Using Reinforcement Learning for Variable Selection in the Branch and Bound Algorithm for Solving Mixed Integer Linear Programs

Shreyas Dutt Mishra

Primary Advisor: Alice Paul (School of Public Health)
CS Advisor: Serdar Kadioglu (Department of Computer Science)
Reader: Robert Y. Lewis (Department of Computer Science)

Thesis
Submitted in Partial Fulfillment of the Honors Requirements for the Computer Science Department.
Acknowledgements

I would like to extend my deepest appreciation to Professor Alice Paul for her exceptional mentorship and consistent support throughout the academic semester. Her insights and encouragement were instrumental in helping me navigate the complexities of the thesis.

Additionally, I am profoundly grateful to Professor Serdar Kadioglu, whose passion for teaching and expertise in prescriptive analytics transformed my view of combinatorial optimization. His compelling lectures and thoughtful guidance sparked my curiosity and inspired me to delve deeper into this field, ultimately leading me to choose it as the topic for my thesis.

I also express my gratitude to Professor Robert Lewis for his willingness to serve as my thesis reader. His course on formal proof and verification has opened new horizons in my academic journey, introducing me to a novel way of problem-solving and has profoundly shaped my approach to analytical thinking.

Finally, I would like to thank my family and friends for their unwavering support and encouragement throughout this journey. Your constant presence and belief in my abilities have been a source of strength and motivation.

Each of you has contributed significantly to my achievements, and I am thankful for the wisdom, encouragement, and love you have shared with me.
Abstract

Mixed Integer Linear Programs (MILPs) are a class of optimization problems that involve both continuous and discrete decision variables, linear constraints, and a linear objective function. Solving MILPs is a critical task in various domains, including supply chain optimization, resource allocation, and scheduling. The Branch and Bound (B&B) algorithm is a fundamental method for solving MILPs, but its efficiency heavily depends on the variable selection strategy employed at each branching step. Traditional approaches, such as Strong Branching and Pseudocost Branching, often struggle to balance the trade-off between computational efficiency and the quality of branching decisions.

This thesis proposes a novel approach to variable selection in B&B algorithms using Reinforcement Learning (RL). We introduce a dual-network architecture, consisting of a policy network and a target network, to learn effective branching policies through interaction with the B&B search process. The RL agent iteratively refines its decision-making strategy by considering the current problem state, available actions, and the potential impact of each branching decision on the optimality gap. The learning process is guided by a reward function that encourages the agent to prioritize branching decisions that lead to significant improvements in the bounds or the discovery of high-quality integer solutions.

We evaluate the performance of our RL-based approach on two challenging combinatorial optimization problems: the Set Cover Problem (SCP) and the Max-Cut Problem (MCP). The Set Cover Problem involves selecting a minimum-cost subset of sets that covers all elements in a given universe, while the Max-Cut Problem seeks to partition the vertices of a weighted graph into two disjoint subsets such that the total weight of edges crossing the partition is maximized. Both problems are formulated as MILPs and serve as suitable testbeds for assessing the effectiveness of our variable selection strategy.

Experimental results on a diverse set of SCP and MCP instances demonstrate that our
RL-based approach consistently outperforms traditional branching heuristics in terms of both solution quality and computational efficiency. By learning problem-specific branching policies, the RL agent is able to guide the B&B search process towards promising regions of the solution space, leading to faster convergence and improved optimality gaps.

Our approach distinguishes itself by demonstrating robust performance independent of the size of the problem instances. Unlike traditional methods, which may scale poorly with increasing problem complexity, our RL-based variable selection strategy maintains consistent efficiency across various scales. Furthermore, the general applicability of this method is underscored by its adaptability to different MILP formulations, showing that the learned branching policies are not tailored to specific problem structures but are broadly effective. This generalizability is critical, as it confirms that our approach can be applied to an expansive array of MILP problems.
### Contents

1 Introduction 1

2 Literature Review 3

3 Background 6
   3.1 Mixed Integer Linear Programming (MILP) 6
   3.2 Branch and Bound 8
   3.3 Traditional Methods 11
   3.4 Q-Learning 12

4 Reinforcement Learning for Variable Selection 13
   4.1 Dual-Network Architecture 13
      4.1.1 Policy Network 13
      4.1.2 Target Network 14
   4.2 Experience Replay Memory 14
   4.3 Learning Mechanism 14
   4.4 State Space Representation 15
   4.5 Reward Function 16
   4.6 Training and Prediction 17
   4.7 Pseudocode 20

5 Experimental Setup 21
   5.1 The Set Cover Problem 21
   5.2 The Max Cut Problem 23

6 Results 27
6.1 The Set Cover Problem ........................................... 27
6.2 The Max Cut Problem ........................................... 32
1 Introduction

The field of combinatorial optimization is pivotal in solving many real-world problems such as supply chain optimization [Jokinen et al., 2015], resource allocation [Liu and Fan, 2018], scheduling [Caumond et al., 2009], etc. The canonical method for solving MILPs is the branch and bound (B&B) algorithm [Land and Doig, 1960] which systematically explores solution spaces, breaking down a complex problem into more manageable sub-problems. However, the algorithm's efficiency is heavily reliant on its heuristic components, particularly variable selection [Achterberg and Wunderling, 2013]. Cutting planes are another powerful method in solving MILPs, where additional constraints derived from the linear programming relaxation, designed to exclude non-integer solutions without excluding any integer feasible solutions, thereby tightening the feasible region and improving the solution process. This thesis aims to introduce an innovative approach to variable selection in B&B algorithms using reinforcement learning (RL), enhancing its decision-making efficacy and solving speed.

MILPs are renowned for being an NP-hard problem [Karp, 1972], which means that there is no algorithm that can solve all instances of MILP problems efficiently in polynomial time. This complexity arises because they involve both continuous and discrete decision variables, combined with linear constraints and an objective function that needs to be optimized. As a result, solving them often requires sophisticated heuristic methods or algorithms that can provide good solutions within a reasonable amount of time, rather than guaranteed optimal solutions. This lack of a universally ‘best’ algorithm for all problem classes makes MILP a particularly important area of study in the field of optimization.

RL emerges as a promising approach here as its strength lies in its ability to learn optimal strategies through interactions with the environment, adapting its actions based on the feedback received in the form of rewards [Sutton and Barto, 1998]. This adaptive
learning process is particularly beneficial as it allows us to discern and refine effective strategies for variable selection tailored to the unique characteristics of each problem instance. Unlike traditional algorithms that rely on predefined rules, RL can continuously improve its decision-making process, potentially uncovering innovative strategies that outperform existing heuristics [Silver et al., 2017].

The remainder of this thesis is organized as follows: Section 2 focuses on Section 3 delves into the foundational concepts necessary for understanding the core methodologies discussed in this thesis. It includes a detailed description of Mixed Integer Linear Programming (MILP), the B&B algorithm, and traditional methods of variable selection such as Pseudocost and Strong Branching. Additionally, this section explores the theoretical underpinnings of Q-learning, a form of reinforcement learning (RL), which is pivotal to our proposed method. Following this, Section 4 introduces our novel reinforcement learning framework tailored for variable selection within the B&B algorithm. We detail our dual-network architecture, consisting of a policy network and a target network, designed to adaptively enhance decision-making and improve computational efficiency in solving MILPs.

Section 5 describes the experimental framework used to validate the effectiveness of the RL-based variable selection strategy. It includes a discussion of the problem instances selected for testing—specifically the Set Cover Problem (SCP) and the Max-Cut Problem (MCP), detailing the rationale behind their selection and the specifics of the computational setup. Section 6 presents the findings of our experiments, comparing the performance of our RL-based approach against traditional variable selection methods across various instances of the SCP and MCP. This section evaluates the impact of our approach on the quality of solutions and the computational resources required, highlighting significant improvements over existing techniques.
2 Literature Review

The advancement of computational techniques for solving MILPs has been a significant research focus in optimization and operations research. Among these techniques, B&B algorithms stand out due to their flexibility and effectiveness in solving a wide range of MILPs. However, the efficiency of B&B algorithms heavily relies on the variable selection (branching) strategy employed, traditionally dominated by heuristic methods like Strong Branching (SB) and Pseudocost (PC) branching. Recent years have seen an increasing interest in leveraging Machine Learning (ML) and Reinforcement Learning (RL) to refine the branching decision process, aiming to overcome the limitations of heuristic approaches. Khalil et al. [2016] introduced an ML framework for variable selection in B&B algorithms, addressing the computational drawbacks of SB and the inefficiency of PC branching. By learning an instance-specific branching strategy through observation and mimicking of SB decisions, their ML-based approach significantly reduced search tree sizes and demonstrated competitive performance against default strategies like CPLEX’s.

Building upon the foundation laid by ML-based approaches, Etheve et al. [2020] advanced the field further with FMSTS, an RL methodology specifically tailored for optimizing the branching strategy in B&B algorithms. By learning from patterns in real-world MILP instances, FMSTS not only surpassed the efficiency of Strong Branching but also showed comparable or superior results to CPLEX, marking the first successful application of RL in fully optimizing branching strategies. The consistency of FMSTS’s local and global optimization metrics highlights the advantages of RL in discovering nuanced, problem-specific branching strategies.

The notion of tailoring algorithm strategies to specific instances has shown its value across various domains, exemplified by the Instance-Specific Algorithm Configuration (ISAC) proposed by Kadioglu et al. [2010]. ISAC is a general configurator that can tune any solver
and choose solver parameters according to the instance to be solved. The approach involves clustering the training instances using normalized features and the g-means algorithm, and then using an instance-oblivious algorithm configurator to find the best parameters for each cluster. At runtime, ISAC determines the cluster closest to the input instance feature vector and solves the instance with the parameters for the respective cluster. In parallel, Kadioglu et al. [2011] have developed a hybrid approach that enhances the performance of SAT solvers by integrating algorithm selection with scheduling. This method, which uses a nearest-neighbor based selector coupled with a lightweight optimization algorithm, divides the solving time between the selected solver and a predefined schedule. Their approach consistently outperformed traditional methods, successfully solving over 95% of instances—a result comparable to the best virtual solver. Although this research primarily addresses SAT problems, the underlying strategies could be adapted to optimize other types of optimization problems, including MILPs. This insight informs our current work, where we apply a similar strategy to adaptively refine variable selection policies for Branch and Bound algorithms through reinforcement learning.

Kadioglu et al. [2012] introduced a dynamic branching scheme for set partitioning problems called Trace. Trace traces features of the underlying MIP model and bases search decisions on the features of the current subproblem to be solved. The approach involves clustering training instances based on normalized feature vectors and using an offline algorithm tuning tool to learn an assignment of branching heuristics to clusters. During the search, Trace determines the nearest cluster to the current subproblem and uses the associated heuristic for branching. Experiments on a diverse set of set partitioning instances showed that Trace outperformed the Cplex default and the best pure branching heuristic, demonstrating the effectiveness of mixing branching heuristics based on instance-specific characteristics. This approach is similar to the DASH method proposed by Liberto
et al. [2016]. By identifying changes in the problem structure during branch-and-bound search and adaptively selecting the most appropriate heuristic for each sub-problem, DASH achieved significant performance improvements over static heuristic selection strategies. This suggests that the branch-and-bound search can benefit from dynamic adaptation based on the characteristics of the current sub-problem, motivating our investigation of reinforcement learning for instance-specific variable selection.

Tang et al. [2020] explored the dimension of RL application in solving MILPs by focusing on enhancing cutting plane methods, particularly Gomory’s cutting plane method. By treating the selection of cutting planes as a Markov Decision Process (MDP) and utilizing a deep RL architecture, they achieved significant improvements over traditional heuristics in solution accuracy and efficiency. This approach confirmed the potential of RL in handling complex decision-making tasks in optimization and suggested its broader applicability in augmenting other components of the B&B algorithm, such as cutting plane selection.

Parsonson et al. [2023] introduced retrobranching, an approach that tackled the inherent challenges of applying RL to B&B variable selection, such as long episodes and large state-action spaces. By retrospectively deconstructing the search tree into shorter trajectories for the RL agent to learn from, retro branching provided denser rewards and improved state predictability. This method not only outperformed the previous state-of-the-art RL branching method (FMSTS) but also demonstrated the ability to discover superior branching policies without relying on expensive expert data. Their work signifies a significant step forward in the application of RL in B&B optimization, showcasing the method’s capacity to learn effective policies through problem decomposition and reward structuring.

Our proposed method extends the concept of retrobranching [Parsonson et al., 2023] by retrospectively deconstructing the search tree into shorter trajectories, enabling the RL agent to learn from denser rewards and improved state predictability. This approach not
only addresses the challenges associated with long episodes and large state-action spaces in applying RL to B&B algorithms but also allows for the discovery of problem-specific branching strategies without relying on expensive expert data.

While our approach draws inspiration from the concept of retrobranching, it differs in the way trajectories are constructed and utilized for learning. In retrobranching, the search tree is retrospectively deconstructed into shorter trajectories by considering each node as a potential starting point and its descendants as the trajectory. This process results in a large number of overlapping trajectories, which are then used to train the RL agent. In contrast, our method generates trajectories by traversing the search tree from the root node to each leaf node, creating a set of unique, non-overlapping trajectories. This approach offers several advantages. First, it reduces the computational overhead associated with processing a large number of overlapping trajectories. Second, it provides a more concise and coherent representation of the search process, as each trajectory represents a complete path from the root to a leaf node. Third, by assigning rewards based on the improvement in the optimality gap at each step along the trajectory, our method offers a more granular and informative feedback signal for the RL agent. This trajectory construction approach, coupled with the incorporation of problem-specific statistics into the state representation, enables our method to learn effective branching strategies that are tailored to the characteristics of the problem being solved.

3 Background

3.1 Mixed Integer Linear Programming (MILP)

MILPs are a class of optimization problems that involve both integer and continuous variables. It is a powerful tool in operational research and decision-making processes because it
provides a framework for modeling complex systems with linear relationships. These problems are known for their computational complexity, especially as the size of the problem (number of variables and constraints) increases.

A MILP problem can be formally defined by the following minimization statement:

\[
\begin{align*}
\text{minimize} & \quad c^T x \\
\text{subject to} & \quad Ax \leq b, \\
& \quad x_j \in \mathbb{Z} \forall j \in J, \\
& \quad x \geq 0
\end{align*}
\]

(1)

where:

- \( x = (x_1, x_2, \ldots, x_n)^T \) is the vector of decision variables, which can be either integer-valued or real-valued.
- \( c = (c_1, c_2, \ldots, c_n)^T \) is the coefficient vector for the objective function.
- \( A \) is a matrix of coefficients for the constraints with dimensions \( m \times n \), where \( m \) is the number of constraints and \( n \) is the number of decision variables.
- \( b = (b_1, b_2, \ldots, b_m)^T \) is the right-hand side vector of the constraints, representing the limits or requirements that must be satisfied. Here, \( m \) is the number of constraints.
- \( J \) is the set of indices for which the decision variables \( x_j \) are required to be integer, i.e., \( J \subseteq \{1, 2, \ldots, n\} \).
- The objective is to minimize the linear function \( c^T x \), subject to the linear constraints \( Ax \leq b \), with some variables constrained to be integers.
3.2 Branch and Bound

The Branch and Bound (B&B) algorithm is a well-established method for solving MILPs, integrating multiple heuristic strategies to efficiently navigate the solution space. At its core, B&B constructs a search tree, with each node representing a subproblem of the original MILP. These subproblems are defined by additional constraints, forming a hierarchical partitioning of the problem space. The edges of the tree represent these partitioning decisions, which divide the MILP into progressively smaller, more manageable subproblems. This iterative division is aimed at narrowing down the search for an optimal solution or, if the algorithm terminates prematurely, securing a solution within a predefined optimality gap. The essence of B&B lies in its ability to evolve this search tree towards the identification of an optimally feasible node through a divide-and-conquer approach.

- **Node Selection:** The process begins by selecting an open node that has not been fully explored and appears to be a promising candidate for further expansion within the tree. This node is the focus node, with potential subproblems to be examined.

- **Branching:** B&B then selects a variable within the focus node’s MILP and imposes further constraints based on the variable’s value in the linear programming (LP) solution. This action bifurcates the focus node into two child nodes, each representing a sub-MILP with a more constrained solution space.

- **Dual Bound Calculation:** For each child node, the algorithm solves a relaxed version of the MILP (often the dual problem), establishing a dual bound. This bound represents the best possible objective value achievable within the subtree originating from the node.

- **Primal Bound Establishment:** Simultaneously, B&B attempts to solve the primal problem to find a feasible solution that adheres to the node’s constraints. This
solution establishes the primal bound, signifying the worst-case feasible objective value within the subtree.

- **Fathoming**: Nodes are fathomed and excluded from further exploration if their relaxed LP solutions are infeasible, inferior to the current best solution (the incumbent), or fail to satisfy the MILP’s integrality requirements.

- **Termination**: The algorithm continues this process until the primal-dual gap closes, indicating that a provably optimal solution to the MILP has been found.

The efficacy of the B&B algorithm is significantly influenced by the heuristics employed at each stage, including the strategies for primal solution, branching decisions, and node selection. While advanced methodologies such as cutting planes and column generation can augment the algorithm’s performance, these enhancements are considered supplementary to the fundamental B&B process. In practical applications, solver implementations like SCIP 2022 manage memory by retaining only nodes eligible for further exploration, thereby streamlining the fathoming process to exclude nodes whose subsequent branches fall outside the predetermined bounds of optimality, are infeasible, or possess an integer-feasible dual solution, effectively sealing off those subtrees from additional scrutiny.
Algorithm 1 Branch and Bound, Part 1: Initialization and Initial Bounds Calculation

0: **Input:** Problem instance
0: **Output:** Best solution found, optimality gap
0: **procedure** INITIALIZE_BRANCH_AND_BOUND(problem)
0: Initialize root node with the problem instance
0: Calculate initial bounds:
0: \( LB \leftarrow \text{Calculate lower bound (using linear relaxation of the MILP)} \)
0: \( UB \leftarrow \text{Estimate upper bound (from a heuristic or previous solution)} \)
0: Calculate initial optimality gap:
0: \( \text{Opt Gap} \leftarrow \frac{UB - LB}{UB} \times 100 \), ensuring \( UB \neq 0 \)
0: return initial bounds \( LB, UB \) and \( \text{Opt Gap} \)
0: **end procedure**

Algorithm 2 Branch and Bound, Part 2: Main Loop and Dynamic Bound Updates

0: **procedure** BRANCH_AND_BOUND_MAIN_LOOP(LB, UB, Opt_Gap)
0: **while** active nodes exist and Opt_Gap > tolerance **do**
0: \( node \leftarrow \text{SelectNodeForBranching()} \)
0: \( childNodes \leftarrow \text{BranchNode}(node) \)
0: **for** child in childNodes **do**
0: \( \text{Solve relaxed problem at child} \)
0: \( LB_{child} \leftarrow \text{New lower bound from relaxed problem at child} \)
0: If \( LB_{child} > LB \), set \( LB \leftarrow LB_{child} \)
0: If feasible solution at child improves \( UB \), set \( UB \leftarrow \text{solution value} \)
0: Update optimality gap: \( \text{Opt Gap} \leftarrow \frac{UB - LB}{UB} \times 100 \)
0: **end for**
0: Prune nodes that cannot improve the solution based on \( LB \) and \( UB \)
0: Update global best solution based on \( UB \)
0: **end while**
0: return best solution found, optimality gap
0: **end procedure**
3.3 Traditional Methods

In B&B algorithms, the selection of branching variables is a pivotal decision that influences both the computational efficiency and the effectiveness of the solution process. Two traditional techniques—Pseudocost Branching (PB) and Strong Branching (SB)—offer contrasting approaches to this decision-making process.

Pseudocost Branching (PB) [Benichou et al., 1971] is a heuristic based on historical data regarding the effectiveness of variables in previous branching decisions. This method approximates the future impact of branching on a variable by considering the average effect observed in past problem-solving instances. PB is particularly valued for its computational efficiency, which facilitates quicker variable selection. However, its reliance on historical data can be a drawback during the initial phases of the algorithm, where such data are limited. This lack of initial precision may lead to suboptimal branching choices, potentially increasing the time required to locate the optimal solution Achterberg et al. [2005].

Strong Branching (SB) [Applegate et al., 1995], on the other hand, is known for its accuracy. This technique evaluates the potential impact of each variable by simulating the branching process on that variable and observing the resultant variation in the objective function. Although this method generally yields more precise and effective branching decisions, it is also notably more resource-intensive. Each candidate variable is assessed in depth, requiring substantial computational time and power.

In summary, while pseudocost branching offers speed but lacks early accuracy, strong branching provides precision at the cost of higher computational demands. Balancing these trade-offs is crucial, and innovations in variable selection methods such as the integration of reinforcement learning could offer new solutions to optimize this balance.
3.4 Q-Learning

Q-learning [Watkins and Dayan, 1992] is a value-based RL method aiming to learn the optimal action-value function $Q^*(s, a)$, mapping state-action pairs to the expected rewards. The learning process iterates to improve the Q-function based on observed rewards and estimated future values.

The update rule is as follows:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot (\text{reward} + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a))$$  \hspace{1cm} (2)

where

- **State-action pair** $(s, a)$: The current state $s$ and action $a$ taken.
- **Reward**: The immediate reward received after taking action $a$ in state $s$.
- **Learning rate** $\alpha$: Determines the extent to which the new Q-value replaces the old value.
- **Discount factor** $\gamma$: The weight of future rewards.

This equation adjusts the Q-value for $(s, a)$ based on the observed reward and the estimated future value, facilitating the convergence to the optimal Q-function. Deep Q-Networks (DQNs) extend Q-learning to complex scenarios by approximating the Q-function with a deep neural network. This network predicts Q-values for all possible actions given a state, enabling the agent to select the most advantageous action. DQNs are great at handling large state spaces and learning complex state-action relationships. However, we can also face instability and divergence which can be mitigated by techniques like experience replay, target networks, and gradient clipping.
4 Reinforcement Learning for Variable Selection

We introduce a novel approach utilizing a deep reinforcement learning framework that employs two deep Q-networks to adaptively select variables, aiming to reduce the overall search space and computational time. The agent’s objective is to maximize the cumulative reward over time. This is achieved through a policy, which maps states to actions, and is learned using a value function approximation.

4.1 Dual-Network Architecture

Our framework employs two neural networks, known as the policy network and the target network, both instantiated from the DQN architecture.

4.1.1 Policy Network

The policy network, denoted as $Q(s, a; \theta)$, with parameters $\theta$, is responsible for estimating the value of taking an action $a$ in a state $s$. The network’s architecture is comprised of fully connected layers with the following specifications:

- Input layer dimension equal to the size of the state space.
- First hidden layer with 64 neurons, utilizing the Leaky ReLU activation function.
- Second hidden layer with 16 neurons, also using Leaky ReLU.
- Output layer with a single neuron corresponding to the state-action value estimate.

The forward propagation of the policy network can be formally described as follows:

$$Q(s, a; \theta) = f_3 (f_2 (f_1 (s; \theta_1); \theta_2); \theta_3),$$

(3)
where $f_1$, $f_2$, and $f_3$ represent the transformations performed by the three layers of the network, respectively.

### 4.1.2 Target Network

The target network, denoted as $Q'(s, a; \theta')$ with parameters $\theta'$, has an identical architecture to the policy network. Its role is to provide a stable target for the policy network’s update. Its parameters are periodically updated to match those of the policy network, preventing the moving target problem which can destabilize learning.

### 4.2 Experience Replay Memory

The framework utilizes a replay memory $D$ to store transitions in the form of tuples $(s, a, r, s')$, representing the previous state, action taken, reward received, and subsequent state, respectively. This memory allows the agent to learn from a diverse set of experiences by sampling randomly from this memory, breaking the correlation between consecutive learning samples.

### 4.3 Learning Mechanism

The agent updates its policy based on the Bellman equation. In each iteration, a mini-batch of experiences is sampled from $D$, and the policy network’s parameters are updated to minimize the loss between predicted and target Q-values, defined as:

$$ L(\theta) = E_{(s,a,r,s') \sim U(D)} \left[ \left( r + \gamma \max_{a'} Q'(s', a'; \theta') - Q(s, a; \theta) \right)^2 \right], $$

where $\gamma$ is the discount factor and $U(D)$ denotes a uniform distribution over the replay memory $D$. 

14
The agent follows an $\epsilon$-greedy strategy to balance exploration with exploitation. With probability $\epsilon$, it explores the environment by selecting an action randomly, facilitating discovery of new strategies. With probability $1 - \epsilon$, it exploits its current knowledge by selecting the action with the highest estimated Q-value.

4.4 State Space Representation

The state space within the B&B algorithm encapsulates all relevant information that could impact the agent’s decision-making process. In our implementation, the state comprises a set of numerical values that describe the current status of the problem, the node within the tree, and specific variable-level characteristics.

The node-level statistics are focused on the local properties of a single node within the tree, detailing the depth level of the node, the bounds on the solution space defined by this node, and a binary indicator of whether this node holds the current best solution. The variable-level statistics at the node include primal beta values and the fractionality of the variables, which are pivotal for deciding the next branching variable. The tree-level statistics capture the dynamics of the exploration process, including the number of steps taken, active nodes, candidate solution quality, lower bound on the objective value, and the current optimality gap.

For the Set Cover Problem, the problem-level statistics include the number of sets and universe items, alongside other pertinent metrics such as the set costs and coverages. At the variable level, each set is described through its cost, its coverage of the universe items, and its average overlap with other sets, quantified by the Jaccard index.

For the Max Cut Problem, the problem-level statistics include the total number of nodes and edges, alongside other critical metrics such as the total, average, maximum, and minimum edge weights, as well as the graph’s density. Additional statistics include
the maximum and minimum node degrees and the average clustering coefficient. At the
variable level, each node is characterized by its degree, the presence of self-loops, and its
clustering coefficient.

Table 1: Statistics for the State Space

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree-level statistics</td>
<td>1. Number of steps taken</td>
</tr>
<tr>
<td></td>
<td>2. Number of active nodes</td>
</tr>
<tr>
<td></td>
<td>3. Number of candidate solution variables</td>
</tr>
<tr>
<td></td>
<td>4. Lower bound</td>
</tr>
<tr>
<td></td>
<td>5. Best integer solution</td>
</tr>
<tr>
<td></td>
<td>6. Initial optimality gap</td>
</tr>
<tr>
<td></td>
<td>7. Current optimality gap</td>
</tr>
<tr>
<td>Node-level statistics for the selected</td>
<td>8. Length of $z_{ub}$ (upper bound set)</td>
</tr>
<tr>
<td>node</td>
<td>9. Length of $z_{lb}$ (lower bound set)</td>
</tr>
<tr>
<td></td>
<td>10. Node-specific primal value</td>
</tr>
<tr>
<td></td>
<td>11. Node depth level</td>
</tr>
<tr>
<td></td>
<td>12. Length of support set</td>
</tr>
<tr>
<td></td>
<td>13. Indicator if the node has the current lower bound</td>
</tr>
<tr>
<td></td>
<td>14. Indicator if the node has the current upper bound</td>
</tr>
<tr>
<td>Variable-level statistics for the selected</td>
<td>15. Primal $\beta$ value</td>
</tr>
<tr>
<td>variable $j$ in the selected node</td>
<td>16. $z$ value</td>
</tr>
</tbody>
</table>

4.5 Reward Function

The reward function plays a crucial role in reinforcement learning, guiding the agent to-
wards desirable behavior. In the context of B&B, the rewards are assigned based on the
change in the optimality gap that results from branching on a variable. A reward structure
is proposed where rewards are categorized based on the reduction of the optimality gap.
The purpose of this reward function is to encourage the agent to explore branches that
lead to significant reductions in the optimality gap, thereby guiding it towards finding the optimal solution more efficiently. A larger reduction results in a higher reward, incentivizing the agent to choose branches that are likely to lead to significant improvements in the bound or the discovery of good integer solutions. Specifically:

- A reward of 1 is given when the optimality gap is reduced by a small amount \((0 < \text{opt\_gap} \leq 0.05)\).
- A reward of 2 is given when the optimality gap is reduced by a moderate amount \((0.05 < \text{opt\_gap} \leq 0.15)\).
- A reward of 3 is given when the optimality gap is reduced by a significant amount \((\text{opt\_gap} > 0.15)\).
- A reward of -1 is given when the optimality gap is not positive, indicating that the branch did not lead to an improvement.

The rewards are designed to balance the exploration of the solution space with the exploitation of promising branches, thereby facilitating the learning process towards an efficient policy for variable selection.

4.6 Training and Prediction

During training, the RL algorithm operates as follows:

1. The agent interacts with the B&B environment by selecting actions based on the current state. It uses an epsilon-greedy strategy, where it either explores by taking random actions or exploits by choosing the action with the highest estimated Q-value. In this case, an action corresponds to selecting a node and a variable to branch on. The agent learns to choose the most promising action based on the estimated Q-values.
2. The selected action is applied to the B&B algorithm, resulting in a new state and a reward. The state transitions (previous state, action, new state, reward) are stored in the replay memory.

3. After collecting a batch of transitions, the agent performs experience replay. It samples a mini-batch of transitions from the replay memory and uses them to update the parameters of the DQN. The Q-learning update rule is applied, which aims to minimize the difference between the predicted Q-values and the target Q-values.

4. The target Q-values are computed using a separate target network, which is periodically synchronized with the main DQN. This stabilizes the learning process by providing a fixed target for the Q-value updates.

5. The training process continues iteratively, with the agent collecting new experiences and updating the DQN parameters until a satisfactory level of performance is achieved.

The predictions are made in the following manner post training:

1. The state vector is fed into the DQN to estimate the Q-values for each state-action pair. These Q-values represent the estimated future rewards for taking each action in the given state.

2. Based on the predicted Q-values, the RL algorithm selects the most promising action to take. This is done using the epsilon-greedy strategy, where the algorithm either explores by choosing a random action with probability epsilon or exploits by selecting the action with the highest estimated Q-value.

3. The selected action corresponds to a specific node and variable to branch on in the B&B algorithm. The RL algorithm communicates this decision to the B&B solver,
which then proceeds to create the corresponding subproblems and update the bounds accordingly.

4. After taking the selected action and observing the resulting state of the B&B algorithm, a reward is calculated based on the improvement in the optimality gap. The reward function is designed to provide positive feedback for actions that lead to a significant reduction in the optimality gap and negative feedback for actions that result in minimal or no improvement.

5. The above steps are repeated iteratively throughout the branch-and-bound search process. At each step, the RL algorithm uses the predictions made by the DQN to make informed decisions, selecting actions that are likely to lead to the optimal solution efficiently.
4.7 Pseudocode

**Algorithm 3 Complete Algorithm for a Single Episode, Part 1: Initialization and Main Loop**

0: Initialize episode parameters: $X, y, \epsilon, \text{Threshold}$
0: Solve the LP relaxation of the MILP to calculate initial Opt_GAP
0: Initialize Q-network $Q$ with weights $\theta$ and target network $\hat{Q}$ with weights $\theta'$
0: Initialize replay memory $D$ with capacity $N$
0: while Opt_GAP > Threshold and not MaxIterations do
  0: $k \leftarrow \text{Uniform}(0, 1)$
  0: if $k \leq \epsilon$ then
    0: Select a branching node and variable (node, $x_j$) via heuristic (e.g., max fraction branching)
  0: else
    0: Initialize $u_{\text{max}} \leftarrow -\infty$
    0: for each available (node, $x_j$) pair do
      0: $s \leftarrow \text{encode state for (node, } x_j\text{)}$
      0: $u \leftarrow Q(s|\theta)$
      0: if $u > u_{\text{max}}$ then
        0: $u_{\text{max}} \leftarrow u$
        0: Select this (node, $x_j$)
      0: end if
    0: end for
  0: end if
  0: Perform branching on the selected (node, $x_j$); solve relaxation; update Opt_GAP
  0: Prune nodes based on bounding conditions
0: end while=0
Algorithm 4 Complete Algorithm for a Single Episode, Part 2: Training and Output

0: if training then
0: Apply \( \epsilon \) decay according to a schedule
0: Perform retrobranching to populate \( D \) with transition tuples \((s_t, a, r, s_{t+1})\)
0: Sample a mini-batch from \( D \)
0: Optimize \( \theta \) by minimizing the loss: 
\[
\sum \left( r + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'|\theta') - Q(s_t, a|\theta) \right)^2
\]
0: Periodically update target network: \( \theta' \leftarrow \theta \)
0: end if
0: return solution coefficients \( \beta \), Opt_Gap = 0

5 Experimental Setup

5.1 The Set Cover Problem

The Set Cover Problem (SCP) is a fundamental NP-hard optimization [Alon et al., 2003] problem with wide-ranging applications in various domains such as resource allocation [Chang et al., 2010] and scheduling [Mesquita and Paias, 2008]. It is a combinatorial problem that involves selecting a minimum number of sets from a given collection of sets, such that their union covers all elements of a given universe.

Formally, the SCP can be defined as follows: Given a universe \( U = 1, 2, \ldots, m \) of \( m \) elements and a collection of \( n \) sets \( S = S_1, S_2, \ldots, S_n \), where each set \( S_i \) has an associated cost \( c_i \) and covers a subset of elements from \( U \), the objective is to find a subset \( C \subseteq S \) such that every element in \( U \) is covered by at least one set in \( C \) and the total cost of sets in \( C \) is minimized.

The SCP can be formulated as an Integer Linear Program (ILP) as follows:
minimize \[ \sum_{i=1}^{n} c_i x_i \] 

subject to \[ \sum_{i:j \in S_i} x_i \geq 1, \quad \forall j \in U, \quad (5) \]

\[ x_i \in \{0, 1\}, \quad \forall i \in \{1, 2, \ldots, n\} \]

where \(x_i\) is a binary decision variable that takes the value 1 if set \(S_i\) is included in the cover and 0 otherwise. The first constraint ensures that each element in the universe is covered by at least one selected set, while the second constraint enforces the binary nature of the decision variables.

The SCP has numerous practical applications. For instance, in wireless sensor networks, the SCP can model the problem of selecting a minimum number of sensors to cover a given area while minimizing the overall energy consumption [Fan and Jin, 2010]. In facility location problems, it can represent the task of choosing the optimal locations for service facilities to cover all customer demand points at the lowest cost.

Due to its NP-hardness, solving large instances of the SCP optimally is computationally challenging. Simple B&B can solve small to medium-sized instances to optimality but may struggle with larger instances. Therefore, heuristic and approximation algorithms are often employed to obtain good-quality solutions within reasonable time limits.

Greedy algorithms are among the most popular heuristics for the SCP [Paschos, 1997]. They iteratively select the set that covers the maximum number of uncovered elements per unit cost until all elements are covered. Although greedy algorithms provide an approximation ratio of \(\ln n\) (where \(n\) is number of items in set) [Feige, 1998] for the SCP, they can be far from optimal in practice. More sophisticated approaches like Lagrangian relaxation, column generation, and meta-heuristics (e.g., genetic algorithms, simulated annealing) have been proposed to tackle larger SCP instances [Caprara et al., 2000]. These methods ex-
ploit the problem structure and incorporate problem-specific strategies to improve solution quality and computational efficiency.

In the context of this thesis, the SCP serves as a suitable testbed for evaluating the effectiveness of RL-based variable selection strategies within the B&B framework. By focusing on this well-known combinatorial optimization problem, we can benchmark the performance of our proposed approach against traditional branching heuristics and assess its potential for solving challenging real-world optimization problems.

We use two distinct problem sets, differing primarily in their constraint-to-variable ratios. The first problem set comprises set cover problems with 50 constraints and 100 variables, while the second consists of problems with 100 constraints and 100 variables. Each problem set contains 60 training instances and 20 test instances. The problem sets were generated through the Ecole [Prouvost et al., 2020] library in python.

We use the SCIP Optimization Suite Bolusani et al. [2024], an integrated framework for solving the LP relaxation. Presolving and Heuristics were disabled to isolate the effect of LP rounding from heuristic influences.

5.2 The Max Cut Problem

The Max-Cut Problem (MCP) is a well-known NP-hard combinatorial optimization problem [Ageev et al., 2014] with applications in various fields such as VLSI design [Barahona et al., 1988], statistical physics, and network science [Poljak and Tuza, 1995]. The problem involves partitioning the vertices of a weighted undirected graph into two disjoint subsets such that the total weight of edges connecting vertices in different subsets, known as the cut size, is maximized.

Formally, the MCP can be defined as follows: Given a weighted undirected graph
Table 2: SCP - Problem Level Statistics for the State Space

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem-level statistics</td>
<td>1. Number of Sets</td>
</tr>
<tr>
<td></td>
<td>2. Number of Items in Universe</td>
</tr>
<tr>
<td></td>
<td>3. Average Set Coverage</td>
</tr>
<tr>
<td></td>
<td>4. Average Item Coverage</td>
</tr>
<tr>
<td></td>
<td>5. Density</td>
</tr>
<tr>
<td>Variable-level statistics for the selected variable j</td>
<td>6. Cost of including set j in the cover.</td>
</tr>
<tr>
<td></td>
<td>7. Number of universe items that set j covers.</td>
</tr>
<tr>
<td></td>
<td>8. Maximum number of sets that cover any single item within set j.</td>
</tr>
<tr>
<td></td>
<td>9. Average Overlap with Other Sets</td>
</tr>
<tr>
<td></td>
<td>10. Average Jaccard index of set j with every other set.</td>
</tr>
</tbody>
</table>

\[ G = (V, E) \] with vertex set \( V = 1, 2, \ldots, n \) and edge set \( E \subseteq V \times V \), where each edge \((i, j) \in E\) has an associated weight \( w_{ij} \), the objective is to find a partition of \( V \) into two disjoint subsets \( S \) and \( V \setminus S \) such that the sum of weights of edges crossing the cut, i.e., \( \sum_{(i,j) \in E} w_{ij}(1 - y_{ij}) \), is maximized.

The MCP can be formulated as an Integer Linear Program as follows:

\[
\begin{align*}
\text{maximize} & \quad \frac{1}{2} \sum_{(i,j) \in E} w_{ij}(1 - y_{ij}) \\
\text{subject to} & \quad y_{ij} \leq x_i + x_j, \quad \forall (i, j) \in E, \\
& \quad y_{ij} \leq 2 - x_i - x_j, \quad \forall (i, j) \in E, \\
& \quad y_{ij} \geq 0, \quad \forall (i, j) \in E, \\
& \quad x_i \in \{0, 1\}, \quad \forall i \in V.
\end{align*}
\]

(6)

In this formulation:

- \( x_i \) is a binary decision variable that takes the value 1 if vertex \( i \) is assigned to the
first subset and 0 if it is assigned to the second subset.

- $y_{ij}$ is introduced to linearize the objective function. It serves as an auxiliary variable to represent whether an edge $(i, j)$ is part of the cut or not. In the objective function, $y_{ij}$ appears as a coefficient for the weight $w_{ij}$ of each edge $(i, j)$. The term $w_{ij}(1 - y_{ij})$ represents the weight of edge $(i, j)$ if it is part of the cut ($y_{ij} = 0$), and 0 if it is not part of the cut ($y_{ij} = 1$).

- The objective function maximizes the cut size by summing the weights of edges $(i, j)$ for which $y_{ij} = 0$, i.e., the endpoints of the edge are in different subsets.

- The first two sets of constraints ensure that $y_{ij} = 1$ if and only if both $x_i$ and $x_j$ are equal to 1, i.e., both vertices are in the same subset. If either $x_i$ or $x_j$ is 0, then $y_{ij}$ can take any value between 0 and 1.

- The third set of constraints ensures that $y_{ij}$ is non-negative.

- The last set of constraints enforces the binary nature of the decision variables $x_i$.

The MCP has diverse applications in various domains. In VLSI design, it can model the problem of partitioning electronic components onto a printed circuit board to minimize the number of connections between the two sides. In statistical physics, the MCP is closely related to the study of spin glasses and the Ising model, where the goal is to find the ground state configuration of a system of interacting spins. In network science, the MCP can be used to identify communities or clusters in social networks, protein-protein interaction networks, and other complex networks.

Solving the MCP to optimality is computationally challenging due to its NP-hardness. Exact algorithms based on branch-and-bound, semidefinite programming (SDP), and polyhedral approaches have been proposed for solving small to medium-sized instances. How-
ever, these methods often struggle with large instances encountered in real-world applications.

Approximation algorithms and heuristics are commonly employed to obtain good-quality solutions for the MCP within reasonable time limits. The Goemans-Williamson algorithm, based on SDP relaxation and randomized rounding, provides an approximation ratio of 0.878 for the MCP [Goemans and Williamson, 1995], which is the best-known approximation guarantee for this problem. Other heuristic approaches include local search methods, greedy algorithms, and meta-heuristics such as simulated annealing and genetic algorithms.

In the context of this thesis, the MCP serves as another suitable problem for evaluating the performance of reinforcement learning-based variable selection strategies within the branch-and-bound framework. By applying our proposed approach to this challenging combinatorial optimization problem, we can assess its effectiveness in guiding the search towards high-quality solutions and compare its performance against traditional branching heuristics and state-of-the-art algorithms.

We use 60 instances of Erdős–Rényi graphs to train the model with 25 and 35 vertices respectively. The model is then tested on 20 instances. The graphs were generated through the NetworkX [Hagberg et al., 2008] library. As stated earlier, we use the SCIP Optimization Suite, an integrated framework for solving MILPs, to solve the LP relaxation. Presolving and Heuristics were disabled to isolate the effect of LP rounding from heuristic influences.
Table 3: MCP - Problem Level Statistics for the State Space

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem-level statistics</td>
<td>1. Number of nodes</td>
</tr>
<tr>
<td></td>
<td>2. Number of edges</td>
</tr>
<tr>
<td></td>
<td>3. Total edge weight</td>
</tr>
<tr>
<td></td>
<td>4. Average edge weight</td>
</tr>
<tr>
<td></td>
<td>5. Maximum edge weight</td>
</tr>
<tr>
<td></td>
<td>6. Minimum edge weight</td>
</tr>
<tr>
<td></td>
<td>7. Graph density</td>
</tr>
<tr>
<td></td>
<td>8. Maximum node degree</td>
</tr>
<tr>
<td></td>
<td>9. Minimum node degree</td>
</tr>
<tr>
<td></td>
<td>10. Number of isolated nodes</td>
</tr>
<tr>
<td></td>
<td>11. Average clustering coefficient</td>
</tr>
<tr>
<td></td>
<td>12. Edge weight standard deviation</td>
</tr>
<tr>
<td>Variable-level statistics for the selected variable $j$</td>
<td>13. Node degree</td>
</tr>
<tr>
<td></td>
<td>14. Self-loop weight (if applicable)</td>
</tr>
<tr>
<td></td>
<td>15. Clustering coefficient</td>
</tr>
</tbody>
</table>

6 Results

All the following graphs show a comparison of the number of iterations taken to reach convergence. In the graphs that show training iterations, we take the rolling mean with a window size of 20 to get a smoother graph as not all instances are of the same difficulty. Similarly, we take the rolling mean with a window size of 4 for graphs that compare the number of iterations for the test instances.

6.1 The Set Cover Problem

The DQN Agent displays a rapid learning curve, with a sharp decrease in the number of iterations required to complete training instances as the number of episodes increases, especially noticeable until around episode 35. After an initial phase of learning, the DQN Agent’s performance stabilizes and remains consistently lower than the starting point, indi-
cating that the agent has successfully learned a generalizable strategy for the given problem. There are some upward fluctuations in the later episodes, but these could represent the agent encountering more complex instances or exploring different strategies. The DQN’s ability to adapt its strategy and improve efficiency over time is clearly demonstrated.

On the other hand, the Maximum Fraction Branching method starts off with a higher number of iterations and shows a slight decline initially but then plateaus. Throughout the episodes, its performance varies with significant peaks and troughs, suggesting that while it may handle some instances well, it struggles with others and does not improve over time - a limitation when compared to the adaptive learning capability of the DQN Agent. We will observe a similar trend in the coming graphs.

In Figure 2, can observe that the DQN Agent trained on 50 variables generally requires fewer iterations to complete the test instances compared to the one trained on 100 variables. This suggests that training on larger, perhaps more complex instances may have provided
the DQN Agent with an equally good generalization capability, allowing it to solve smaller instances efficiently.

In Figure 3, a rapid improvement is visible until it stabilizes around episode 42. After this point, the agent consistently requires far fewer iterations, compared to the beginning of training, with some fluctuations that might be attributed to the variance in the difficulty of training instances or exploration during learning. The steady and maintained performance of the DQN Agent after the initial episodes indicates a successful learning and optimization process, highlighting the agent’s ability to generalize from the experience gained during training. It suggests that the DQN Agent has effectively learned a strategy that is generally applicable to this class of problems.
Figure 3

Comparison of Iterations taken to Complete Training Instances (100 Variables and 100 Constraints)

Figure 4

Comparison of Iterations taken to Complete Test Instances (100 Variables and 100 Constraints)
In Figure 4, we again observe that the DQN Agent trained on 100 variables generally requires fewer iterations to complete the test instances compared to the one trained on 50 variables. This suggests that training models with similar instance sizes as the test cases tend to provide better results.
6.2 The Max Cut Problem

In Figure 5, we observe a similar case as in the SCP results. The model is able to learn a strategy around the 40th episode, suggesting a solidification of its problem-solving strategy.
In Figure 6, the DQN Agent trained on 25 vertices consistently performs with the fewest iterations across all test instances, showcasing its effectiveness for the instance size it was trained on. The iterations for this agent seem to range mostly between approximately 200 and 600, demonstrating a robust generalization to the test instances.

Interestingly, the DQN Agent trained on 35 vertices does not perform as well on these 25-vertex test instances. It requires a higher number of iterations, typically between 400 and 800, suggesting that training on larger instances does not necessarily translate to better performance on smaller instances. This might be due to overfitting to features specific to larger instances.
In Figure 7, approximately by episode 30, the iterations have decreased to about 1200 or fewer. This downward trend suggests that the agent has learned a successful strategy.
In Figure 8, the DQN Agent trained on 35 vertices generally requires fewer iterations to solve the instances than the DQN Agent trained on 25 vertices. This suggests that having a training set that closely matches the test instance size can be advantageous for the agent’s performance, which we also observed in SCP. The DQN Agent trained on 35 vertices shows a range of iterations needed from around 1600 to just over 2000, while the agent trained on 25 vertices requires iterations in a slightly higher range, roughly between 1400 and 1800. Despite being trained on smaller instances, the DQN Agent trained on 25 vertices performs well, but never outperforms the agent trained on 35 vertices. This could indicate that the training on 25 vertices was sufficient to learn a general strategy effective for larger instances as well, but perhaps with less consistency.
References


Suresh Bolusani, Mathieu Besançon, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros


Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In *Principles and Practice of Con-


