On Learning for Node Selection in the Branch-and-Bound Algorithm using Reinforcement Learning

J. J. Groenheide

Master's Thesis
Delft University of Technology

On Learning for Node Selection in the Branch-and-Bound Algorithm using Reinforcement Learning

by

J. J. Groenheide

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday 15th of July, 2024 at 15:00 CET.

Student number:	4915623	
Project duration:	6th of November, 2023 – 8th of July, 2024	
Thesis committee:	Dr. N. Yorke-Smith,	TU Delft, supervisor
	Dr. D. de Laat,	TU Delft

Preface

As I am writing this preface, just a few hours before the deadline - because of course - I would like to take a quick moment to look back on this journey. I have had an incredible time as a student at the TU Delft, starting all the way back in 2018 when I had only just turned 17, to now 6 years later. It has been a luxury and a privilege to be the student of so many amazing professors over the years, and to meet so many amazing peers and friends. And all of it has lead up to this final project, which I started over 8 months ago. I remember seeing the open research position for *learning to optimise*, and immediately feeling drawn to it. I had experience with combinatorial optimisation, and had an absolute blast with the deep reinforcement learning course. Like I did over 6 years ago, when I blindly chose to do a Bachelor in Computer Science and Engineering at the TU Delft without considering any other options, I also chose this thesis research without considering anything else. And just like back then, it worked out beautifully for me. The benefit of being born with golden hat.

With that said, I would like to thank my supervisor, professor Neil Yorke-Smith, for putting up with my weekly rambling sessions, as I had few other people willing and able to listen to everything I had been working on that week. Of those few other people, I most importantly want to thank Lara Scavuzzo for not only taking the time to discuss some particularly difficult questions with me, but also for giving me the idea for this research both through her previous work and during the meeting we had at the very start of this whole journey. Additionally, I would like to thank Reuben Gardos Reid for picking up some meetings, giving my supervisor some well-deserved rest, as well as professor David de Laat for reading this thesis and being part of my thesis defence committee. On a more personal note, I would like to thank my family, who always pretended to understand what I was talking about at the dinner table. Finally, I would like to thank my girlfriend for staying up with me while I write this, as well as for all the support these last 8 months. I would not have been able to do it without you.

While I would love to say that this experience has taught me a lot about having realistic expectations and setting approachable goals, that would be a lie. I'm still just as stubborn as ever, and I'm afraid it will stay that way forever. I'm really grateful for this amazing experience, however. I'm grateful for all the guidance, but most importantly for all of the challenges that I was allowed to overcome on my own. I hope my life can continue to be this good.

J. J. Groenheide
Maasdijk, July 2024

Abstract

The branch-and-bound algorithm is used by solvers to efficiently find the optimal solution of discrete optimisation problems. It does so by sequentially partitioning parts of the search space based on the solution to the linear relaxation of the problem. This sequential decision-making is performed by the variable selection and node selection heuristics. The sequential nature of these heuristics makes them suitable for trajectory-based learning in the form of imitation learning and reinforcement learning. Learning problem-specific heuristics in this way has become increasingly popular in recent years.
Despite their similarities, the two heuristics have very different dynamics during learning, and success has mainly been achieved for variable selection. In this work, we evaluate the node selection problem and formulate a *learning to select* paradigm for both imitation and reinforcement learning. We find that *learning to select* is generally more difficult due to the small margin of possible improvement over the current baselines, and the lack of informative features to distinguish nodes during ranking. These challenges are exacerbated by focusing on sibling comparisons, which are generally the most difficult due to the high similarity between the nodes. Sibling comparisons are also arguably the most important in node selection, however, due to the importance of plunging to reduce context switching overhead. The results indicate that both approaches fail to learn meaningful decision-making policies based on the limited fixed-size feature representation of the nodes. A code repository for reproducing and extending the experiments is publicly available at https://github.com/jgroenheide/rl2select.

Contents

1 Introduction

With the Earth's population surpassing 8 billion people in November of 2022, it is more important than ever to optimally schedule, design, and route the logistics of the world. The capacity of the global merchant fleet has grown by more than 43 percent in the last decade, reaching 2.2 million dead-weight tons in 2022[1]. In that same year, the global production of agricultural commodities reached 9.6 billion tonnes, a growth of 56 percent since the start of the millennium[2]. Simultaneously, sustainability requirements are forcing innovation in all sectors. Advanced planning and scheduling (APS) software is becoming more common. Finding optimal solutions to these kinds of tasks has been done for centuries, but never in a way as structured and on as large a scale as today [Paschos, 2014]. This has created a demand for solvers that can provide optimal solutions to a wide range of problems at a very high rate.

Integer programming is a prominent way of solving combinatorial optimisation problems [Hoffman and Padberg, 2001]. It offers a way to encode combinatorial optimisation problems as an integer assignment problem. The benefit of encoding problems in this way is that the integer assignment problem can be efficiently solved using the branch-and-bound algorithm [Land and Doig, 1960]. This algorithm relies on the natural relaxation of the integer assignment problem to a linear program, which can be efficiently solved using the simplex algorithm. The solution to the relaxation can be used to bound the optimal solution value and prune parts of the solution space. Different solvers have sprung up around the mixed-integer linear programming (MILP) approach, each implementing slightly different variants of the branch-and-bound algorithm and the mechanisms that help complete the search as quickly as possible, like presolve, cutting planes, and heuristics [Koch et al., 2022].

At its core, the branch-and-bound algorithm consists of two main heuristics, which sequentially determine how the solution space is traversed. The *variable selection rule* decides which variable is chosen for branching. The solution space is then divided into two child nodes, with each child node representing a partition of the original problem with a new bound on the chosen variable. The new nodes are added to a list of open nodes, one of which must be chosen to continue the search. Which sub-problem is solved at each step is determined by the *node selection rule*. The node selection rule can choose from any unsolved sub-problem in any sub-tree of the branch-and-bound tree, but it is common for special priority to be given to child and sibling nodes of the last-solved LP due to overhead incurred from context-switching [Sabharwal et al., 2012].

The node selection rule is dependent on a node comparison operator to order the open nodes, similar to a priority queue. This node comparison operator is deterministic and depends only on the features of the nodes being compared. These two aspects make it particularly well-suited for a statistical learning approach. Recent work has created a learned node comparison rule through the use of imitation learning on a graph neural network [Labassi et al., 2022]. While this approach is similar to the state-of-the-art in *learning to branch* [Gasse et al., 2019, Scavuzzo et al., 2022], the node selection problem has not reached the same level of success. In this work we will highlight the differences between these heuristics, the implications for the learning process, and explore the effect of learning a node comparison rule using reinforcement learning.

[1]https://unctad.org/system/files/official-document/tdstat47_FS14_en.pdf
[2]https://fao.org/3/cc9205en/cc9205en.pdf

1.1. Research questions

While variable selection and node selection appear to fulfil similar goals, they actually work quite differently in practice. Variable selection directly influences the sub-problems that are created at each node, and as such the search tree that is created. Depending on the order in which variables are branched on, the path to a primal solution can be different. It can therefore be used to minimise the length of the path from the root node to the solution. Node selection only determines the order in which nodes created by the variable selection heuristic are explored. This makes variable selection much more influential in determining the efficiency of the solver [Achterberg, 2007].

The influence of node selection is limited by the fact that, regardless of search order, all nodes created by the variable selection rule with a lower bound better than or equal to the optimal objective value will have to be expanded to prove optimality of the solution. Meanwhile, any node with a lower bound that is worse than the optimal objective value can be pruned at some point during the search. The only way node selection can increase the number of nodes solved in a static search is by choosing a node that could have otherwise been pruned. With this in mind, we can easily form a greedy heuristic that is always optimal for minimising the number of nodes in a static branch-and-bound tree. By always choosing the node with the best lower bound we are guaranteed to never select a node with a lower bound worse than the optimal objective value. This approach is known as best-first search.

It is then interesting to note that this "optimal greedy heuristic" is not the default for most solvers. Indeed, they generally use a much more advanced procedure involving plunging with early plunge-abort, periodically performing best-first search, and otherwise following the highest-priority selection according to some heuristic. The reason for this is two-fold. **Firstly**, there is non-negligible overhead when switching between sub-problems. This makes repeatedly solving child-nodes of the current focus node during short plunges more efficient than constantly switching contexts to solve the new best bound node. **Secondly**, solvers do not work on a static branch-and-bound tree. Instead they use a combination of many different heuristics, cut techniques, and advanced variable selection rules that use information gathered during the search to speed up the solving process beyond the influence of the node selection rule itself. When these components are active, the solving time can be influenced in subtle ways by the order in which nodes are explored [Achterberg et al., 2008].

This puts into perspective previous research on learning for node selection and node comparison, which has mostly focused on imitating a diving oracle that plunges towards the optimal solution, without any regard for how this might affect the behaviour of other components of the search. Additionally, this approach ignores the effects of an imbalanced search tree on the proving phase of the solving process. We argue that the current direction of research for learning to select and compare is not making meaningful progress because of this lack of consideration for the softer components of solving, and uses incorrect metrics to describe the achieved results. For this section, we look towards answering the following research questions.

1. What kind of behaviour makes a good node selection heuristic?
2. How much room for improvement is there for node selection?
3. How accurate is number of nodes solved as a success metric?

The most important thing to realise is that node selection, heuristics, and the branch-and-bound algorithm are not an exact science. The components interact with each other in subtle, sometimes undefined ways. What works well for some problems might not work for others. We can relate the success of the node selector to many different things that make sense in theory, but sometimes a seemingly poor node selection decision can still improve the solving time because of the interaction with other components of the search. Because of this, we think reinforcement learning will be a better fit for node selection than imitation learning.

We posit that reinforcement learning, which is unbounded by the performance of an oracle, will be able to learn more nuanced interactions of the solving process and act accordingly. As such, our main question is that of whether a reinforcement learning approach is able to achieve better results than a policy trained with imitation learning. To test this, we will answer the following sub-questions.

4. Which proxy-metrics can be used as reward signal during learning?
5. What kind of behaviour is learned by an agent based on this reward?
6. Can an agent learn to optimise the efficiency of other components?

1.2. Contributions

Previous work has mainly focused on learning to imitate the decisions of a diving oracle, with the goal of finding a (near-)optimal solution early on, which should enable a high amount of pruning later in the search. In this research we assume that optimal node comparison behaviour cannot be approximated by an oracle because of the subtle interactions node selection can have with other components of the search, and as such is better learned through exploration using reinforcement learning.

As part of this research, we will explore the differences between node selection and variable selection, design different proxy-metrics for quantifying success, and evaluate the potential of reinforcement learning in *learning to select*. We use a CPU-friendly fixed-size feature array and a MLP architecture based on the baseline of previous work [Labassi et al., 2022]. Contrary to previous work, however, we only focus on sibling comparisons, which leads to a more natural formulation of the MDP environment and rewards. Learned policies are implemented and evaluated in the SCIP solver [Achterberg et al., 2005], a leading open-source MIP solver which has become the default for much of the research on *learning to optimise*. We compare these results against SCIP's default node selection rule, a greedy best-first search selector, and a random selector.

The results show that both imitation learning and reinforcement learning fail to convincingly decide between sibling nodes, which we attribute to the lack of distinct features between sibling nodes. We speculate that the fixed-size feature array is not informative enough for the network to make an informed decision, which is particularly visible in primal difficult problem sets where the global upper bound remains poor throughout the search. Simultaneously, we show that there is still a lot of room for improvement within node selection, with SCIP's default rule being outperformed by the random selector on a number of occasions.

1.3. Thesis outline

Chapter 2.1 will provide an introduction to the topics of combinatorial optimisation and integer programming, the branch-and-bound algorithm, machine learning, imitation learning, and reinforcement learning. Chapter 2.2 contains an overview of the related work in the field of learning for combinatorial optimisation. In Chapter 3 we will discuss the limitations of the standard MDP representation for *learning to select* when applying reinforcement learning, and explain our reasoning for limiting the node comparisons to sibling nodes. The experimental setup and the results can be found in Chapter 4. Finally we discuss the findings of the research in Chapter 5.

2 Preliminaries

In this chapter we cover the preliminary information for this research, including an introduction of the different techniques used, and the related work in the field of *learning to optimise*.

2.1. Background

Mixed-integer linear programming (MILP) is a way of representing mathematical optimisation problems where some or all of the variables are constrained to take integer values. As is common in the literature, we assume mixed-integer linear programs to be of the following form:

$$\begin{aligned} \min_{\mathbf{x},\mathbf{y}} \quad & \mathbf{c}^\top \mathbf{x} + \mathbf{h}^\top \mathbf{y} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} + \mathbf{G}\mathbf{y} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0}, \quad \mathbf{x} \in \mathbb{Z} \\ & \mathbf{y} \geq \mathbf{0}, \quad \mathbf{y} \in \mathbb{R}, \end{aligned}$$

where $(\mathbf{c}|\mathbf{h}) \in \mathbb{R}^n$, $[\mathbf{A}|\mathbf{G}] \in \mathbb{R}^{m \times n}$, and $\mathbf{b} \in \mathbb{R}^m$ denote the objective coefficients, the coefficient matrix, and the right-hand side respectively. The additional integrality constraints turn the polynomially solvable linear program into an NP-Hard problem, because the optimal solutions are no longer restricted to the vertices of the solution space. This means that, despite the additional constraints, the search space is exponentially larger than that of the relaxed linear program. Still, the solution to the linear program can be used as a bound on the solution of the integer problem. If the solution to the linear relaxation is integral, its objective value is used as an upper bound on the optimal solution value. If the solution has fractional variables, the objective value forms a lower bound on the optimal solution value in this sub-problem. If the lower bound of a sub-problem ever exceeds the global upper bound, we have a numerical guarantee that the optimal solution will not be in this part of the search space. By repeatedly splitting the search space and solving the linear relaxation of the sub-problems that are created, nodes can be pruned before they are fully explored.

This approach, later named the Branch-and-Bound algorithm, was first introduced in a 1960 journal article titled *"An automatic method of solving discrete programming problems"* [Land and Doig, 1960]. Already in the original introduction of the approach, the node selection is performed based on a best-first search approach of always expanding the node with the best lower bound on the problem. Simultaneously, the authors mention computational constraints, and suggest "carrying down the tree to a pre-determined cut-off value", which is still the most widely used approach for node selection to this day. Their prescience did not extend to branching variable selection, however, as they initially used a heuristic based on the distance from each variable value to the nearest integer. Other variable selection approaches were discussed in the appendices of their work, though none compare to the current state-of-the-art variable branching techniques. For a more extensive overview of the branch-and-bound algorithm and its components, we refer the reader to Achterberg, 2007.

While it has been proven that there cannot exist a universally effective heuristic for all problems, some heuristics are more generally effective than others. The best universal *branching variable selection heuristic* currently known is strong-branching, which is a computationally expensive branching strategy based on calculating and scoring the bound improvement in each branching candidate before taking a step. The computational cost of strong-branching makes it difficult to use in practice, but it does offer a strong baseline for approximation and imitation [Bengio et al., 2021]. This has been the motivation behind the development of techniques like partial strong-branching and reliability branching [Fischetti and Monaci, 2012, Achterberg et al., 2005]. While these heuristics are able to make highly effective branching decisions, their performance still falls behind that of full strong-branching.

In an attempt to better approximate the full strong-branching decisions, research has investigated creating a fast approximation provided by a machine learning model, learned through imitation with strong-branching decisions as oracle [Alvarez et al., 2014, Khalil et al., 2016, Gasse et al., 2019]. Node selection lacks such a strong baseline for imitation. Instead, research for node selection has focused on imitating a diving oracle that always chooses a node on the path from the root to the optimal solution [He et al., 2014, Song et al., 2018, Yilmaz and Yorke-Smith, 2021].

One of the prerequisites for this type of learning is that the environment must be able to be described using a Markov decision process (MDP). This was shown to be possible for the branch-and-bound algorithm by He et al., 2014. A MDP describes a set of states and actions, with transitions defined for each state-action pair. Importantly, MDPs must follow the Markov property, which states that transitions only depend on the current state and action. This ensures that the outcome of an action can be entirely judged by the future trajectory, without taking into account the history of the trajectory. This allows us to learn from each individual action, instead of the entire trajectory, which greatly increases the learning efficiency. The MDP control problem is the problem of finding a state-dependent action distribution that maximises the expected reward. The action distribution is referred to as the policy [Howard, 1960].

We consider two main approaches for solving the MDP control problem. The first is imitation learning, which is a type of supervised learning for sequential decision making. Samples are taken from the trajectories of an oracle policy. The actions of the oracle policy are assumed to be optimal regarding the expected cumulative reward, without having to quantify the exact value of each action. This is often done with the goal of automating human decisions, or to reduce computational costs. The policy is trained to predict (imitate) the actions of the oracle as closely as possible, while having access to a more limited state.

The second approach is reinforcement learning. Instead of learning the relation between the state and some external information known only by the oracle, we directly aim to learn the relation between the state and the reward. Since we no longer have examples of state-action pairs that lead to high reward, however, we need to gather this information by evaluating the policy and exploring different state-action pairs. Through an inherent desire to optimise the reward, the reinforcement learning agent will learn which actions to take in which states to maximise the obtained reward. For a further overview of reinforcement learning techniques, we refer the reader to Sutton and Barto, 2018.

2.2. Related work

Learning for combinatorial optimisation has been around for over a decade, with both variable and node selection being approached using imitation learning for the first time in 2014. For branching variable selection this research was done by Alvarez et al., 2014. Before this work, some attempts at *learning to branch* had used restarts to learn instance-specific branching information, but this was the first time it had been attempted to learn a general branching rule through an off-line training phase. They use a fixed-size feature array and an *Extremely Randomised Trees* (ExtRaTrees) model [Geurts et al., 2006] to approximate strong-branching decisions. Later, two separate works propose an on-line variant that learns to imitate calculated strong-branching decisions early in the search, before applying the learned approximation once the accuracy has been proven [Alvarez et al., 2016, Khalil et al., 2016]. More recent work has attempted to generalise the learned variable branching rules to heterogeneous sets of problems [Zarpellon et al., 2020]. They report much better generalisation across instance types, but fall short of SCIP's default performance.

The node selection problem has received relatively little attention compared to branching variable selection. The first research on *learning to select* with imitation learning was performed using a SVM$^{\text{rank}}$ model trained using the DAgger algorithm [He et al., 2014]. They learn to imitate the actions of a diving oracle, which has become the standard for *learning to select*. The research combined a learned node selection rule with an early-pruning mechanism. This work was later improved upon by switching to a MLP model, alongside a new type of diving oracle [Song et al., 2018]. The new approach was referred to as *retrospective imitation learning*. The resulting trajectories are almost identical to the trajectories of the original diving oracle, however. This work was then superseded by switching to a plunging policy approach, where the selection decision is reduced to the direct child nodes of the current focus node [Yilmaz and Yorke-Smith, 2021]. They use a similar MLP architecture, but show that the performance of the learned rule can be greatly improved with this approach.

For both branching variable and node selection, the performance of the learned policies was limited by the fixed-size feature representation, which necessarily loses information during aggregation. This was changed with the introduction of a GNN-based architecture, which takes as input a variable-constraint bipartite graph representation of the entire sub-problem at each node [Gasse et al., 2019]. The variable-constraint bipartite graph representation constitutes the current state-of-the-art for both *learning to branch* and *learning to select* [Scavuzzo et al., 2022, Labassi et al., 2022]. The current state-of-the-art for *learning to select* uses a Siamese GNN architecture to learn a good node comparison function, which is used by the node selection rule to order the open nodes. The Siamese architecture is a natural choice for node comparison, as it ensures the model defines a total order on the nodes of the search process. Since these developments, questions have been raised about the feasibility of using a GPU-reliant architecture to perform historically CPU-based processes. As such, the performance of GNN models was reevaluated on CPU-based machines, and a hybrid model was created which uses a MLP model during later stages of the search [Gupta et al., 2020].

Few attempts have been made to replace the imitation learning component in these works. Reinforcement learning was first applied to *learning to branch* using (among others) an actor-critic MLP architecture trained using PPO updates [Scavuzzo, 2020]. Unfortunately, the reinforcement learning approach was not able to consistently outperform the imitation learning approach in either performance or inference time, leading to the hypothesis that the imitation learning approach had already managed to reach near-optimal behaviour, or a strong local optimum. Similar research was performed simultaneously by Etheve et al., 2020, as well as in unpublished work by DS4DM[1].

This research was followed up by the introduction of an improved definition of the MDP environment for *learning to branch*, referred to as the *TreeMDP*, which improves on the way rewards are distributed across branching variable decisions [Scavuzzo et al., 2022]. When using a *TreeMDP*, the return of a branching decision is based entirely on the sub-tree that is created from the node in which that decision takes place. This is valid as long as the pruning bound is not externally changed after entering a sub-tree. While branching rules trained using the *TreeMDP* outperform the standard temporal MDPs and improve convergence, they are still unable to beat the performance of the imitation learning approach.

The work of Mattick and Mutschler, 2023, constitutes the only attempt at applying reinforcement learning to node selection we are aware of thus far. They introduce a novel approach for representing the state of the search tree based on a complex bi-simulation framework that aims to quantify interrelations between nodes through embedded message passing. The approach relies on simulating the entire branch-and-bound tree as a graph, with a score assigned to the leaf nodes, which are then used as a probability distribution over the open nodes. A reinforcement learning agent is then trained to choose actions from this probability distribution. We instead propose to return to a per-node approach.

One area where reinforcement learning has been successful is in *learning to approximate*. Bello et al., 2016, report significant improvements in creating near-optimal solutions for TSP by employing reinforcement learning instead of imitation learning, when applied to an identical pointer network architecture [Vinyals et al., 2015]. The reinforcement learning is implemented using an actor-critic approach. Khalil et al., 2017, then further improved on this approach using a `structure2vec` architecture, trained using deep Q-learning. This proves that, with the right features and architecture, reinforcement learning is able to make meaningful decisions in mathematical optimisation processes. For this reason, we continue to believe that reinforcement learning can make improvements in *learning to select*, and *learning to optimise* in general.

[1]Canada Excellence Research Chair in Data Science for Real-Time Decision-Making at Polytechnique Montréal. URL: https://cerc-datascience.polymtl.ca.

3 Methodology

Labassi et al., 2022, showed that the node comparison operator can be replaced with a learned policy that is able to compare any two nodes. The environment they use for this is not suitable for reinforcement learning, however. Instead, we create a new representation of the environment and adjust the feature set accordingly. These adjusted features are used to train a policy according to a reward signal through reinforcement learning. This learned policy is then applied within the SCIP solver. In this chapter we will go over the methodology and design choices that were made during this process.

3.1. Defining the environment

During solving, newly created nodes are inserted into the priority queue of open nodes through a series of comparisons with the nodes already present. These comparisons are performed by the node comparison operator and called by the ordering algorithm of the solver to create a partial order on the open nodes. Not every comparison has to be correct to optimally traverse the search tree, however. This is because some nodes might be removed from the queue before they are expanded, meaning their exact placement is irrelevant. Additionally, some orderings can be inferred from the transitive property of the node comparison operator. Solvers use this to minimise the number of comparisons needed to determine the next node to select. As a side effect of this optimisation, however, the selection reward cannot be distributed over the node comparisons.

Previous work using imitation learning gets around this by only considering trajectories containing exclusively comparisons involving a node on the path to the optimal solution. When such an optimal node is added to the queue of open nodes, all comparisons should be in favour of the new node, after which it is selected. By manually constructing the trajectories in this way, the dynamics of the ordering algorithm are completely circumvented. This same approach cannot be used with reinforcement learning, where trajectories are generated from evaluating the policy. Instead, we use the approach of [Yilmaz and Yorke-Smith, 2021] and only compare between the child nodes of the current focus node. By reducing the node selection to a single comparison, we completely remove the influence of the ordering algorithm from our environment. Additionally, limiting the node selection to a single comparison allows us to directly assign the selection reward to this comparison. With only a single comparison per node selection step, we can formulate our MDP with states $\mathbf{s} = (\text{node}_1, \text{node}_2)$ and actions $\mathbf{a} \in$ [-1,0,1] for left, equal, and right node preference. The transitions are defined between tree states, which are determined by SCIP's solver implementation.

The downside of this approach is that the policy is forced to continue until a leaf node is reached, even after a mistake is made. This is not a problem when learning to imitate the trajectories of a diving oracle, which always guarantee a perfect dive towards the solution. But in reinforcement learning, where mistakes will undoubtedly happen in the generated trajectories, this can lead to a large, unbalanced search tree that can be hard to recover from. To tackle this issue, we apply the policy in a node selection rule with a deterministic early plunge-abort mechanism, instead of the highest-priority node selection rule used in previous work. After plunging, the next node is chosen using a best-first search heuristic, with the intention to provide the best new starting position without relying on an external node comparison rule.

3.2. Defining the reward function

The only true success metric for *learning to optimise* is the solving time. Unfortunately, the solving time is hardware dependent and cannot easily be distributed over the trajectory. Instead, previous research has used the number of solved nodes in the completed branch-and-bound tree as a proxy metric. The number of nodes is easily distributed among node and variable selection decisions, as each solving step simply adds one new node to the number of solved nodes. While this reward makes sense for variable selection heuristics, where the branching decisions directly determine the sub-problems that are created, and therefore the size of the resulting sub-trees, node selection does not have this ability. In fact, node selection does not influence the efficiency of the created sub-problems in any way beyond the influence it has on other components of the search. This makes the number of solved nodes a much less informative reward for *learning to select* than for *learning to branch*.

To the best of our knowledge, there has only been one work on *learning to select* with a reward signal [Mattick and Mutschler, 2023]. They propose a reward signal based on the ratio between the achieved optimality gap of the agent and the default SCIP performance after 45 seconds of solving. This reward is then shifted to evenly match the [-1, 1] range. For this reward to work, however, instances cannot be completed to optimality within the 45 seconds solving time, as this would lead to a divide-by-zero error. As such, these instances are removed during training. This is justified by stating that small problems will give less informative reward signals, but this makes it unsuitable for our research, as we aim to train instances to completion. Instead, we look towards designing alternative reward signals.

Before designing a new reward signal, it is important to describe the desired behaviour. We can quantify a good node selection rule by the following three criteria. A good node selection rule should:

1. Avoid nodes that can later be pruned.
2. Minimise overhead from context switching.
3. Choose nodes that help other components.

Criteria 1 can be formulated into a per-step reward based on the inequality between the chosen node's lower bound and the optimal objective value. If the lower bound of a chosen node is higher than the optimal objective value, the pruning bound would have surpassed the lower bound at some point during the search, and the node could have been pruned[1]. By applying a penalty for choosing a node like this, the agent will learn to avoid these nodes while also finishing the search as quickly as possible. As previously mentioned, Criteria 1 is naturally satisfied by a best-first search approach, since it will never select a node with a lower bound above the optimal objective value. The "lb-obj" reward is more general, however, allowing for more optimisation in the order in which nodes are chosen.

One of the main issues with using the number of nodes ("nnodes") as a proxy metric for the solving time is that it does not include any penalty for incurring overhead (Criteria 2). While it is possible to apply a penalty for choosing nodes that are not child- or sibling-nodes of the current focus node, context switching is strictly necessary for efficiently traversing the branch-and-bound tree. When blindly following this criteria, the policy would collapse to some form of unbounded plunging or depth-first search. Instead, a penalty would need to be added to other reward signals based on whether a context-switching node was chosen. What we believe to be a better solution, however, is to apply the learned node comparison operator in a node selection rule that performs plunging as part of its procedure.

Finally, learning to satisfy Criteria 3 is the novel focus of this research. Node decisions that are valuable to the performance of other components are difficult to define before-hand, but with the right reward signal we believe that a reinforcement learning agent will be able to learn how to optimally feed these components. We measure the success of the components based on the efficiency with which they find feasible solutions. Larger upper bound improvements should be rewarded more heavily, however. We award the agent with a reward equal to the achieved global upper bound improvement, normalised by the total achievable upper bound improvement after presolve, and multiplied by a discount factor to incentivise finding good solutions early in the solve. This reward is naturally maximised by the plunging oracle of previous work using imitation learning, but with additional freedom to optimise the trajectory.

[1]Cut techniques can create a situation where the lower bound of a node is improved beyond the optimal objective value when the LP-relaxation is strengthened with additional cuts. For the sake of the argument, however, we will assume a static environment without cutting planes.

3.3. Defining the state representation

We use a modified version of the state representation of Labassi et al., 2022, which is itself a simplification of the state representation of He et al., 2014. To evaluate the variety in the state representation, we aggregate all of the created imitation samples to the minimum value, the maximum value, and the average value for each feature in the state representation. Since the number of sibling comparisons in a single instance is relatively small, we use the sampling procedure of Yilmaz and Yorke-Smith, 2021, which creates trajectories from the root to the top-k solutions. Samples are taken from the search tree created by SCIP's *BestEstimate* heuristic. At every node visited in this process, we check for each of the two children nodes which of the top-k solutions of that instance are present in the sub-tree of that node. Contrary to the approach of previous work, however, when both of the children lead to one of the top-k solutions, the associated action is the node leading to the highest ranking solution. This reduces the action space of the samples to just "left" and "right". Because of the symmetric property of the node comparison operator, $\text{comp}(\text{node1}, \text{node2}) = 1 - \text{comp}(\text{node2}, \text{node1})$, changing the order of the nodes in the comparison also changes the associated action, such that the same node is prioritised again. We use this to create an equal class balance between "left" and "right" actions by randomly swapping the nodes in each sample with a 50% probability.

We evaluate each of the problem benchmarks (discussed in section 4.1) separately to highlight the different behaviours among them. The number of samples for each benchmark set is based on the average number of samples that can be generated per instance. 50 for GISP, 25 for CFLP, and 5 for MKP. These statistics are for the combined samples of the training and validation set, generated for the top-10 solutions. The results can be found in Table 3.1.

	GISP			CFLP			MKP		
	Min	Max	Mean	Min	Max	Mean	Min	Max	Mean
prio_down	0	0	0	0	0	0	0	0	0
prio_up	0	0	0	0	0	0	0	0	0
bound_lp_diff	0	1	0.548	0	1	0.508	-1	1	0.508
root_lp_diff	-0.75	0.208	-0.198	-0.992	0.886	-0.156	-1	1	-0.073
pseudo_cost	0	69.334	27.796	0	603.290	46.088	0	45.000	0.179
n_inferences	0	34.75	2.196	0	66.714	4.180	0	83.333	3.460
node_type_child	1	1	1	1	1	1	1	1	1
node_type_sibling	0	0	0	0	0	0	0	0	0
node_type_leaf	0	0	0	0	0	0	0	0	0
estimate	-0.086	0.922	0.419	0	1.337	1.023	0	1	1.000
node_lb	0	1	0.820	0	1.059	1.008	0	1	1.000
relative_bound	0	0.981	0.391	0	0.984	0.301	0	0.900	6.19e-3
relative_depth	1	1	1	1	1	1	1	1	1
node_depth	0	32	10.068	0	34	10.672	0	21	2.518
focus_depth	0	32	10.068	0	34	10.672	0	21	2.518
plunge_depth	0	9	0.693	0	11	0.667	0	8	0.234
global_ub	0	0.723	0.664	0	1.129	1.023	0	0.999	0.992
bound_gap	0	0.531	0.269	0	0.129	0.019	0	0.034	7.86e-3
ub_is_infinite	0	0	0	0	0	0	0	0	0
gap_is_infinite	0	0	0	0	0	0	0	0	0

Table 3.1: Original feature statistics

We evaluate the results to create an adjusted feature array for the sibling comparison setting. The original feature array contains 8 features that are constant. The `prio_down` and `prio_up` features are never active, because the priority setting for each variable is set to `AUTO` by default. We also see that the infinity indicators `ub_is_infinite` and `gap_is_infinite` are never active. While this is not guaranteed, all instances for all problem sets are apparently easy enough for SCIP's presolve to find at least one primal solution in the root node. Interestingly, these features are independent of the type of comparison, meaning they are also never active for general node comparison.

Unsurprisingly, the node type is always `node_type_child` when performing child comparisons, making this feature redundant. Similarly, the `relative_depth` between nodes will always be 1, since sibling nodes are always at the same depth. Finally, the `node_depth` and `focus_depth` are always the same, meaning one of the two can be removed. After removing these features, we are left with 11 of the original 20 features.

We propose to add a binary indicator `is_prio_child`, which is true if the node is marked as the priority child of the current focus node. We also normalise the `plunge_depth` by the maximum plunge depth as calculated by SCIP's *BestEstimate* selection rule, and the `node_depth` by the maximum depth of the entire search process. The adjusted feature statistics can be found in Table 3.2.

	GISP			CFLP			MKP		
	Min	Max	Mean	Min	Max	Mean	Min	Max	Mean
bound_lp_diff	0	1	0.548	0	1	0.508	-1	1	0.508
root_lp_diff	-0.75	0.208	-0.198	-0.992	0.886	-0.156	-1	1	-0.073
pseudo_cost	0	69.334	27.796	0	603.290	46.088	0	45.000	0.179
n_inferences	0	34.75	2.196	0	66.714	4.180	0	83.333	3.460
estimate	-0.086	0.922	0.419	0	1.337	1.023	0	1	1.000
node_lb	0	1	0.820	0	1.059	1.008	0	1	1.000
relative_bound	0	0.981	0.391	0	0.984	0.301	0	0.900	6.19e-3
is_prio_child	0	1	0.5	0	1	0.5	0	1	0.5
node_depth	0	1	0.568	0	1	0.422	0	1	0.650
plunge_depth	0	2	0.102	0	1	0.051	0	1.5	0.018
global_ub	0	0.723	0.664	0	1.129	1.023	0	0.999	0.992
bound_gap	0	0.531	0.269	0	0.129	0.019	0	0.034	7.86e-3

Table 3.2: Adjusted feature statistics

8 of the 12 remaining features will be the same between child nodes, which leaves only 4 distinct features to distinguish the two nodes in a sibling comparison. Developing an informative set of features for node selection will continue to be an issue, as there is simply not a lot of defining features for the nodes without using a bipartite graph representation of each sub-problem. To quantify the loss of information in the new representation, both the original and the adjusted feature arrays were used to train a model based on node comparisons and sibling comparisons. The results are shown in Table 3.3.

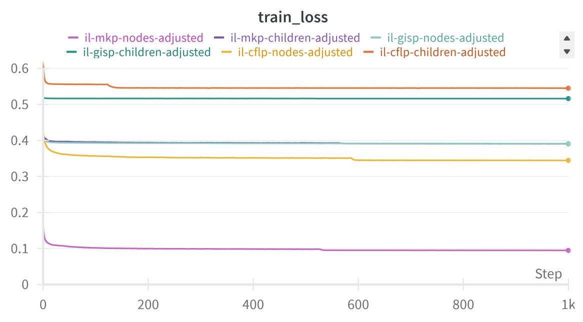
	Nodes		Children	
	Original	Adjusted	Original	Adjusted
GISP	82.9%	82.9%	74.9%	75.0%
CFLP	85.5%	83.2%	73.6%	74.0%
MKP	92.6%	96.5%	61.2%	85.2%

Table 3.3: Imitation accuracy for original and adjusted features.

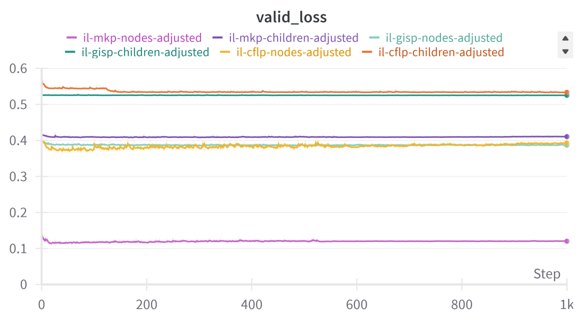
As expected, there is very little difference between the imitation accuracy of the original and the adjusted node representation for child comparisons[2]. More surprisingly, this is also the case for node comparisons. This indicates that the reduced feature set is as expressive as the original. Only for the MKP benchmark do we see a large change in accuracy, from 61.2% to 85.2%. We believe this increase is due to the newly added `is_prio_child` feature, which appears to be highly informative for MKP.

We note the general drop in accuracy between nodes and children. This indicates that child comparisons are indeed more difficult to imitate than general node comparisons, which makes sense considering the reduced number of distinct features between sibling node representations. The decrease

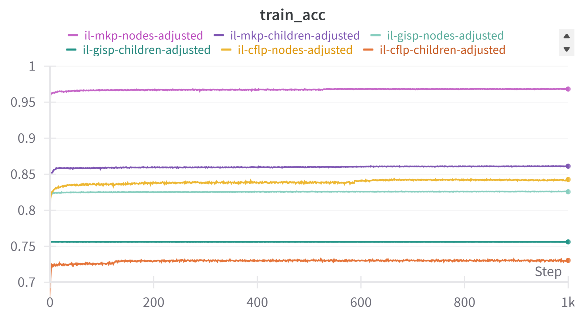
[2]Samples were generated separately, meaning small variances in performance can be attributed to variance in the samples.



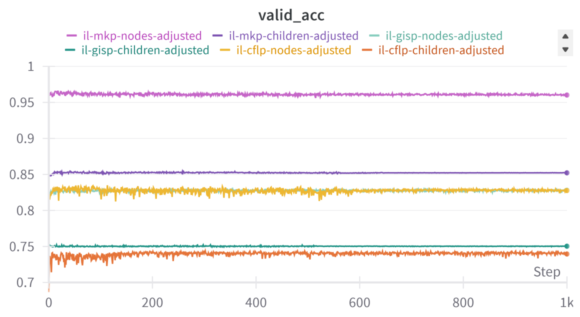
(a) IL training loss (Binary cross-entropy)



(b) IL validation loss (Binary cross-entropy)



(c) IL training accuracy (compared to oracle)



(d) IL validation accuracy (compared to oracle)

Figure 3.1: Learning behaviour for imitation learning baselines.

in accuracy does not lead to a decrease in performance, however, as policies trained on node comparisons perform equal to those trained on child comparisons during evaluation. This is to be expected though, since during evaluation we exclusively perform sibling comparisons. Despite the decrease in accuracy, we believe it will be important to continue reporting the sibling comparison accuracy in *learning to select* research. Because of the importance of plunging to minimise context switching overhead, it will be important to focus on optimising the sibling comparison performance alongside general node comparison accuracy. Figure 3.1 shows the learning behaviour for all IL baselines[3].

3.4. Training procedure

Our training procedure is based on the architecture of Scavuzzo et al., 2022. They use a plain REINFORCE algorithm with an entropy bonus to train a GNN model. We instead use a Siamese MLP model, which was shown to be effective for learning a good node comparison function by Labassi et al., 2022, achieving over 95% test accuracy in imitating the node comparison decisions of a diving oracle on all benchmark sets. The implementation uses PyTorch [Paszke et al., 2019] and PySCIPOpt [Maher et al., 2016] for interfacing with SCIP Optimization Suite 9.0 [Bolusani et al., 2024]. We obtain state-action-reward samples from running the current policy. Changes to the model are made based on the return of the state-action until the end of the episode. Episodes can run very long, however, so we only take a small subset of samples to learn on; roughly 5%. We stop the training once the optimal solution is found, since node selection is most impactful when the global upper bound is poor. The approach is implemented asynchronously, with n agents able to run in parallel on different seeds, which helps to stabilise learning by decorrelating the data [Mnih et al., 2016]. We refer to the individual papers for more information on the exact implementation.

[3]In case of printing in black-and-white. Experiments are reported from highest to lowest.
Loss results: CFLP children, GISP children, GISP nodes, MKP children, CFLP nodes, MKP nodes.
Accuracy results: MKP nodes, MKP children, CFLP nodes, GISP nodes, GISP children, CFLP children.

4 Experimental results

We start this chapter with an introduction of the benchmark problems that were used for this research, the parameters that were chosen, and the generation methods that were used. We then explain the training procedure for our reinforcement learning agents, and introduce some initial findings about their behaviour. We then move on to large-scale tests and the results thereof. We discuss the observed behaviour of the agent for each phase, before moving on to any potential additions or ablations of the approach.

4.1. Benchmarks

We evaluate our results on three separate problem sets comprising different NP-hard instance families. In line with previous work on the node comparison problem by Labassi et al., 2022, these problem sets are chosen to be primal difficult, since the impact of node selection is highest when the global upper bound is poor. The first benchmark is taken directly from their work, but refactored for better readability. The other two benchmarks are taken from the field of *learning to select*. These benchmarks create plenty of variety in the types of instances to learn on.

4.1.1. Generalised independent set problem

An independent set is a set of vertices within a network such that no two vertices are adjacent. When the aim is to find a maximal size independent set in a graph, we are dealing with the NP-Hard maximum independent set problem. A graph can have many independent sets, however. To limit the number of optimal solutions, we can add a value to each vertex, with the objective to find an independent set that maximises the total value of the vertices contained. This problem is referred to as the maximum-weight independent set problem. To further increase the complexity, we can make a subset of edges removable for a fixed cost. This increases the total number of possible independent sets in the solution space, while keeping the number of optimal sets low. This variant of the problem was first introduced in a 1997 article on forest harvesting [Hochbaum and Pathria, 1997]. The problem is posed as finding a balance between lumber quality and wildlife habitat preservation. We use a heavily refactored version of the instance generator of Chmiela et al., 2021, to generate instances.

$$\boxed{\begin{aligned}
\max_{\mathbf{x},\mathbf{y}} \quad & \sum_{i\in V} w_i x_i - \sum_{(i,j)\in E2} c_{ij} y_{ij} \\
\text{subject to} \quad & x_i + x_j \le 1, \quad \forall (i,j) \in E1 \\
& x_i + x_j - y_{ij} \le 1, \quad \forall (i,j) \in E2 \\
& x_i \in \{0,1\}, \quad \forall i \in V \\
& y_{ij} \in \{0,1\}, \quad \forall (i,j) \in E2
\end{aligned}}$$

4.1.2. Single source capacitated facility location problem

The facility location problem is a well-known combinatorial problem based on a real-world issue of supply and demand. Customers want their demands to be met by facilities that can supply them. Which facilities supply which customers is based on a variable cost incurred by transferring goods between those locations, and whether the facility is opened, which also incurs a fixed cost. The variant of this problem we consider here is a capacitated single-source variant, which means that facilities have a limited capacity on the amount of supply they can deliver, and each customer can only be supplied by one facility. These additional constraints increase the primal difficulty of the problem. For an overview of the problem, we refer the reader to Holmberg et al., 1999. The code for the instance generator was taken from the work of Gasse et al., 2019, who in turn cite Cornuejols et al., 1991.

$$\boxed{\begin{aligned}
\min_{\mathbf{x},\mathbf{y}} \quad & \sum_{i\in M}\sum_{j\in N} c_{ij}x_{ij} + \sum_{i\in M} f_i y_i \\
\text{subject to} \quad & \sum_{j\in N} a_j x_{ij} \le b_i y_i, \quad \forall i \in M \\
& \sum_{i\in M} x_{ij} \ge 1, \quad \forall j \in N \\
& x_{ij} - y_i \le 0, \quad \forall i \in M, \forall j \in N \\
& x_{ij} \in \{0,1\}, \quad \forall i \in M, \forall j \in N \\
& y_i \in \{0,1\}, \quad \forall i \in M
\end{aligned}}$$

4.1.3. Multiple knapsack problem

In the standard knapsack problem we are given a set of items N, each with a value v_i and a weight w_i. The goal is then to choose a subset of items $\hat{N}$ such that the total value is maximised while keeping the total weight within some capacity value c. If the capacity of the knapsack is subdivided into multiple sections, we speak of the multiple knapsack problem, as each section of the total capacity can be seen as a separate knapsack. In this generalisation of the knapsack problem we consider a set of knapsacks M, each having a separate capacity value c_j. Note that the linear relaxation of the multiple knapsack problem is the same as that of the standard knapsack problem when $\sum_{j\in M} c_j = c$. The generator is taken from the work of Scavuzzo et al., 2022, who cite Fukunaga, 2011.

$$\boxed{\begin{aligned}
\max_{\mathbf{x}} \quad & \sum_{i\in M}\sum_{j\in N} v_j x_{ij} \\
\text{subject to} \quad & \sum_{j\in N} w_j x_{ij} \le c_i \quad \forall i \in M \\
& \sum_{i\in M} x_{ij} \le 1, \quad \forall j \in N \\
& x_{ij} \in \{0,1\}, \quad \forall i \in M, \forall j \in N
\end{aligned}}$$

4.1.4. Dataset generation

For each of the aforementioned problem types we simultaneously generate and solve instances. This gives us greater control over the difficulty of instances that are saved, based on the solving statistics. Solving every generated instance to optimality is also necessary to collect optimal solutions of the instances, which are used for training and evaluation. All instances are solved using SCIP's default settings, with a time limit of 30 seconds. Feasible problems are saved if the number of nodes in the solve are between 100 and 1000 nodes.

For the instance generation we use the following parameters. Generalised independent set instances are generated on Erdos-Renyi graphs of 60 nodes for standard instances and 80 nodes for transfer instances with an edge probability of 0.5. Each edge is made removable with a 60% chance. Capacitated facility location instances are generated with 25 customers and 25 facilities, and a capacity/demand ratio of 3. Transfer instances are generated with 60 customers instead. Multiple knapsack instances are generated with 100 items and 4 knapsacks for standard instances and 8 knapsacks for transfer instances using weakly-correlated weights and values for the items.

These parameters were chosen based on the average number of nodes and the percentage of passing instances, with the goal of keeping the average number of nodes low, while maximising the number of instances that pass. In addition to speeding up the generation process for convenience, having a high number of passing instances is also important for ensuring the instances are actually difficult to solve. Since instances are only solved for one seed, there might be cases where an easy instance required a lot of nodes because SCIP was initialised with a difficult seed, but on average the instance is much easier. While this could be prevented by solving each instance multiple times during generation, we instead favour generating larger datasets.

4.2. Training

The reinforcement learning agent is trained on a set of 4000 instances, and validated every 50 episodes on a stable set of 50 validation instances. Model parameters are updated after each training batch of 10 training episodes. The model is saved if the 1-shifted geometric mean on the number of nodes in the final tree improves upon the current best achieved tree size. We run to a maximum of 1000 epochs, or 3 days of computation time. All experiments were run on two single-threaded Intel Xeon Platinum 8358P CPUs running at 2.60GHz. The CPUs were provided by the TU Delft through a remote virtual machine. Training was initially done in a static environment, which means disabling primal heuristic, cut generation, and conflict analysis. We experimented with enforcing a static variable selection rule, but this did not lead to improved learning behaviour. Even with static variable selection, however, there was still variance in the created search tree based on the order in which nodes were selected, of which we were unable to identify the source.

Training was done for each of the three reward signals discussed in Section 3.2. Namely, the tree size ("nnodes"), the lower bound-objective value inequality ("lb-obj"), and the global upper bound improvement ("gub+"). Initial tests were done using the "nnodes" reward signal, trained in a static environment. This did not create the desired learning behaviour, however. The agent would have a high variance in the training loss, without meaningful validation improvement. Our hypothesis for this behaviour is that the REINFORCE algorithm penalises correct node selection decisions at the top of the tree, when the tree size penalty is very high, while ignoring correct decisions that are further down the tree. As mentioned previously, the node selection rule only influences the total tree size by maximising pruning and through its impact on other components of the search. In a static environment these other components are turned off, leaving only pruning decisions to be learned, which is done mostly near the leaves of the tree. Unfortunately, switching to an active environment did not improve the learning behaviour, which we attribute to the lack of distinct features for the agent to draw on.

The "lb-obj" reward works similarly to the tree size reward, but only penalises nodes that have a lower bound higher than the optimal objective value. While this reward is more suitable for learning in a static environment, as it creates less variance in the training loss, it still lacks meaningful validation score improvement. This should not be the case, since theoretically the "lb-obj" value can be minimised based solely on the `node_lb` feature. However, it seems the reward distribution still pushes the policy away from this behaviour, possibly based on decisions where both nodes of the comparison would receive a penalty.

Finally, the "gub+" reward is the most tree size independent, focusing purely on finding good primal solutions early in the search. The return value of each action is the discounted sum of normalised upper bound improvements, normalised over the total obtainable upper bound improvement between the root node solution and the optimal objective value. Even with an extremely slow discount factor of 0.997, however, the rewards are too sparse to create a strong reward signal. Despite different learning behaviour, evaluating the learned policies gave identical results for all benchmarks. We attribute this behaviour to a lack of distinguishing features and the presence of strong local optima.

4.3. Evaluation

We report the performance of each node selection rule as the geometric mean over a set of 50 test and 50 transfer instances, with a time limit of 150 seconds. The results of each instance are averaged over all the completed runs from 5 distinct seeds, which reduces variance and discrepancies from crashed runs. Results that contain instances that ended because of timeout are indicated with an asterisk. All heuristics are evaluated in both a static and non-static search environment, regardless of which environment they were trained on. We consider three non-policy baselines. We evaluate SCIP's *BestEstimate* node selection heuristic (SCIP default), which is the default node selection heuristic for SCIP. We also evaluate the greedy best-first search heuristic, which always chooses the best bound node in the tree (B*FS). Note that this is not the *BestFirstSearch* node selector as implemented in SCIP. Finally, we evaluate a random node comparison function, which randomly chooses between the two nodes of the comparison with a 50% probability (Random). Every learned policy, as well as the Random node selector, are added to a node selector using a modified version of the *BestEstimate* node selection heuristic, which always chooses the highest priority child while plunging, and the best bound node otherwise. Only the node comparison is performed by the policy. We first report the performance of an untrained policy (Untrained). We then report the performance of the imitation learning benchmarks (IL policies), which have been trained according to the description in section 3.3. Finally, the performance of the reinforcement learning agents, trained with different reward signals, is reported (RL policies). The results for the policies have been grouped to avoid repetitive data, since all policies reached the same local optimum, regardless of training procedure.

	Test				Transfer			
	Static		Active		Static		Active	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
SCIP default	1994×/1.23	**1.36×/1.14**	812×/1.47	**2.05×/1.12**	11803×/1.18	9.09×/1.17	5521×/1.30	**10.00×/1.17**
B*FS	**1855×/1.32**	1.50×/1.19	**809×/1.54**	2.81×/1.21	11388×/1.19	10.76×/1.19	**5407×/1.35**	17.29×/1.23
Random	1976×/1.25	1.46×/1.15	885×/1.48	2.22×/1.13	**10657×/1.17**	**8.95×/1.18**	5620×/1.31	10.10×/1.17
Untrained	2055×/1.25	1.73×/1.17	1299×/1.36	2.62×/1.15	11923×/1.19	11.10×/1.19	7417×/1.31	13.34×/1.19
IL Policy	2055×/1.25	1.78×/1.17	1299×/1.36	2.69×/1.15	11923×/1.19	11.32×/1.19	7417×/1.31	13.99×/1.19
RL Policy	2055×/1.25	1.78×/1.17	1299×/1.36	2.68×/1.15	11923×/1.19	11.36×/1.19	7417×/1.31	13.66×/1.19

Table 4.1: Evaluation results for GISP benchmark.

	Test				Transfer			
	Static		Active		Static		Active	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
SCIP default	**24076×/1.96**	**44.17×/1.99**	805×/3.47	8.19×/1.80	7119×/2.50	**43.14×/1.94**	1178×/3.08	20.67×/1.91
B*FS	28978×/1.61	67.31×/1.61	**375×/5.65**	8.24×/2.04	**5771×/2.80**	52.33×/2.09	**406×/5.62**	**17.65×/2.21**
Random	33226×/1.61	60.38×/1.71	598×/4.49	**7.73×/1.84**	9766×/1.92	60.00×/1.70	875×/4.31	20.19×/2.11
Untrained	*80738×/1.41	117.01×/1.41	989×/3.71	9.02×/1.87	*21324×/1.95	86.01×/1.75	1095×/4.17	22.15×/2.07
IL Policies	*80726×/1.41	117.14×/1.41	989×/3.71	9.22×/1.87	*21248×/1.95	86.14×/1.75	1095×/4.17	22.17×/2.07
RL Policies	*80767×/1.41	116.92×/1.42	989×/3.71	8.94×/1.87	*21345×/1.95	85.93×/1.75	1162×/4.59	22.68×/2.25

Table 4.2: Evaluation results for CFLP benchmark.

	Test				Transfer			
	Static		Active		Static		Active	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
SCIP default	**1155×/4.40**	**0.35×/3.86**	**508×/2.80**	**0.54×/1.50**	4846×/3.54	4.58×/3.01	1060×/4.18	2.30×/2.20
B*FS	3152×/7.67	0.94×/7.16	618×/6.12	0.63×/2.67	**4579×/5.39**	**3.94×/4.46**	**630×/11.14**	**1.72×/4.46**
Random	5993×/6.15	1.91×/6.56	2346×/4.24	1.41×/2.83	32834×/3.68	30.03×/3.27	16185×/3.54	16.86×/2.73
Policy	*116015×/2.08	72.49×/2.15	9424×/3.52	6.54×/3.11	TIMEOUT	TIMEOUT	*82801×/1.86	110.86×/1.78
IL Policies	*117543×/2.09	72.33×/2.19	9594×/4.04	6.83×/3.44	TIMEOUT	TIMEOUT	*75099×/2.15	107.07×/1.97
RL Policies	*115832×/2.11	71.39×/2.16	8463×/4.34	6.12×/3.39	TIMEOUT	TIMEOUT	*86805×/1.61	115.98×/1.57

Table 4.3: Evaluation results for MKP benchmark.

4.4. Discussion

While the behaviour of each of the three problem benchmarks varies, the learned policies perform worse than the baselines in all cases. This was the expected outcome, since machine learning models have not beaten the state-of-the-art in node selection before. It is surprising, however, that the untrained policy performs equal to the learned policies. This would suggest that either no learning has occurred at all, or the randomly initialised policy is immediately in a local optimum. We hypothesise that the feature representation is not expressive enough to meaningfully decide between the nodes, leading to worse-than-random decision making. We will discuss the behaviour of the policies in the context of each of the benchmarks.

GISP is the most well-behaved benchmark, with small differences between the learned policies and the baselines. This might be explained by the performance of the Random baseline, which performs almost identical to the other two baselines. This indicates that the node selection decisions are not highly impactful in determining the efficiency of the solve for GISP instances. We also see the expected behaviour from the baselines. If we ignore the strong performance of the Random selector for the static transfer set, SCIP's default rule has the best time performance in all cases, while B*FS successfully minimises the number of nodes in the tree, at the cost of higher solving time due to context switching.

The CFLP benchmark shows similar behaviour, but with some outliers. B*FS reduces the number of nodes in all evaluations except for static test instances, where SCIP's default has a lower geometric mean, though with a considerably higher standard deviation. SCIP's *BestEstimate* node selector is also outperformed in static evaluations by the Random selection rule and B*FS respectively. We attribute this to the much larger number of nodes needed in the solve, which indicates that the lower bound is quite weak for these instances, leading to a low amount of pruning. In fact, the active transfer evaluation is the only one where the policies outperform SCIP's default selection rule on the number of nodes, though they still fall behind in solving time. In general, we see a large increase in the solving time for CFLP instances in the static environment, indicating that CFLP benefits greatly from other components of the search. We attribute the respectable performance of the policies in active evaluations to the efficiency of other components of the search, and consequently the poor performance in static evaluations to the lack thereof. While evaluating in a static environment is not a realistic setting, an active environment can sometimes mask the poor performance of a node selection rule, which is very clearly visible in this case.

MKP is the most unwieldy benchmark. Not only does it contain the largest differences in performance between the selection rules, but within each evaluation we also see shockingly large variance. The instances are extremely easy for the baselines, with test instances being solved on average within 2 seconds by even the Random selection rule, but taking more taking more than 70 seconds during policy evaluation. We also see that every run of every transfer instance under static evaluation timed out after 150 seconds. We again relate these results to the performance of the Random heuristic, which for the transfer instances takes almost 10 times as long as the other two baselines. This indicates that MKP instances are very difficult to solve by random selection, meaning the node selection decisions are highly impactful. We attribute this to a poor lower bound again, as indicated by the domination of the B*FS selector on the transfer instances. This exposes a highly intriguing aspect of *learning to select*, which is the benchmark selection. There appears to be an inherent difficulty to problems for learning a node selection rule, which is separate from the difficulty of solving the instances using SCIP's default implementation. While a full analysis of the learning behaviour is beyond the scope of this research, it lays the groundwork for further investigation and improvement.

5 Conclusions

The aim of this research was to investigate the differences between variable selection and node selection within the context of *learning to optimise*, and investigate the applicability of reinforcement learning for the node selection problem. We divided the research into a number of sub-questions (Section 1.1).

We first analyse the influence of the node selection rule on the branch-and-bound tree and identify the ways it can improve the efficiency of the solving process as part of answering the preliminary research questions. We define the behaviour of a good node selection rule in Section 3.2, determining that the best way to minimise the solving time is to maximise pruning, avoid context switching, and choose informative nodes for other components of the search. We also show that node selection has gone through relatively little development since the introduction of the branch-and-bound procedure in 1960, where the node selection was already performed using best-first search and plunging. How the plunge is performed still provides some room for improvement, but in general node selection is not very impactful in the efficiency of the solver. This is highlighted by the performance of the Random plunger heuristic evaluated in Section 4.3. The results also show that the number of nodes used in the solve does not always equate to the best solving time. Despite these results, the number of nodes needed to complete the solve remains the best proxy metric for solving time, assuming that context switching overhead is taken into consideration during the design. This answers the final question of the preliminary investigation. Based on these findings we then design an environment for learning.

The objective in *learning to select* can be reduced to creating a ranking of the open nodes in the search, such that the best one can be chosen. In the SCIP solver, this ranking is created by an ordering algorithm, which calls the node comparison operator when two nodes need to be compared. While this simplifies the node selection decision to an efficient set of binary comparisons, it adds hidden dynamics to the environment due to the implicit comparisons inferred from previous comparisons. What's more, reward signals that quantify the number of nodes used in the solve cannot easily be distributed over node comparisons, since not all comparisons are impactful to the search. We solve this issue by following previous work by Yilmaz and Yorke-Smith, 2021. By limiting the search to the immediate children of the focus node, we can directly assign the node selection reward to the single node comparison that is made at each step. We incorporate this approach into a selection rule with a deterministic early plunge abort mechanism to avoid creating large imbalanced search trees during unbounded plunges.

Based on this definition of the environment we answer question 4 by introducing two additional reward signals in Section 3.2. We introduce the lb-obj inequality reward, which penalises node decisions where the chosen node has a worse lower bound than the optimal objective value, and the upper bound improvement reward, which gives a normalised reward based on the improvement of the global upper bound. These rewards are naturally optimised by B*FS and the plunging oracle of previous work respectively, but are less restrictive, allowing for further optimisation. While question 5 can then be answered theoretically based on the expected learned behaviour, the experiments do not allow us to identify what kind of behaviour is actually learned based on these rewards, as all policies converge to the same local optimum. This is also the case for question 6, which unfortunately means the potential for reinforcement learning to learn more nuanced interactions of node selection decisions with other components of the search remains inconclusive.

The results of this research show that a fixed-size feature array is not informative enough for a reinforcement learning agent to meaningfully decide between sibling nodes. This same outcome can be seen for the imitation learning setting (Section 3.3), where the performance does not significantly improve anymore after just a few epochs. At this point, all the information in the feature representation has been exhausted and there is no further improvement to be made. It should be possible to consider general node comparisons in the environment, as long as the dynamics of the ordering algorithm are changed such that the node selection decisions can be directly related to the comparisons that were impactful to that decision, without considering other comparisons that preceded it. The easiest way to achieve this might be to return to a ranking model approach, where all comparisons are explicitly performed at every step. This issue is not simply solved by expanding the scope of the comparisons to involve all open nodes, however, since plunging is an important part of any effective node selection rule due to context switching overhead. We also show in Section 3.3 that a policy trained based on general node comparisons achieves equal performance to a policy trained on child comparisons when evaluated in a plunging node selector. This indicates that sibling comparisons will continue to be difficult, even for policies that perform well on general node comparisons. As such, we believe this will continue to be a challenge for node selection, as the defining features of the sub-problems are limited.

Future work might investigate applying the node bipartite graph representation to a GNN model, which has been shown to be more expressive in representing sub-problems. The usefulness of this approach is limited by the necessity of GPU access for training and applying the GNN model on historically CPU-based machines. Additionally, the training setup might be changed to produce better results. Alternatively, we might look to enhance the feature representation using diving statistics gathered during a preliminary dive starting in each of the two child nodes, or even all open nodes in the search. Another possible direction for future work is to learn a better approximation for SCIP's *estimate* function, which is an important component in both the node selection and node comparison of the default *BestEstimate* approach. We believe that improving the accuracy of this measure would also improve the performance of the *BestEstimate* selector, and as such the efficiency of the solving process.

Based on the results of the sub-questions, we posit that the reward signals are either too small or too detached from the node selection decisions for a reinforcement learning agent to learn meaningful decision making. This is in addition to the feature representation being too limited. We also conclude that the node selection problem creates a generally weaker learning environment than the variable selection problem because of the inherent difference in the state representation. Variable selection is defined using state transitions between sub-problems (nodes), while node selection transitions are defined between states of the entire branch-and-bound tree (generally represented based on the open nodes of the search). Variable selection decisions directly influence the efficiency of the newly created sub-problems. As a result, minimising the sub-tree size during plunging also minimises the global tree size, as mentioned in Etheve et al., 2020. This property is used by the *TreeMDP* representation of the variable selection problem to decompose the trajectory and better distribute rewards. In contrast, the quality of a node selection decision is not related to the decisions made in the ancestors of the focus node, and the sequential states of the branch-and-bound tree cannot be decomposed in any way.

Despite the challenges of *learning to select*, we believe that it will offer the best approach for developing problem-specific node selection rules in the future. We hope that the research performed in this thesis, along with the insights and baselines contained in it, can serve as a starting point for future work. As part of this research we have created a code repository for performing further *learning to select* research. Based on code from the field of *learning to branch* [Gasse et al., 2019, Scavuzzo et al., 2022], it has been rewritten to allow for clean and extendable node selection research. The code can be found at https://github.com/jgroenheide/rl2select, DOI: 10.4121/c93be76f-bb24-488e-895d-98c9810b3364.

Bibliography

Achterberg, T. (2007). Constraint integer programming.

Achterberg, T., Berthold, T., Koch, T., & Wolter, K. (2008). Constraint integer programming: A new approach to integrate cp and mip. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-68155-7_4

Achterberg, T., Koch, T., & Martin, A. (2005). Branching rules revisited. *Operations Research Letters*, *33*, 42–54. https://doi.org/10.1016/j.orl.2004.04.002

Alvarez, A. M., Wehenkel, L., & Louveaux, Q. (2016). Online learning for strong branching approximation in branch-and-bound.

Alvarez, A. M., Louveaux, Q., & Wehenkel, L. (2014). A supervised machine learning approach to variable branching in branch-and-bound in ecml.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., & Bengio, S. (2016). Neural combinatorial optimization with reinforcement learning.

Bengio, Y., Lodi, A., & Prouvost, A. (2021). Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, *290*, 405–421. https://doi.org/10.1016/j.ejor.2020.07.063

Bolusani, S., Besançon, M., Bestuzheva, K., Chmiela, A., Dionísio, J., Donkiewicz, T., van Doornmalen, J., Eifler, L., Ghannam, M., Gleixner, A., Graczyk, C., Halbig, K., Hedtke, I., Hoen, A., Hojny, C., van der Hulst, R., Kamp, D., Koch, T., Kofler, K., … Xu, L. (2024, February). *The SCIP Optimization Suite 9.0* (ZIB-Report No. 24-02-29). Zuse Institute Berlin. https://nbn-resolving.org/urn:nbn:de:0297-zib-95528

Chmiela, A., Khalil, E. B., Gleixner, A., Lodi, A., & Pokutta, S. (2021). Learning to schedule heuristics in branch-and-bound.

Cornuejols, G., Sridharan, R., & Thizy, J. (1991). A comparison of heuristics and relaxations for the capacitated plant location problem. *European Journal of Operational Research*, *50*, 280–297. https://doi.org/10.1016/0377-2217(91)90261-S

Etheve, M., Alès, Z., Bissuel, C., Juan, O., & Kedad-Sidhoum, S. (2020). Reinforcement learning for variable selection in a branch and bound algorithm. In *Lecture notes in computer science* (pp. 176–185). Springer International Publishing. https://doi.org/10.1007/978-3-030-58942-4_12

Fischetti, M., & Monaci, M. (2012). Branching on nonchimerical fractionalities. *Operations Research Letters*, *40*, 159–164. https://doi.org/10.1016/j.orl.2012.01.008

Fukunaga, A. S. (2011). A branch-and-bound algorithm for hard multiple knapsack problems. *Annals of Operations Research*, *184*, 97–119. https://doi.org/10.1007/s10479-009-0660-y

Gasse, M., Chételat, D., Ferroni, N., Charlin, L., & Lodi, A. (2019). Exact combinatorial optimization with graph convolutional neural networks. *33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*, 15580–15592. https://doi.org/10.5555/3454287.3455683

Geurts, P., Ernst, D., & Wehenkel, L. (2006). Extremely randomized trees. *Machine Learning*, *63*, 3–42. https://doi.org/10.1007/s10994-006-6226-1

Gupta, P., Gasse, M., Khalil, E. B., Kumar, M. P., Lodi, A., & Bengio, Y. (2020). Hybrid models for learning to branch.

He, H., III, H. D., & Eisner, J. M. (2014). Learning to search in branch and bound algorithms. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, & K. Q. Weinberger (Eds.). Curran Associates, Inc.

Hochbaum, D. S., & Pathria, A. (1997). Forest harvesting and minimum cuts: A new approach to handling spatial constraints. *Forest Science*, *43*, 544–554. https://doi.org/10.1093/forestscience/43.4.544

Hoffman, K. L., & Padberg, M. (2001). Combinatorial and integer optimization. In S. I. Gass & C. M. Harris (Eds.), *Encyclopedia of operations research and management science* (pp. 94–102). Springer US. https://doi.org/10.1007/1-4020-0611-X_129

Holmberg, K., Rönnqvist, M., & Yuan, D. (1999). An exact algorithm for the capacitated facility location problems with single sourcing. *European Journal of Operational Research*, *113*, 544–559. https://doi.org/10.1016/S0377-2217(98)00008-3

Howard, R. A. (1960). *Dynamic programming and markov processes*. The M.I.T. Press.

Khalil, E. B., Bodic, P. L., Song, L., Nemhauser, G., & Dilkina, B. (2016). Learning to branch in mixed integer programming. *Proceedings of the AAAI Conference on Artificial Intelligence*, *30*. https://doi.org/10.1609/aaai.v30i1.10080

Khalil, E. B., Dai, H., Zhang, Y., Dilkina, B., & Song, L. (2017). Learning combinatorial optimization algorithms over graphs. http://arxiv.org/abs/1704.01665

Koch, T., Berthold, T., Pedersen, J., & Vanaret, C. (2022). Progress in mathematical programming solvers from 2001 to 2020. https://doi.org/10.1016/j.ejco.2022.100031

Labassi, A. G., Chételat, D., & Lodi, A. (2022). Learning to compare nodes in branch and bound with graph neural networks.

Land, A. H., & Doig, A. G. (1960). An automatic method of solving discrete programming problems. *Econometrica*, *28*, 497. https://doi.org/10.2307/1910129

Maher, S., Miltenberger, M., Pedroso, J. P., Rehfeldt, D., Schwarz, R., & Serrano, F. (2016). Pyscipopt: Mathematical programming in python with the scip optimization suite. https://doi.org/10.1007/978-3-319-42432-3_37

Mattick, A., & Mutschler, C. (2023). Reinforcement learning for node selection in branch-and-bound.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., & Kavukcuoglu, K. (2016, July). Asynchronous methods for deep reinforcement learning. In M. F. Balcan & K. Q. Weinberger (Eds.). PMLR. https://proceedings.mlr.press/v48/mniha16.html

Paschos, V. T. (2014). *Applications of combinatorial optimization*. John Wiley & Sons.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library.

Sabharwal, A., Samulowitz, H., & Reddy, C. (2012). Guiding combinatorial optimization with uct. https://doi.org/10.1007/978-3-642-29828-8_23

Scavuzzo, L. (2020). Learning variable selection rules for the branch-and-bound algorithm using reinforcement learning.

Scavuzzo, L., Chen, F., Chetelat, D., Gasse, M., Lodi, A., Yorke-Smith, N., & Aardal, K. (2022). Learning to branch with tree mdps. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, & A. Oh (Eds.). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2022/file/756d74cd58592849c904421e3b2ec7a4-Paper-Conference.pdf

Song, J., Lanka, R., Zhao, A., Bhatnagar, A., Yue, Y., & Ono, M. (2018). Learning to search via retrospective imitation.

Sutton, R. S., & Barto, A. G. (2018, November). *Reinforcement learning: An introduction* (Second Edition). The M.I.T. Press.

Vinyals, O., Fortunato, M., & Jaitly, N. (2015). Pointer networks.

Yilmaz, K., & Yorke-Smith, N. (2021). A study of learning search approximation in mixed integer branch and bound: Node selection in scip. *AI*, *2*, 150–178. https://doi.org/10.3390/ai2020010

Zarpellon, G., Jo, J., Lodi, A., & Bengio, Y. (2020). Parameterizing branch-and-bound search trees to learn branching policies.