Intelligent Variable Selection for Branch & Bound Methods

Priya Shanmugasundaram
priya_s2@dell.com
CFS AI Research
Dell Inc.

Saurabh Jha
saurabh_jha2@dell.com
CFS AI Research
Dell Inc.

Sailendu Patra
sailendu_patra@dell.com
CFS AI Research
Dell Inc.

Abstract

Combinatorial optimisation (Wolsey and Nemhauser [22]) is applied to a wide variety of real-world problems like job scheduling, capacity planning and supply-chain management. These problems are usually modelled as Mixed Integer Programming (MIP) Problems and solved using the Branch and Bound (B&B) (Land and Doig [12]) paradigm. Branch and Bound method partitions the solution space (branching) by creating constrained sub-problems (bounding) and explores those subsets of the solution space which are highly likely to produce optimal solutions. The efficiency of the Branch and Bound method in finding optimal solutions is heavily influenced by the variable and node selection heuristics used for branching (Linderoth and Savelsbergh [13]). In this paper, we propose a novel deep reinforcement learning based variable selection strategy. The proposed solution shows significant improvement over traditional branching strategies like Strong Branching (SB) (Applegate et al. [3]), which have been traditionally used for variable selection and also outperforms the current state of the art RL-based branching strategies which use PPO (Schulman et al. [20]) and DQN (Hessel et al. [11]). The results of our experiments show that the proposed solution performs consistently better than the traditional and RL-based strategies across two different NP-hard problems.

1 Introduction

Mixed Integer Programming (MIP) formulation is an essential step in solving real-world combinatorial optimisation problems. Real-world optimisation problems are complex in nature and are often NP-hard due to the number of integer variables involved in these problems. An increase in the number of integer variables causes an expansion in the solution space, thus, making the task of finding a solution set that minimises the optimisation objective while obeying the constraints, tedious and computationally costly.

Branch and Bound (Land and Doig [12]) (B&B) method is a widely used strategy to solve Mixed Integer Programming problems. Branch and Bound method performs an iterative tree search of the solution space by partitioning the solution space into disjoint subsets creating sub-problems. The method selects sub-problems in such a fashion that it avoids exploration of subsets unlikely to produce optimal solutions and actively explores subsets of the solution space that are more likely to produce optimal solutions. Different branching strategies, pruning and cutting rules are used to search the solution space efficiently, to fasten the search and obtain optimal solutions.

Branching strategies are key to solving branch and bound problems, as robust branching can result in reduction in the effective size of the solution space and obtain optimal solutions at a faster rate. Variable selection based branching (Lodi and Zarpellon [14]) determines the next variable to be used for partitioning the subspace and node selection based branching determines the next subset of the solution space to be explored. The focus of this paper will be on variable selection based branching.

Foundation Models for Decision Making Workshop at Neural Information Processing Systems, 2022

Traditional strategies for variable selection like Pseudo-Cost Branching (PCB) and Strong Branching (SB) (Applegate et al. [3]) are manually designed using domain knowledge for each problem instance and do not generalise to other kinds of problems. Learning to branch using machine learning methods helps in generating variable selection strategies that are non-myopic and can generalise to a broad set of problems. Supervised Learning and Imitation Learning based approaches are a common choice for learning to branch which are trained using expert demonstrations. These methods face issues when applied to real-world problems as the data they are exposed to differ from the expert demonstrations that were used during training phase.

Reinforcement Learning has also been adopted for variable selection in the literature. However, it suffers from cold-start problems and cannot perform optimally until sufficient experience tuples are collected, resulting in sub-optimal early performance. In this paper, a novel deep reinforcement learning based variable selection strategy devoid of above mentioned drawbacks is proposed. The key contributions of the paper are as follows,

- A soft-actor critic based deep reinforcement learning framework for branching variable selection has been developed which can be used in branch & bound methods to solve Mixed Integer Programming (MIP) problems.
- The proposed solution uses experience replay, entropy regularized policy updates and double critics to enhance learning stability.
- The proposed solution uses expert demonstration data during beginning of training to avoid cold-start problem.
- The proposed solution uses MaxLP Gain construction heuristic to traverse the search tree and samples experience trajectories, to improve sample efficiency and avoid partial observability.
- We assess the performance of our proposed strategy on non-adaptive heuristic based and adaptive state of the art RL based branching strategies across different kind of MIP formulations and observe that our solution generalizes well to the different scenarios.

2 Related Work

Traditional Variable Selection Strategies Most Infeasible Branching (MIB) (Achterberg et al. [1]), Pseudo-Cost Branching (PC) and Strong Branching (SB) (Applegate et al. [3]) are traditional methods which score the candidate variables based on their effectiveness in partitioning the solution space. *Most Infeasible Branching* method chooses the variable with the high fractional parts (close to 0.5) and has been shown in Achterberg et al. [1] to perform worse than when the variables were to be selected randomly. *Pseudo-cost Branching* (Falk [8]) estimates an expected gain for each candidate variable based on the history of branching. The method suffers from cold-start due to lack of history and requires the pseudo-costs to be initialised manually which might induce bias. *Strong Branching* is an efficient strategy which selects candidate variables that result in maximum improvement the dual bound at every stage. However, this strategy is computationally costly even for small problems and becomes intractable even for a slight increase in the size of the problem.

Learning based Variable Selection Strategies As traditional variable branching strategies are computationally heavy and require manual intervention, machine learning based strategies are being studied to develop improved and efficient strategies. Expert demonstrations of Strong Branching (SB) strategy are used to train *Supervised Learning* based techniques like offline extra tree regression (Geurts et al. [10]) and online linear regression (Marcos Alvarez et al. [16]) to predict branching scores and *Imitation Learning* based agents which learn to branch by mimicking Strong Branching behavior. However, these methods require extensively labelled expert data sets for training and cannot generalise well to heterogeneous problems. *Reinforcement Learning* has also been used to solve the variable branching problem, Sun et al. [21] designs a primal-dual policy network with negative unit reward for every time step to encourage the agent to learn to solve the problem with minimum number of steps. In Parsonson et al. [17]), Double Deep Q Network architecture with a Superior Network is used to deal with the large action spaces. In Ahn et al. [2], the authors leverage deep reinforcement learning to design solvers for combinatorial optimization problems that can adaptively learn to minimize the number of stages involved in obtaining the solution by deferring the assessment of some vertices to later iterations. In Chen and Tian [6], the authors formulate the optimization problem as a rewriting problem and perform iterative rewriting an existing solution until optimality.

3 Background

3.1 Mixed Integer Programming (MIP)

Definition 1 Mixed Integer Programming (MIP) Formulation Given A $\in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$, and a subset $I \subseteq \{1, 2, ..., n\}$, the mixed integer program $MIP = (A, b, c, I)$ linear in x is defined as follows,

$$z^* = min\{c^T x | Ax \leq b, x \in \mathbb{R}^n, x_j \in \mathbb{Z}, \forall j \in I\} \tag{1}$$

where the vectors in X are feasible solutions of the MIP if $X = \{x \in \mathbb{R}^n | Ax \leq b, x \in \mathbb{R}^n, x_j \in \mathbb{Z}, \forall j \in I\}$, and $x^* \in X$ is an optimal solution if $c^T x^* = z^*$. The problem is NP-Hard due to the integer requirements on x. MIP problems are solved by recursively splitting the problem into sub-problems using Branch and Bound method, where the LP relaxation of the problem is used to circumvent the integerality requirements and to obtain the lower bound for the sub-problems.

Definition 2 LP Relaxation for a (MIP) Problem The LP relaxation of the MIP is defined as follows,

$$\check{z} = min\{c^T x | Ax \leq b, x \in \mathbb{R}^n\} \tag{2}$$

The lower-bound for the sub-tree is provided by solving the above LP-relaxation and if the objective value of the LP relaxation $\check{z} \geq \hat{z}$ where $\hat{z} = c^T \hat{x}$ of the current best solution $\hat{x}$, then the node can be discarded. The optimal integer solution will always lie between the determined upper-bound and lower-bound for the node to be explored. The choice of the variable x_j to be used for calculating these bounds, the choice of the nodes to be explored and the cutting rules are different strategies used in solving MIP problems.

3.2 Branch & Bound (B&B) Algorithm

For any optimisation problem defined as $O = (W, f)$ with an objective function $f : W \to \mathbb{R}$ and W as the search space which is the set of all valid solutions to O, the branch and bound algorithm aims to find an optimal solution $x^* \in argmin_{x \in W} f(x)$. The search space W is partitioned into disjoint sub-spaces by constructing a search tree T, which is then iteratively partitioned by selecting a subspace $G \subset W$ from the unexplored sub-spaces and checking for $\hat{x}' \in G$ has $f(\hat{x}') < f(\hat{x})$ for a feasible $\hat{x} \in W$ stored globally. If above condition is satisfied then the incumbent solution is updated and the subspace G is branched into sub-spaces $G_1, G_2, ..., G_k$ and the search continues iteratively. If $f(\hat{x}') \geq f(\hat{x})\ \forall x \in G$, the subspace is abandoned and not searched further. Once the search is complete with no unexplored sub-spaces, the best incumbent solution is returned and the search terminates.

4 Proposed Solution

Variable Selection plays an important role in branch and bound methods as it prescribes the variable on which branching is to be performed. Selecting a wrong variable for branching can result in increase in the size of the search tree and can increase the computational complexity significantly. Existing methods score the different variables and perform branching greedily by ordering the variables in terms of the score. The goal is to develop a high-quality far-sighted variable selection strategy that is capable of generalising to unseen expert data. The proposed solution leverages reinforcement learning to achieve above goals by modelling the variable selection process as a Markov Decision Process (MDP) (Puterman [19]).

4.1 MDP Formulation

The sequential decision making process for variable selection is formulated as a Markov Decision Process. With the environment represented by the solver, the brancher agent states, actions, transition function and the reward function are described as follows,

State Space S At each instance t, the problem state s_t consists of the bi-partite graph representation for the MIP of the focus node and the index set I_t of the branching candidates which posses constraints on integers and have a non-integer solution. The state s_t is defined as below,

$$s_t = \{(X_t, E_t, C_t), I_t\} \tag{3}$$

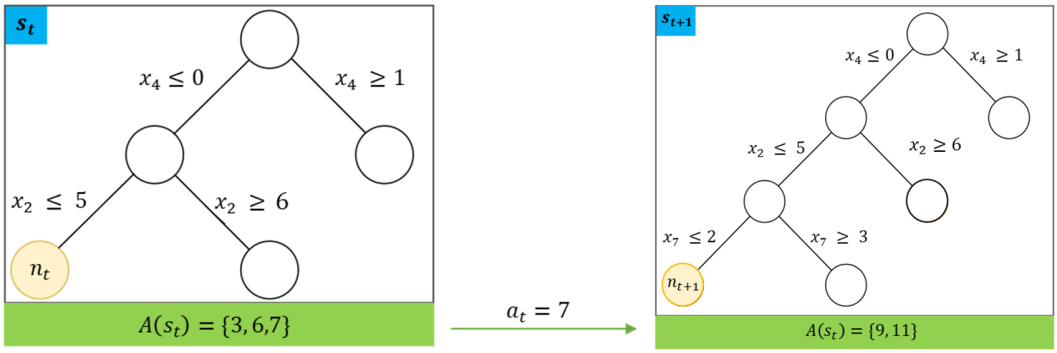

Figure 1: Branching State Transition

where $X \in \mathbb{R}^{n \times d_x}$ is a feature matrix describing the variables and the features from the objective function and the constraints. The nodes of the bi-partite graph represent the n variables to be optimised and the m constraints to be met. An edge e_{ij} is created if the variable x_i has a corresponding $A_{ij} \neq 0$ in the constraint c_j with features d_e representing the constraint's constant term resulting in $E \in \mathbb{R}^{m \times n \times d_e}$, the edge feature matrix. $C \in \mathbb{R}^{m \times d_c}$ represents the constraint matrix where the constraints are encoded into d_c features like in (Gasse et al. [9]). Thus, the state s_t presents a holistic representation of the variables, the objective function and the constraints involved in the MIP at any instance t.

Action Space A The action a_t represents the branching variable selected at instance t from the action space where I_t is the index set of branching variable candidates.

$$A(s_t) = \{i \in I_t\} \tag{4}$$

Transition Function P The state transition function $P(s_{t+1}|s_t, a_t)$ generates the next state provided the current state and action. The next state s_{t+1} is given by using a node selection policy π_n which selects the next node n_t to be branched upon as shown in fig. 1, whose features will be represented by s_{t+1}. However, the environment changes independently of the action of the brancher agent as it cannot perceive the actions of the node selection policy resulting in partial observability. To circumvent partial observability, multiple trajectory paths are constructed such that the root-leaf pairs are used as the source-destination pair and each node on the search tree is mapped to a single trajectory. This generated trajectory information is then stored in the replay buffer and used during the training process.

Reward Function R The goal of our solution is to be able to obtain an optimal solution faster with minimum branching steps as possible. Thus, we define the reward R_t at instance t as follows,

$$R_t = -1 \tag{5}$$

where for every step a unit penalty is issued. The cumulative reward of an episode will represent the number of steps to optimal solution as a penalty, thereby encouraging the method to solve in minimum number of steps.

4.2 Solution

A novel deep reinforcement learning algorithm for variable branching based on Soft Actor Critic Networks (Christodoulou [7]) is proposed. The soft actor critic algorithm aims to learn an optimal actor π^* which maximises the expected discounted reward and the entropy H associated with it.

$$\pi^* = argmax_\pi \sum_{t=0}^{T} \mathbb{E}_{(s_t, a_t)} \left[\gamma^t r(s_t, a_t) + \alpha H(\pi(.|s_t)) \right] \tag{6}$$

where $H(\pi(.|s_t)) = -\log \pi(.|s_t)$, α is the temperature parameter that manages the exploration-exploitation trade off. The actor network and the critic network are modelled using Graph Convolutional Neural Network (GCNN) as the state s_t is represented as a bipartite graph. Two successive passes to perform a single convolution as below,

$$c'_p \leftarrow f_c \left(c_p, \sum_{q}^{(p,q)\in E} g_c(x_q, e_{pq}, c_p) \right), \tag{7}$$

$$x'_q \leftarrow f_x \left(x_q, \sum_{p}^{(p,q)\in E} g_x(x_q, e_{pq}, c'_p) \right), \tag{8}$$

$\forall p \in C, \forall q \in X$, followed by a double-layer perceptrons with ReLU activation. The actor network uses softmax to estimate the action probability distribution π and the critic network uses the masked sum to estimate the state-action value Q. The action generated by the soft-actor agent a_t represents the branching variable and the next node n_t to be branched at instance t is selected by a node selection policy. The sub-tree with n_t as the root node is traversed by selecting the leaf with the largest LP gain until a leaf node not yet fathomed is reached. Trajectories are generated for each of the nodes present in the sub-tree and are added to the experience replay buffer. The node selection policy is used when the soft actor-critic agent interacts with the environment and generates a variable for branching during both training and inference phases. However, the maximum LP-gain trajectory generation happens only during training phase having no additional inference-time overhead.

During the initial training phases, the brancher agent is naive due to lack of experience causing random actions to be taken, which can result in sub-optimal initial performance. The experience buffer filled with expert demonstration data at the beginning of training process will help close the performance gap during this time. The proposed solution is provided with tuples of expert demonstration data $\{s_t, a_t, r_t, s_{t+1}, done_t\}$ in the replay buffer during the beginning of training which is gradually phased it out in a linear matter as new experience tuples are collected by the brancher agent, to avoid cold-start problem during beginning of training.

Algorithm 1: Soft Actor-Critic for Variable Selection in Branch & Bound Methods

for *episode = 1 to E* **do**
  Initialize experience replay buffer D with batch b filled with expert demonstration tuples;
  Initialize online policy weights θ_π and critic weights $\theta_{Q_1}, \theta_{Q_2}$;
  Set target value network weights equal to online parameters $\theta'_{Q_1} \leftarrow \theta_{Q_1}, \theta'_{Q_2} \leftarrow \theta_{Q_2}$;
  Obtain initial observation from environment s_0;
  for *t = 1 to T* **do**
    Execute action $a_t \sim \pi_{\theta_\pi}(.|s_t)$ by branching on variable with index a_t;
    Observe next state $s_{t+1}, a_t, r_t, s_{t+1}$;
    Store (s_t, a_t, s_{t+1}, r_t) in D;
    Set s_t = s_{t+1} using DFS node selection policy which gives next node n_t;
    Explore $Traj \leftarrow$ MaxLPGain($s_t, a_t, n_t, r_t, s_{t+1}$);
    Store $Traj$ in D;
    for i *= 1 to M updates* **do**
      Sample a random batch of S samples (s_t, a_t, s_{t+1}, r_t) from D;
      Set $y_t = r_t + \gamma \min_{j=1,2} Q_{\theta'_{Q_j}}(s_{t+1}, a_{t+1}) - \alpha \log \pi_{\theta_\pi}(a_{t+1}|s_{t+1})$;
      where $a_{t+1} \leftarrow \pi_{\theta_\pi}(.|s_{t+1})$;
      Minimize critic loss for both $j = 1, 2$ $L(\theta_i) = \frac{1}{S} \sum_S (Q_{t,j}(s_t, a_t) - y_t)^2$;
      if $t\%d$ == *0* **then**
        Update policy π with gradient $\nabla_{\theta_\pi} J(\pi_t) = \nabla_{\theta_\pi} \min_{j=1,2} Q_{\theta_{Q_j}}(s_{t+1}, a_{t+1}) - \alpha \log \pi_{\theta_\pi}(a_{t+1}|s_{t+1})$;
        Update target critic network parameters as $\theta_{Q'_j} = \tau\theta_{Q_j} + (1-\tau)\theta_{Q'_j}$;
      end
    end
  end
end

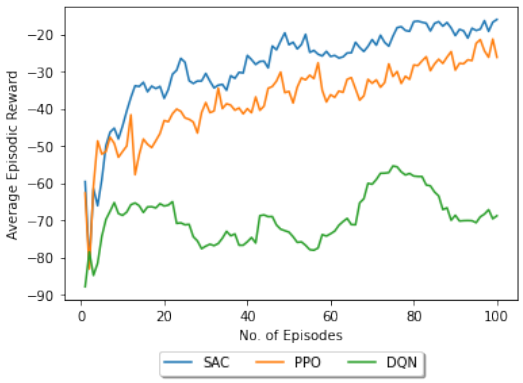

Figure 2: Comparison for MIP Setting 1

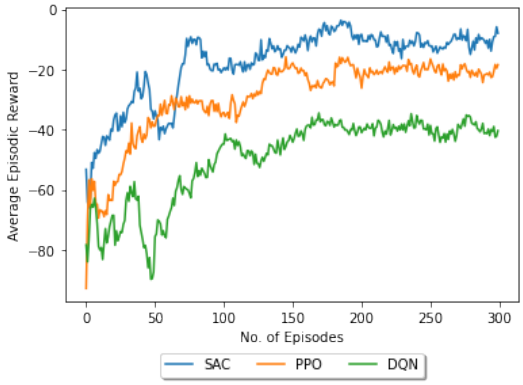

Figure 3: Comparison for MIP Setting 2

5 Results and Discussion

In this section, the results of our experiments are evaluated and examined for insights and improvement areas. The performance of our proposed solution is studied under two different NP-hard problem settings where we compare the performance of our solution with existing solutions of different levels of control complexity as follows,

1. Heuristic-based control – Describes the variable selection methods that are hand-crafted and non-adaptive. The solution is compared against the traditional variable selection method, strong branching (SB);
2. Adaptive Control - Describes the variable selection methods where the actions are prescribed based on recommendations made by reinforcement learning based controllers. The performance of our solution which leverages SAC is compared to the methods which use PPO & DQN.

The experiments were run using open-source Ecole (Prouvost et al. [18]) and PySCIPOpt (Maher et al. [15]) libraries with SCIP solver. Two different NP-hard problems were considered: 1) Set Covering (Balas and Ho [4]) referred to as MIP setting 1 and 2) Maximum Independent Set (Bergman et al. [5]) referred to as MIP setting 2, across which the SB and RL-based controllers PPO & DQN were evaluated in comparison to our solution. The comparison to SOTA tuned commercial solvers is not made, as we do not claim to be competitive with them at this stage. The PPO and DQN agents have similar MDP formulation and reward definition so that they can be compared effectively. The node selection policy used for them is DFS search and we can see that DFS search is causing the algorithm to perform poorly as DFS becomes computationally heavy for practical MIPs. As can be seen in figs 2 & 3, the average reward of our solution improves over PPO by 27% and 22% in the MIP setting 1 and MIP setting 2 respectively. It also significantly outperforms DQN control by 70% and 58% for both the MIP settings.

The proposed strategy exhibits steady initial learning behavior unlike the PPO and DQN based branching strategies, this can be attributed to the expert demonstration data used in our solution to avoid cold-start. The behavior of the different algorithms in the MIP problem 2 are in-line with the performance of the algorithms on the MIP problem 1. It can be seen that the episodic reward for SAC in the MIP problem 2 starts at lower levels than that of the PPO and DQN. However, it exhibits steady episode-over-episode learning behavior until around 100 episodes, after which the agent's learning stabilises. DQN exhibits very slow learning as it ascends with much lower slope and takes until around 160 episodes to have stabilised learning.

The efficiency of our framework in solving the CO problems can be inferred from the average number of nodes and the average number of LPs in Table 1. The proposed strategy is able to reduce the average number of nodes and average number of LPs significantly. SAC reduces the average number of nodes over PPO by 30% and 16% over the MIP problem 1 and MIP problem 2 respectively. It also shows significant impact in the average number of nodes over DQN by around 40% and 20% for the different problems. The average numbers of LPs exhibit a decreasing trend across the different

Table 1: Performance Metrics separated by Algorithms

MIP Setting 1				
Metric	**SAC**	**PPO**	**DQN**	**SB**
Average Reward	-19.72	-29.76	-68.34	−−
Average # Nodes	7.35	10.48	12.61	6.43
Average # LPs	198	238	262	170
MIP Setting 2				
Metrics	**SAC**	**PPO**	**DQN**	**SB**
Average Reward	-12.93	-19.88	-42.71	−−
Average # Nodes	4.27	4.58	5.68	3.76
Average # LPs	18.7	21.3	25.3	19.6

problem settings. Also, it performs as good as the SB control significantly across the different MIP problems. Thus, the improvement trend of our proposed strategy over PPO and DQN control is consistent across the different MIP formulations, elucidating that our method is robust and scalable to different kind of problems and significantly outperforms state of the art control strategies.

6 Conclusion

In this paper, we proposed variable selection strategy for Branch & Bound Methods leveraging Soft Actor Critic. Our method is capable of performing variable selection for different kinds of Mixed Integer Programming problems as it is supplemented by demonstration data to avoid cold-start and trajectory generation that alleviates partial observability and increases sample efficiency. The proposed method outperforms existing rule-based and reinforcement learning based control strategies like Proximal Policy Optimization (PPO) and Deep Q Networks (DQN) and the traditional variable selection method Strong Branching control (SB) across different MIP problems. The solution is being studied for larger MIP problems and detailed experiments to observe its scores, wins and performance comparisons with different constructions heuristics are being performed. The findings will be communicated in a detailed version of this paper.

References

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

[2] Sungsoo Ahn, Younggyo Seo, and Jinwoo Shin. Learning what to defer for maximum independent sets. *CoRR*, abs/2006.09607, 2020. URL `https://arxiv.org/abs/2006.09607`.

[3] D. Applegate, R. Bixby, V. Chvatal, and B. Cook. Finding cuts in the tsp (a preliminary report). Technical report, 1995.

[4] Egon Balas and Andrew Ho. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. In *Combinatorial Optimization*, pages 37–60. Springer, 1980.

[5] David Bergman, Andre A Cire, Willem-Jan Van Hoeve, and John Hooker. *Decision diagrams for optimization*, volume 1. Springer, 2016.

[6] Xinyun Chen and Yuandong Tian. Learning to progressively plan. *CoRR*, abs/1810.00337, 2018. URL `http://arxiv.org/abs/1810.00337`.

[7] Petros Christodoulou. Soft actor-critic for discrete action settings. *arXiv preprint arXiv:1910.07207*, 2019.

[8] Patrick G Falk. Experiments in mixed integer linear programming in a manufacturing system. *Omega*, 8(4):473–484, 1980.

[9] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *CoRR*, abs/1906.01629, 2019. URL `http://arxiv.org/abs/1906.01629`.

[10] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[11] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[12] Ailsa H Land and Alison G Doig. An automatic method for solving discrete programming problems. In *50 Years of Integer Programming 1958-2008*, pages 105–132. Springer, 2010.

[13] 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.

[14] Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *Top*, 25(2):207–236, 2017.

[15] Stephen Maher, Matthias Miltenberger, João Pedro Pedroso, Daniel Rehfeldt, Robert Schwarz, and Felipe Serrano. PySCIPOpt: Mathematical programming in python with the SCIP optimization suite. In *Mathematical Software – ICMS 2016*, pages 301–307. Springer International Publishing, 2016. doi: 10.1007/978-3-319-42432-3_37.

[16] Alejandro Marcos Alvarez, Louis Wehenkel, and Quentin Louveaux. Online learning for strong branching approximation in branch-and-bound. 2016.

[17] Christopher WF Parsonson, Alexandre Laterre, and Thomas D Barrett. Reinforcement learning for branch-and-bound optimisation using retrospective trajectories. *arXiv preprint arXiv:2205.14345*, 2022.

[18] Antoine Prouvost, Justin Dumouchelle, Maxime Gasse, Didier Ch'etelat, and Andrea Lodi. Ecole: A library for learning inside milp solvers. *ArXiv*, abs/2104.02828, 2021.

[19] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[20] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[21] Haoran Sun, Wenbo Chen, Hui Li, and Le Song. Improving learning to branch via reinforcement learning. 2020.

[22] Laurence A Wolsey and George L Nemhauser. *Integer and combinatorial optimization*, volume 55. John Wiley & Sons, 1999.