average number of nodes (lower is better) to solve 100 random instances of the set-cover problem.

In their original paper, Gasse et al. collect 100,000 training samples and train their GCNN for 100 epochs over the training data. Due to limited computational resources, we generated 10,000 samples and trained our model for 10 epochs. This was enough to dramatically improve the baseline quality of our branching policy. Figure 5 shows the average number of nodes (lower is better) used to solve a random set cover instance before and after warm-starting the GCNN.

Our implementation of evolutionary strategies was based off Tang et al.'s learning to cut implementation [8] which involved using a batch size of 5 (i.e. independent noisy performance estimates), and a learning rate of 0.01. Instead of using random instances to train their algorithm, they used a smaller training dataset of 20 instances for their sample problem. However, we found that their approach didn't scale to our larger GCNN architecture. We increased the batch size to 50, and dramatically reduced the learning rate (LR) to 0.0001. Figure 4 shows a small ablation test for the impact of these changes. We can see that without both tweaks, ES leads to policies that perform worse than the baseline pre-trained GCNN. To help reduce noise in the gradient estimate for ES, we also ensure that all batches use the same randomly generated test instances, as opposed to independently sampling instances for each. This helped reduce the noise from the difficulty of the test instances in determining ascent directions.

4 Empirical Results

To test the performance of our model trained using reinforcement learning, we follow the procedure from [2] and evaluate the branching policy on 20 new instances of the set cover problem. We run each instance with five different random seeds to capture variance that comes from random tie breaking within the branch and bound algorithm beyond branching.

All of the experiments were run on a laptop computer with a 2.7 GHz Quad-Core Intel Core i7 processor and no GPU. Our reinforcement learning approach (denoted GCNN + ES) was run overnight, which translated to roughly 50 iterations of the ES algorithm. We note that the algorithm had not converged at termination, and could likely have benefited from a longer training period or computing infrastructure better suited for deep learning.

We report the 1-shifted geometric mean for both the number of nodes explored to solve the branch and bound tree and the total computation time. We look at the geometric mean, as opposed to the arithmetic mean, as both the solve time and number of nodes have a left-skew (many instances are solved in pre-solve) and thus the geometric mean provides a performance measure that's less sensitive to large outliers but still includes their effects. Since each instance-by-instance performance has high variance, we report the average variance of performance across the same instance over multiple random seeds. For example, 11.44 (11.1%) for number of nodes indicates that on average the algorithm solves an instance of the set cover problem in 11.44 nodes, and that when solving one instance the number of nodes used varies on average by 11.1 percent. We also report the number of wins across the 100 instances, defined as the number of instances where each algorithm solved the instance in the smallest amount of time. Table 1 summarizes our empirical trials for small set covering instances.

Table 1: Policy evaluation on test set cover instances

Algorithm	Time	Wins	Nodes
Full Strong Branching	6.16 (11.1%)	0/100	**11.44 (11.1%)**
Pseudocost Branching	2.66 (15.2%)	23/100	20.27 (15.2%)
GCNN	2.07 (14.8%)	29/100	16.48 (16.6%)
GCNN + ES	**2.04 (13.7%)**	**48/100**	16.11 (14.7%)

Overall, we can see that the model trained with reinforcement learning does marginally better than the pre-trained GCNN alone (though the results are stll within the margin of error). It won roughly 50% of the instances. It's important to note, that the GCNN was only trained using our warm star procedure, and not the 100,000 samples used in the original Gasse et al. paper due to computational limitations. While these results don't provide evidence that our model outperforms the state of the art, it does provide some evidence that training a model with evolutionary strategies can improve on a pre-trained baseline.

5 Conclusion

In this project, we demonstrated the potential of reinforcement learning to improve branching variable selection in integer programming. Our ES algorithm was able to improve upon a GCNN trained on expert demonstrations from Full Strong Branching. While this improvement provides a promising signal of the power of RL in this context, computational limitations impeded a fair comparison to the state of the art. For one, we were unable to train the GCNN to the extent of Gasse et al. [2], using only a tenth of the expert demonstrations. Our training processss was also terminated pre-maturely, prior to convergence

of an optimal policy. Finally, we were only able to evaluate our policy on small instances of the set covering problem. While we initially planned on running experiments with large instances ($\sim$ 2000 rows), doing so would have required nearly a week of computation time on our computing infrastructure as each instance can take up to an hour to solve. Thus it remains to be seen, whether ES can improve upon a fully trained GCNN branching policy, and whether such a policy generalizes well to large settings. These caveats aside, we believe this approach provides a promising step towards the integration of reinforcement learning into heuristics employed during integer programming branch and bound algorithms.

References

[1] Michele Conforti, Gerard Cornuejols, and Giacomo Zambelli. *Integer programming*. Springer, 2014.

[2] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks, 2019.

[3] Prateek Gupta, Maxime Gasse, Elias B. Khalil, M. Pawan Kumar, Andrea Lodi, and Yoshua Bengio. Hybrid models for learning to branch, 2020.

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

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[6] Antoine Prouvost, Justin Dumouchelle, Lara Scavuzzo, Maxime Gasse, Didier Chételat, and Andrea Lodi. Ecole: A gym-like library for machine learning in combinatorial optimization solvers, 2020.

[7] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017.

[8] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming: Learning to cut. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 9367–9376. PMLR, 13–18 Jul 2020.