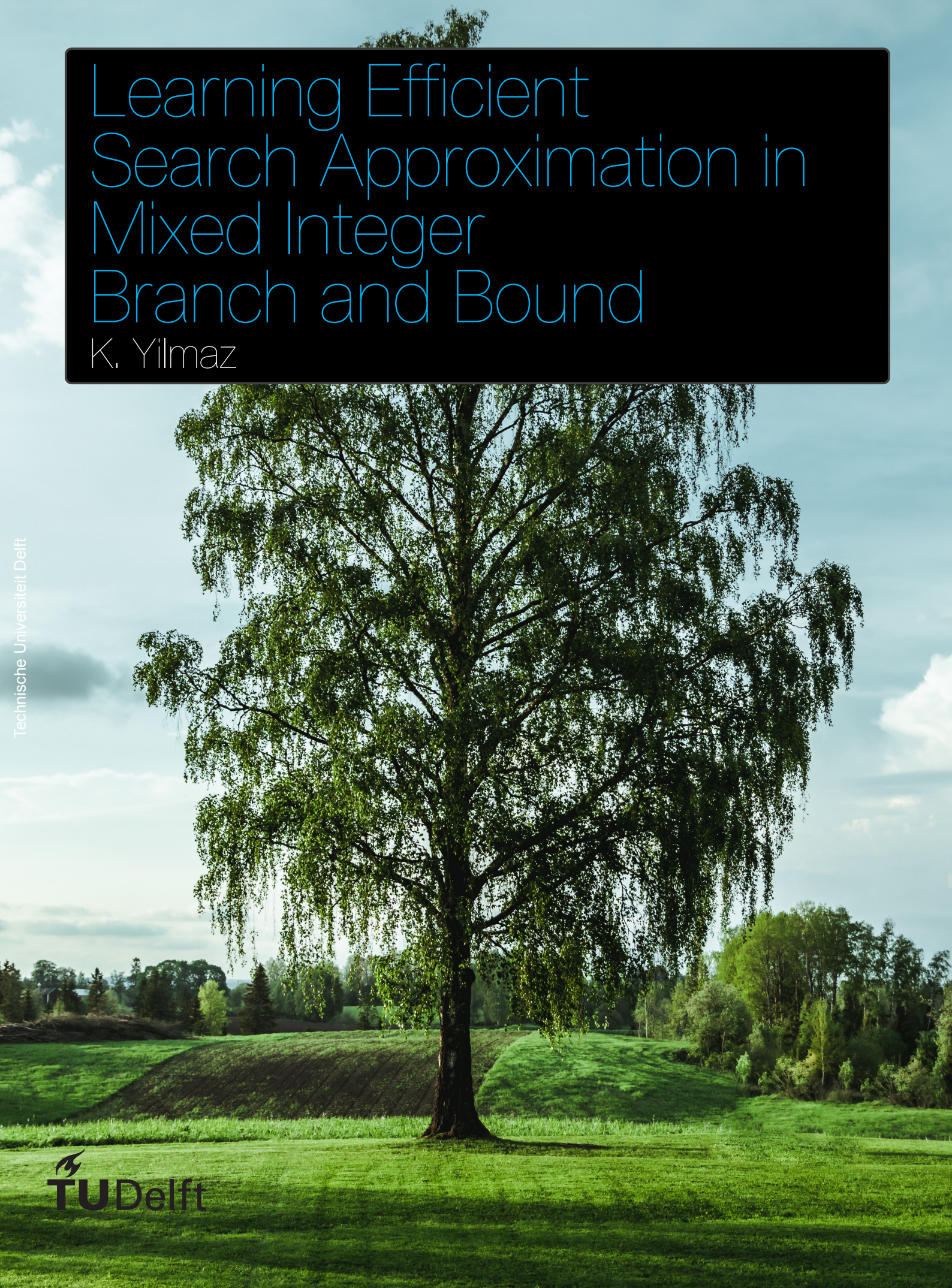
Learning Efficient Search Approximation in Mixed Integer Branch and Bound

K. Yilmaz



Technische Universiteit Delft

TU Delft

Learning Efficient Search Approximation in Mixed Integer Branch and Bound

by

K. Yilmaz

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 16, 2020 at 13:45.

Student number: 4385616
Project duration: September 1, 2019 – July 16, 2020
Thesis committee: Dr. N. Yorke-Smith, TU Delft, supervisor
Prof. K. Aardal, TU Delft
Dr. D. Gijswijt, TU Delft

TU Delft

Abstract

In line with the growing trend of using machine learning to improve solving of combinatorial optimisation problems, one promising idea is to improve node selection within a mixed integer programming branch-and-bound tree by using a learned policy. In contrast to previous work using imitation learning, our policy is focused 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 is approximate by committing to pruning of nodes; one that is exact and backtracks from a leaf to use a different strategy. We apply the policy within the popular open-source solver SCIP. Empirical results on four MIP datasets indicate that our node selection policy leads to solutions more quickly than the state-of-the-art in the literature, but not as quickly as the state-of-practice SCIP node selector. While we do not beat the highly-optimised SCIP baseline in terms of solving time on exact solutions, our approximation-based policies have a consistently better optimality gap than all baselines if the accuracy of the predictive model adds value to prediction. Further, the results also indicate that, when a time limit is applied, our approximation method finds better solutions than all baselines in the majority of problems tested.

Preface

I started this thesis in September 2019 after being inspired by the Vehicle Routing Problem project done a few months prior in the course Intelligent Decision Making Project at the Technical University of Delft.

During my research, countless hours were spent figuring out how the open-source solver SCIP worked, interfacing it with a Python framework, trying different methods and running experiments. Staying up late to check the results of the experiments became the norm. Ultimately, I found a simple, elegant method that works and I am proud to present it to you.

I found a new appreciation for researchers who do this for a living, as this work was the hardest test of my academic life. At the cost of my blood, sweat and tears, doing this research gave me the opportunity to explore this relatively new field of combining machine learning and mixed integer programming, and I can say it was more than worth it.

I could not have done this without the guidance of my supervisor Dr. Neil Yorke-Smith. I am grateful for the detailed feedback by Lara Scavuzzo Montaña and my peers in the Algorithmics group. Last, but not least, I would like to thank Robbert Eggermont for the university cluster access.

K. Yilmaz
Delft, July 2020

Contents

1 Introduction

Hard constrained optimisation problems (COPs) exist in many different applications. Examples include airlines, that schedule flights, such that profit is maximised and constraints involving aircraft maintenance and crew scheduling are satisfied [6]. Additionally, the experimental multicore CPU of Intel, called the ”Single-chip Cloud Computing”, consists of 24 dual-core tiles where jobs are mapped to cores that maximise efficiency and meet the temperature constraints of every core [22]. Perhaps the most common paradigm for modelling and solving COPs 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 [15].

A growing trend is to use machine learning (ML) to improve COP solving. Bengio et al. [9] 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 [16]. A policy is a function that maps states to actions, where in this work 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. [16].

1.1. Research objective

The research goal of this thesis is to find a node selection (exact) and pruning (approximation) policy to improve over current heuristics. Current heuristics include *Depth-First Search* (*DFS*), *BestEstimate* and *RestartDFS* (see Chapter 2). Additionally, we want to improve on the work of He et al. [16], which is the only ML-based node selection and pruning policy.

Our motivation is that current heuristics are too simplistic to fully assess the richness of the MIP solving process. Additionally, we want to add to the ML-based node selection literature, as the current body of literature is inadequate.

The research questions are:

1. **Can a node selector be created that uses a policy obtained from machine learning, that can improve on the solving times of the default SCIP node selector and the current state-of-the-art?**

2. **Can a node pruner be created that uses a policy obtained from machine learning, that can improve on the solving times or solution quality of the default SCIP node selector and the current state-of-the-art?**

3. **How does the machine learning model testing accuracy affect the results of the node selector?**

4. **Is it possible to leverage the obtained ML-based node selector and pruner, which are trained to solve smaller problems, then used to solve bigger problems, and outperform the default SCIP node selector and the current state-of-the-art?**

1.2. Contributions

This thesis contributes a novel approach to improve 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. [16], our policy learns only 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 method. 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 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.

We study two settings: the first is approximate 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. The second setting is exact: when reaching a leaf the solver backtracks up the branch-and-bound tree to use a different strategy.

We apply the learned policy within the popular open-source solver SCIP [15]. The results indicate that our node selector finds (optimal) solutions more quickly than He et al. [16], but not as quickly as the current default SCIP node selector, called *BestEstimate*. However, the results also indicate that our approximation method finds better initial solutions than *BestEstimate*, albeit in a higher solving time. Overall, our approximation-based 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 approximate method finds better solutions than all the baselines in three of the five problem classes tested and for one problem class not statistically significantly worse than the baseline.

1.3. Organisation

The outline of this thesis is as follows: Chapter 2 contains the preliminaries needed to understand this thesis, Chapter 3 presents the literature survey, Chapter 4 specifies the imitation learning approach, Chapter 5 contains the experimental setup, Chapter 6 shows the results, Chapter 7 concludes with future directions and in Appendix A we include the preprint[1] for this thesis.

[1] The preprint is also published at `http://arxiv.org/abs/2007.03948`

2

Preliminaries

2.1. Machine learning

Supervised learning, unsupervised learning, a combination of both and reinforcement learning are all different methods of machine learning. In supervised learning, the goal is to create a model that can predict a value for a certain input. Creating this model is called training. Training a model is done by using training data where the value to predict, also called label, is known in advance. Moreover, the goal is to minimise a loss function, typically expressed as:

$$L = \min_{\theta} \frac{1}{N} \sum_{i=1}^{N} l(f(x_i; \theta), y_i) + \lambda R(f) \tag{2.1}$$

where L is the loss value, θ represents model parameters, l is the loss function, f is the predictor, x is the instance, y is the label, λ is the regularisation parameter and R is the regularisation function.

Predicting this label is called classification and the algorithm that implements it is a classifier. The aim is to train the model enough so that it can predict reasonably well on yet unseen data. A popular performance measure is the root mean square error (RMSE):

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{N} (y_i - \hat{y}_i)^2}{N}} \tag{2.2}$$

where N is the sample size, y_i is the actual label and $\hat{y}_i$ is the predicted label. Neural networks are popular classifiers that are able to successfully model complex non-linear functions. An example on the application of a neural network is facial recognition.

Reinforcement learning (RL) is based on a Markov Decision Process (MDP). An MDP is a framework for decision making and has an environment, which consists of: states, possible actions, a reward function and a state transition function. A so-called agent starts in a certain state, can perform actions that result in the agent being in a different state and the goal for the agent is to be in the terminal state, such that accumulated reward is maximised. The reward function maps states and possibly actions to a number, in order to either penalise or reward the agent for the decision he made. The state transition function maps states, actions and subsequent states to a probability. If this probability for a state and action to a next state is one, then the decision process is deterministic and if not, it is stochastic.

RL is used to learn an optimal policy, which is a function that maps states to actions, such that the accumulated reward is maximised. Different methods exist to solve reinforcement learning problems, such as temporal difference learning [26] and Q-learning [26]. RL can be model-based or model-free, the former learning the model (transition and reward function) first to find an optimal policy and the latter not learning the model, but directly computing a policy.

An example of applied RL is AlphaGo [25], which used RL to learn an optimal policy for playing Go and has beat world champions.

2.2. Imitation learning

Reinforcement learning is designed to learn an approximately optimal policy: a function that maps states to actions, such that the accumulated reward is maximised [26]. A problem that can arise in trying to find a policy using reinforcement learning is that the reward function is unknown. In some cases an expert or oracle can provide demonstrations. The demonstrations show what action the expert has taken in a specific state. In this case, a policy can be learned by using inverse reinforcement learning. This is called imitation learning [1] or apprenticeship learning. Here, supervised learning is used with the features being the states of the demonstrations and the labels being the action the expert took.

Imitation learning is used in our method extensively to power the training of our classifier, which results in a policy used in the solving process of mixed integer programming problems.

2.3. Mixed integer programming

Mixed integer programming (MIP) is a familiar approach to constraint optimisation problems. A mixed integer program 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:

$$\begin{aligned} &\text{minimise } y := c^T x \\ &\text{subject to } Ax \geq b \\ &\qquad x \in \mathbb{Z}^k \times \mathbb{R}^{n-k},\ k > 0 \end{aligned} \tag{2.3}$$

where y is the objective value and x is the vector of decision variables to decide on. In a MIP, at least one variable has integer domain; if all variables have continuous domains then the problem is a linear program (LP). A is an m x n constraint matrix with m constraints and n variables; c is a n x 1 vector.

Since general MIP problems cannot be solved in polynomial time, a helpful idea is to relax the integer constraints to allow all variables to take real values: an LP relaxation of the problem (2.3). A series of LP relaxations 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 2.3 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:

$$I_G = \begin{cases} \frac{|B_P - B_D|}{\min(|B_P|, |B_D|)|}, & \text{if } \text{sign}(B_P) = \text{sign}(B_D) \\ \infty, & \text{otherwise} \end{cases} \tag{2.4}$$

where B_P is the primal bound, B_D is the dual bound, and $\text{sign}(\cdot)$ returns the sign of its argument. The integrality gap is 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.

2.4. Branch and bound

Branch and bound [20] is the most common constructive search approach to solving MIP problems. In this method, 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. Choosing to branch on variable i and choosing the left child is also referred to as branching on variable i on downwards direction. Choosing the right child is also referred to as going in upwards direction. The main steps of a standard MIP branch-and-bound algorithm are shown in Algorithm 1.

See Figure 2.1 for a branch and bound example solving a real problem.

Algorithm 1: Algorithm to 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
6      N ← PQ.poll()
7      if N.dualBound ≥ P_B 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*
```

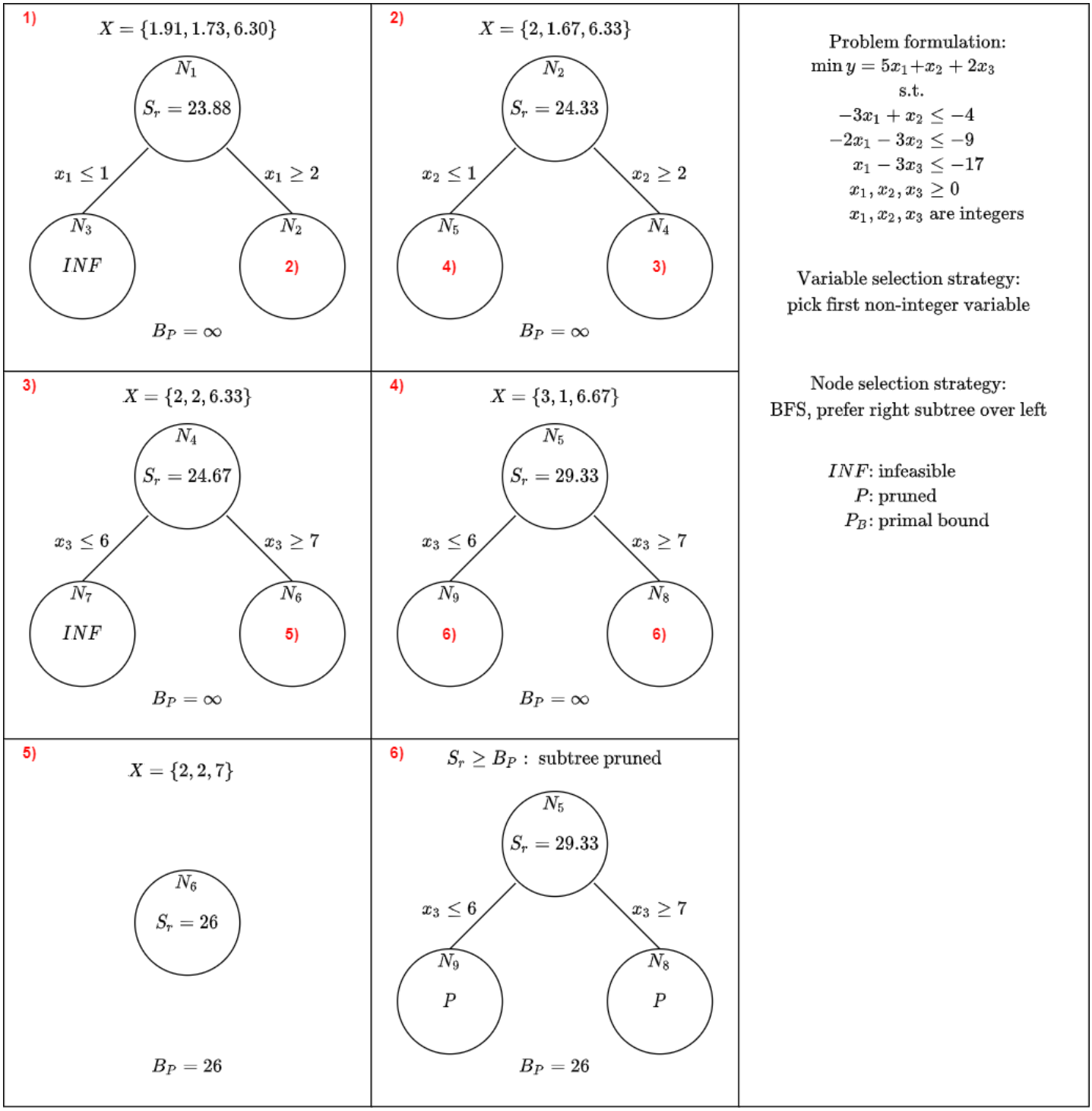


Figure 2.1: Branch and bound example.

Defining a comparator for the node priority queue PQ is handled by the node selector (used in PQ.poll() and PQ.add() in Algorithm 1). We explain further below.

Choosing on which variable to branch (variableSelection() in Algorithm 1) is not trivial and affects the time to find solutions and prove optimality. Different approaches to this variable selection are discussed in Chapter 3. For example, in the SCIP solver[1], the default variable selection heuristics (the 'brancher') is: reliability branching on pseudo-cost values. 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. As described in Chapter 4, the *prioChild* property is used as a feature in our work.

In more detail, the left child priority value is calculated by SCIP as:

$$P_L = I_L(V_r - V + 1) \tag{2.5}$$

and the right child priority value as:

$$P_R = I_R(V - V_r + 1) \tag{2.6}$$

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 [2]. 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.

2.5. Node selection and pruning

In MIP solvers such as SCIP, a node (and its entire sub-tree) is pruned when the solution of the relaxation is worse than the current primal bound (line 7 and 15 in Algorithm 1). However, we can further leverage node pruning to create an approximation 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. However, we must be aware that there is no guarantee that the optimal solution is not pruned.

Deciding which node is prioritised over another node to explore 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*, see Algorithm 2), breadth-first search (*BFS*), *RestartDFS* (restarting *DFS* at the root after a fixed amount of newly-explored nodes, see Algorithm 3) and *BestEstimate*. The latter is the default node selector in the SCIP solver from version 6. The algorithm is quite involved and for that reason we did not include the algorithm here, but we do include a brief description[2]:

- Plunging is defined as successively selecting a child node.
- The plunge depth is the successive times a child is selected as the next node.
- *BestEstimate* can do two different operations:
  - Plunge into the tree as long as the current plunge depth is within the minimum and maximum plunge depth. These bounds are calculated dynamically. While plunging, prefer children over siblings and siblings over leaves. Also, prefer priority children over best estimate children and priority siblings over best estimate siblings.
  - If the plunge depth is not within this range, then select the node with best estimate.

The estimate of a node p is defined as:

$$E(p) = L_p + \sum_{i \in V_f} \min(c_i^-(p), c_i^+(p)) \tag{2.7}$$

[1] `scip.zib.de`

[2] *BestEstimate* is detailed at `www.scipopt.org/doc/html/nodesel__estimate_8c_source.php`.

where L_p is the lower (dual) bound of node p, V_f are the fractional variables obtained after solving the LP of node p, $c_i^-(p)$ (respectively $c_i^+(p)$) is the pseudocost of branching on variable i in downwards direction (upwards direction). Note that a history must be kept to calculate pseudocosts for variables. The pseudocosts are defined as [4]:

$$c_i^-(p) = f_i^-(p)\frac{\sum_{q \in Q_i^-} \Delta(q)/f_i^-(q)}{|Q_i|^-} \tag{2.8}$$

and:

$$c_i^+(p) = f_i^+(p)\frac{\sum_{q \in Q_i^+} \Delta(q)/f_i^+(q)}{|Q_i|^+} \tag{2.9}$$

where $f_i^-(p)$ is the fractionality of variable i at node p with relation to the lower bound, Q_i^- is the history of nodes where variable i was the branched variable in downwards direction, $\Delta(p)$ is the gain in objective value on node p. Note that if $|Q_i^-| = 0$, then there is no history of branching on variable i in downwards direction, and so $c_i(p)$ is set to $f_i^-(p)$. The value for $c_i^+(p)$ is similarly defined.

The fractionalities are defined as:

$$f_i^-(p) = x_i(p) - \lfloor x_i(p) \rfloor \tag{2.10}$$

and:

$$f_i^+(p) = \lceil x_i(p) \rceil - x_i(p) \tag{2.11}$$

where $x_i(p)$ is the value of variable i at node p after solving the LP.

Summarising the node selection heuristics, they can be grouped into two general strategies [5]. The first strategy 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. The second has the advantage to prune more nodes and decrease the search space. In this thesis 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 to be selected.

2.6. Problem sets

In this section, we present the NP-hard problem definitions with their MIP formulations of set cover, maximum independent set, capacitated facility location and combinatorial auctions. We evaluate our node selector by solving instances of these problems, described in Chapter 5.

2.6.1. Set cover

A set cover instance consists of a universe of elements U, a collection of sets $\mathcal{S}$, such that $\bigcup_{S \in \mathcal{S}} S = U$ and a cost $C(S)$ for each $S \in \mathcal{S}$. The objective is to find a subset $\mathcal{S}' \subseteq \mathcal{S}$, such that $\bigcup_{S' \in \mathcal{S}'} S' = U$ and $\sum_{S' \in \mathcal{S}'} C(S')$ is minimised. The MIP formulation is the following [27]:

$$\begin{aligned} \text{minimise} \quad & \sum_{S \in \mathcal{S}} C(S) x_S \\ \text{subject to} \quad & \sum_{S : e \in S} x_S \geq 1 \text{ for all } e \in U \\ & x_S \in \{0, 1\} \quad \text{for all } S \in \mathcal{S} \end{aligned} \tag{2.12}$$

where x_S is a decision variable. If $x_S = 1$, then $S \in \mathcal{S}'$.

Algorithm 2: *DFS* node selection algorithm.

```
   output: Next node S to select
 1 Function selectNode():
 2 |  S ← getPrioChild()
 3 |  if S does not exist then
 4 |  |  S ← getPrioSibling()
 5 |  |  if S does not exist then
   |  |  |  // Filters leaves and uses nodeComp() to compare leaves with
   |  |  |     each other, selects the best
 6 |  |  |  S ← getBestLeaf()
 7 |  |  end
 8 |  end
 9 |  return S
10 end
   input  : N_1, N_2 to compare
   output: -1 if N_1 comes before N_2
           1 if N_1 comes after N_2
           0 if equal
11 Function nodeComp(N_1, N_2):
12 |  d_1 ← N_1.getDepth()
13 |  d_2 ← N_2.getDepth()
14 |  if d_1 > d_2 then
15 |  |  return -1
16 |  else if d_1 < d_2 then
17 |  |  return 1
18 |  else
   |  |  // N.getLowerBound() returns the lower (dual) bound of node N
19 |  |  l_1 ← N_1.getLowerBound()
20 |  |  l_2 ← N_2.getLowerBound()
21 |  |  if l_1 < l_2 then
22 |  |  |  return -1
23 |  |  else if l_1 > l_2 then
24 |  |  |  return 1
25 |  |  else
26 |  |  |  return 0
27 |  |  end
28 |  end
29 end
```

Algorithm 3: *RestartDFS* node selection algorithm.

```
   // Initialise global variables
1  nProcessedLeaves ← 0
2  selectBestFreq ← 100 // Constant in SCIP, every 100 leaves it selects
       the node with the smallest lower bound
3
   output: Next node S to select
4  Function selectNode():
5      S ← getPrioChild()
6      if S does not exist then
7          nProcessedLeaves++
8          nNodes ← getNumberOfNodes()
9          if nNodes > selectBestFreq then
10             selectBestFreq ← 0
               // Gets the node with smallest lower bound
11             S ← getBestBoundNode()
12         else
13             S ← getPrioSibling()
14             if S does not exist then
                   // Filters leaves and uses nodeComp() to compare leaves
                      with each other, selects the best
15                 S ← getBestLeaf()
16             end
17         end
18     end
19     return S
20 end
   input  : N_1, N_2 to compare
   output: −1 if N_1 comes before N_2
           1 if N_1 comes after N_2
           0 if equal
21 Function nodeComp(N_1, N_2):
       // Numbers are successively assigned
22     n_1 ← N_1.getNumber()
23     n_2 ← N_2.getNumber()
24     if n_1 > n_2 then
           // N_1 was created after N_2
25         return −1
26     else if n_1 < n_2 then
           // N_1 was created before N_2
27         return 1
28     else
           // Can never happen
29         return 0
30     end
31 end
```

2.6.2. Maximum independent set

Given an undirected graph $\mathcal{G} = (V, E)$, a subset $S \subseteq V$ is independent if there is no edge $e = (v_i, v_j)$ in E for every pair $v_i, v_j \in S$. This subset S is a maximum independent set if $|S|$ is maximised. Let $\mathcal{C}$ be a collection of cliques of graph $\mathcal{G}$, such that $C \in \mathcal{C}$ is not necessarily a maximal clique and every vertex $v \in V$ is in at least one $C \in \mathcal{C}$. It is easy to see that for every clique $C \in \mathcal{C}$, at most one vertex $v \in C$ can be in the independent set S [28]. The MIP formulation is the following:

$$\begin{aligned} &\text{maximise } \sum_{v \in V} x_v \\ &\text{subject to } \sum_{v \in C} x_v \leq 1 \text{ for all } C \in \mathcal{C} \\ &\qquad x_v \in \{0, 1\} \text{ for all } v \in V \end{aligned} \tag{2.13}$$

where x_v is a decision variable. If $x_v = 1$, then $v \in S$.

2.6.3. Capacitated facility location

Given a set of facilities F, a set of customers C, shipping costs $\{s_{f,c} : f \in F, c \in C\}$, demands $\{d_c : c \in C\}$, facility opening costs $\{o_f : f \in F\}$ and facility capacities $\{p_f : f \in F\}$. The objective is to meet the demands of every customer, while minimising the total cost. A facility f can only supply customers once it is opened and by opening it incurs an opening cost o_f. A facility f can supply at most p_f items and it can partially supply one customer. Once a facility f (partially) supplies a customer c, it incurs a (partial) shipping cost $s_{f,c}$. The MIP formulation is the following [11]:

$$\begin{aligned} &\text{minimise } \sum_{f \in F} \sum_{c \in C} s_{f,c} d_c y_{f,c} + \sum_{f \in F} o_f x_f \\ &\text{subject to } \sum_{f \in F} y_{f,c} = 1 && \text{for all } c \in C \\ &\qquad \sum_{c \in C} d_c y_{f,c} \leq p_f x_f && \text{for all } f \in F \\ &\qquad y_{f,c} \geq 0 && \text{for all } f \in F, c \in C \\ &\qquad x_f \in \{0, 1\} && \text{for all } f \in F \end{aligned} \tag{2.14}$$

where $y_{f,c}$ is the fraction of the demand facility f supplies to customer c and x_f is a decision variable, where $x_f = 1$ means that facility f is opened.

2.6.4. Combinatorial auctions

Given a set of items I, a collection of items as packages $\mathcal{J} = \{J : J \subseteq I\}$ and a bid for each package b_J. The auctioneer must determine which package to sell, such that total bid of each sold package is maximised and that sold packages $\mathcal{J}' \subseteq \mathcal{J}$ are mutually exclusive, that is: $\bigcap_{J' \in \mathcal{J}'} J' = \emptyset$.

The MIP formulation is the following [23]:

$$\begin{aligned} &\text{maximise } \sum_{J \in \mathcal{J}} b_J x_J \\ &\text{subject to } \sum_{J \in \mathcal{J}} a_{J,i} x_J \leq 1 \text{ for all } i \in I \\ &\qquad x_J \in \{0, 1\} \quad \text{for all } J \in \mathcal{J} \end{aligned} \tag{2.15}$$

where $a_{J,i}$ is a binary constant which determines that item i is in package J and x_J is a decision variable. If $x_J = 1$, then the auctioneer sells package J.

3

Literature Survey

In this chapter, we present a literature survey on methods where imitation learning is used to improve the solving process of MIPs.

3.1. Branching

Deciding on what variable to branch on in the branch and bound process is called branching, as was mentioned in Section 2.4. 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, are by Lodi and Zarpellon [21].

Strong branching [8] is a popular branching strategy, among other strategies such as most infeasible branching, pseudo-cost branching, reliability branching [4] – used as the default in SCIP – and hybrid branching [3]. Strong branching creates the smallest trees, as Achterberg et al. [4] reported that strong branching required around 20 times less nodes to solve a problem than most infeasible branching and around 10 times less 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. [7] were 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 2.3. 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 [14]. 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. [18] extended Alvarez et al. [7] 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 [17]. Their algorithm solved 70% more hard problems (over 500,000 nodes, cut-off time 5 hours) 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] propose 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. [7] and Khalil et al. [18] 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. [7] and 70% faster than the method of Khalil et al. [18], both for hard problems (1,500 auctions).

Seeing their success, we adopt the same variable features as Gasse et al. [12].

3.2. Node selection and pruning

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. [16] 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. [16] 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. [16] by constraining the node selection space to direct children only at non-leaf nodes. Furthermore, 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. Lastly, we include branched variable features, obtained from Gasse et al. [12]. As seen in Chapter 6, our approach easily outperforms that of He et al. [16], in both their original implementation and a re-implementation in SCIP 6.

4

Approach

Recall that our goals are to obtain an exact and approximate MIP node selection policy using machine learning, and to use it in a MIP solver. The policies should lead to promising solutions more quickly in the branch-and-bound tree, while the approximate policy should prune as few good solutions as possible.

4.1. Imitation learning

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 4.1 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. We constrain the action space for a_t to only select one child node or both. 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.

4.2. Sampling

In order to train a policy by imitation learning, we require training data from the expert. Our sampling process of state-action pairs is similar to the prior work of He et al. [16], with two major differences. The first is that our policy learns only 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 method. The second difference is that we generalise the expert demonstration 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.

He et al. [16] check whether the current node is in the path to the best solution. In our work, we check whether the left and right children of the current node are in a path that leads to 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.

See Figure 4.1 for a sampling example of a real problem. We assume that the top k solutions are given. For every node, we check whether each of its children are optimal. A child is optimal if it is in the path to at least one of the best k solutions. An easy way to confirm this, is by checking whether the solution of the branched variable is possible to obtain by the bound set on the branched variable. For example, at the root node, we know that the two possible solutions for the branched variable $x_1 \in \{2, 3\}$. Since only the bound $x_1 \geq 2$ makes it possible to reach at least one of the optimal values for x_1, we set the action taken at that node to be R. This also becomes the label (or class) for that node that the ML model tries to predict. At the second (right) node, the optimal values for $x_2 \in \{2, 1\}$ and so both $x_2 \leq 1$ and $x_2 \geq 2$ bounds make it possible to reach the top 2 solutions and thus the label is B.

Table 4.1: Features that define a state. Variable features from Gasse et al. [13].

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.

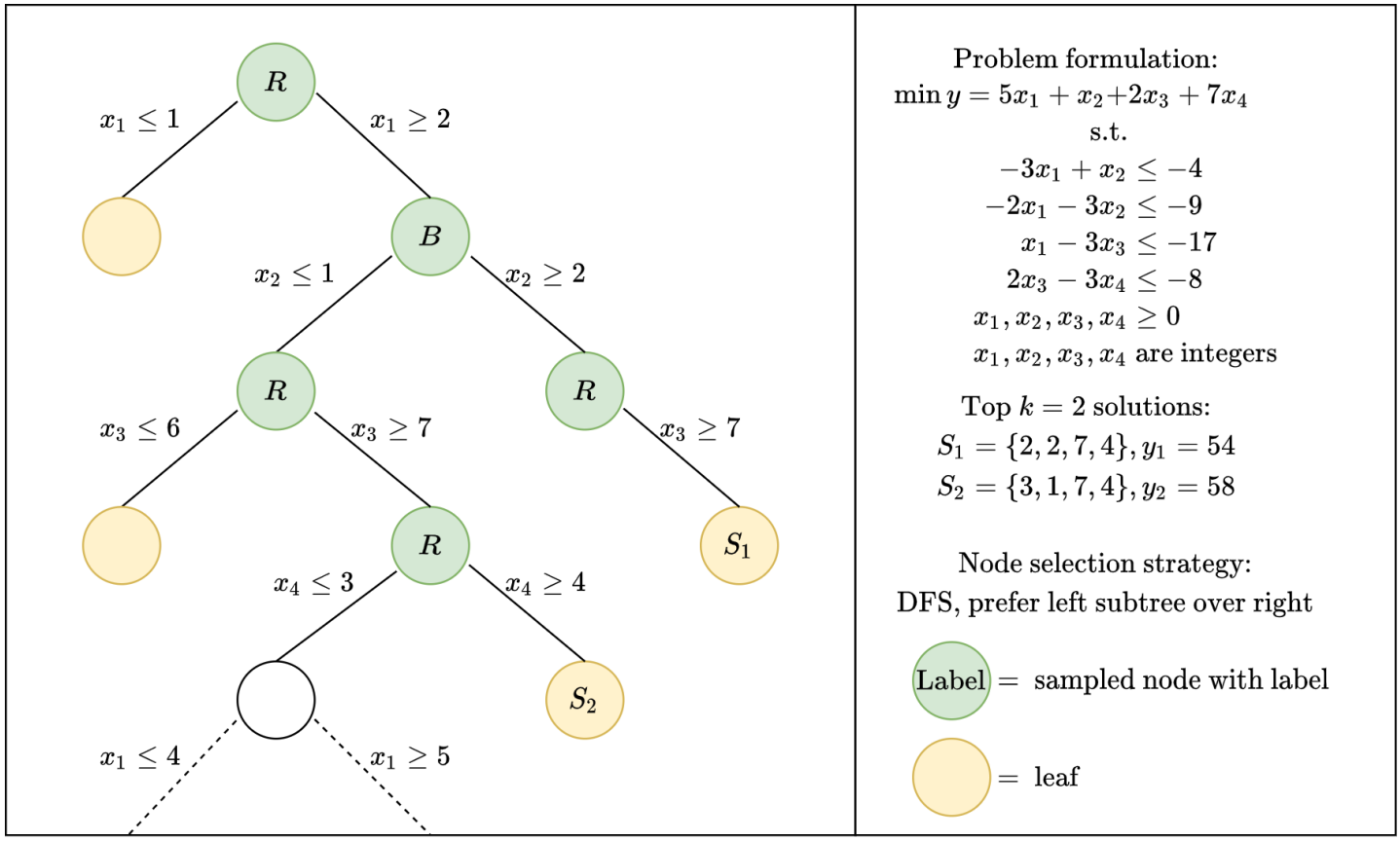


Figure 4.1: Node sampling example. The top k solutions are assumed to be given. The left sibling of node S_1 is left out.

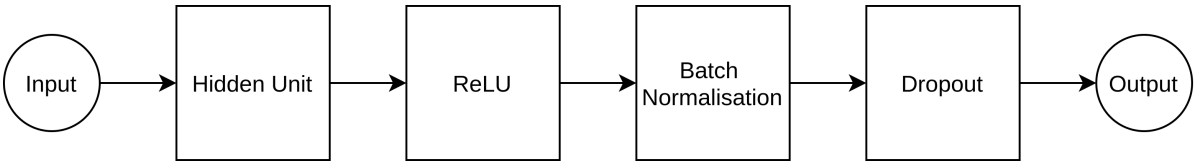


Figure 4.2: Operations within a hidden layer of the network.

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.

4.3. Neural network architecture

Our machine learning model is a standard fully-connected multi-layer perceptron (MLP) 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. See Figure 4.2 for an overview of the operations within a hidden layer. Our motivation for choosing a neural network is that deep learning tends to outperform other methods as the amount of data increases. The amount of data is not an issue, because we can generate possibly unlimited amount of data (see Chapter 5).

We obtain the model architecture parameters and learning rate ρ using the hyperparameter optimisation algorithm hyperopt[1] [10]. 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 [19].

[1] `github.com/hyperopt/hyperopt`

4.4. Policy configuration

We define three different settings that need to be set for every ML policy. These are *on_both*, *on_leaf* and *prune_on_both*, see Table 5.1 for an overview.

During policy evaluation, the action B ('both') can result in different operations. We define *PrioChild*, *Second* and *Random* as possible operations for the *on_both* setting. *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 for the *on_leaf* setting. These are *RestartDFS*, *BestEstimate* and *Score*. The first two are baseline node selectors from SCIP [15]; *Score* selects the node which obtained the highest score so far as calculated by our node selection policy.

Lastly, if we use the node pruning policy and the action B ('both') is selected, then we may not want to prune the ultimately decided opposing sub-tree. For this, we have the *prune_on_both* settings, which can be either set to True or False.

4.5. Node pruning policy

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, we use our learned policy in two ways: the first is approximate 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. See Algorithm 4 for the general ML policy node selection and pruning algorithm. The actual pruning only occurs if the user initialises the policy with it, regardless of the flag (*setPruneFlag()*). Note that the function *nodeComp()* is left out here, because the logic of *nodeComp()* depends on the *on_leaf* parameter. For *on_leaf* $\in$ {*RestartDFS*, *BestEstimate*}, we use the same rule as described in Chapter 2. For *on_leaf* = *Score*, we return the node with the highest policy score, as a history of node scores are kept (see *setScore()* in line 24 and 25 in Algorithm 4).

Algorithm 4: ML policy node selection algorithm.

```
input  : Current node N, String on_both, Function on_leaf, Boolean prune_on_both
output: Next node S to select
Function selectNode():
    C ← N.getChildren()
    if C is empty then
        return on_leaf(N)
    end
    f ← extractFeatures(N)
    scores ← applyMLPredictor(f)
    a ← argmax scores // Action with highest score
    pruneOtherChild ← false
    if a is B then
        if on_both is "PrioChild" then
            a ← getPrioChild()
        else if on_both is "Second" then
            scores' ← scores \ a // Remove a from scores
            a ← argmax scores'
        else
            a ← chooseRandomAction()
        end
        pruneOtherChild ← prune_on_both
    end
    a' ← L if a is R else R
    S ← getChild(C, a)
    S' ← getChild(C, a')
    S.setScore(scores[a])
    S'.setScore(scores[a'])
    setPruneFlag(S, false)
    setPruneFlag(S', pruneOtherChild)
    return S
end
```

5

Experimental setup

In this chapter, we explain how we designed our experiments to evaluate our method. We present the problem sets we use, the frameworks we use to solve them and train our machine learning model. Moreover, we define the baselines that we compare our method to and then detail the experiments.

5.1. Problem sets

The following standard NP-hard problem instances are tested: set cover, maximum independent set, capacitated facility location and combinatorial auctions. These problems are derived from the generator provided by Gasse et al. [12]. The instances are different from each other in terms of constraints structure, existence of continuous variables, existence of non-binary integer variables, and direction of optimisation.

5.2. Frameworks

For the MIP branch-and-bound framework we use SCIP version 6.0.2.[1] 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[2] [24], and interfaces with SCIP's C code via PySCIPOpt 2.1.6. 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.

5.3. Training

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. For every problem, we use the $k = 10$ best solutions to gather the state-action pairs, as this provides a good balance between high quality solutions and sampled state-action pairs. Additionally, 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.

5.4. Baselines

We compare the policy evaluation results with various node selectors in SCIP, namely *BestEstimate* (the SCIP default), *RestartDFS* and *DFS*. Additionally, we compare our results with the node selector and pruner from He et al. [16], 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. [16] have three policies: selection only (S), pruning only (P) and both (B). For exact solutions, we only use (S). For the first

[1] Note that SCIP version 7, released after we commenced this work, does not bring any major improvements to its MIP solving.
[2] `www.pytorch.org`

solution found at a leaf, we use (S) and (B). For the experiments with a time limit, we use (P) and (B).

5.5. Policy configurations

We evaluate a number of different settings for our node selection and pruning policy, as seen in Table 5.1. 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 5.1: Parameter settings for our node selection and pruning policy.

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

5.6. Experiments

In more detail, we report three different kinds of experiments:

1. We evaluate the policy on every problem by checking the average solving time of each node selector.
2. We check the solution quality in terms of the optimality gap and the solving time of the first solution found at a leaf node. 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 inbuilt in SCIP.
3. 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 show the initial optimality gap obtained by the solver before branch-and-bound is applied, i.e., from the solver's pre-solve prior to search.

For each experiment, we apply the policies on two different difficulty levels:

1. Easy instances, which can be solved within 15 minutes.
2. Hard instances, where we set a solving time limit to one hour. Here, for all three experiments we substitute the optimality gap for the integrality gap, because the optimal solution is not known for every hard instance. Additionally, for the first experiment, instead of checking the solving time, we check the integrality gap.

The experiments 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. We use the shifted geometric mean (SGM, shift = 1) as the average across all metrics:

$$\text{SGM} = \exp\left(\frac{\sum_{i=1}^{n} \ln \max(1, v_i + 1)}{n}\right) - 1 \tag{5.1}$$

where v_i is non-negative for all i. Using SGM is standard practice for MIP benchmarks[3].

[3]See, e.g., `plato.asu.edu/bench.html`

6 Results

Table 6.1 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 Section 6.2 for an explanation. The best performing ML model is the model with the settings that achieve the lowest validation loss.

Table 6.1: Machine learning parameters and prediction results for every problem. The baseline accuracy is predicting everything as the majority class.

Problem	Base acc	Test acc	H	U	p_d	ρ
Set cover	0.575	0.764	1	49	0.445	0.253
Max ind set (10)	0.923	0.922	1	25	0.266	0.003
Max ind 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

6.1. Set cover

These instances consist of 2,000 variables and 1,000 constraints forming a pure binary minimisation problem. We sampled 17,254 state-action pairs on the training instances, 2,991 on the validation instances and 3,218 on the test instances. The model achieves a testing accuracy of 76.4%, with a baseline accuracy of 57.5%.

See Table 6.2 for the average solving time and explored nodes of various node selection strategies. *BestEstimate* achieves the lowest average solving time at 26.4 seconds; ML_RB comes next at 35.7 seconds. 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 can not reject the null hypothesis of equal means.

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 0.05 for any of our ML policies (lowest observed p-value: 0.34). Recall that the values in the tables are the shifted geometric means, while the t-tests by their nature compares (arithmetic) means. The difference between the solving time of *BestEstimate* and any ML policy here is not statistically significant due to outliers.

See Figure 6.1 for boxplots of the baseline node selectors and our ML node selectors. The boxplots show that BestEstimate has smaller outliers, resulting in a lower average solving time.

Figure 6.2 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. Note that the only parameter that matters

Table 6.2: Set cover instances: average solving time and explored nodes for various node selection strategies. Pair-wise t-tests: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$ compared to *BestEstimate*.

Strategy	Solving time (s)	Explored nodes
BestEstimate	**26.39**	**4291**
DFS	47.10 *	9018 *
He	499.00 ***	47116 ***
He6	85.73 **	18386 **
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 *

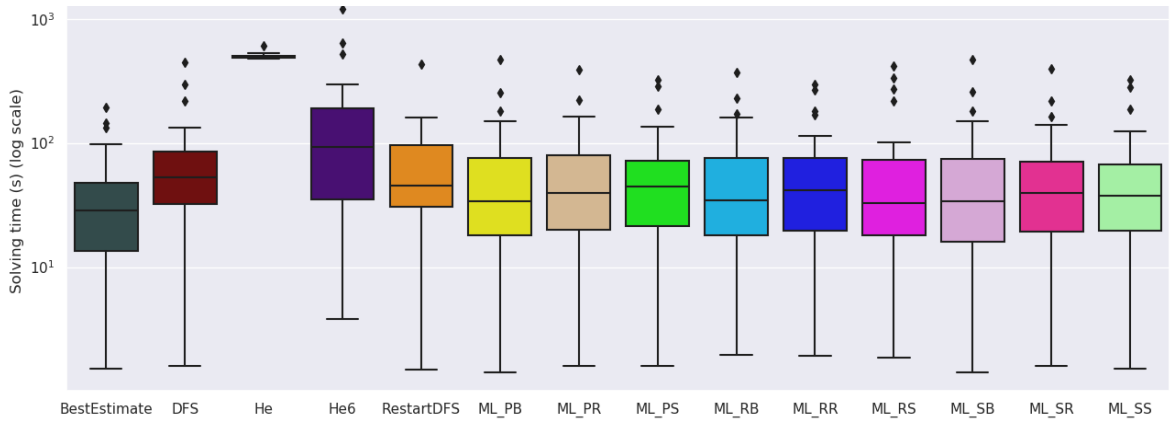


Figure 6.1: Set cover instances: optimality gap for each node selection strategy. Note log scale on y-axis.

for the ML solver is the first parameter, namely *on_both*. The second parameter *on_leaf* and the third parameter *prune_on_both* do not influence the solving time or quality of the first solution as the search terminates at the first found leaf that is found. The policy of He et al. [16] are not included here, due to its 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.

See Table 6.3 and Figure 6.3 for the average optimality gap of the baselines using 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 average solving time and average optimality gap across all instances. In this case, ML_SRF has the lowest harmonic mean and also achieves the lowest average optimality gap. We conducted a pair-wise 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.

The average optimality gap of the first found feasible solution is 2.746. This shows that applying branch-and-bound to find a solution has a significant difference.

6.2. Maximum independent set

These instances consist of 1,000 variables and around 4,000 constraints forming a pure binary maximisation problem. For this particular problem, we noticed that for $k = 10$, the class imbalance was significant. To combat this, we increased the value k to 40. For $k = 10$, we sampled 29,801 state-action

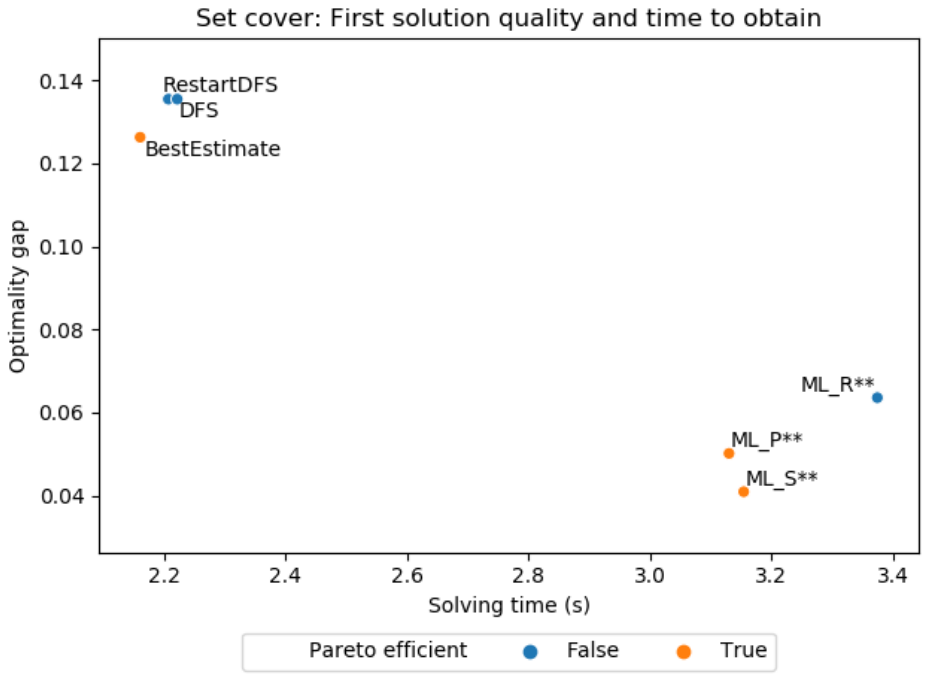


Figure 6.2: Average solving time against the average optimality gap of the first solution found at a leaf node on set cover instances

Table 6.3: Set cover: model with *on_both* = Second, *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 2.746. Pair-wise t-tests: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$.

Strategy	Optimality gap
BestEstimate	0.1767 **
DFS	0.0718 ***
He6 (prune only)	0.7988 ***
He6 (both)	1.1040 ***
ML_SRF	**0.0278**
RestartDFS	0.0741 ***

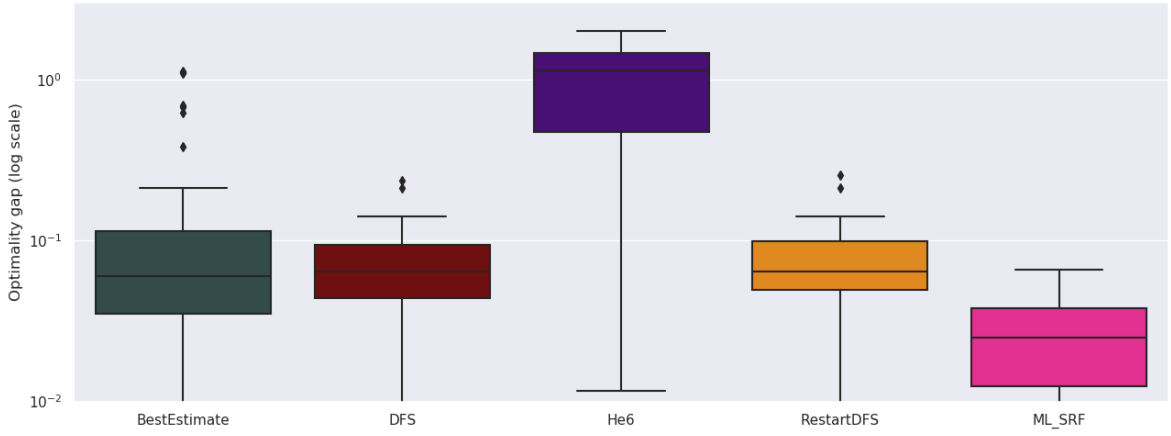


Figure 6.3: Set cover instances: optimality gap for each node selection strategy with a short time limit. Note log scale on y-axis.

pairs on the training instances, 5,820 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 6.4 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 6.4: Maximum independent set instances: average solving time and explored nodes for various node selection strategies. Pair-wise t-tests: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$ compared to *BestEstimate*.

Strategy	Solving time (s)	Explored nodes
BestEstimate	158.42	6344
DFS	**155.27**	**6340**
He	394.56 **	30965 **
He6	204.68	7992
ML_PB (10)	260.34 **	10029 *
ML_PB (40)	227.45 *	8100
ML_PR (10)	255.49 **	10191 *
ML_PR (40)	222.59 *	8581
ML_PS (10)	245.92 *	10184 ·
ML_PS (40)	207.28 ·	7878
ML_RB (10)	274.17 **	10296 *
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 **	10661 *
ML_SB (40)	283.02 **	10491 *
ML_SR (10)	252.59 **	9936 *
ML_SR (40)	258.68 **	10120 *
ML_SS (10)	242.37 *	9921 ·
ML_SS (40)	274.73 **	11251 *
RestartDFS	183.24	7854

See Figure 6.4 for boxplots of the baseline node selectors and the top 4 performing ML node selector for $k = 10$ and $k = 40$.

Figure 6.5 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 6.5 and Figure 6.6 examine 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 pair-wise 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 0.1 for all baselines, except *BestEstimate* (p-value: 0.33).

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

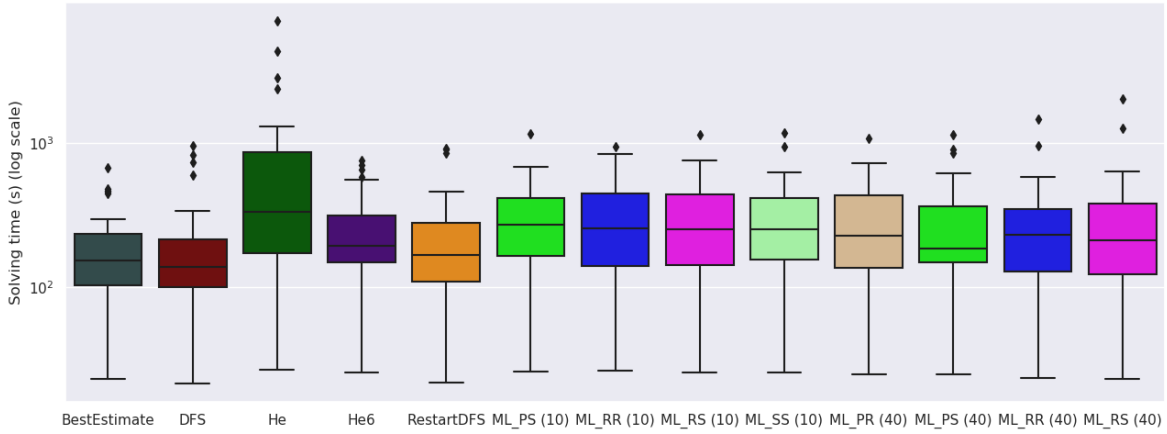


Figure 6.4: Maximum independent set instances: optimality gap for each node selection strategy. Note log scale on y-axis.

similar to initial optimality gap. The other policies find significantly better solutions.

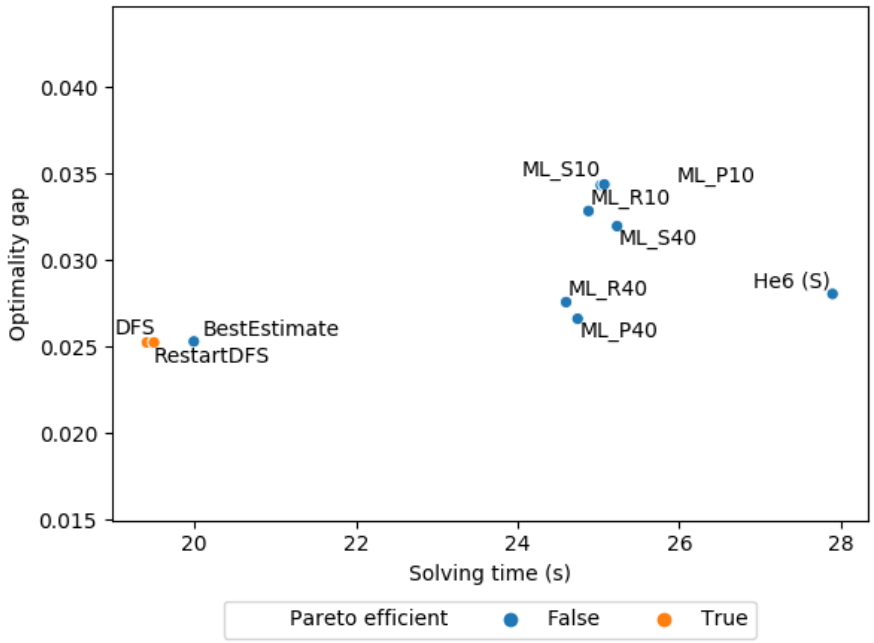


Figure 6.5: Average solving time against the average optimality gap of the first solution found at a leaf node on maximum independent set instances.

Table 6.5: 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: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$.

Strategy	Optimality gap
BestEstimate	0.0174
DFS	**0.0134** *
He6 (both)	0.9930 *
He6 (prune only)	0.9902 *
ML_PRF	0.0211
RestartDFS	0.0134 *

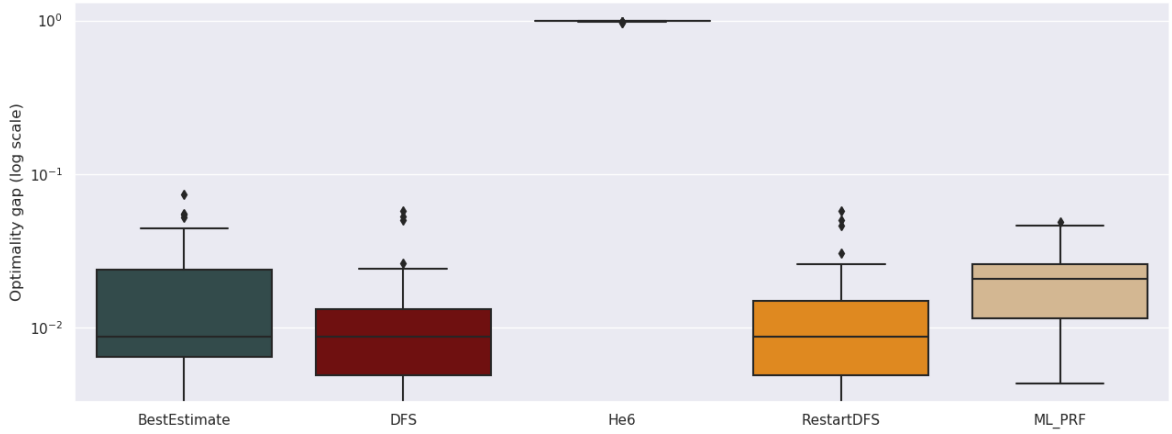


Figure 6.6: Maximum independent set instances: optimality gap for each node selection strategy with a short time limit. Note log scale on y-axis.

6.3. 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, 3,531 on the validation instances and 3,431 on the testing instances. The model achieves a testing accuracy of 90.1%, with a baseline accuracy of 73.1%.

See Table 6.6 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 6.6: Capacitated facility location instances: average solving time and explored nodes for various node selection strategies. Pair-wise t-tests: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$ compared to *BestEstimate*.

Strategy	Solving time (s)	Explored nodes
BestEstimate	122.79	1674
DFS	690.40 ***	17155 ***
He	1754.10 ***	5733 ***
He6	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 ***

See Figure 6.7 for boxplots of the baseline node selectors and our ML node selectors. The boxplots show that BestEstimate has smaller outliers, resulting in a lower average solving time, but not as pronounced as with set cover.

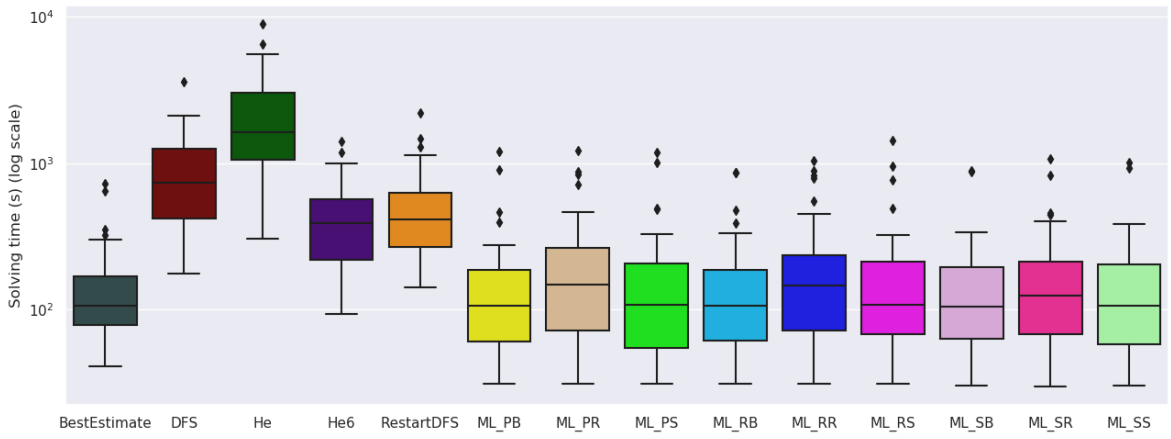


Figure 6.7: Capacitated facility location instances: optimality gap for each node selection strategy. Note log scale on y-axis.

Figure 6.8 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 6.7 and Figure 6.9 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 pair-wise 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.

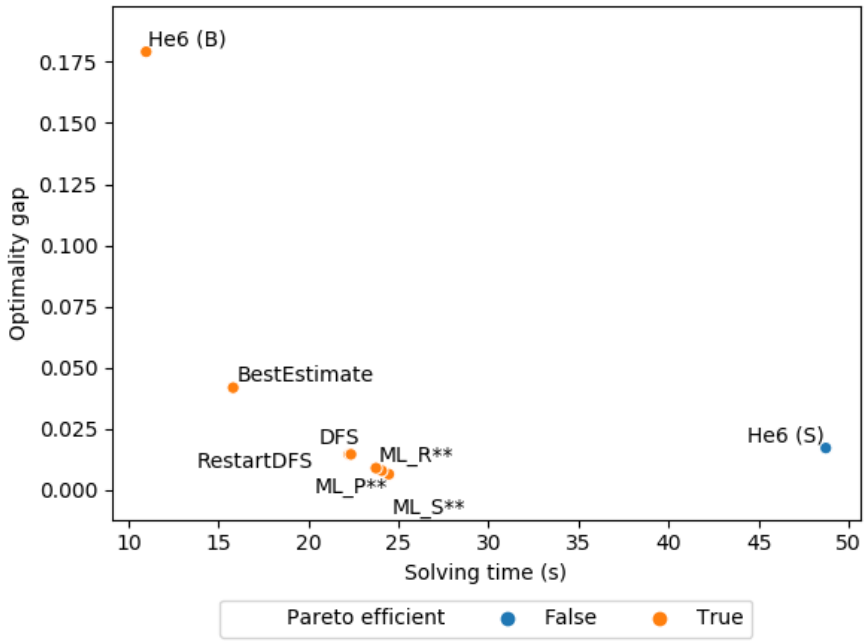


Figure 6.8: Average solving time against the average optimality gap of the first solution found at a leaf node on capacitated facility location instances.

Table 6.7: 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: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$.

Strategy	Optimality gap
BestEstimate	0.0821 *
DFS	0.0619 *
He6 (both)	0.1516 *
He6 (prune only)	0.1503 *
ML_SST	**0.0065**
RestartDFS	0.0590 *

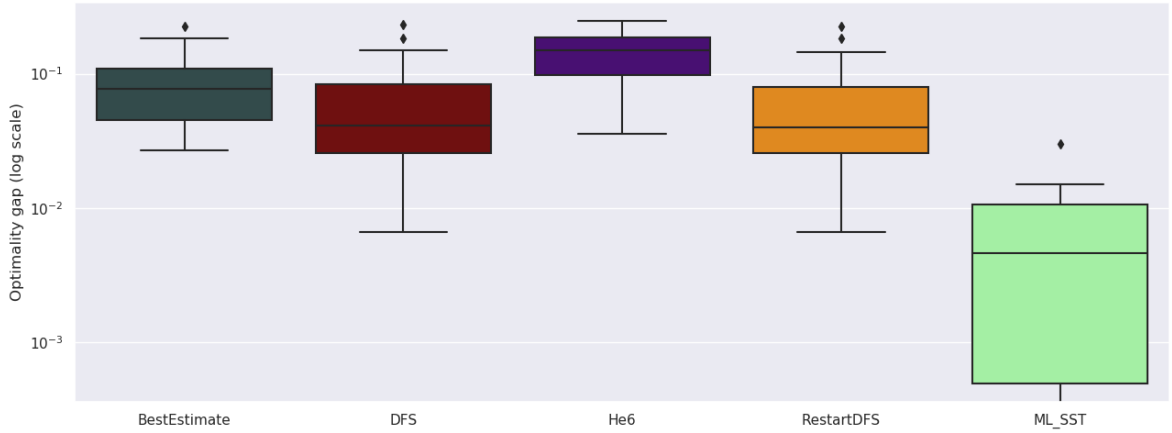


Figure 6.9: Capacitated facility location instances: optimality gap for each node selection strategy with a short time limit. Note log scale on y-axis.

6.4. Combinatorial auctions

These instances consist of 1,200 variables and around 475 constraints forming a pure binary maximisation problem. We sampled 13,554 state-action pairs on the training instances, 2,389 on the validation instances and 2,170 on the testing instances. The model achieves a testing accuracy of 71.7%, with a baseline accuracy of 57.0%.

See Table 6.8 for the average solving time and explored nodes of various node selection strategies. *BestEstimate* achieves the lowest average solving time at 19.7 seconds. 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.

See Figure 6.10 for boxplots of the baseline node selectors and our ML node selectors. The boxplots show that BestEstimate has significantly smaller outliers and a lower variance, resulting in a lower average solving time.

Figure 6.11 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 6.9 and Figure 6.12 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 pair-wise 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

Table 6.8: Combinatorial auctions instances: average solving time and explored nodes for various node selection strategies. Pair-wise t-tests: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$ compared to *BestEstimate*.

Strategy	Solving time (s)	Explored nodes
BestEstimate	**19.68**	**3489**
DFS	23.48	4490
He	40.11 ***	4519
He6	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

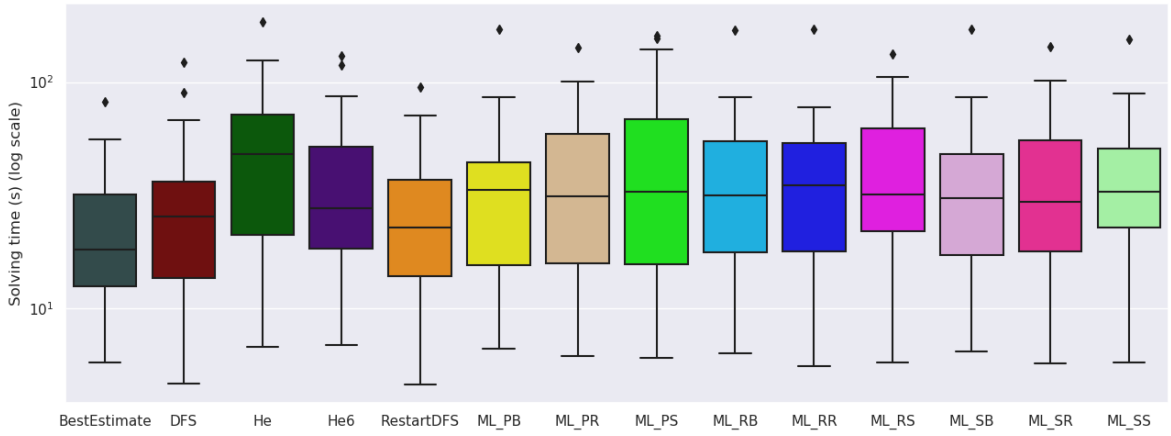


Figure 6.10: Combinatorial auctions instances: optimality gap for each node selection strategy. Note log scale on y-axis.

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.

6.5. 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 size of these hard instances are in terms of 4,000 variables and 2,000 constraints, while the easier set cover instances had 2,000 variables and 1,000 constraints. We evaluated 10 hard instances due to computational limitations, and focused on *BestEstimate* as a baseline on the node selection policy.

For the node selection policies, we set the time limit to one hour per problem. Figure 6.13 shows the number of solved instances per policy, and Figure 6.14 shows boxplots of the integrality gaps (see Equation 2.4) for each policy. We use integrality gap here, because we do not know the optimal objective value for all instances.

Table 6.10 shows the average solving time, explored nodes and integrality gap of various node selection strategies. *BestEstimate* achieves the lowest average solving time at 2256.7 seconds and integrality gap at 0.0481. ML_SB has the lowest number of explored nodes at 109298. We conducted

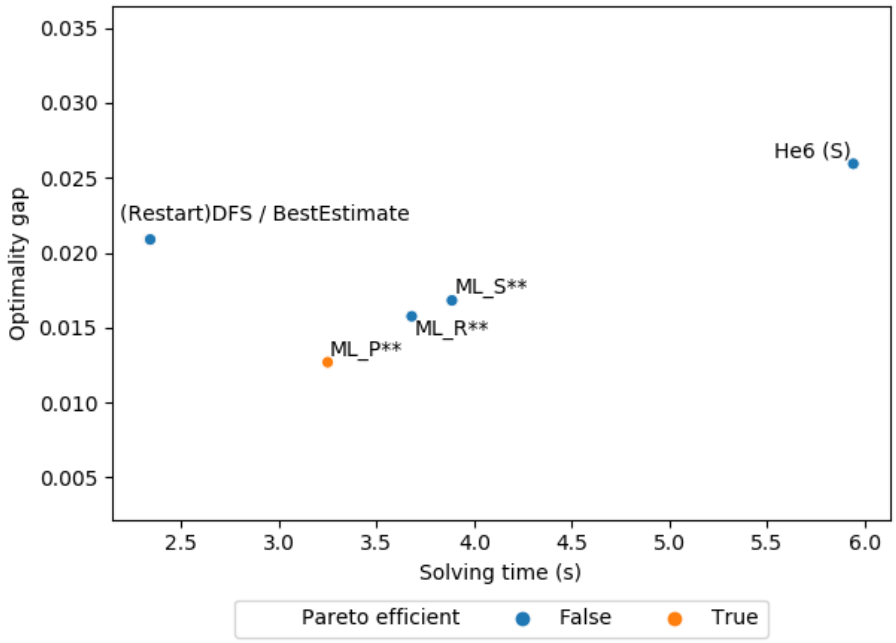


Figure 6.11: Average solving time against the average optimality gap of the first solution found at a leaf node on combinatorial auctions instances.

Table 6.9: 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: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$.

Strategy	Optimality gap
BestEstimate	0.0174 *
DFS	**0.0126**
He6 (both)	0.4678 *
He6 (prune only)	0.2873 *
ML_PST	0.0127
RestartDFS	0.0126

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 all our ML policies (lowest observed p-value: 0.64).

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 0.05 for ML_PB, ML_RB and ML_SB.

Lastly, we have 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.

For the pruning policies, Figure 6.15 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 6.11 and Figure 6.16 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 average solving time and average integrality gap of all ML policies. ML_PST achieves a significantly lower integrality gap compared to the baselines. We conducted a pair-wise 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.

In all instances, the solver could only obtain an integrality gap of infinity on the first feasible solution.

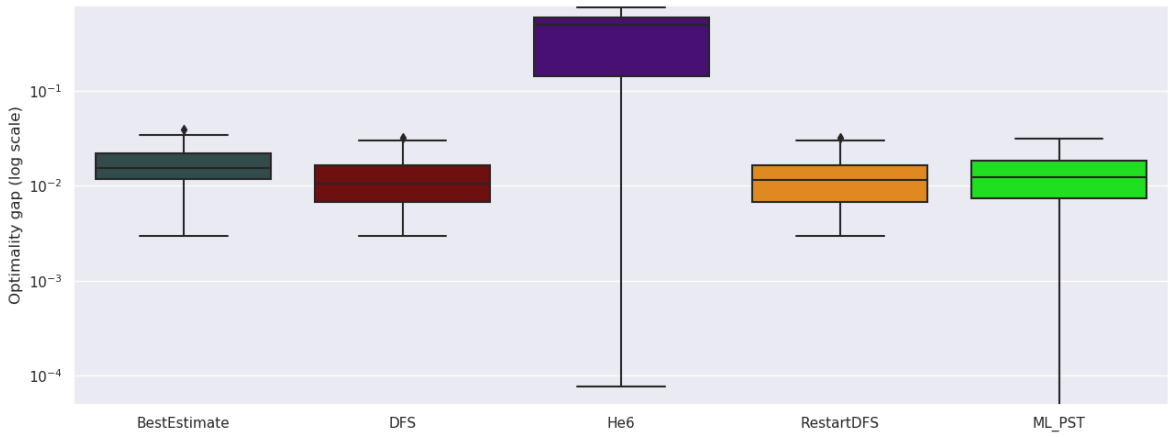


Figure 6.12: Combinatorial auctions instances: optimality gap for each node selection strategy with a short time limit. Note log scale on y-axis.

Table 6.10: Hard set cover instances: average solving time, explored nodes and integrality gap for various node selection strategies. Pair-wise t-tests: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$ compared to *BestEstimate*.

Strategy	Solving time (s)	Explored nodes	Integrality gap
BestEstimate	**2256.69**	161438	**0.0481**
ML_PB	2497.56	112869 *	0.0714
ML_PR	2531.32	150055	0.0763
ML_PS	2279.41	157367	0.1152
ML_RB	2441.60	111629 *	0.0683
ML_RR	2552.68	152871	0.0794
ML_RS	2352.93	153727	0.1289 ·
ML_SB	2427.25	**109298** *	0.0706
ML_SR	2685.09	161825	0.0734
ML_SS	2381.28	163631	0.1278 ·

This means we can not compare the initial integrality gap to the found integrality gaps during branch and bound.

6.6. Discussion

We examined three experiments, namely measuring the solving time for an exact solution, measuring the solving time and optimality gap for the first solution found at a leaf node in the branch and bound tree, and setting a low instance-specific time limit to measure what the optimality gap is.

For the first experiment, our method performed better than the baselines for capacitated facility location problem, but worse on the three purely binary problems. The best ML policy is ML_RB, although ML_SB has an almost equal solving time, while still exploring fewer nodes.

For the second 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.

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 they do against the baselines. These policies were chosen based on the (lowest) harmonic mean between the solving time and optimality gap as was conducted in the second experiment. 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 significantly worse than *DFS* and *RestartDFS*.

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* = *BestEs-*

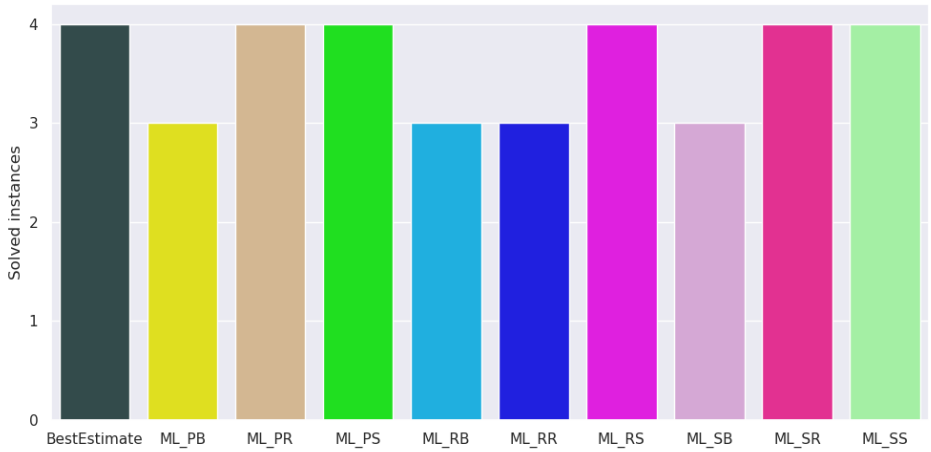


Figure 6.13: Hard set cover instances: Number of solved instances for each node selection strategy. Total number of instances is ten.

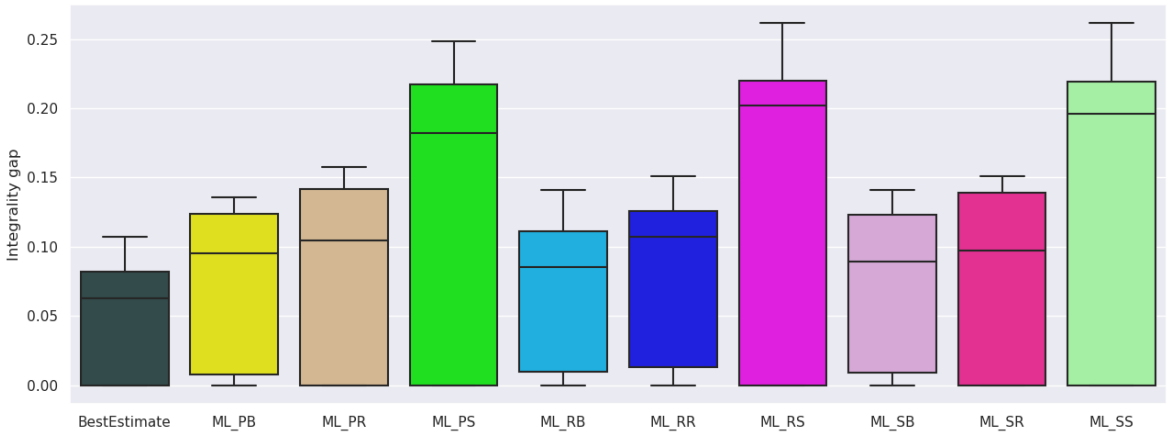


Figure 6.14: Hard set cover instances: integrality gap of each node selection strategy.

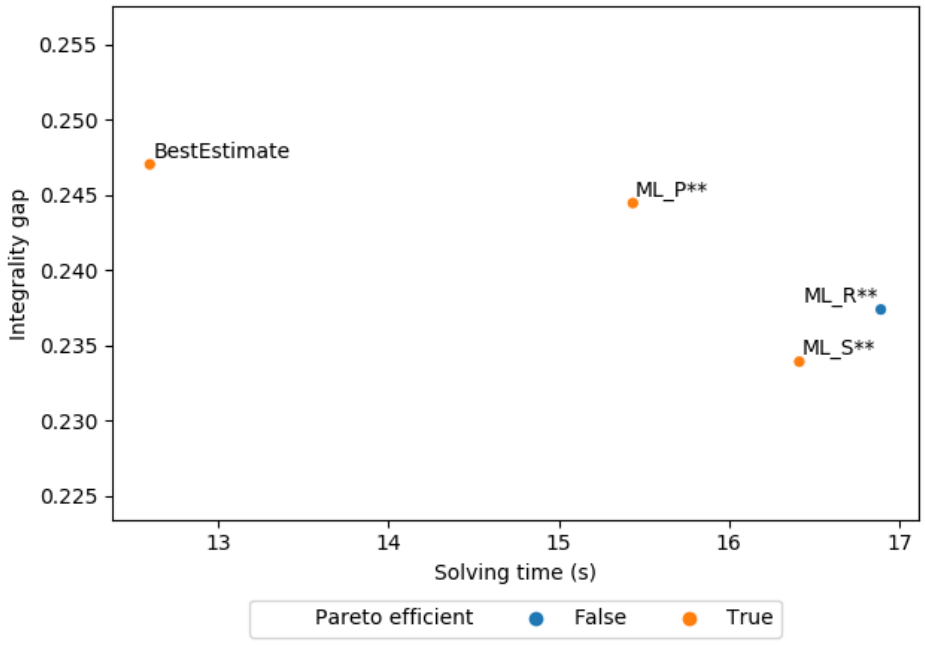


Figure 6.15: Average solving time against the average integrality gap of the first solution found at a leaf node on hard set cover instances.

Table 6.11: Set cover (hard): 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: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$.

Strategy	Integrality gap
BestEstimate	1.3678 *
DFS	0.3462 *
He6 (both)	1.8315 *
He6 (prune only)	1.6835 *
ML_PST	**0.2445**
RestartDFS	0.3489 *

timate 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 it is up to the user to choose 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; when it does not, inferior performance can be expected.

Lastly, 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 overhead of the Python interface.

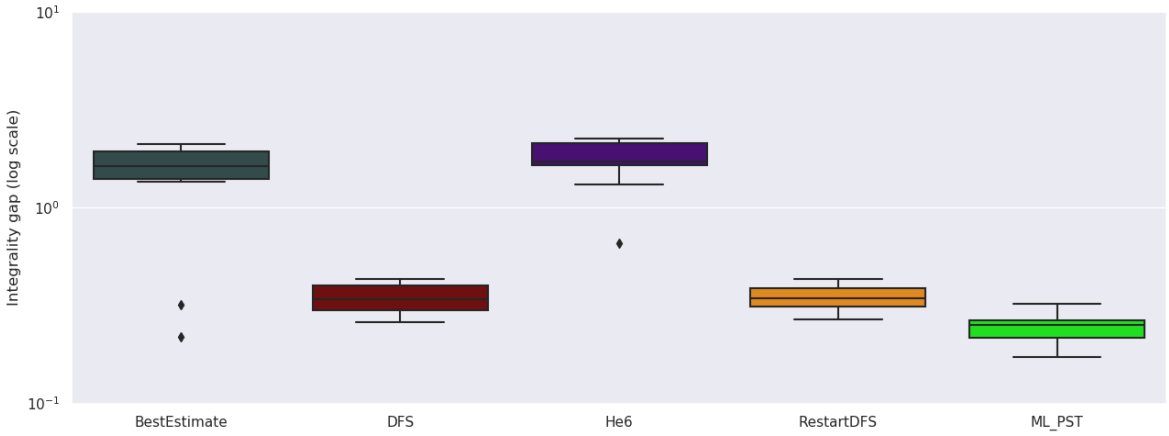


Figure 6.16: Hard set cover instances: Integrality gap of various approximation strategies. Note log scale on y-axis. ML_PST outperforms the compared methods.

7

Conclusion

This thesis shows that approximate solving of mixed integer programs can be improved by a node selection policy obtained with offline imitation learning. 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 approximate settings.

To answer our first research question, the empirical results on four MIP datasets indicate that our node selector leads to solutions more quickly than the state-of-the-art in the literature [16], but not as quickly as the state-of-practice SCIP node selector. While we do not beat the highly-optimised SCIP baseline in terms of solving time on exact solutions, our approximation-based policies have a consistently better optimality gap than all baselines if the accuracy of the predictive model adds value to prediction; this addresses our third research question. Further, to address our second and fourth research question, the results also indicate that our approximation method finds better solutions within a given time limit than all baselines in four of the five problem classes examined, including the harder set cover instances.

This thesis shows that learned policies can be Pareto-equivalent or superior to state-of-practice MIP node selection heuristics: heuristics that have been honed by hand over many years. It adds to the body of literature that demonstrates how machine learning 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 [5]. By contrast, in general for binary instances, the greater challenge is to close the dual bound, and our learned policy also performs well here.

7.1. Discussion

For future work, 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.

We mentioned that during pre-processing certain features are removed that are constant throughout the entire dataset. This has the consequence of a differing number of input units in the neural network architecture for every problem. The bigger issue is that this work focused on training a machine learning model for every problem. Future work could include a method to unify a machine learning model that works for all problems. This would make ML-based node selection a more accessible feature for current MIP solvers, like SCIP.

A limitation of this study is that we only performed experiments with the MIP solver SCIP. We chose SCIP, because it is open-source and has a rich amount of documentation. However, another MIP solver Gurobi (proprietary) is generally faster than SCIP and it would be interesting to see how the ML-based node selection policy compares to Gurobi.

While PySCIPOpt makes it easy to implement and test node selectors, there is a performance penalty associated to it. This interfacing overhead can play a significant role in the solving time. Implementing our method directly in the C code of SCIP can reduce the solving time we obtained across all experiments.

Our method on the maximum independent set problem did substantially worse than the other problems. We are not the first to experience trouble with this problem, since this problem was also the worst performer for the brancher of Gasse et al. [13]. It might be interesting to study why this problem is problematic for ML-based solutions.

Lastly, reinforcement learning, in contrast to imitation learning, is an interesting research direction to create a node selection policy.

Bibliography

[1] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proc. of 21st International Conference on Machine Learning (ICML'04)*, 2004. doi: 10.1145/1015330.1015430.

[2] Tobias Achterberg. *Constraint integer programming*. PhD thesis, Technische Universität Berlin, 2007.

[3] Tobias Achterberg and Timo Berthold. Hybrid branching. In *Proc. of 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09)*, volume 5547 of *Lecture Notes in Computer Science*, pages 309–311. Springer, 2009. doi: 10.1007/978-3-642-01929-6_23.

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

[5] Tobias Achterberg, Timo Berthold, Thorsten Koch, and Kati Wolter. Constraint integer programming: A new approach to integrate cp and mip. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 6–20. Springer, 2008.

[6] Claudine Biova Agbokou. *Robust airline schedule planning: review and development of optimization approaches*. PhD thesis, Massachusetts Institute of Technology, 2004.

[7] Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A supervised machine learning approach to variable branching in branch-and-bound. In *Proc. of 7th European Machine Learning and Data Mining Conference (ECML-PKDD'14)*, 2014.

[8] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. Finding cuts in the TSP (a preliminary report). Technical Report 5, Center for Discrete Mathematics & Theoretical Computer Science, 1995.

[9] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *CoRR*, abs/1811.06128, 2018. URL http://arxiv.org/abs/1811.06128.

[10] James Bergstra, Daniel Yamins, and David D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proc. of 30th International Conference on Machine Learning (ICML'13)*, pages 115–123, 2013.

[11] Michele Conforti, Gérard Cornuéjols, Giacomo Zambelli, et al. *Integer programming*, volume 271. Springer, 2014.

[12] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Proc. of 2019 Neural Information Processing Systems (NeurIPS'19)*, pages 15554–15566, 2019.

[13] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *arXiv preprint arXiv:1906.01629*, 2019.

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

[15] Ambros Gleixner, Michael Bastubbe, Leon Eifler, Tristan Gally, Gerald Gamrath, Robert Lion Gottwald, Gregor Hendel, Christopher Hojny, Thorsten Koch, Marco E. Lübbecke, Stephen J. Maher, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Franziska Schlösser, Christoph Schubert, Felipe Serrano, Yuji Shinano, Jan Merlin Viernickel, Matthias Walter, Fabian Wegscheider, Jonas T. Witt, and Jakob Witzig. The SCIP Optimization Suite 6.0. Technical report, Optimization Online, 2018. URL `http://www.optimization-online.org/DB_HTML/2018/07/6692.html`.

[16] He He, Hal Daumé III, and Jason Eisner. Learning to search in branch and bound algorithms. In *Proc. of 2014 Neural Information Processing Systems Conference (NeurIPS'14)*, pages 3293–3301, 2014.

[17] Thorsten Joachims. Training linear SVMs in linear time. In *Proc. of 12th International Conference on Knowledge Discovery and Data Mining (KDD'06)*, pages 217–226, 2006. doi: 10.1145/1150402.1150429.

[18] Elias Boutros Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proc. of 30th AAAI Conference on Artificial Intelligence (AAAI'16)*, pages 724–731, 2016.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proc. of 3rd International Conference on Learning Representations, (ICLR'15)*, 2015.

[20] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[21] A. Lodi and G. Zarpellon. On learning and branching: a survey. *TOP*, 25:207—236, 2017. doi: 10.1007/s11750-017-0451-6.

[22] Michele Lombardi, Michela Milano, and Andrea Bartolini. Empirical decision model learning. *Artificial Intelligence*, 244:343–367, 2017.

[23] Mark Michael. *Generalized Combinatorial Auction for Mixed Integer Linear Programming*. PhD thesis, University of Toronto, 2014.

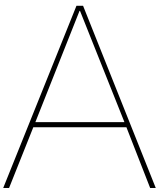
[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proc. of 2019 Neural Information Processing Systems (NeurIPS'19)*, pages 8024–8035, 2019.

[25] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

[26] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT Press, Cambridge, MA, 2018.

[27] Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

[28] Abraham Michiel Verweij. *Selected applications of integer programming: A computational study*. PhD thesis, Utrecht University, 2000.



Appendix

LEARNING EFFICIENT SEARCH APPROXIMATION IN MIXED INTEGER BRANCH AND BOUND

A PREPRINT

Kaan Yilmaz
Algorithmics group
Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
The Netherlands
M.K.Yilmaz@student.tudelft.nl

Neil Yorke-Smith*
Algorithmics group
Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
The Netherlands
n.yorke-smith@tudelft.nl

9 July 2020

ABSTRACT

In line with the growing trend of using machine learning to improve solving of combinatorial optimisation problems, one promising idea is to improve node selection within a mixed integer programming branch-and-bound tree by using a learned policy. In contrast to previous work using imitation learning, our policy is focused 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 is approximate by committing to pruning of nodes; one that is exact and backtracks from a leaf to use a different strategy. We apply the policy within the popular open-source solver SCIP. Empirical results on four MIP datasets indicate that our node selection policy leads to solutions more quickly than the state-of-the-art in the literature, but not as quickly as the state-of-practice SCIP node selector. While we do not beat the highly-optimised SCIP baseline in terms of solving time on exact solutions, our approximation-based policies have a consistently better optimality gap than all baselines if the accuracy of the predictive model adds value to prediction. Further, the results also indicate that, when a time limit is applied, our approximation method finds better solutions than all baselines in the majority of problems tested.

***K*eywords** mixed integer programming · node selection · machine learning · approximate pruning · imitation learning · SCIP

1 Introduction

Hard constrained optimisation problems (COPs) exist in many different applications. Examples include airline scheduling [Bayliss et al., 2017] and CPU efficiency maximisation [Lombardi et al., 2017]. Perhaps the most common paradigm for modelling and solving COPs 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 [Gleixner et al., 2018].

*Contact author

A growing trend is to use machine learning (ML) to improve COP solving. Bengio et al. [2018] 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 [He et al., 2014]. A policy is a function that maps states to actions, where in this work 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. [2014].

This paper contributes a novel approach to improve 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. [2014], our policy learns only 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 method. 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 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.

We study two settings: the first is approximate 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. The second setting is exact: when reaching a leaf the solver backtracks up the branch-and-bound tree to use a different strategy.

We apply the learned policy within the popular open-source solver SCIP [Gleixner et al., 2018]. The results indicate that our node selector finds (optimal) solutions more quickly than He et al. [2014], but not as quickly as the current default SCIP node selector, called *BestEstimate*. However, the results also indicate that our approximation method finds better initial solutions than *BestEstimate*, albeit in a higher solving time. Overall, our approximation-based 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 approximate method finds better solutions than all the baselines in three of the five problem classes tested and for one problem class not statistically significantly worse than the baseline.

The outline of this preprint paper 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, Section 5 reviews related work, and Section 6 concludes with future directions.

2 Background

2.1 Imitation learning

Reinforcement learning is designed to learn an approximately optimal policy: a function that maps states to actions, such that the accumulated reward is maximised [Sutton and Barto, 2018]. A problem that can arise in trying to find a policy using reinforcement learning is that the reward function is unknown. In some cases an expert or oracle can provide demonstrations. The demonstrations show what action the expert has taken in a specific state. In this case, a policy can be learned by using inverse reinforcement learning. This is called imitation learning [Abbeel and Ng, 2004] or apprenticeship learning. Here, supervised learning is used with the features being the states of the demonstrations and the labels being the action the expert took.

2.2 Mixed integer programming

Mixed integer programming (MIP) is a familiar approach to constraint optimisation problems. A mixed integer program 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:

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

where y is the objective value and x is the vector of decision variables to decide on. In a MIP, at least one variable has integer domain; if all variables have continuous domains then the problem is a linear program (LP). A is an m x n constraint matrix with m constraints and n variables; c is a n x 1 vector.

Since general MIP problems cannot be solved in polynomial time, a helpful idea is to relax the integer constraints to allow all variables to take real values: an LP relaxation of the problem (1). 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:

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

where B_P is the primal bound, B_D is the dual bound, and sign$(\cdot)$ returns the sign of its argument. The integrality gap is 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.

2.3 Branch and bound

Branch and bound [Land and Doig, 1960] is the most common constructive search approach to solving MIP problems. In this method, 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.

The main steps of a standard MIP branch-and-bound algorithm are shown in Algorithm 1.

Defining a comparator for the node priority queue PQ is handled by the node selector (used in PQ.poll() and PQ.add() in Algorithm 1). We explain further below.

Choosing on which variable to branch (variableSelection() in Algorithm 1) is not trivial and affects the time to find solutions and prove optimality. Different approaches to this variable selection are discussed in Section 5. For example, in the SCIP solver[2], the default variable selection heuristics (the 'brancher') is: reliability branching on pseudo-cost values. 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. As described in Section 3, the *prioChild* property is used as a feature in our work.

In more detail, the left child priority value is calculated by SCIP as:

$$P_L = I_L(V_r - V + 1) \tag{3}$$

and the right child priority value as:

$$P_R = I_R(V - V_r + 1) \tag{4}$$

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 [Achterberg, 2007]. 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.

2.4 Node selection and pruning

In MIP solvers such as SCIP, a node (and its entire sub-tree) is pruned when the solution of the relaxation is worse than the current primal bound (line 7 and 15 in Algorithm 1). However, we can further leverage node pruning to create an approximation 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. However, we must be aware that there is no guarantee that the optimal solution is not pruned.

Deciding which node is prioritised over another node to explore 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 root 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 going deep into the search tree.[3]

[2] `scip.zib.de`

[3] *BestEstimate* is detailed at `www.scipopt.org/doc/html/nodesel__estimate_8c_source.php`.

Algorithm 1: Algorithm to 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

$R.dualBound \leftarrow -\infty$ `// Initialise dual bound`
$PQ \leftarrow \{R\}$ `// Node priority queue`
$B_P \leftarrow \infty$ `// Primal bound`
$S^* \leftarrow$ null `// Optimal solution`
while *PQ is not empty* **do**
  $N \leftarrow PQ$.poll();
  if N.dualBound $\geq P_B$ **then**
    `// Parent of N had a relaxed solution worse than current best integer feasible solution, skip solving relaxation and prune`
    continue
  end
  $S_r \leftarrow$ solveRelaxation(N);
  if *S_r is not feasible* **then**
    `// Infeasible relaxation can not lead to a feasible solution for the original problem`
    continue;
  end
  $O_r \leftarrow S_r$.objectiveValue;
  if $O_r > B_P$ **then**
    `// This subtree cannot contain any solution better than the current best (pruning)`
    continue;
  end
  if *S_r is integer feasible* **then**
    $B_P \leftarrow O_r$;
    $S^* \leftarrow S_r$;
    `// Found incumbent solution no worse than current best`
    continue;
  end
  $V \leftarrow$ variableSelection(S_r);
  $a \leftarrow$ floor(V.value);
  $L \leftarrow$ copyAndAddConstraint(N, $V \leq a$);
  $R \leftarrow$ copyAndAddConstraint(N, $V \geq a + 1$);
  $L.dualBound \leftarrow O_r$;
  $R.dualBound \leftarrow O_r$;
  PQ.add(L);
  PQ.add(R);
end
return S^*;

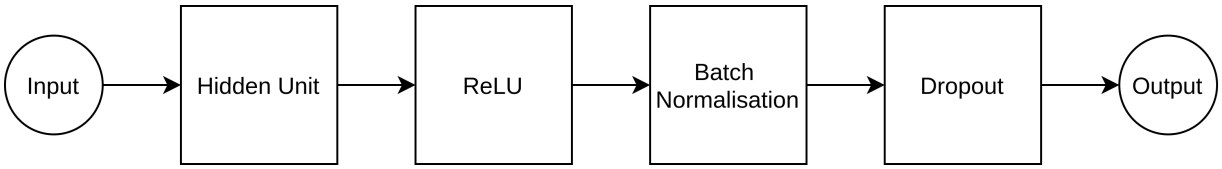


Figure 1: Operations within a hidden layer of the network.

Summarising the node selection heuristics, they can be grouped into two general strategies [Achterberg et al., 2008]. The first strategy 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. The second has the advantage to prune more nodes and decrease the search space. In this paper 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 to be selected.

3 Approach

Recall that our goal is to obtain an approximate 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. [2019]. 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. We constrain the action space for a_t to only select one child node or both. 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. Our sampling process of state-action pairs is similar to the prior work of He et al. [2014], with two major differences. The first is that our policy learns only 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 method. The second difference 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.

He et al. [2014] check whether the current node is in the path to the best solution. In our work, we check whether the left and right children of the current node are in a path that leads to 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.

Our machine learning model is a standard fully-connected multi-layer perceptron (MLP) 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. See Figure 1 for an overview of the operations within a hidden layer.

We obtain the model architecture parameters and learning rate ρ using the hyperparameter optimisation algorithm hyperopt[4] [Bergstra et al., 2013]. 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 [Kingma and Ba, 2015].

During policy evaluation, the action B ('both') can result in different operations. See Table 2 for an overview. 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*

[4] `github.com/hyperopt/hyperopt`

Table 1: Features that define a state. Variable features from Gasse et al. [2019]

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.

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 [Gleixner et al., 2018]; *Score* selects the node which obtained the highest score so far as calculated by our node selection policy.

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, we use our learned policy in two ways: the first is approximate 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.

4 Results and Discussion

The following standard NP-hard problem instances were tested: set cover, maximum independent set, capacitated facility location and combinatorial auctions. These problems are derived from the generator provided by Gasse et al. [2019]. The instances are different 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.[5] 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[6] [Paszke et al., 2019], 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* (the SCIP default), *RestartDFS* and *DFS*. Additionally, we compare our results with the node selector and pruner from He et al. [2014], 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. [2014] 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 use (S) and (B). For the experiments with a time limit, we use (P) and (B).

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. For every problem, we use the $k = 10$ best solutions to gather the state-action pairs. Additionally, 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*.

In more detail, we report three different kinds of experiments:

1. We evaluate the policy on every problem by checking the average solving time of each node selector.
2. We check the solution quality in terms of the optimality gap and the solving time of the first solution found at a leaf node. 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 inbuilt in SCIP.
3. 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 show the initial optimality gap obtained by the solver before branch-and-bound is applied, i.e., from the solver's pre-solve prior to search.

[5] Note that SCIP version 7, released after we commenced this work, does not bring any major improvements to its MIP solving.
[6] `www.pytorch.org`

Table 2: Parameter settings for our node selection and pruning policy.

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

Table 3: Machine learning parameters and prediction results for every problem. The baseline accuracy is predicting everything as the majority class.

Problem	Base acc	Test acc	H	U	p_d	ρ
Set cover	0.575	0.764	1	49	0.445	0.253
Max ind set (10)	0.923	0.922	1	25	0.266	0.003
Max ind 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

For each experiment, we apply the policies on two different difficulty levels:

1. Easy instances, which can be solved within 15 minutes.
2. Hard instances, where we set a solving time limit to one hour. Here, for all three experiments we substitute the optimality gap for the integrality gap, because the optimal solution is not known for every hard instance. Additionally, for the first experiment, instead of checking the solving time, we check the integrality gap.

Table 3 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 Section 4.2 for an explanation. The best performing ML model is the model with the settings that achieve the lowest validation loss.

The experiments 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 = 1) as the average across all metrics. This is a standard practice for MIP benchmarks.[7]

4.1 Set cover

These instances consist of 2,000 variables and 1,000 constraints forming a pure binary minimisation problem. We sampled 17,254 state-action pairs on the training instances, 2,991 on the validation instances and 3,218 on the test instances. The model achieves a testing accuracy of 76.4%, with a baseline accuracy of 57.5%.

See Table 4 for the average solving time and explored nodes of various node selection strategies. *BestEstimate* achieves the lowest average solving time at 26.4 seconds; ML_RB comes next at 35.7 seconds. 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 can not reject the null hypothesis of equal means. 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. Again, we can not 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). Recall that the values in the tables are the shifted geometric means, while the t-tests by their nature compares (arithmetic) means. The difference between the solving time of *BestEstimate* and any ML policy here is not statistically significant due to outliers.

Figure 2 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. Note that the only parameter that is influential for the ML solver is the first parameter, namely *on_both*. The second parameter *on_leaf* and the third parameter *prune_on_both* do not influence the solving time or quality of the first solution as the search terminates at the first found leaf that is found. The policy of He et al. [2014] is not included here, due to its 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.

See Table 5 for the mean optimality gap of the baselines using 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 average

[7] See, e.g., `plato.asu.edu/bench.html`

Table 4: Set cover instances: average solving time and explored nodes for various node selection strategies. Pair-wise t-tests: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$ compared to *BestEstimate*.

Strategy	Solving time (s)	Explored nodes
BestEstimate	**26.39**	**4291**
DFS	47.10 *	9018 *
He	499.00 ***	47116 ***
He6	85.73 **	18386 **
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 *

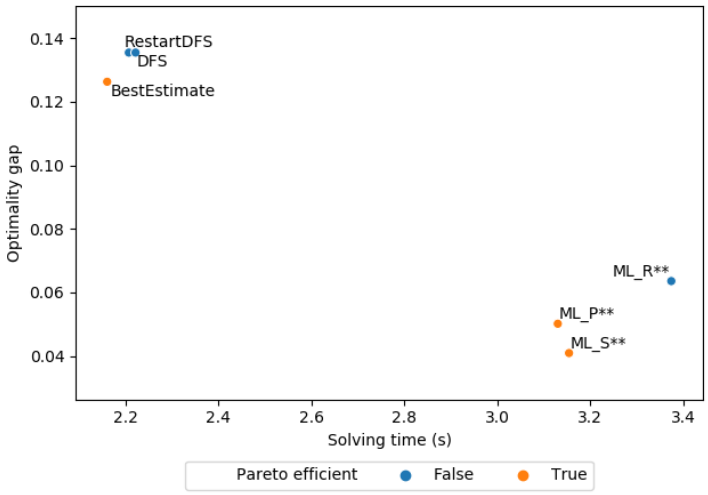


Figure 2: Average solving time against the average optimality gap of the first solution found at a leaf node on set cover instances

Table 5: Set cover: model with *on_both* = Second, *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 2.746. Pairwise t-tests against the ML policy: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$.

Strategy	Optimality gap
BestEstimate	0.1767 **
DFS	0.0718 ***
He6 (prune only)	0.7988 ***
He6 (both)	1.1040 ***
ML_SRF	**0.0278**
RestartDFS	0.0741 ***

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

Strategy	Solving time (s)	Explored nodes
BestEstimate	158.42	6344
DFS	**155.27**	**6340**
He	394.56 **	30965 **
He6	204.68	7992
ML_PB (10)	260.34 **	10029 *
ML_PB (40)	227.45 *	8100
ML_PR (10)	255.49 **	10191 *
ML_PR (40)	222.59 *	8581
ML_PS (10)	245.92 *	10184 ·
ML_PS (40)	207.28 ·	7878
ML_RB (10)	274.17 **	10296 *
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 **	10661 *
ML_SB (40)	283.02 **	10491 *
ML_SR (10)	252.59 **	9936 *
ML_SR (40)	258.68 **	10120 *
ML_SS (10)	242.37 *	9921 ·
ML_SS (40)	274.73 **	11251 *
RestartDFS	183.24	7854

solving time and average optimality gap across all instances. In this case, ML_SRF has the lowest harmonic mean and also achieves the lowest average 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.

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

4.2 Maximum independent set

These instances consist of 1,000 variables and around 4,000 constraints forming a pure binary maximisation problem. For this particular problem, we noticed that for $k = 10$, the class imbalance was significant. To combat this, we increased the value k to 40. For $k = 10$, we sampled 29,801 state-action pairs on the training instances, 5,820 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 6 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.

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

Figure 3 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 7 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

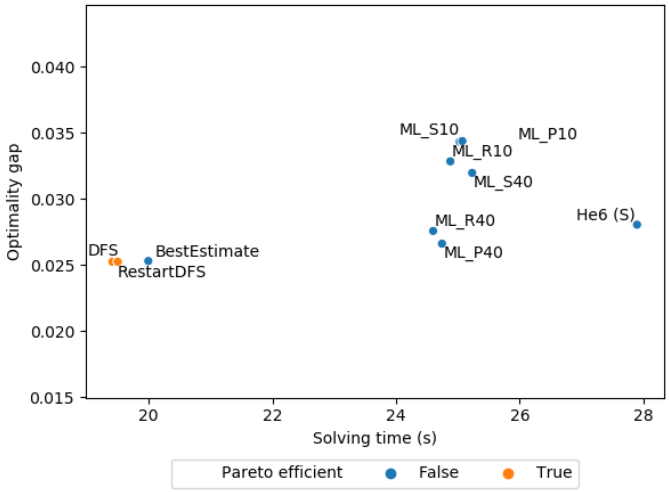


Figure 3: Average solving time against the average optimality gap of the first solution found at a leaf node on maximum independent set instances.

Table 7: 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: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$.

Strategy	Optimality gap
BestEstimate	0.0174
DFS	**0.0134** *
He6 (both)	0.9930 ***
He6 (prune only)	0.9902 ***
ML_PRF	0.0211
RestartDFS	0.0134 *

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.

4.3 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, 3,531 on the validation instances and 3,431 on the testing instances. The model achieves a testing accuracy of 90.1%, with a baseline of 73.1%.

See Table 8 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 8: Capacitated facility location instances: average solving time and explored nodes for various node selection strategies. Pair-wise t-tests: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$ compared to *BestEstimate*.

Strategy	Solving time (s)	Explored nodes
BestEstimate	122.79	1674
DFS	690.40 ***	17155 ***
He	1754.10 ***	5733 ***
He6	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 4 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 9 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.

4.4 Combinatorial auctions

These instances consist of 1,200 variables and around 475 constraints forming a pure binary maximisation problem. We sampled 13,554 state-action pairs on the training instances, 2,389 on the validation instances and 2,170 on the testing instances. The model achieves a testing accuracy of 71.7%, with a baseline of 57.0%.

See Table 10 for the average solving time and explored nodes of various node selection strategies. *BestEstimate* achieves the lowest average solving time at 19.7 seconds. 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.

Figure 5 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 11 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.

4.5 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 size of these hard instances are in terms of 4,000 variables and

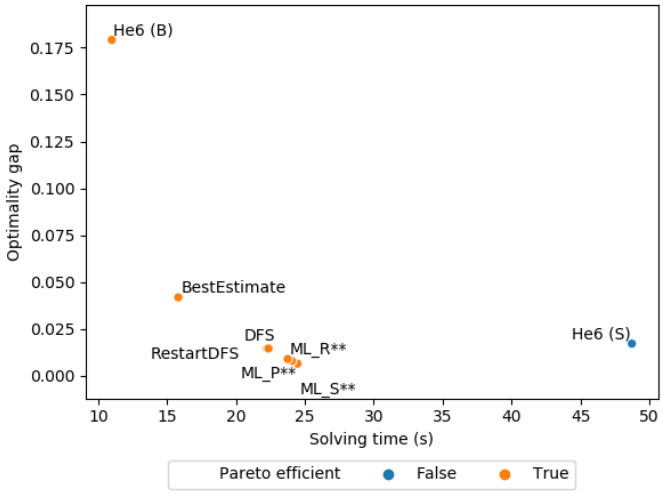


Figure 4: Average solving time against the average optimality gap of the first solution found at a leaf node on capacitated facility location instances.

Table 9: 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: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$.

Strategy	Optimality gap
BestEstimate	0.0821 ***
DFS	0.0619 ***
He6 (both)	0.1516 ***
He6 (prune only)	0.1503 ***
ML_SST	**0.0065**
RestartDFS	0.0590 ***

Table 10: Combinatorial auctions instances: average solving time and explored nodes for various node selection strategies. Pair-wise t-tests: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$ compared to *BestEstimate*.

Strategy	Solving time (s)	Explored nodes
BestEstimate	**19.68**	**3489**
DFS	23.48	4490
He	40.11 ***	4519
He6	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

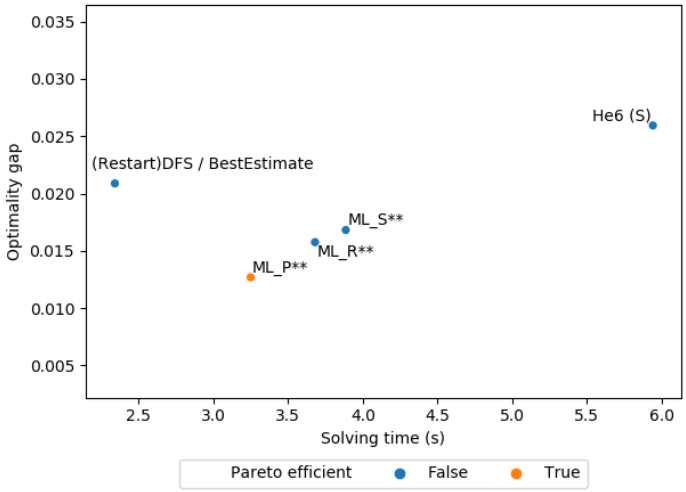


Figure 5: Average solving time against the average optimality gap of the first solution found at a leaf node on combinatorial auctions instances.

Table 11: 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: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$.

Strategy	Optimality gap
BestEstimate	0.0174 *
DFS	**0.0126**
He6 (both)	0.4678 ***
He6 (prune only)	0.2873 ***
ML_PST	0.0127
RestartDFS	0.0126

2,000 constraints, while the easier set cover instances had 2,000 variables and 1,000 constraints. We evaluated 10 hard instances due to computational limitations, and focused on *BestEstimate* as a baseline on the node selection policy.

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 12 shows the average solving time, explored nodes and integrality gap of various node selection strategies. *BestEstimate* achieves the lowest average solving time at 2256.7 seconds and integrality gap at 0.0481. ML_SB has the lowest number of explored nodes at 109298. 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 all our ML policies (lowest observed p-value: 0.64). 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 0.05 for ML_PB, ML_RB and ML_SB. Lastly, we have 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.

For the pruning policies, Figure 8 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 13 and Figure 9 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 average solving time and average 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. In all instances, the solver

Table 12: Hard set cover instances: average solving time, explored nodes and integrality gap for various node selection strategies. Pair-wise t-tests: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$ compared to *BestEstimate*.

Strategy	Solving time (s)	Explored nodes	Integrality gap
BestEstimate	**2256.69**	161438	**0.0481**
ML_PB	2497.56	112869 *	0.0714
ML_PR	2531.32	150055	0.0763
ML_PS	2279.41	157367	0.1152
ML_RB	2441.60	111629 *	0.0683
ML_RR	2552.68	152871	0.0794
ML_RS	2352.93	153727	0.1289 ·
ML_SB	2427.25	**109298** *	0.0706
ML_SR	2685.09	161825	0.0734
ML_SS	2381.28	163631	0.1278 ·

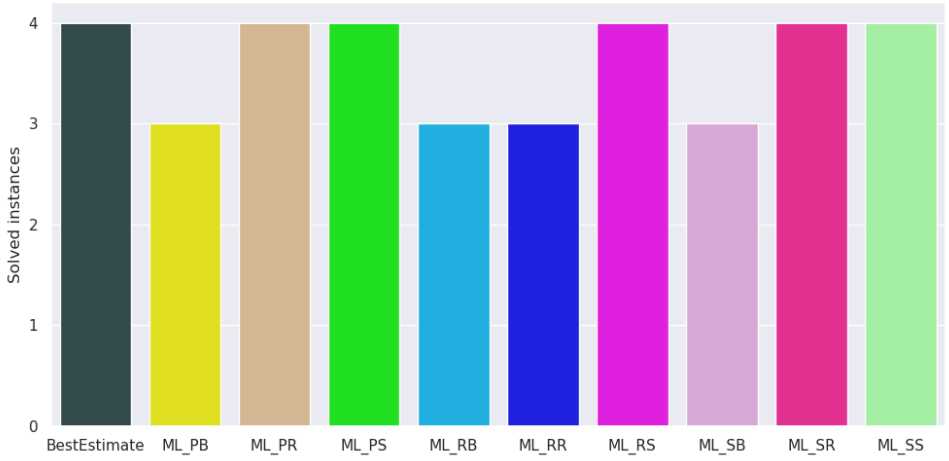


Figure 6: Hard set cover instances: number of solved instances for each node selection strategy. Total number of instances is ten.

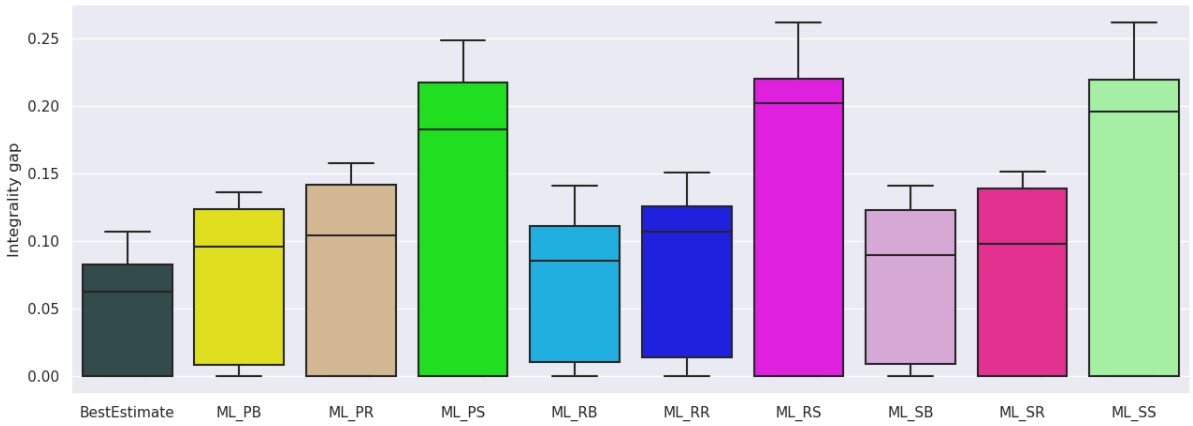


Figure 7: Hard set cover instances: integrality gap of each node selection strategy.

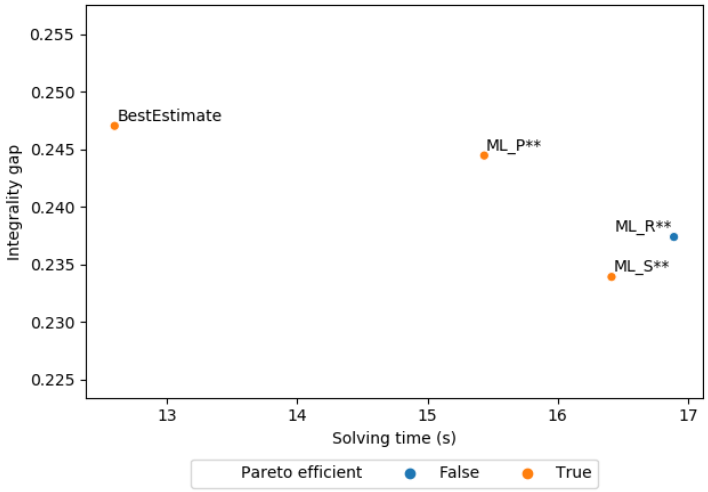


Figure 8: Average solving time against the average integrality gap of the first solution found at a leaf node on hard set cover instances.

Table 13: Set cover (hard): 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: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '·' $p < 0.1$.

Strategy	Integrality gap
BestEstimate	1.3678 ***
DFS	0.3462 ***
He6 (both)	1.8315 ***
He6 (prune only)	1.6835 ***
ML_PST	**0.2445**
RestartDFS	0.3489 ***

could only obtain an integrality gap of infinity on the first feasible solution. This means we can not compare the initial integrality gap to the found integrality gaps during branch and bound.

4.6 Discussion

We examined three experiments, namely measuring the solving time for an exact solution, measuring the solving time and optimality gap for the first solution found at a leaf node in the branch and bound tree, and setting a low instance-specific time limit to measure what the optimality gap is.

For the first experiment, our method performed better than the baselines for capacitated facility location problem, but worse on the three purely binary problems. The best ML policy is ML_RB, although ML_SB has an almost equal solving time, while still exploring fewer nodes.

For the second 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.

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 they do against the baselines. These policies were chosen based on the (lowest) harmonic mean between the solving time and optimality gap as was conducted in the second experiment. In four of the five problem sets, our policies had a statistically significantt lower optimality gap than the baselines, while on combinatorial auctions, ML_PST did not perform statistically significantly worse than *DFS* and *RestartDFS*.

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*}

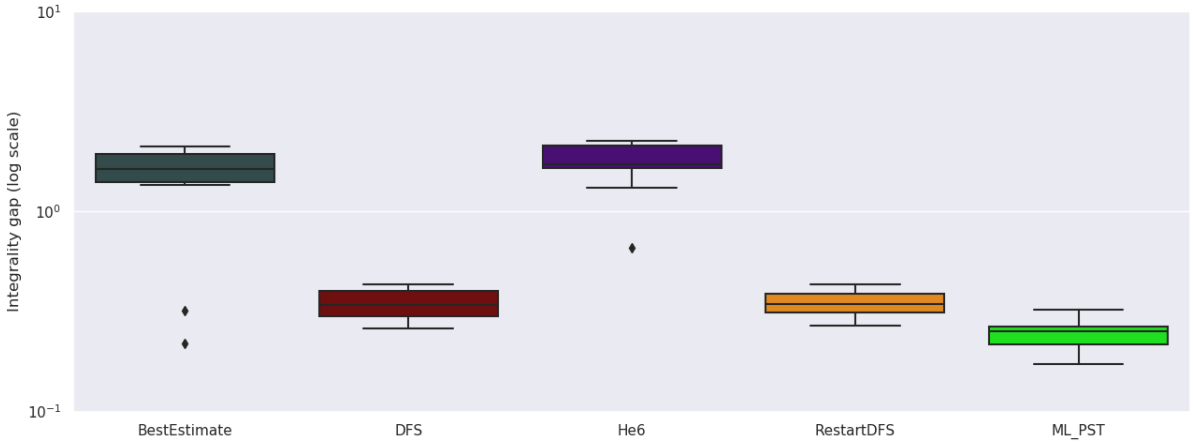


Figure 9: Hard set cover instances: Integrality gap of various approximation strategies. Note log scale on y-axis. ML_PST outperforms the compared methods.

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; when it does not, inferior performance can be expected.

Lastly, 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 overhead of the Python interface.

5 Related Work

5.1 Branching

Deciding on what variable to branch on in the branch and bound process is called branching, as was mentioned in Section 2.3. 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 [2017].

Strong branching [Applegate et al., 1995] is a popular branching strategy, among other strategies such as most infeasible branching, pseudo-cost branching, reliability branching [Achterberg et al., 2005] – used as the default in SCIP – and hybrid branching [Achterberg and Berthold, 2009]. Strong branching creates the smallest trees, as Achterberg et al. [2005] reported that strong branching required around 20 times less nodes to solve a problem than most infeasible branching and around 10 times less 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. [2014] 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 [Geurts et al., 2006]. 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. [2016] extended Alvarez et al. [2014] 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 [Joachims, 2006]. Their algorithm solved 70% more hard problems (over 500,000 nodes, cut-off time 5 hours) 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. [2019] 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. [2014] and Khalil et al. [2016] 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. [2014] and 70% faster than the method of Khalil et al. [2016], both for hard problems (1,500 auctions).

Seeing their success, we adopt the same variable features as Gasse et al. [2019].

5.2 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. [2014] 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. [2014] 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. [2014] by constraining the node selection space to direct children only at non-leaf nodes. Furthermore, 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. Lastly, we include branched variable features, obtained from Gasse et al. [2019]. As seen in Section 4, our approach easily outperforms that of He et al. [2014], in both their original implementation and a re-implementation in SCIP 6.

6 Conclusion

This paper shows that approximate solving of mixed integer programs can be improved by a node selection policy obtained with offline imitation learning. 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 approximate settings.

Empirical results on four MIP datasets indicate that our node selector leads to solutions more quickly than the state-of-the-art in the literature [He et al., 2014], but not as quickly as the state-of-practice SCIP node selector. While we do not beat the highly-optimised SCIP baseline in terms of solving time on exact solutions, our approximation-based policies have a consistently better optimality gap than all baselines if the accuracy of the predictive model adds value to prediction. Further, the results also indicate that our approximation method finds better solutions within a given time limit than all baselines in four of the five problem classes examined.

This paper shows that learned policies can be Pareto-equivalent or superior to state-of-practice MIP node selection heuristics: heuristics that have been honed by hand over many years. It adds to the body of literature that demonstrates how machine learning 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 [Achterberg et al., 2008]. By contrast, in general for binary instances, the greater challenge is to close the dual bound, and our learned policy also performs well here.

For future work, 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.

We mentioned during pre-processing that certain features are removed that are constant throughout the entire dataset. This has the consequence of different number of input units in the neural network architecture for every problem. The bigger issue is that this work focused on training a machine learning model for every problem. Future work could include a method to unify a machine learning model that works for all problems. This would make ML-based node selection a more accessible feature for current MIP solvers, like SCIP.

A limitation of this study is that we only performed experiments with the MIP solver SCIP. We chose SCIP, because it is open-source and has a rich amount of documentation. However, another MIP solver Gurobi (proprietary) is generally faster than SCIP and it would be interesting to see how the ML-based node selection policy compares to Gurobi.

Lastly, reinforcement learning, in contrast to imitation learning, is an interesting research direction to create a node selection policy.

Acknowledgements

Thanks to Robbert Eggermont and Lara Scavuzzo.

References

P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proc. of 21st International Conference on Machine Learning (ICML'04)*, 2004. doi: 10.1145/1015330.1015430.

T. Achterberg. *Constraint integer programming*. PhD thesis, Technische Universität Berlin, 2007.

T. Achterberg and T. Berthold. Hybrid branching. In *Proc. of 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09)*, volume 5547 of *Lecture Notes in Computer Science*, pages 309–311. Springer, 2009. doi: 10.1007/978-3-642-01929-6_23.

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

T. Achterberg, T. Berthold, T. Koch, and K. Wolter. Constraint integer programming: A new approach to integrate cp and mip. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 6–20. Springer, 2008.

A. M. Alvarez, Q. Louveaux, and L. Wehenkel. A supervised machine learning approach to variable branching in branch-and-bound. In *Proc. of 7th European Machine Learning and Data Mining Conference (ECML-PKDD'14)*, 2014.

D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Finding cuts in the TSP (a preliminary report). Technical Report 5, Center for Discrete Mathematics & Theoretical Computer Science, 1995.

C. Bayliss, G. D. Maere, J. A. D. Atkin, and M. Paelinck. A simulation scenario based mixed integer programming approach to airline reserve crew scheduling under uncertainty. *Annals of OR*, 252(2):335–363, 2017. doi: 10.1007/s10479-016-2174-8.

Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *CoRR*, abs/1811.06128, 2018. URL `http://arxiv.org/abs/1811.06128`.

J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proc. of 30th International Conference on Machine Learning (ICML'13)*, pages 115–123, 2013.

M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Proc. of 2019 Neural Information Processing Systems (NeurIPS'19)*, pages 15554–15566, 2019.

P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 6.0. Technical report, Optimization Online, 2018. URL `http://www.optimization-online.org/DB_HTML/2018/07/6692.html`.

H. He, H. Daumé III, and J. Eisner. Learning to search in branch and bound algorithms. In *Proc. of 2014 Neural Information Processing Systems Conference (NeurIPS'14)*, pages 3293–3301, 2014.

T. Joachims. Training linear SVMs in linear time. In *Proc. of 12th International Conference on Knowledge Discovery and Data Mining (KDD'06)*, pages 217–226, 2006. doi: 10.1145/1150402.1150429.

E. B. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina. Learning to branch in mixed integer programming. In *Proc. of 30th AAAI Conference on Artificial Intelligence (AAAI'16)*, pages 724–731, 2016.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Proc. of 3rd International Conference on Learning Representations, (ICLR'15)*, 2015.

A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3): 497–520, 1960.

A. Lodi and G. Zarpellon. On learning and branching: a survey. *TOP*, 25:207—236, 2017. doi: 10.1007/s11750-017-0451-6.

M. Lombardi, M. Milano, and A. Bartolini. Empirical decision model learning. *Artificial Intelligence*, 244:343–367, 2017.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proc. of 2019 Neural Information Processing Systems (NeurIPS'19)*, pages 8024–8035, 2019.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT Press, Cambridge, MA, 2018.