A Study of Learning Search Approximation in Mixed Integer Branch and Bound Node Selection in SCIP

Yilmaz, Kaan ; Yorke-Smith, Neil

DOI
10.3390/ai2020010

Publication date
2021

Document Version
Final published version

Published in
AI

AI

MDPI

Article

A Study of Learning Search Approximation in Mixed Integer Branch and Bound: Node Selection in SCIP

Kaan Yilmaz and Neil Yorke-Smith *

Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, 2600 GA Delft, The Netherlands; kaan.yilmaz.96@hotmail.com
* Correspondence: n.yorke-smith@tudelft.nl

Abstract: In line with the growing trend of using machine learning to help solve combinatorial optimisation problems, one promising idea is to improve node selection within a mixed integer programming (MIP) branch-and-bound tree by using a learned policy. Previous work using imitation learning indicates the feasibility of acquiring a node selection policy, by learning an adaptive node searching order. In contrast, our imitation learning policy is focused solely on learning which of a node's children to select. We present an offline method to learn such a policy in two settings: one that comprises a heuristic by committing to pruning of nodes; one that is exact and backtracks from a leaf to guarantee finding the optimal integer solution. The former setting corresponds to a child selector during plunging, while the latter is akin to a diving heuristic. We apply the policy within the popular open-source solver SCIP, in both heuristic and exact settings. Empirical results on five MIP datasets indicate that our node selection policy leads to solutions significantly more quickly than the state-of-the-art precedent in the literature. While we do not beat the highly-optimised SCIP state-of-practice baseline node selector in terms of solving time on exact solutions, our heuristic policies have a consistently better optimality gap than all baselines, if the accuracy of the predictive model is sufficient. Further, the results also indicate that, when a time limit is applied, our heuristic method finds better solutions than all baselines in the majority of problems tested. We explain the results by showing that the learned policies have imitated the SCIP baseline, but without the latter's early plunge abort. Our recommendation is that, despite the clear improvements over the literature, this kind of MIP child selector is better seen in a broader approach to using learning in MIP branch-and-bound tree decisions.

Keywords: mixed integer programming; node selection; machine learning; approximate pruning; imitation learning; SCIP

Citation: Yilmaz, K.; Yorke-Smith, N. A Study of Learning Search Approximation in Mixed Integer Branch and Bound: Node Selection in SCIP. *AI* **2021**, *2*, 150–178. https://doi.org/10.3390/ai2020010

Received: 15 March 2021
Accepted: 7 April 2021
Published: 12 April 2021

1. Introduction

Hard constrained optimisation problems (COPs) exist in many different applications. Examples include airline scheduling [1] and CPU efficiency maximisation [2]. Perhaps the most common approach for modelling and solving COPs in practice is mixed integer linear programming (MILP, or simply MIP). State-of-the-art MIP solvers perform sophisticated pre-solve mechanisms followed by branch-and-bound search with cuts and additional heuristics [3].

A growing trend is to use machine learning (ML) to improve COP solving. Bengio et al. [4] survey the potential of ML to assist MIP solvers. One promising idea is to improve node selection within a MIP branch-and-bound tree by using a learned policy. A policy is a function that maps states to actions, where in this context an action is the next node to select. However, research in ML-based node selection is scarce, as the only available literature is the work of He et al. [5]. The objective of this article is to explore learning approximate node selection policies.

Node selection rules are important because the solver must find the balance between exploring nodes with good lower bounds, which tend to be found at the top of the branch-

and-bound search tree, and finding feasible solutions fast, which often happens deeper into the tree. The former task allows for the global lower bound to improve (which allows us to prove optimality), whereas the latter is the main driver for pruning unnecessary nodes.

This article contributes a novel approach to MIP node selection by using an offline learned policy. We obtain a node selection and pruning policy with imitation learning, a type of supervised learning. In contrast to He et al. [5], our policy learns solely to choose which of a node's children it should select. This encourages finding solutions quickly, as opposed to learning a breadth-first search-like policy. Further, we generalise the expert demonstration process by sampling paths that lead to the best k solutions, instead of only the top single solution. The motivation is to obtain different state-action pairs that lead to good solutions compared to only using the top solution, in order to aid learning within a deep learning context.

We study two settings: the first is heuristic that approximates the solution process by committing to pruning of nodes. In this way, the solver might find good or optimal solutions more quickly, however with the possibility of overlooking optimal solutions. This first setting has similarities to a diving heuristic. By contrast, the second setting is exact: when reaching a leaf the solver backtracks up the branch-and-bound tree to use a different path. In the first setting, the learned policy is used as an heuristic method. This is akin to a diving heuristic, with the difference that it uses the default branching rule as a variable fixing strategy. In the second setting, the learned policy is used as a child selector during plunging. The potential of learning here is to augment node selection rules, such that the child selection process is better informed than the simple heuristics that solvers typically use.

We apply the learned policy within the popular open-source solver SCIP [3], in both settings. The results indicate that, while our node selector finds (optimal) solutions a little slower on average than the current default SCIP node selector, *BestEstimate*, it does so much more quickly than the state-of-the-art in ML-based node selectors. Moreover, our heuristic method finds better initial solutions than *BestEstimate*, albeit in a higher solving time. Overall, our heuristic policies have a consistently better optimality gap than all baselines if the accuracy of the predictive model adds value to prediction. Further, when a time limit is applied, our heuristic method finds better solutions than all the baselines (including *BestEstimate*) in three of five problem classes tested and ties in a fourth problem class.

While these results indicate the potential of ML in MIP node selection, we uncover several limitations. This analysis comes from showing that the learned policies have imitated the SCIP baseline, but without the latter's early plunge abort. Therefore our recommendation is that, despite the clear improvements over the literature, this kind of MIP child node selector is better seen in a broader approach to using learning in MIP branch-and-bound tree decisions.

The outline of this article is as follows: Section 2 contains preliminaries, Section 3 specifies the imitation learning approach to node selection and pruning, Section 4 reports the results on benchmark MIP instances (with additional instances reported in the Appendix A), Section 5 discusses the results, Section 6 reviews related work, and Section 7 concludes with future directions.

2. Background

Mixed integer programming (MIP) is a familiar approach to constraint optimisation problems. A MIP requires one or more variables to decide on, a set of linear constraints that need to be met, and a linear objective function, which produces an objective value without loss of generality to be minimised. Thus we have, for y the objective value and x the vector of decision variables to decide on:

$$\min y := c^T x \text{ s.t. } Ax \geq b \cdot x \in \mathbb{Z}^k \times \mathbb{R}^{n-k}, k > 0 \tag{1}$$

where A is an $m \times n$ constraint matrix with m constraints and n variables; c is a $n \times 1$ vector.

Algorithm 1: Solve a minimisation MIP problem using branch and bound.

input : Root R, which is a node representing the original problem
output: Optimal solution if one exists

```
1  R.dualBound ← −∞ // Initialise dual bound
2  PQ ← {R}         // Node priority queue
3  B_P ← ∞          // Primal bound
4  S* ← null        // Optimal solution
5  while PQ is not empty do
       // Remove and return the node in front of the queue
6      N ← PQ.poll();
7      if N.dualBound ≥ B_P then
           // Parent of N had a relaxed solution worse than current best
              integer feasible solution, skip solving relaxation and
              prune
8          continue
9      end
10     S_r ← solveRelaxation(N);
11     if S_r is not feasible then
           // Infeasible relaxation can not lead to a feasible solution
              for the original problem
12         continue;
13     end
14     O_r ← S_r.objectiveValue;
15     if O_r > B_P then
           // This subtree cannot contain any solution better than the
              current best (pruning)
16         continue;
17     end
18     if S_r is integer feasible then
19         B_P ← O_r;
20         S* ← S_r;
           // Found incumbent solution no worse than current best
21         continue;
22     end
23     V ← variableSelection(S_r);
24     a ← floor(V.value);
25     L ← copyAndAddConstraint(N, V ≤ a);
26     R ← copyAndAddConstraint(N, V ≥ a +1);
27     L.dualBound ← O_r;
28     R.dualBound ← O_r;
29     PQ.add(L);
30     PQ.add(R);
31 end
32 return S*;
```

At least one variable has integer domain in a MIP; if all variables have continuous domains then the problem is a linear program (LP). Since general MIP problems cannot be solved in polynomial time, the LP relaxation of (1) relaxes the integer constraints. A series of LP relaxation can be leveraged in the MIP solving process. For minimisation problems, the solution of the relaxation provides a lower bound on the original MIP problem.

Equation (1) is also referred to as the primal problem. The primal bound is the objective value of a solution that is feasible, but not necessarily optimal. This is referred to as a 'pessimistic' bound. The dual bound is the objective value of the solution of an LP

relaxation, which is not necessarily feasible. This is referred to as an 'optimistic' bound. The *integrality gap* is defined as:

$$I_G = \begin{cases} \frac{|B_P - B_D|}{\min(|B_P|, |B_D|)|}, & \text{if } \text{sign}(B_P) = \text{sign}(B_D) \\ \infty, & \text{otherwise} \end{cases} \quad (2)$$

where B_P is the primal bound, B_D is the dual bound, and $\text{sign}(\cdot)$ returns the sign of its argument. Note this definition is that used by the SCIP solver [3], and requires care to handle the infinite gaps case.

Related to the integrality gap is the *optimality gap*: the difference between the objective function value of the best found solution and that of the optimal solution. Both the integrality gap and the optimality gap are monotonically reduced during the solving process. The solving process combines inference, notably in the form of inferred constraints ('cuts'), and search, usually in a branch-and-bound framework.

Branch and bound [6] is the most common constructive search approach to solving MIP problems. The state space of possible solutions is explored with a growing tree. The root node consists of all solutions. At every node, an unassigned integer variable is chosen to branch on. Every node has two children: candidate solutions for the lower and upper bound respectively of the chosen variable. Note that a node (and its entire sub-tree) is pruned when the solution of the relaxation at that node is worse than the current primal bound. The main steps of a standard MIP branch-and-bound algorithm are given in Algorithm 1.

Choosing on which variable to branch is not trivial and affects the size of the resulting search tree, and therefore the time to find solutions and prove optimality. For example, in the SCIP solver, the default variable selection heuristic (the 'brancher') is hybrid branching [7]. Other variable selection heuristics are for example pseudo-cost branching [8] and reliability branching on pseudo-cost values [9]. The brancher can inform the node selector which child it prefers; it is up to the node selector, however, to choose the child. The child preferred by the brancher, if any, is called the priority child.

SCIP's brancher prefers the priority child by using the *PrioChild* property. It is used as a feature in our work. The intuition is to maximise the number of inferences and pushing the solution away from the root node values. In more detail, the left child priority value is calculated by SCIP as: $P_L = I_L(V_r - V + 1)$ and the right child priority value as: $P_R = I_R(V - V_r + 1)$ where I_L (respectively I_R) is the average number of inferences at the left child (right child), V_r is the value of the relaxation of the branched variable at the root node and V is the value of the relaxation of the branched variable at the current node. An inference is defined as a deduction of another variable after tightening the bound on a branched variable [10]. If $P_L > P_R$, then the left child is prioritised over the right child, if $P_L < P_R$, then the right child is prioritised. If they are equal, then none are prioritised. Note that while this rule for priority does not necessarily hold for all branchers in general, it does hold for the standard SCIP brancher.

Choosing on which node to prioritise for exploration over another node is defined by the node selector. As is the case for branching, different heuristics exist for node selection. Among these are depth-first search (*DFS*), breadth-first search (*BFS*), *RestartDFS* (restarting *DFS* at the best bound node after a fixed amount of newly-explored nodes) and *BestEstimate*. The latter is the default node selector in the SCIP solver from version 6. It uses an estimate of the objective function at a node to select the next node and it prefers diving deep into the search tree (see www.scipopt.org/doc/html/nodesel__estimate_8c_source.php, accessed on 1 September 2020).

In more detail, *BestEstimate* assigns a score to each node, which takes into account the quality of its dual bound, plus an objective value penalty for lack of integrality. In particular, the penalty is calculated using the variable's pseudo-costs, which act as an indicator of the per unit objective value increase for shifting a variable downwards or upwards. SCIP actually uses, *BestEstimate* with plunging, which applies depth-first search (selecting

children/sibling nodes) until this is no longer possible or until a diving abort mechanism is triggered. SCIP then chooses a node according to the *BestEstimate* score.

Node selection heuristics can be grouped into two general strategies [11]. The first is choosing the node with the best lower bound in order to increase the global dual bound. The second strategy is diving into the tree to search for feasible solutions and decrease the primal bound. This has the advantage to prune more nodes and decrease the search space. In this article we use the second strategy to develop a novel heuristic using machine learning, leveraging local variable, local node and global tree features, in order to predict as far as possible the best possible child node to be selected.

We can further leverage node pruning to create a heuristic algorithm. The goal is then to prune nodes that lead to bad solutions. Correctly pruning sub-trees that do not contain an optimal solution is analogous to taking the shortest path to an optimal solution, which obviously minimises the solving time. It is generally preferred to find feasible solutions quickly, as this enables the node pruner to prune more sub-trees (due to bounding), with the effect of decreasing the search space. There is no guarantee, however, that the optimal solution is not pruned.

3. Approach

Recall that our goal is to obtain a MIP node selection policy using machine learning, and to use it in a MIP solver. The policy should lead to promising solutions more quickly in the branch-and-bound tree, while pruning as few good solutions as possible.

Our approach is to obtain a node selector by imitation learning. A policy maps a state s_t to an action a_t. In our case s_t consists of features gathered within the branch-and-bound process. The features consist of branched variable features, node features and global features. The branched variable features are derived from Gasse et al. [12]. See Table 1 for the list of features. Note that we define a separate *left_node_lower_bound* and *right_node_lower_bound*, instead of a general node lower bound, because during experimentation, we obtained two different lower bounds among the child nodes.

The actions are the node selection decisions at a node. As opposed to unconstrained node selectors—which can choose any open node s- we constrain our node selector's action space a_t to the selection of direct child nodes. This leads to the restricted action space $\{L, R, B\}$, where L is the left child, R is the right child and B are both children.

In order to train a policy by imitation learning, we require training data from the expert. We obtain this training data, in the form of sampled state-action pairs, by running a MIP solver (specifically SCIP), and recording the features from the search process and the node selection decisions at each sampled node. We now explain the process.

3.1. Data Collection and Processing

Our sampling process is similar to the prior work of He et al. [5], with two major differences. The first is that our policy learns *only* to choose which of a node's children it should select; it does not consider the sub-tree below either child. This encourages finding solutions quickly, as opposed to attempting to learn a breadth-first search-like method.

The second difference from previous work is that we generalise the node selection process by sampling paths that lead to the best k solutions, instead of only the top solution. The reason for this is to obtain different state-action pairs that lead to good solutions compared to only using the top solution, in order to aid learning within a deep learning context. In more detail, the best k solutions are chosen in the same order as output from SCIP; we therefore adopt SCIP's solution ranking which is based on the optimality gap. Thus, whereas He et al. [5] check whether the current node is in the path to the best solution, we check whether the left and right children of the current node are in a path that leads to one of the best k solutions. If that is the case, then we associate the label of the current node as 'B'; if not, we check the left child or right child and associate the appropriate label ('L' or 'R' respectively). If neither are in such a path, then the node is not sampled.

Pre-processing of the dataset is done by removing features that do not change and standardising every non-categorical feature. We then feed it to the imitation learning component, described next, after the example.

Table 1. Features that define a state. Variable features from Gasse et al. [12].

Category	Feature	Description
Variable features	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
Node features	left_node_lb	Lower (dual) bound of left subtree
	left_node_estimate	Estimate solution value of left subtree
	left_node_branch_bound	Branch bound of left subtree
	left_node_is_prio	Branch rule priority indication of left subtree
	right_node_lb	Lower (dual) bound of right subtree
	right_node_estimate	Estimate solution value of right subtree
	right_node_branch_bound	Branch bound of right subtree
	right_node_is_prio	Branch rule priority indication of right subtree
Global features	global_upper_bound	Best feasible solution value found so far
	global_lower_bound	Best relaxed solution value found so far
	integrality_gap	Current integrality gap
	gap_is_infinite	Gap is infinite indicator
	depth	Current depth
	n_strongbranch_lp_iterations	Total number of simplex iterations used so far in strong branching
	n_node_lp_iterations	Total number of simplex iterations used so far for node relaxations
	max_depth	Current maximum depth

3.2. Example of Data Collection

We here give an example of the data collection and processing on a toy problem with two variables. Let $k = 3$ and suppose the known best 3 solutions are: $(x_1, x_2) \in (1,2), (0,0), (4,3)$. The sampling process begins by activating the solver.

1. The solver solves the relaxation of current (root) node and variable selection tells it to branch on variable x_1 with bounds $x_1 \leq 2$ and $x_1 \geq 3$. In this case we know that both

children are in the path of best 3 solutions, so the label here is 'B'. Note that we just found a state-action pair and so we save it.
2. The solver continues its process as usual and decides to enter the left child.
3. Again, the solver solves the relaxation and applies variable selection, the following two child nodes are generated with bounds: $x_2 \leq 3$ and $x_2 \geq 4$. This case is interesting, because only the left child can lead to best 3 solutions, so the label here is 'L'. Again we save this state-action pair.
4. Since we do not assume control of the solver in the sampling process, let us suppose the solver enters the right child (recall the best 3 solutions are not here, but the solver does not know that).
5. Two additional child nodes are generated: $x_1 \leq 0$ and $x_1 \geq 1$. The left node has the following branched variable conditions in total: $x_1 \leq 2$, $x_2 \geq 4$ and $x_1 \leq 0$. and right node has: $x_1 \leq 2$, $x_2 \geq 4$ and $x_1 \geq 1$. None of these resolve to any of the best 3 solutions. Thus we do not sample anything here.
6. The solver continues to select the next node.
7. We continue to proceed with the above until the solver stops. Now we have sampled state-action pair from one training instance. This process is repeated for many training instances.

3.3. Machine Learning Model

Our machine learning model is a standard fully-connected multi-layer perceptron with H hidden layers, U hidden units per layer, ReLU activation layers and Batch Normalisation layers after each activation layer, following a Dropout layer with dropout rate p_d. Figure 1 gives a visual overview of the operations within a hidden layer.

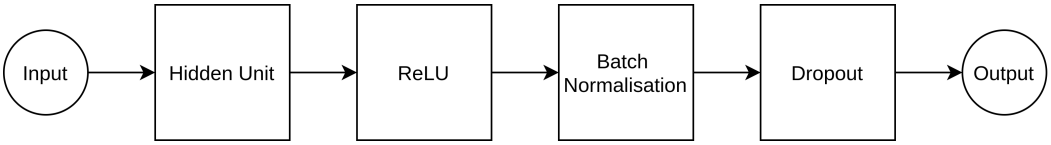

Figure 1. Operations within a hidden layer of the network.

We obtain the model architecture parameters and learning rate ρ using the hyperparameter optimisation algorithm [13]. Since during pre-processing features that have constant values are removed, the number of input units can change across different problems. For example, in a fully binary problem, the features *left_node_branch_bound* and *right_node_branch_bound* are constants (0 and 1 respectively), while for a general mixed-integer problem this is not the case. The number of output units is three. The cross-entropy loss is optimised during training with the Adam algorithm [14].

During policy evaluation, the action B ('both') can result in different operations, as seen in Table 2. We define *PrioChild*, *Second* and *Random* as possible operations. *PrioChild* selects the priority child as indicated by the variable selection heuristic (i.e., the brancher); *Second* selects the next best scoring action from the ML policy; *Random* selects a random child. Additionally, when the solver is at a leaf and there is no child to select, then we define three more operations. These are *RestartDFS*, *BestEstimate* and *Score*. The first two are baseline node selectors from SCIP [3]; *Score* selects the node which obtained the highest score so far as calculated by our node selection policy.

Table 2. Parameter settings for our node selection and pruning policy. Policies are denoted by ML_ followed by up to three letters: ML_{on_both}{on_leaf}{prune_on_both}. Table 4 enumerates the resulting policies.

Parameter	Domain
on_both	{PrioChild, Second, Random}
on_leaf	{RestartDFS, BestEstimate, Score}
prune_on_both	{True, False}

Obtaining the node pruning policy is similar to obtaining the node selection policy. The difference is that the node pruning policy also prunes the child that is ultimately not selected by the node selection policy. If *prune_on_both* = True, then this results in diving only once and then terminating the search. Otherwise, the nodes initially not selected after the action *B* are still explored. The resulting solving process is thus approximate, since we cannot guarantee that the optimal solution is not pruned.

To summarise, as seen in Figure 2, we use our learned policy in two ways: the first is heuristic by committing to pruning of nodes, whereas the second is exact: when reaching a leaf, we backtrack up the branch-and-bound tree to use a different strategy. Source code is available at: www.doi.org/10.4121/14054330 (accessed on 1 September 21); note that due to licensing restrictions, SCIP must be obtained separately from www.scipopt.org (accessed on 1 September 2020).

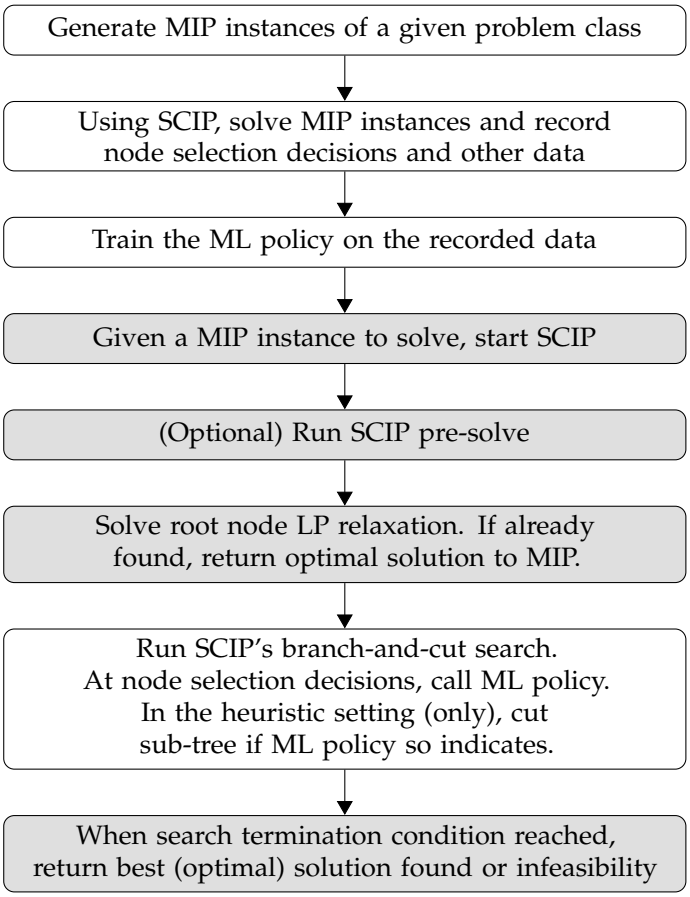

Figure 2. Flowchart of the ML-based approximate node pruning, for a given problem class. Shaded boxes are standard SCIP steps. The first three steps are training, which is performed once for a problem class; the remaining steps are the online solving, which is performed for a given input MIP instance to be solved.

4. Empirical Results

This section studies empirically the node selection policies described in Section 3. The goal is to explore the effectiveness of the learning and of the learned policy within a MIP solver.

The following standard NP-hard problem classes were tested: set cover, maximum independent set, capacitated facility location and combinatorial auctions. The instances were generated by the generator of Gasse et al. [12] with default settings. The classes differ from each other in terms of constraints structure, existence of continuous variables, existence of non-binary integer variables, and direction of optimisation.

For the MIP branch-and-bound framework we use SCIP version 6.0.2. (Note that SCIP version 7, released after we commenced this work, does not bring any major improvements to its MIP solving.) As noted earlier, SCIP is an open source MIP solver, allowing us access to its search process. Further, SCIP is regarded as the most sophisticated and fastest such MIP solver. The machine learning model is implemented in PyTorch [15], and interfaces with SCIP's C code via PySCIPOpt 2.1.6.

For every problem, we show the learning results, i.e., how well the policy is learned, and the MIP benchmarking results, i.e., how well the MIP solver does with the learned policy. We compare the policy evaluation results with various node selectors in SCIP, namely *BestEstimate* with plunging (the SCIP default), *RestartDFS* and *DFS*. Additionally, we compare our results with the node selector and pruner from He et al. [5], with both the original SCIP 3.0 implementation by those authors (*He*) and with the a re-implementation in SCIP 6 developed by us (*He6*). He et al. [5] has three policies: selection only (S), pruning only (P) and both (B). For exact solutions, we only use (S). For the first solution found at a leaf, we compare (S) and (B). For the experiments with a time limit, we compare (P) and (B). Table 3 summarises the node selectors compared.

Table 3. Summary of methods compared in the experiments.

Method	Origin	Exact?
BestEstimate	SCIP [3]	yes
DFS	SCIP [3]	yes
RestartDFS	SCIP [3]	yes
He (select only)	He et al. [5] original	yes
He (prune only)	He et al. [5] original	no
He (both)	He et al. [5] original	no
He6 (select only)	He et al. [5] re-implemented	yes
He6 (prune only)	He et al. [5] re-implemented	no
He6 (both)	He et al. [5] re-implemented	no
ML_$\cdots$ heuristic	this article	no
ML_$\cdots$ exact	this article	yes

We train on 200 training instances, 35 validation instances and 35 testing instances across all problems. These provide sufficient state-action pairs to power the machine learning model. The number of obtained samples (state-action pairs) differs per problem. The process of obtaining samples from each training instance was described in Section 3. For every problem, we use the $k = 10$ best solutions to gather the state-action pairs; for maximum independent set we also experiment with $k = 40$. The higher the value of k, the more data we can collect and the better one can generalise the ML model; we discuss further in Section 5. We selected $k = 10$ having tried lower values ($k = 1, 2, 5$) and found inferior results in initial experiments. Due to computational limitations on training time, higher values of k were not feasible across the board.

Based on initial trials and the hyperparameter optimisation, we use a batch size of 1024, dynamically lower the learning rate after 30 epochs and terminate training after another 30 epochs if no improvement was found. During training, the validation loss is optimised. The maximum number of epochs is 200.

We evaluate a number of different settings for our node selection and pruning policy, as seen in Table 2. This leads to nine different configurations for the node selection policy and twelve different configurations for the node pruning policy. Note that for the node pruning policy, when *prune_on_both* is true, then optimisation terminates when a leaf is found; thus the parameter value for *on_leaf* does not matter. We refer to our policies as ML_{on_both}{on_leaf}{prune_on_both}. For example, ML_PB denotes the node pruning policy that uses *PrioChild* for *on_both* and *BestEstimate* for *on_leaf*. Table 4 enumerates the set of configurations for the learned policies.

Table 4. The node selection (top) and pruning (bottom) policy configurations in full.

Method	on_both	on_leaf	prune_on_both
ML_PR	PrioChild	RestartDFS	false
ML_SR	Second	RestartDFS	false
ML_RR	Random	RestartDFS	false
ML_PB	PrioChild	BestEstimate	false
ML_SB	Second	BestEstimate	false
ML_RB	Random	BestEstimate	false
ML_PS	PrioChild	Score	false
ML_SS	Second	Score	false
ML_RS	Random	Score	false
ML_PRF	PrioChild	RestartDFS	false
ML_SRF	Second	RestartDFS	false
ML_RRF	Random	RestartDFS	false
ML_PBF	PrioChild	BestEstimate	false
ML_SBF	Second	BestEstimate	false
ML_RBF	Random	BestEstimate	false
ML_PSF	PrioChild	Score	false
ML_SSF	Second	Score	false
ML_RSF	Random	Score	false
ML_P·T	PrioChild	–	true
ML_S·T	Second	–	true
ML_R·T	Random	–	true

In the following we report three sets of experiments, and then a subsequent fourth experiment:

1. **First solution.** In the first experiment, we examine the solution quality in terms of the optimality gap and the solving time of the first solution found at a leaf node. Recall that the optimality gap is the difference between the objective function value of the best found solution and that of the optimal solution. Note that it is possible an infeasible leaf node is found, in that case, a solution is returned that was found prior to the branch-and-bound process, through heuristics in-built in SCIP.
2. **Optimal solution.** In the second experiment, we evaluate the policy on every problem in terms of the arithmetic mean solving time of each node selector. That is, the total time to find an optimal solution and prove its optimality.
3. **Limited time.** In the third experiment, we select one ML policy, based on the (lowest) harmonic mean between the solving time and optimality gap. For each instance, we run the solver on each baseline with a time limit equal to the solving time of the selected ML policy and present the obtained optimality gaps. We also report the initial optimality gap obtained by the solver before branch-and-bound is applied.
4. **Imitation.** In the fourth experiment, we analyze the behaviour of the learned policy during the first plunge, and compare it to SCIP's default *BestEstimate* rule. We do this in detail on one set of instances.

By *solving time* we mean the gross difference in time between starting SCIP and terminating it. When we terminate SCIP depends on the task of the experiment:

- Solve to optimality and record solving time: termination condition is finding an optimal solution.
- Solve until we find a solution in a leaf node (or by SCIP's built-in heuristics if the leaf node reached is infeasible) and record the solving time and optimality gap: termination condition is finding a first leaf node.
- Solve until a certain time limit and record optimality gap: termination condition is the time limit itself.

We apply the policies on instances of two different difficulties.

First, easy instances, which can be solved within 15 min. Second, hard instances, where we set a solving time limit to one hour. Here, for all experiments we substitute the integrality gap for the optimality gap, because the optimal solution is not known for every hard instance. An alternative approach is to compare directly the primal solution quality (see Section 5). Additionally with hard instances, for Experiment 1, instead of checking the solving time, we check the integrality gap.

Table 5 provides an overview of the machine learning parameters and results. The baseline accuracy (column 2) is what the accuracy would have been if each sample is classified as the majority class. The test accuracy (column 3) is the classification accuracy on the test dataset. Note that $k = 40$ is included in the maximum independent set instances: see Appendix A.1. The best performing ML model is the model with the settings that achieve the lowest validation loss.

Table 5. ML parameters and prediction results. The baseline accuracy is predicting everything as the majority class.

Problem	Base	Test	H	U	p_d	ρ
Set cover	0.575	0.764	1	49	0.445	0.253
Maximum independent set (10)	0.923	0.922	1	25	0.266	0.003
Maximum independent set (40)	0.895	0.899	1	42	0.291	0.003
Capacitated facility location	0.731	0.901	3	20	0.247	0.008
Combinatorial auctions	0.570	0.717	1	9	0.169	0.002

The experiments reported in Sections 4.2–4.4 are run on a machine with an Intel i7 8770K CPU at 3.7–4.7 GHz, NVIDIA RTX 2080 Ti GPU and 32GB RAM. For the hard instances, the default SCIP solver settings are used. For the other instances, pre-solving and primal heuristics are turned off, to better capture the effect of the node selection policy. We use the shifted geometric mean (shift of 1) as the average across all metrics. This is standard practice for MIP benchmarks [10].

4.1. Summary of Results

The experiments can be summarised as follows:

- **Set cover:** *BestEstimate* has the lowest mean solving time. ML_RB is second, but is not statistically significantly slower. The policies of He et al. [5] are slow. When a time limit has imposed, ML_SRF has the lowest harmonic mean of solving time and optimality gap; this is statistically significant compared to all others.
- **Other instances:** On maximum independent set, *DFS* dominates and the ML policies are relatively poor. Only on this dataset are our policies slower (slightly) than *He6*; in all other cases they dominate. On capacity facility location and on combinatorial auctions, the ML policies and *BestEstimate* respectively are fastest, but not statistically significantly so.
- **Hard set cover:** *BestEstimate* has the lowest mean solving time, but no ML policy was statistically significantly slower. When a time limit is imposed, ML_PST has the lowest harmonic mean of solving time and optimality gap; this is statistically significant compared to all others.

- **Imitation quality:** When the policies are analysed in detail, the *PrioChild* policies ML_P·· are found to accurately imitate *BestEstimate*.

4.2. Set Cover Instances

These instances consist of 2000 variables and 1000 constraints forming a pure binary minimisation problem. We sampled 17,254 state-action pairs on the training instances, 2991 on the validation instances and 3218 on the test instances. The model achieves a testing accuracy of 76.4%, with a baseline accuracy of 57.5%.

Figure 3 shows the mean solving time against the mean optimality gap of the first solution obtained by the baselines and ML policies at a leaf node. Note that the only parameter that is influential for the ML solver is the first parameter, i.e., *on_both*. (The second parameter *on_leaf* and the third *prune_on_both* do not influence the solving time or quality of the first solution, since the search terminates at the first found leaf that is found.) The policy of He et al. [5] is not included here due to its substantial outliers. We see here that our ML policy obtains a lower optimality gap at the price of a higher solving time for the first solution.

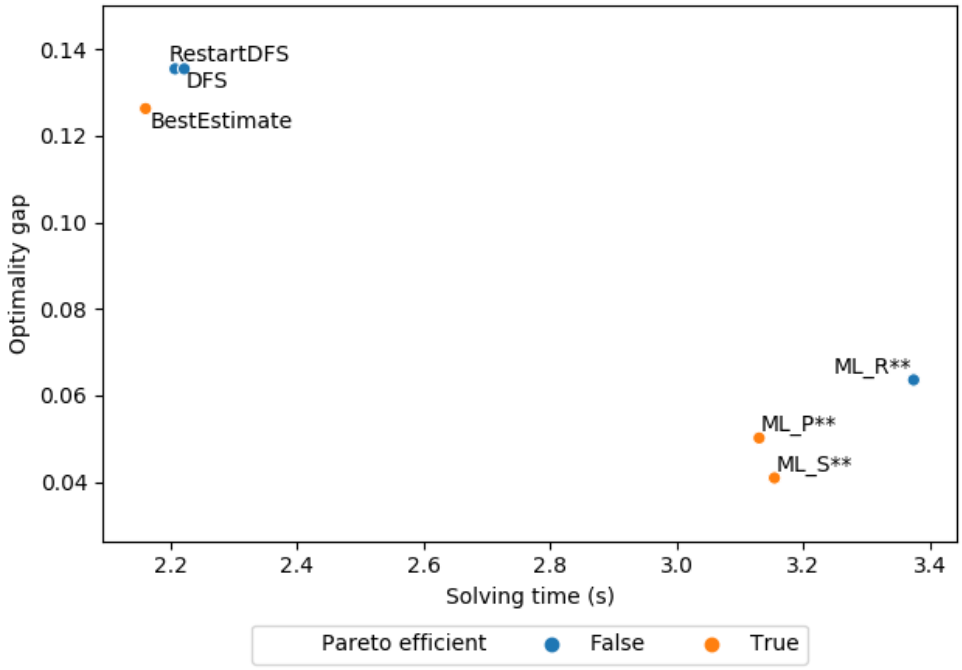

Figure 3. Set cover: mean solving time vs. mean optimality gap of the first solution found at a leaf node. Orange points are Pareto-efficient, blue are not. '**' $p < 0.01$.

Table 6 reports the mean solving time and explored nodes of various exact node selection strategies. *BestEstimate* achieves the lowest mean solving time at 26.4 s; ML_RB comes next at 35.7 s. The policies of He et al. [5]–both the original and our re-implementation–are markedly slower than all other methods.

We conducted a pair-wise *t*-test between the mean solving time of *BestEstimate* and the mean solving time of the other policies. We can only reject the null hypothesis of equal means with *p*-value below 0.1 for ML_RR (*p*-value: 0.08). For the rest of our ML policies, we cannot reject the null hypothesis of equal means. We also conducted a pair-wise *t*-test between the mean number of explored nodes of *BestEstimate* and the mean number of explored nodes of the other policies. Again, we cannot reject the null hypothesis of equal means with *p*-value below 0.05 for any of our ML policies (lowest observed *p*-value: 0.34).

Table 7 reports the mean optimality gap of the baselines under a *time limit* for each instance. The time limit for each instance is based on the solving time of the ML policy that achieved the lowest harmonic mean between the mean solving time and mean optimality

gap across all instances. This time limit fosters comparison because it is certain to be neither excessively low, so that the different methods would accomplish little, nor excessively high, so that all methods would find an optimal solution. In the case of the set cover instances, ML_SRF has the lowest harmonic mean and also achieves the lowest mean optimality gap. We conducted a pairwise *t*-test between the mean optimality gap of our best ML policy and the mean optimality gap of each baseline. We can reject the null hypothesis of equal means with *p*-value below 0.005 for all baselines. This shows that ML_SRF's smaller optimality gap is significant.

The initial optimality gap obtained by the solver before branch-and-bound is 2.746. This shows that applying branch-and-bound–with whatever node selection strategy–to find a solution has a significant difference.

Table 6. Set cover instances: mean solving time and explored nodes for various node selection strategies. Pair-wise *t*-tests compared to *BestEstimate*: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$.

Strategy	Solving Time (s)	Explored Nodes
BestEstimate	26.39	4291
DFS	47.10 *	9018 *
He (S)	499.00 ***	47,116 ***
He6 (S)	85.73 **	18,386 **
ML_PB	36.82	4573
ML_PR	38.82	5241
ML_PS	39.73	5295
ML_RB	35.68	4663
ML_RR	40.09 ·	5666
ML_RS	37.13	4799
ML_SB	35.90	4408
ML_SR	36.64	4776
ML_SS	37.61	4923
RestartDFS	45.30 *	8420 *

Table 7. Set cover: ML model with *on_both* = Second, *on_leaf* = RestartDFS and *prune_on_both* = False against baselines, with equal time limits for each problem. Pairwise *t*-tests against the ML policy: '***' $p < 0.001$, '**' $p < 0.01$.

Strategy	Optimality Gap
BestEstimate	0.1767 **
DFS	0.0718 ***
He6 (P)	0.7988 ***
He6 (B)	1.1040 ***
ML_SRF	0.0278
RestartDFS	0.0741 ***

4.3. Three Further Problem Classes

The appendix gives detailed results for the maximum independent set, capacitated facility location, and combinatorial auctions problems. Briefly, on maximum independent set, baseline *DFS* dominates, and the ML policies are relatively poor; this is the only problem class where *He6* outperforms our policies slightly (although we outperform the original *He* easily). On capacitated facility location, our ML policies give the best results on solving time and on explored nodes, although the results are not significantly better than *BestEstimate* according to the *t*-test. On combinatorial auctions, *BestEstimate* dominates, although its explored nodes are not significantly fewer than our ML policies.

When a time limit is applied, our best ML policy either achieves the best optimality gap (as with set cover problems), or is Pareto-equivalent to the baseline which does (interestingly, this is *DFS*, never *BestEstimate* in these problems). This in in contrast to *He6*, which in all problems is significantly poorer in optimality gap obtained.

4.4. Set Cover: Hard Instances

To assess how the ML policies perform on hard instances, we use the same trained model of the ML policies that were previously trained on the easier set cover instances. The hard instances consist of 4000 variables and 2000 constraints, while the easier set cover instances had 2000 variables and 1000 constraints. We evaluated 10 hard instances (due to computational limitations) and focused on *BestEstimate* as a baseline on the node selection policy. Primal heuristics and pre-solving were enabled.

For the pruning policies, Figure 4 shows the solving time plotted against the integrality gap of the first solution obtained by the baselines and ML policies. As before, we see the same trend where the ML policies find a lower gap at the cost of a higher solving time. See Table 8 and Figure 5 for the integrality gap of the baselines using a time limit for each instance. In this case, ML_PST has the lowest harmonic mean between the mean solving time and mean integrality gap of all ML policies. ML_PST achieves a significantly lower integrality gap compared to the baselines: we conducted a pairwise *t*-test between the mean optimality gap of our best ML policy and the mean optimality gap of each baseline. We can reject the null hypothesis of equal means with *p*-value below 0.001 for all baselines.

For the node selection policies, we set the time limit to one hour per problem. Figure 6 shows the number of solved instances per policy, and Figure 7 shows boxplots of the integrality gaps for each policy. We use integrality gap here, because we do not know the optimal objective value for all instances. Table 9 shows the mean solving time, explored nodes and integrality gap of various node selection strategies. *BestEstimate* achieves the lowest mean solving time at 2256.7 seconds and integrality gap at 0.0481. ML_SB has the lowest number of explored nodes at 109298. A caveat over the reduced number of nodes of the best ML policies versus *BestEstimate* comes from the solver timing out on many instances–the instances being hard. Fewer nodes could be due to the overhead that the ML classifier requires at each node.

We conducted a pair-wise *t*-test between the mean solving time of *BestEstimate* and the mean solving time of the other policies. We cannot reject the null hypothesis of equal means with *p*-value below 0.1 for all our ML policies (lowest observed *p*-value: 0.64). We also conducted a pair-wise *t*-test between the mean number of explored nodes of *BestEstimate* and the mean number of explored nodes of the other policies. We can reject the null hypothesis of equal means with *p*-value below 0.05 for ML_PB, ML_RB and ML_SB. Lastly, we conducted a pair-wise *t*-test between the mean integrality gap of *BestEstimate* and the mean integrality gap of the other policies. We can reject the null hypothesis of equal means with *p*-value below 0.1 for ML_RS and ML_SS. Summarising, ML_SB has a statistically significant fewer number of nodes, while not having a statistically significant higher time or integrality gap than *BestEstimate*.

Unlike the easier set cover instances, in all the hard instances, the solver could only obtain an integrality gap of infinity on the first feasible solution. Hence we cannot compare the initial integrality gap to the found integrality gaps during branch and bound. A possible reason for the solver's integrality gap of infinity could be that the first solution is found before the root LP relaxation was solved.

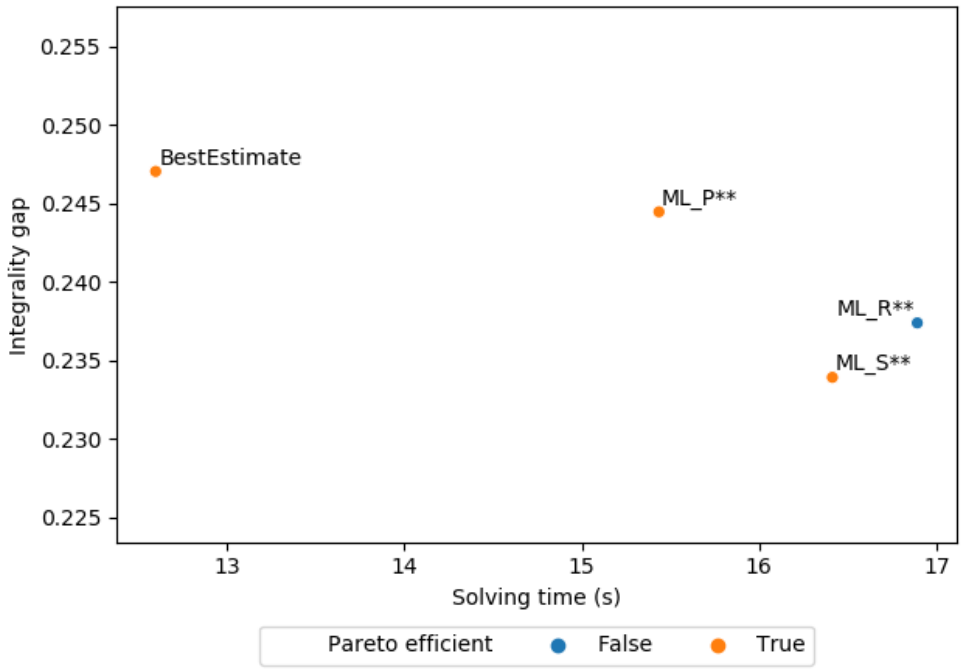

Figure 4. Hard set cover: mean solving time vs. mean integrality gap of the first solution found at a leaf node. Orange points are Pareto-efficient, blue are not. '**' $p < 0.01$.

Table 8. Hard set cover: ML model with *on_both* = PrioChild, *on_leaf* = Score and *prune_on_both* = True against the baselines, with equal time limits for each problem. Pairwise *t*-tests against the ML policy: '***' $p < 0.001$.

Strategy	Integrality Gap
BestEstimate	1.3678 ***
DFS	0.3462 ***
He6 (B)	1.8315 ***
He6 (P)	1.6835 ***
ML_PST	0.2445
RestartDFS	0.3489 ***

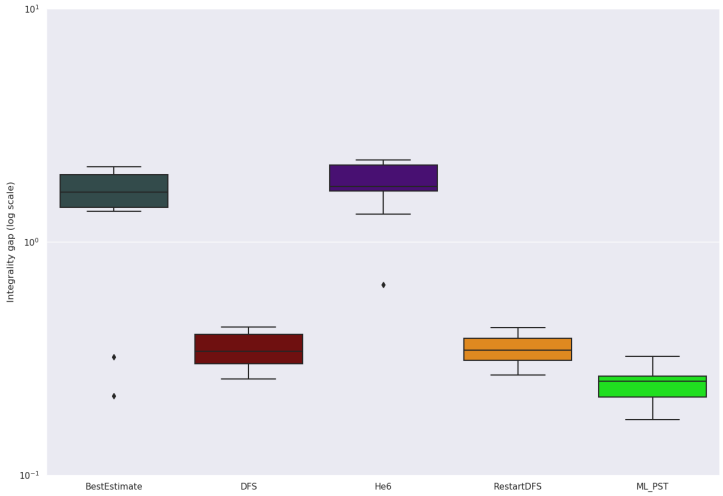

Figure 5. Hard set cover instances: integrality gap of various heuristic strategies. Note log scale on y-axis. ML_PST outperforms the compared methods.

Table 9. Hard set cover instances. Pair-wise *t*-tests against *BestEstimate*: '*' $p < 0.05$, '·' $p < 0.1$.

Strategy	Time (s)	Nodes	Integrality Gap
BestEstimate	2256.69	161,438	0.0481
ML_PB	2497.56	112,869 *	0.0714
ML_PR	2531.32	150,055	0.0763
ML_PS	2279.41	157,367	0.1152
ML_RB	2441.60	111,629 *	0.0683
ML_RR	2552.68	152,871	0.0794
ML_RS	2352.93	153,727	0.1289 ·
ML_SB	2427.25	109,298 *	0.0706
ML_SR	2685.09	161,825	0.0734
ML_SS	2381.28	163,631	0.1278 ·

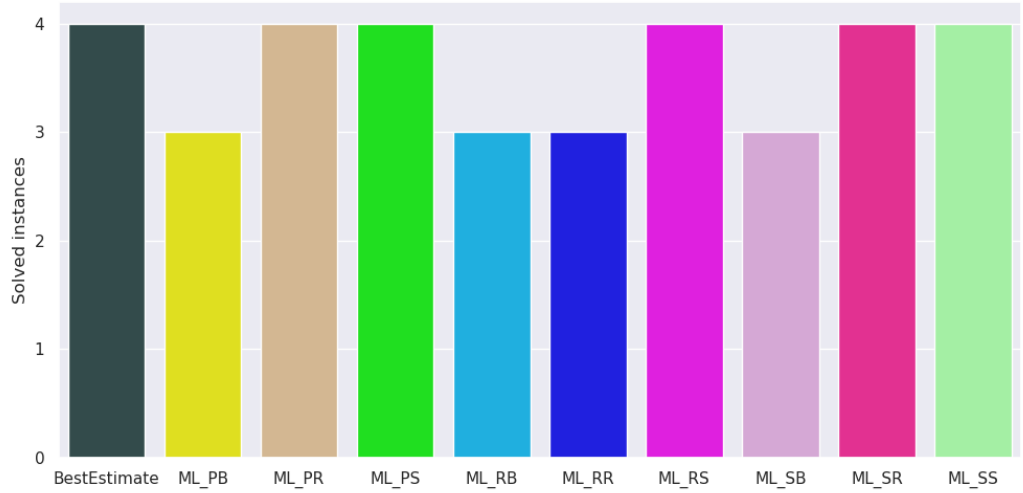

Figure 6. Hard set cover instances: number of solved instances (out of 10) for each node selection strategy.

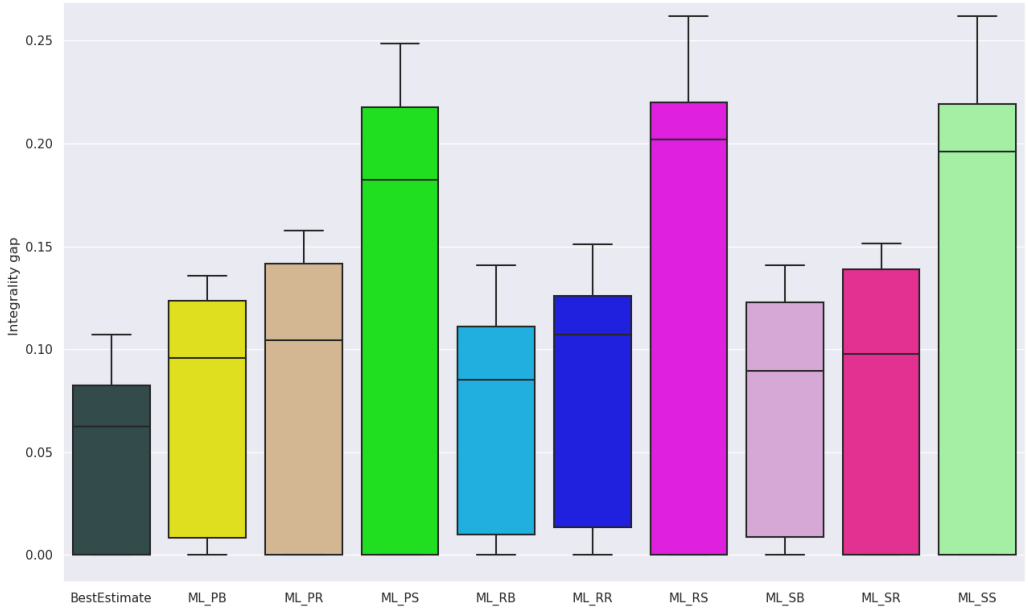

Figure 7. Hard set cover instances: integrality gap of each node selection strategy.

4.5. Success of the Learned Imitation

The results reported so far correspond to the first three experiments: optimality gap and solving time of first solution at a leaf node; total solving time to find an optimal solution and prove its optimality; and optimality gap within a fixed solving time. The results are on set cover (easy and hard instances), and three further problem classes.

We further analyze the behaviour of the learned policy during the first plunge, in detailed comparison to SCIP's default *BestEstimate* rule. Recall that, when plunging, *BestEstimate* chooses the priority child according to the *PrioChild* property (see Section 2), which is one of the input features to our method. It is also important to note that, while the learned policy plunges until finding a leaf node, *BestEstimate* may decide to abort the plunge early. This early abort decision is made according a set of pre-established parameters such as a maximum plunge depth (for more details we refer to the SCIP documentation).

In order to study the trade-off between better feasible solutions and the computational cost of obtaining them, we analyze the attained optimality gap against depth of the final node in the first plunge. This in contrast to our previous experiment, where we considered solving time as the second metric. We compare *BestEstimate* and the learned policy with on_both set to *PrioChild*. Further, we present experiments per instance, instead of aggregating them.

Figure 8 shows the results for the 35 (small) test instances. The effect of *BestEstimate*'s plunge abort becomes apparent. On instances where the first leaf has depth smaller than 14, both policies perform identically. On the contrary, on the remaining instances, *BestEstimate* hits the maximum plunge depth, hence aborting the plunge. These results show that the better optimality gap achieved by the learned policy comes at the cost of processing more nodes, i.e., plunging deeper into the tree. We also note that these results were obtained setting on_both to *PrioChild*, however no node was labelled with 'both' during our experiments. In spite of this, the learned policy chose the priority child on 99.6% of the occasions, demonstrating its strong imitation of that heuristic.

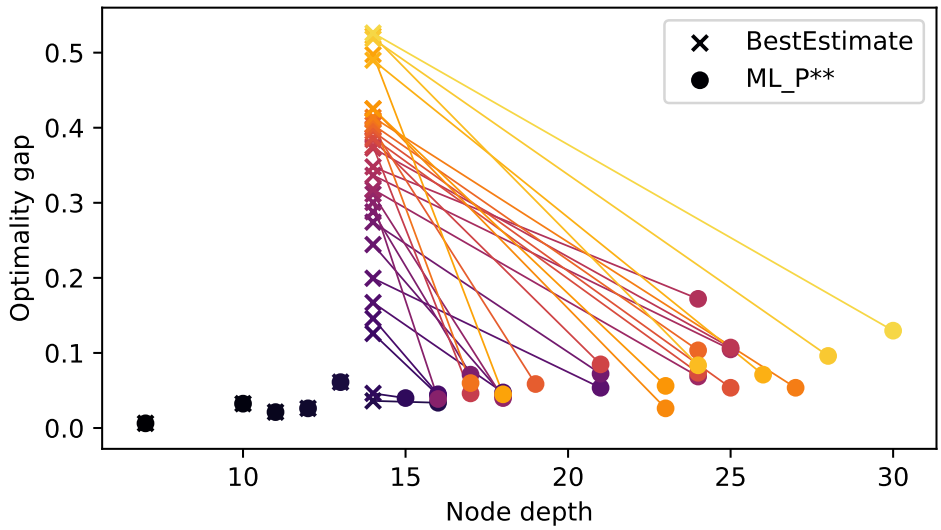

Figure 8. Set cover: node depth vs. optimality gap attained at the final node, after one plunge. Each point-pair corresponds to one instance: one for *BestEstimate* (×) and one for the learned policy (•). '**' $p < 0.01$.

5. Discussion and Limitations

This section reflects on our experiments and results, and discusses design decisions and possible alternative to explore.

5.1. Experiments

Summarising Section 4, we undertook three performance experiments–namely measuring the solving time and optimality gap for the first solution found at a leaf node in the branch and bound tree, measuring the total solving time for an exact solution, and setting a low instance-specific time limit to measure the optimality gap–and one exploratory experiment.

For the first experiment, the *PrioChild* ML policy (ML_P··) performed Pareto-equivalent on four of the five problem sets, performing inferior to the baselines only on maximum independent set instances. We turned off SCIP's primal heuristics in order to see the ability of the methods to find feasible solutions.

For the second experiment, our method performed better than the baselines in terms of mean solving time for capacitated facility location problem, but worse on the three purely binary problems. The best ML policy is ML_RB, although ML_SB has only slightly higher time, while exploring fewer nodes.

For the third experiment, we chose ML_SRF on set cover, ML_PRF on maximum independent set, ML_SST on capacitated facility location, and ML_PST on combinatorial auctions and hard set cover instances, in order to measure how well these best ML policies perform against the baselines. These policies were chosen based on the (lowest) harmonic mean between the solving time and optimality gap. In four of the five problem sets, our policies had a statistically-significant lower optimality gap than the baselines, while on combinatorial auctions, ML_PST did not perform statistically worse than *DFS* and *RestartDFS*.

For Experiments 2 and 3, we kept primary heuristics off, except for hard instances. It would be interesting to repeat the easy instances with primal heuristics on. This is because these heuristics improve the primal side of the bounds improvement while the branching rule takes care of dual side–and the node selection must balance both depending on the situation. The interesting question, which is answered in part by looking at the hard set cover instances, is whether the balance of primal and dual will indicate that the node selection should restrain its diving.

Overall we conclude that the *on_both* = *Random* configuration of the policy usually performs worse than the other configurations. *on_both* $\in$ {*PrioChild*, *Second*} both do well. The policies from both the *on_leaf* $\in$ {*Score*, *RestartDFS*} configurations perform better than those from the *on_leaf* = *BestEstimate* configuration. For both *prune_on_both* configurations, the policy performed well. Recall that when *prune_on_both* is True, then the search is terminated after the first leaf, saving solving time but resulting in a higher optimality gap. That both *prune_on_both* configurations lead to effective policies means that we offer the user the choice between a lower optimality gap and higher solving time, or the other way around.

Our method is effective when the ML model is able to meaningfully classify optimal child nodes correctly. By contrast, in the case of the maximum independent set problem, the classification was poor (base acc.: 0.895, test acc.: 0.899, gain: 0.004). Hence, when the predictive model adds value to the prediction, there is potential for effective decision making using the policy; and contrariwise when it does not.

We note that the feature extraction was the biggest contributor to the overall solving time. Applying the predictor had a rather small impact. This means that it is possible to achieve lower solving times by incorporating the entire process in the original C code of SCIP, avoiding the Python interface. However, Experiment 4 suggests that even a lower overhead in invoking the ML policy will not pay off.

Second, instead of actually pruning nodes, one could only assign low priorities to children considered inferior, such that they are never selected after a backtrack. This would automatically make the solver with the learned policy exact since it would never commit to pruning. However, the objective for our work was to explore learning approximate node selection policies. Second, node pruning has the advantage that the search does not visit the subtree of that node at all. Third, by pruning we follow the direction of He et al. [5] and can compare directly to the previous state-of-art in the literature. Indeed, as seen above,

our approach easily outperforms that of He et al. [5], in both their original implementation and a re-implementation in SCIP 6.

5.2. Using Best k Solutions

As discussed earlier, we set $k = 10$ for all our experiments bar one, with the hypotheses that, in general, higher values of k allows the ML model better potential to generalise. We found that lower k (below 10) gave inferior results in initial experiments. There is potential to vary k according to the problem class being solved. In particular, the distribution of nodes labelled 'L', 'R' and 'B'. If k is too high, it could be the number of 'B's become too high, leading to a class imbalance in training the ML model. On the other hand, another artefact of k being too low can be class imbalance, as we saw for maximum independent set instances.

We make three further remarks. First, we rank solutions using SCIP's solution ranking which is based on the optimality gap. It could be interesting to look also at the depth of the solutions in the tree, especially for the first setting of the heuristic policy use. Second, the k best solutions can vary in their objective value, possibly by relatively large amounts. An alternative to using any number of best solutions is to select an optimality gap bound. However, such an approach is more complicated than using the top-k. Third, in this article we trained on instances where the optimum was known. In the case it is not known, one could choose the best k solutions in terms of their integrality gap.

In Section 7 we discusses additional possible further work for choosing k.

5.3. Heuristic and Exact Settings

Recall that our learned policies were studied in two settings. In the first setting, the learned policy is used as a heuristic method. This is akin to a diving heuristic, with the difference that it uses the default branching rule as a variable fixing strategy. In more detail, this diving heuristic uses the branching rule to decide on which variable the disjunction will be made and then uses the child selector to choose the value to which the variable will be fixed (i.e., 0 or 1). In MIP branch-and-bound trees it is known that good branching rules are not good diving rules (attributed to T. Berthold). This is because the branching rule explicitly tries to balance the quality of the two children, in order to focus mostly on improving the lower bound, whereas a diving rule will try to construct an unbalanced (one-sided) tree. It remains to compare the learned policy against actual diving heuristics. We hypothesise that it will be inferior in terms of total time, because our heuristic uses the default branching rule as a variable fixing strategy.

In the second setting–the exact setting–the learned policy is used as a child selector during plunging. The potential of learning here is to augment node selection rules, such that the child selection process is better informed than the simple heuristics that solvers typically use. Recall that in the case of SCIP, the heuristic is *BestEstimate* with plunging. Indeed, our fourth experiment showed that, when used as a child selector, the learned policy acts almost exactly like SCIP's *PrioChild* rule. The main difference comes from the fact that SCIP's plunging has an abort mechanism. The policy could not learn this because policy learns to select a child, and then the node selection rule dives using this policy until no children are left. That the policy acts like *PrioChild* is no surprise given that this is the rule that was used to generate samples and it is also one of the features fed to the learner. Table shows that, in terms of solving time, the learned policy is not better than *BestEstimate* with plunging and its abort mechanism.

We conclude that that there is limited potential for improvement by selecting the best child during plunging. If the branching rule is working well, the two children should be quite balanced. By contrast, an interesting question is choosing a good node *after* plunging stops.

6. Related Work

We position our work in the literature on using machine learning in MIP branching, node selection, and other aspects of MIP solving.

6.1. Branching

Deciding on what variable to branch on in the branch and bound process is called branching, as was introduced in the main text of the paper. Good branching techniques make it possible to reduce the tree size, resulting in fast solving times. A survey on branching, and the use of learning to improve it, is by Lodi and Zarpellon [16].

Strong branching [17] is a popular branching strategy, among other strategies such as most-infeasible branching, pseudo-cost branching [8], reliability branching [9]–used as the default in SCIP–and hybrid branching [7]. Strong branching creates the smallest trees, as Achterberg et al. [9] reported that strong branching required around 20 times fewer nodes to solve a problem than most infeasible branching and around 10 times fewer nodes than pseudo-cost branching. However, strong branching is the most expensive to calculate, because two LP-relaxations are solved for every variable to assign scores.

Nonetheless, exact scores are not required to find the best variable to branch on. Therefore, it is interesting to approximate the score of strong branching, which can be done using machine learning. Alvarez et al. [18] was the first to use supervised learning to learn a strong branching model. The features they used to train the ML model consist of static problem features, dynamic problem features and dynamic optimisation features. The static problem features derive from c, A and b as stated in Equation (1). The dynamic problem features derive from the solution $\hat{x}$ of the current node in the branch and bound tree and the dynamic optimisation features derive from statistics of the current variable. They used the Extremely Randomized Trees (ExtraTree) classifier [19]. The results show that supervised learning successfully imitated strong branching, being 9% off relative to gap size, but 85% faster to calculate. Although strong branching was successfully imitated, it was still behind reliability branching in terms of gap size and runtime.

Khalil et al. [20] extended Alvarez et al. [18] work by adding new features to the machine learning model and by learning a pairwise ranking function instead of a scoring function. The ranking function they used is a ranking variant of Support Vector Machine (SVM) classifier [21]. Their algorithm solved 70% more hard problems (over 500,000 nodes, cut-off time 5 h) than strong branching alone. However, the time spent per node (18 ms) is higher than pseudo-cost branching (10 ms) and combining strong branching with pseudo-cost branching (15 ms). This is due to calculating the large number of features on every node.

To overcome complex feature calculation, Gasse et al. [12] proposes features based on the bipartite graph structure of a general MILP problem. The graph structure is the same for every LP relaxation in the branch-and-bound tree, which reduces the feature calculation cost. They use a graph convolutional neural network (GCNN) to train and output a policy, which decides what variable to branch on. Furthermore, they used cutting planes on the root node to restrict the solution space. Their GCNN model performs better than both Alvarez et al. [18] and Khalil et al. [20] for generalising branching, using few demonstration examples for the set covering, capacitated facility location, and combinatorial auction problems. Moreover, GCNN solved the combinatorial auction problem 75% faster than the method of Alvarez et al. [18] and 70% faster than the method of Khalil et al. [20], both for hard problems (1500 auctions). Seeing their success, we adopt the same variable features as Gasse et al. [12].

Other recent works learning branching rules are Zarpellon et al. [22], Gupta et al. [23], Yang et al. [24].

Nair et al. [25] develop a combination of branching rule and primal heuristic. The authors apply imitation learning to acquire a MIP brancher, based on full strong branching [9]. Separately, the authors apply generative modelling and supervised learning to acquire a 'diving rule'. Together, SCIP using the learned brancher and learned diving rule out-

performs default SCIP by an order of magnitude or more in terms of the primal integral. It is noteworthy that Nair et al. [25] train and test on relatively large MIP instances and use extensive amount of parallelised GPU computation in order to train their neural network models.

6.2. Node Selection and Pruning Policy

While learning to branch has been studied quite extensively, learning to select and prune nodes has received insufficient attention in the literature.

He et al. [5] used machine learning to imitate good node selection and pruning policies. The method of data collection in that work is by first solving a problem and provide its solution to the solver. Afterwards, the problem is solved again, but now that the solver knows the solution, it will take a shorter path to the solution. The features for learning the node selection policy are derived from the nodes in this path and the features for the node pruning policy are derived from the nodes that were not explored further. This was done for a limited amount of problems as the demonstrations.

He et al. [5] trained their machine learning algorithm on four datasets, called MIK, Regions, Hybrid and CORLAT. They were able to achieve prune rates of 0.48, 0.55, 0.02 and 0.24 for each dataset respectively. Prune rate shows the amount of nodes that did not have to be explored further relative to the total amount of nodes seen. Their solving time reached a speedup of 4.69, 2.30, 1.15 and 1.63 compared to a baseline SCIP version 3 heuristic respectively for each dataset. Note that the lowest speedup seems to correlate with a low prune rate.

Our work differs from He et al. [5] by constraining the node selection space to direct children only at non-leaf nodes. Second, we use the top k solutions to sample state-action pairs. By using more than one solution, we can create additional state-action pairs from which the neural network can learn and create a predictive model. Third, we take advantage of branched variable features, obtained from Gasse et al. [12]. As seen in Section 4, our approach easily outperforms that of He et al. [5], in both their original implementation and a re-implementation in SCIP 6.

Other recent works learning decisions related to node selection and pruning focus on learning primal heuristics, sometimes for a specific problem class [4,25,26].

7. Conclusions

This article shows that approximate solving of mixed integer programs can be achieved by a node selection policy obtained with offline imitation learning. Node selection is important because through it, the solver balances between exploring search nodes with good lower bounds (crucial for proving global optimality), and finding feasible solutions fast. In contrast to previous work using imitation learning, our policy is focused on learning to choose which of its children it should select. We apply the policy within the popular open-source solver SCIP, in exact and heuristic settings.

Empirical results on five MIP datasets indicate that our node selector leads to solutions more quickly than the state-of-the-art in the literature [5]. While our node selector is not as fast on average as the highly optimised state-of-practice in SCIP in terms of solving time on exact solutions, our heuristic policies have a consistently better optimality gap than all baselines if the accuracy of the predictive model is sufficient. Second, the results also indicate that our heuristic method finds better solutions within a given time limit than all baselines in the majority of the problem classes examined. Third, the results show that learned policies can be Pareto-equivalent or superior to state-of-practice MIP node selection heuristics. However, these results come with a caveat, summarized below.

This work adds to the body of literature that demonstrates how ML can benefit generic constraint optimisation problem solvers. In MIP terminology, our learned policy constitutes a diving rule, focusing on finding a good integer feasible solution. The performance on non-binary problem classes like capacitated facility location is particularly noteworthy. This is because, unlike purely binary problems, for non-binary instances, MIP primal

heuristics struggle to obtain decent primal bounds [11]. By contrast, in general for binary instances, the greater challenge is to close the dual bound, and our learned policy also performs well here.

The results are explained by the conclusion that the learned policies have imitated SCIP brancher's preferred rule for node selection, but without the latter's early plunge abort. This is a success for imitation learning, but does not overall improve the best state-of-practice baseline from which it has learned. Pareto-efficiency between total solving time and integrality gap is is important, yet total solving time and the primal integral quality are the most crucial in practice. However, an interesting question to pursue is learning to choose a good node *after* plunging stops.

Despite the clear improvements over the literature, then, this kind of MIP child selector is better seen in a broader approach to using learning in MIP branch-and-bound tree decisions [25]. While forms of supervised learning find success [4], reinforcement learning is also interesting [12], and could be used for node selection.

Nonetheless, for future work on node selection by imitation learning, more study could be undertaken for choosing the meta-parameter k. Values too low add only few state-action pairs, which naturally degrades the predictive power of neural networks. On the other hand, values too high add noise, as paths to bad solutions add state-action pairs that are not useful. An interesting direction is to exploit an oracle (solver) to decide whether a node is 'good' or 'bad', e.g., if the node falls onto a path of a solution within say 5% of the optimum value. This more expensive data collection method might eliminate choosing a specific k.

The way in which the ML policies were trained can be explored further. For instance, one could consider the two components of *PrioChild* separately, or could train on standard SCIP with and without early plunge abort. Comparing learned policies with and without primal heuristics and with and without pre-solve also has scientific value.

Lastly, Section 3 explained how during pre-processing certain features are removed which are constant throughout the entire dataset. This has the consequence of a different number of input units in the neural network architecture for every problem. Moreover, future work could include a method to unify a ML model that is effective for all problem classes. This would make ML-based node selection a more accessible feature for current MIP solvers such as SCIP; another promising direction in this regard is experimentation via the 'gym' of Prouvost et al. [27].

Author Contributions: Conceptualization, K.Y. and N.Y.-S.; methodology, K.Y.; software, K.Y.; analysis, K.Y.; writing—original draft preparation, K.Y.; writing—review and editing, N.Y.-S.; visualization, K.Y.; supervision, N.Y.-S.; project administration, N.Y.-S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under grant 952215, and by the Dutch Research Council (NWO) Groot project OPTIMAL.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Source code for the methods of this article available is available at www.doi.org/10.4121/14054330 (accessed on 1 September 2020). Problem instance were obtained from the generator of Gasse et al. [12] with default settings.

Acknowledgments: Thanks to Lara Scavuzzo Montaña for a number of contributions. Thanks to Robbert Eggermont for computational support. We thank the anonymous reviewers of this article, and those who commented on the arXiv preprint of this work since July 2020.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Results on Three Further Problems Sets

Besides easy and hard set cover instances, as reported in Section 4, we studied our ML-based node selection and pruning policies on three further standard datasets. These are maximum independent set, capacitated facility location and combinatorial auctions. Instances were obtained from the generator of Gasse et al. [12] with default settings.

Appendix A.1. Maximum Independent Set

These instances consist of 1000 variables and around 4000 constraints forming a pure binary maximisation problem. For this particular problem, we noticed that for $k = 10$, the class imbalance was significant. To mitigate this imbalance, we increased the value k to 40 (see the discussion in Section 5). For $k = 10$, we sampled 29,801 state-action pairs on the training instances, 5820 on the validation instances and 4,639 on the testing instances. The class distribution is: (Left: 92%, Right: 4%, Both: 4%). For $k = 40$, we sampled 82,986 state-action pairs on the training instances, 14,460 on the validation instances and 14,273 on the testing instances. The class distribution is: (Left: 89%, Right: 4%, Both: 7%).

Both the $k = 10$ and $k = 40$ models achieve a testing accuracy that is very close to the baseline accuracy, which results in a model that is not able to generalise. See Table A1 for the average solving time and explored nodes of various node selection strategies. We conducted a pair-wise t-test between the mean solving time of *BestEstimate* and the mean solving time of the other policies. We can reject the null hypothesis of equal means with p-value below 0.1 for all our ML policies. We have also conducted a pair-wise t-test between the mean number of explored nodes of *BestEstimate* and the mean number of explored nodes of the other policies. We can reject the null hypothesis of equal means with p-value below 0.1 for 12 of 18 of our ML policies.

Table A1. Maximum independent set instances: average solving time and explored nodes for various node selection strategies. Pair-wise t-tests against *BestEstimate*: '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$

Strategy	Solving Time (s)	Explored Nodes
BestEstimate	158.42	6344
DFS	155.27	6340
He (S)	394.56 **	30965 **
He6 (S)	204.68	7992
ML_PB (10)	260.34 **	10,029 *
ML_PB (40)	227.45 *	8100
ML_PR (10)	255.49 **	10,191 *
ML_PR (40)	222.59 *	8581
ML_PS (10)	245.92 *	10,184 ·
ML_PS (40)	207.28 ·	7878
ML_RB (10)	274.17 **	10,296 *
ML_RB (40)	222.80 *	8006
ML_RR (10)	244.01 **	9654 *
ML_RR (40)	213.81 ·	8196
ML_RS (10)	232.67 *	9582 ·
ML_RS (40)	207.71 ·	8137
ML_SB (10)	285.21 **	10,661 *
ML_SB (40)	283.02 **	10,491 *
ML_SR (10)	252.59 **	9936 *
ML_SR (40)	258.68 **	10,120 *
ML_SS (10)	242.37 *	9921 ·
ML_SS (40)	274.73 **	11,251 *
RestartDFS	183.24	7854

For both $k = 10$ and $k = 40$, *DFS* achieved the lowest average solving time on node selection at 155.3 s, while the $k = 10$ ML_RS model achieved 232.7 s and the $k = 40$ ML_PS model an average solving time of 207.3 s.

Figure A1 shows that the first solution quality and solving time of ML policies are all near each other and dominated by *RestartDFS* and *DFS*. Note that in the plot the suffix ('**') is replaced by the value of k. Table A2 examines how the node pruner compares to the baselines, when the baselines have a set time limit. In this case, ML_PRF has the lowest harmonic mean between the average solving time and average optimality gap of all ML policies. The ML policy has a higher average optimality gap than the baselines for this problem. We conducted a pairwise t-test between the mean optimality gap of our best ML policy and the mean optimality gap of each baseline. We can reject the null hypothesis of equal means with p-value below 0.05 for all baselines, except BestEstimate (p-value: 0.334). The initial optimality gap obtained by the solver before branch-and-bound is 0.999. This shows that He6 policy prunes aggressively at the start, because the average optimality gap obtained by He6 is similar to initial optimality gap. The other policies find significantly better solutions.

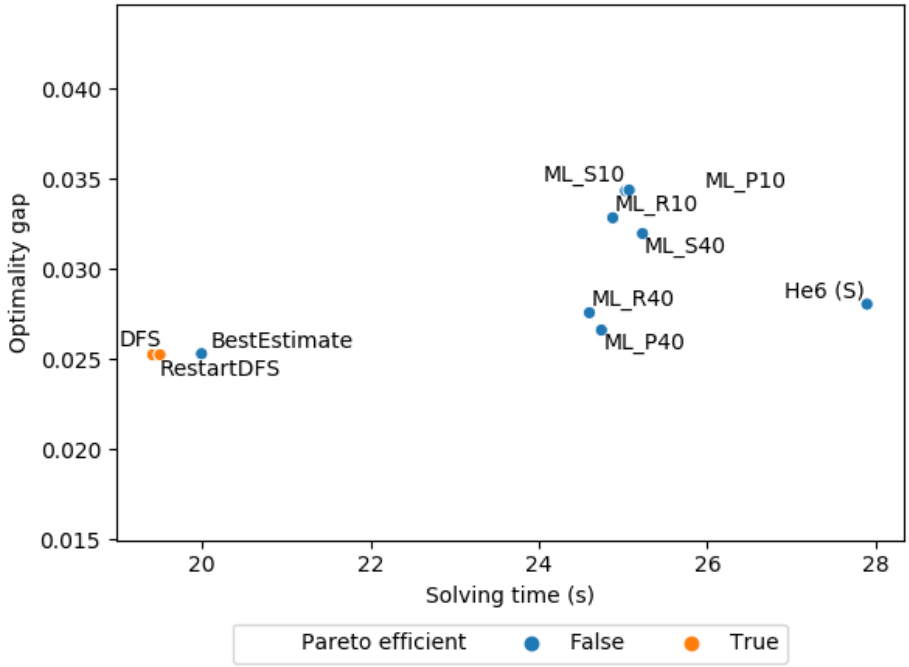

Figure A1. Maximum independent set: average solving time against the average optimality gap of the first solution found at a leaf node.

Table A2. Maximum independent set: model ($k = 40$) with *on_both* = PrioChild, *on_leaf* = RestartDFS and *prune_on_both* = False against baselines, with equal time limits for each problem. The initial optimality gap obtained by the solver before branch-and-bound is 0.999. Pairwise t-tests against the ML policy: '***' $p < 0.001$, '*' $p < 0.05$.

Strategy	Optimality Gap
BestEstimate	0.0174
DFS	0.0134 *
He6 (B)	0.9930 ***
He6 (P)	0.9902 ***
ML_PRF	0.0211
RestartDFS	0.0134 *

Appendix A.2. Capacitated Facility Location

These instances consist of 150 binary variables, 22,500 continuous variables and 300 constraints, forming a mixed-integer minimisation problem. We sampled 17,266 state-action pairs on the training instances, 3531 on the validation instances and 3431 on the testing instances. The model achieves a testing accuracy of 90.1%, with a baseline of 73.1%.

See Table A3 for the average solving time and explored nodes of various node selection strategies. ML_RB achieves the lowest average solving time at 111.7 seconds and ML_SS the lowest average explored nodes at 1099. We conducted a pair-wise *t*-test between the mean solving time of *BestEstimate* and the mean solving time of the other policies. We can not reject the null hypothesis of equal means with *p*-value below 0.1 for any our ML policies (lowest observed *p*-value: 0.14). We have also conducted a pair-wise *t*-test between the mean number of explored nodes of *BestEstimate* and the mean number of explored nodes of the other policies. We can not reject the null hypothesis of equal means with *p*-value below 0.1 for any our ML policies (lowest observed *p*-value: 0.18).

Table A3. Capacitated facility location instances: average solving time and explored nodes for various node selection strategies. Pair-wise *t*-tests against *BestEstimate*: '***' $p < 0.001$.

Strategy	Solving Time (s)	Explored Nodes
BestEstimate	122.79	1674
DFS	690.40 ***	17155 ***
He (S)	1754.10 ***	5733 ***
He6 (S)	373.32 ***	6799 ***
ML_PB	114.41	1189
ML_PR	148.21	1740
ML_PS	118.36	1207
ML_RB	111.67	1163
ML_RR	147.85	1773
ML_RS	125.99	1344
ML_SB	111.69	1109
ML_SR	133.68	1462
ML_SS	115.39	1099
RestartDFS	444.36 ***	9977 ***

Figure A2 shows the average solving time against the average optimality gap of the first solution obtained by the baselines and ML policies at a leaf node. We see here that the ML policies are clustered and obtain a lower optimality gap than the baselines. Note that *RestartDFS* and *DFS* have a very similar optimality gap and solving time, so they are stacked on top of each other. See Table A4 for the optimality gap of the baselines using a time limit for each instance. In this case, ML_SST has the lowest harmonic mean between the average solving time and average optimality gap of all ML policies. ML_SST also achieves a significantly lower average optimality gap than the baselines. We conducted a pairwise *t*-test between the mean optimality gap of our best ML policy and the mean optimality gap of each baseline. We can reject the null hypothesis of equal means with *p*-value below 0.001 for all baselines. The initial optimality gap obtained by the solver before branch-and-bound is 0.325. All policies find a significantly better solution than the first found feasible solution.

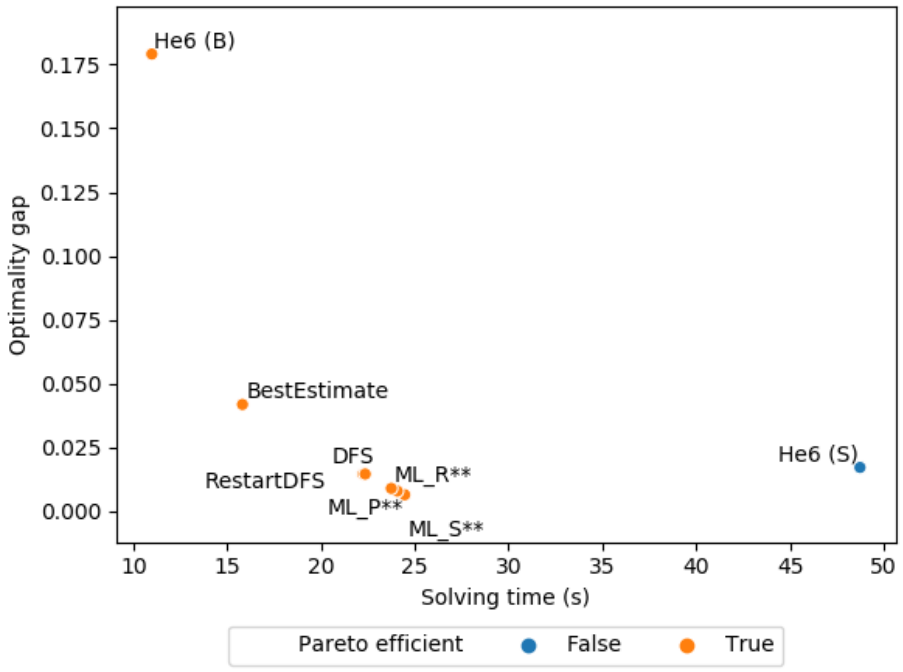

Figure A2. Capacitated facility location: average solving time against the average optimality gap of the first solution found at a leaf node. '**' $p < 0.01$.

Table A4. Capacitated facility location: model with *on_both* = Second, *on_leaf* = Score and *prune_on_both* = True against baselines, with equal time limits for each problem. The initial optimality gap obtained by the solver before branch-and-bound is 0.325. Pairwise *t*-tests against the ML policy: '***' $p < 0.001$.

Strategy	Optimality Gap
BestEstimate	0.0821 ***
DFS	0.0619 ***
He6 (B)	0.1516 ***
He6 (P)	0.1503 ***
ML_SST	0.0065
RestartDFS	0.0590 ***

Appendix A.3. Combinatorial Auctions

These instances consist of 1200 variables and around 475 constraints forming a pure binary maximisation problem. We sampled 13,554 state-action pairs on the training instances, 2389 on the validation instances and 2170 on the testing instances. The model achieves a testing accuracy of 71.7%, with a baseline of 57.0%.

See Table A5 for the average solving time and explored nodes of various node selection strategies. *BestEstimate* achieves the lowest average solving time at 19.7 s. We conducted a pair-wise *t*-test between the mean solving time of *BestEstimate* and the mean solving time of the other policies. We can reject the null hypothesis of equal means with *p*-value below 0.05 for all our ML policies. We have also conducted a pair-wise *t*-test between the mean number of explored nodes of *BestEstimate* and the mean number of explored nodes of the other policies. We can reject the null hypothesis of equal means with *p*-value below 0.1 for ML_PS and ML_RS.

Table A5. Combinatorial auctions instances: average solving time and explored nodes for various node selection strategies. Pair-wise *t*-tests against *BestEstimate*: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$.

Strategy	Solving Time (s)	Explored Nodes
BestEstimate	**19.68**	**3489**
DFS	23.48	4490
He (S)	40.11 ***	4519
He6 (S)	30.44 **	6358 *
ML_PB	29.77 *	4288
ML_PR	29.82 **	4419
ML_PS	32.51 **	4811 ·
ML_RB	30.08 *	4494
ML_RR	30.89 **	4715
ML_RS	32.68 **	5190 ·
ML_SB	28.94 *	4187
ML_SR	29.83 **	4458
ML_SS	30.52 **	4567
RestartDFS	22.35	4299

Figure A3 shows the average solving time against the average optimality gap of the first solution obtained by the baselines and ML policies at a leaf node. We see here that the ML policies are not as clustered. The ML_P** strategy is the only strategy that delivers Pareto efficient result, having a both a lower optimality gap and a lower solving time. Note that *BestEstimate*, *RestartDFS* and *DFS* have a very similar optimality gap and solving time, so they are stacked on top of each other. See Table A6 for the optimality gap of the baselines using a time limit for each instance. In this case, ML_PST has the lowest harmonic mean between the average solving time and average optimality gap of all ML policies. ML_PST achieves a very similar optimality gap compared to the baselines. We conducted a pairwise *t*-test between the mean optimality gap of our best ML policy and the mean optimality gap of each baseline. We can reject the null hypothesis of equal means with *p*-value below 0.05 for *BestEstimate* and *He6*, but not *DFS* (*p*-value: 0.94) and *RestartDFS* (*p*-value: 0.98). The initial optimality gap obtained by the solver before branch-and-bound is 0.914. All policies find a significantly better solution than the first found feasible solution.

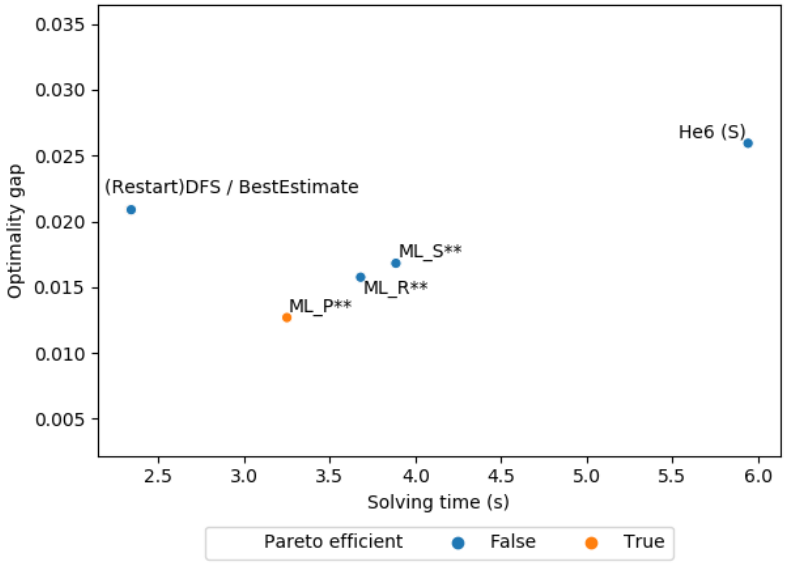

Figure A3. Combinatorial auctions: average solving time against the average optimality gap of the first solution found at a leaf node. '**' $p < 0.01$.

Table A6. Combinatorial auctions: model with *on_both* = PrioChild, *on_leaf* = Score and *prune_on_both* = True against baselines, with equal time limits for each problem. The initial optimality gap obtained by the solver before branch-and-bound is 0.914. Pairwise *t*-tests against the ML policy: '***' $p < 0.001$, '*' $p < 0.05$.

Strategy	Optimality Gap
BestEstimate	0.0174 *
DFS	**0.0126**
He6 (B)	0.4678 ***
He6 (P)	0.2873 ***
ML_PST	0.0127
RestartDFS	0.0126

References

1. Bayliss, C.; Maere, G.D.; Atkin, J.A.D.; Paelinck, M. A simulation scenario based mixed integer programming approach to airline reserve crew scheduling under uncertainty. *Ann. OR* **2017**, *252*, 335–363. [CrossRef]
2. Lombardi, M.; Milano, M.; Bartolini, A. Empirical decision model learning. *Artif. Intell.* **2017**, *244*, 343–367. [CrossRef]
3. Gleixner, A.; Bastubbe, M.; Eifler, L.; Gally, T.; Gamrath, G.; Gottwald, R.L.; Hendel, G.; Hojny, C.; Koch, T.; Lübbecke, M.E.; et al. *The SCIP Optimization Suite 6.0*; Technical Report, 2018. ZIB-Report 18-26, Zuse Institute Berlin. Available online: http://www.optimization-online.org/DB_HTML/2018/07/6692.html (accessed on 1 September 2020).
4. Bengio, Y.; Lodi, A.; Prouvost, A. Machine learning for combinatorial optimization: A methodological tour d'horizon. *Eur. J. Oper. Res.* **2021**, *290*, 405–421. [CrossRef]
5. He, H.; Daumé, H., III; Eisner, J. Learning to Search in Branch and Bound Algorithms. In Proceedings of the 2014 International Conference on Neural Information Processing Systems Conference (NeurIPS'14), Montreal, QC, Canada, 3–6 December 2014; pp. 3293–3301.
6. Land, A.H.; Doig, A.G. An automatic method of solving discrete programming problems. *Econometrica* **1960**, *28*, 497–520. [CrossRef]
7. Achterberg, T.; Berthold, T. Hybrid Branching. In Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09), Pittsburgh, PA, USA, 20 May 2009; Volume 5547, pp. 309–311.
8. Bénichou, M.; Gauthier, J.; Girodet, P.; Hentges, G.; Ribière, G.; Vincent, O. Experiments in mixed-integer linear programming. *Math. Program.* **1971**, *1*, 76–94. [CrossRef]
9. Achterberg, T.; Koch, T.; Martin, A. Branching rules revisited. *Oper. Res. Lett.* **2005**, *33*, 42–54. [CrossRef]
10. Achterberg, T. Constraint Integer Programming. Ph.D. Thesis, Technische Universität Berlin, Berlin, Germany, 2007.
11. Achterberg, T.; Berthold, T.; Koch, T.; Wolter, K. Constraint Integer Programming: A New Approach to Integrate CP and MIP. In Proceedings of the 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR'08), Paris, France, 25 May 2008; Volume 5015, pp. 6–20.
12. Gasse, M.; Chételat, D.; Ferroni, N.; Charlin, L.; Lodi, A. Exact Combinatorial Optimization with Graph Convolutional Neural Networks. In Proceedings of the 2019 International Conference on Neural Information Processing Systems (NeurIPS'19), Vancouver, BC, Canada, 17 December 2019; pp. 15554–15566.
13. Bergstra, J.; Yamins, D.; Cox, D.D. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In Proceedings of the 30th International Conference on Machine Learning (ICML'13), Atlanta, GA, USA, 9 June 2013; pp. 115–123.
14. Kingma, D.P.; Ba, J. Adam: A Method for Stochastic Optimization. In Proceedings of the 3rd International Conference on Learning Representations (ICLR'15), San Diego, CA, USA, 14 May 2015.
15. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Proceedings of the 2019 International Conference on Neural Information Processing Systems (NeurIPS'19), Vancouver, BC, Canada, 20 December 2019; pp. 8024–8035.
16. Lodi, A.; Zarpellon, G. On learning and branching: A survey. *TOP* **2017**, *25*, 207—236. [CrossRef]
17. Applegate, D.; Bixby, R.; Chvátal, V.; Cook, W. *Finding Cuts in the TSP (A Preliminary Report)*; Technical Report 5; Center for Discrete Mathematics & Theoretical Computer Science: Piscataway, NJ, USA, 1995.
18. Alvarez, A.M.; Louveaux, Q.; Wehenkel, L. A supervised machine learning approach to variable branching in branch-and-bound. In Proceedings of the 7th European Machine Learning and Data Mining Conference (ECML-PKDD'14), Nancy, France, 3 September 2014.
19. Geurts, P.; Ernst, D.; Wehenkel, L. Extremely randomized trees. *Mach. Learn.* **2006**, *63*, 3–42. [CrossRef]

20. Khalil, E.B.; Le Bodic, P.; Song, L.; Nemhauser, G.; Dilkina, B. Learning to branch in mixed integer programming. In Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI'16), Phoenix, AZ, USA, 25 February 2016; pp. 724–731.
21. Joachims, T. Training linear SVMs in linear time. In Proceedings of the 12th International Conference on Knowledge Discovery and Data Mining (KDD'06), Philadelphia, PA, USA, 10 August 2006; pp. 217–226.
22. Zarpellon, G.; Jo, J.; Lodi, A.; Bengio, Y. Parameterizing Branch-and-Bound Search Trees to Learn Branching Policies. *arXiv* **2020**, arXiv:2002.05120.
23. Gupta, P.; Gasse, M.; Khalil, E.B.; Mudigonda, P.K.; Lodi, A.; Bengio, Y. Hybrid Models for Learning to Branch. In Proceedings of the Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, Virtual / Vancouver, BC, Canada, 6 December 2020.
24. Yang, Y.; Boland, N.; Dilkina, B.; Savelsbergh, M. Learning Generalized STRONG branching for Set Covering, Set Packing, and 0-1 Knapsack PROBLEMS. Technical Report, 2020. Available online: http://www.optimization-online.org/DB_HTML/2020/02/7626.html (accessed on 1 January 2021).
25. Nair, V.; Bartunov, S.; Gimeno, F.; von Glehn, I.; Lichocki, P.; Lobov, I.; O'Donoghue, B.; Sonnerat, N.; Tjandraatmadja, C.; Wang, P.; et al. Solving Mixed Integer Programs Using Neural Networks. *arXiv* **2020**, arXiv:2012.13349.
26. Ding, J.; Zhang, C.; Shen, L.; Li, S.; Wang, B.; Xu, Y.; Song, L. Accelerating Primal Solution Findings for Mixed Integer Programs Based on Solution Prediction. In Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI'20), New York, NY, USA, 4 February 2020; pp. 1452–1459.
27. Prouvost, A.; Dumouchelle, J.; Scavuzzo, L.; Gasse, M.; Chételat, D.; Lodi, A. Ecole: A Gym-like Library for Machine Learning in Combinatorial Optimization Solvers. *arXiv* **2020**, arXiv:2011.06069.