Lifted Branching: Learning to Improve Branching Strategies

Simon Renard[a,*], Quentin Louveaux[b], Bernard Fortz[c]

[a] *Département d'informatique, Université libre de Bruxelles, 1050 Bruxelles, Belgium*
[b] *Institut Montefiore, Université de Liège, 4000 Liège, Belgium*
[c] *HEC Liège, Université de Liège, 4000 Liège, Belgium*

Abstract

Branch-and-bound (B&B) is a widely used algorithm for solving mixed-integer linear programs (MILPs). One of the elements of its efficiency is the branching strategy. While strong branching empirically produces small B&B trees, its high computational cost makes it impractical for real-world applications. Recent advances in machine learning explored imitation learning to replicate the effectiveness of strong branching. However, these approaches are limited to mimicking existing strategies rather than discovering new ones. In this work, we propose lifted branching, a novel framework that iteratively improves an existing branching strategy by using imitation learning to learn from an improved version of the strategy. Lifted branching is designed to generate smaller B&B trees while avoiding the computational overhead of strong branching, making it suitable for offline training and efficient real-time deployment. We demonstrate the effectiveness of our approach through extensive experiments on four different problem classes, showing that our learned lifted branching models can outperform state-of-the-art strategies, such as reliability pseudo-branching and learned strong branching, depending on the problem of interest.

Keywords: Branch-and-bound, Integer programming, Machine learning in OR

*Corresponding author.
Email addresses: simon.renard@ulb.be (Simon Renard), q.louveaux@uliege.be (Quentin Louveaux), bernard.fortz@uliege.be (Bernard Fortz)

1. Introduction

Industries today face complex combinatorial problems on a daily basis. Examples include planning the electricity production of different power plants (Montero et al., 2022; Carroll et al., 2017), routing packets in a communication network (Abdullah et al., 2018; Callebaut et al., 2023), planning delivery routes for different drivers (Golden et al., 2023). These examples represent only a fraction of the applications; many others are listed in Petropoulos et al. (2024). These combinatorial problems consist of finding an optimal solution in a discrete space. Efficiently solving these problems is a major concern. However, they are generally very hard to solve in the sense that they are NP-hard (Karp, 2010).

These combinatorial problems can often be formulated as mixed integer linear programs (MILPs). Branch-and-bound (B&B; Land and Doig 2010) is the most used algorithm used to solve MILPs. It consists in recursively exploring the solution space while solving simple relaxations. During the process, some bounds are used to prune the B&B tree and thus limit the number of steps needed. In addition to being an exact procedure, it finds feasible solutions along the way and is able to give an indication of their quality.

The success of B&B relies on several heuristics, such as node selection or branching strategy, which can have a great impact on the final size of the B&B tree and thus the solving time. The *node selection* strategy chooses which solution space to explore at each step. The *branching strategy* chooses how the space is partitioned.

In order to improve the efficiency of B&B, recent research (Alvarez et al., 2017; Gasse et al., 2019) has taken advantage of the fact that the MILPs which are solved on a daily basis are often very similar, differing in some parameters only. Based on that observation, the authors use some machine learning (ML) based approach in different steps of B&B. In particular, a successful application (Gasse et al., 2019) is the use of an ML model to imitate a branching strategy that empirically leads to small B&B trees but is too time-consuming to be used in practice. Other works, such as Etheve et al.

(2020); Parsonson et al. (2023); Scavuzzo et al. (2022), use a reinforcement learning approach to discover new branching strategies that might be specific to the problem in focus.

On the one hand, trying to imitate an existing strategy does not allow the discovery of new strategies. On the other hand, the use of reinforcement learning struggles to give better strategies than the default one in the solvers. The primary objective of this research is to formulate a machine learning framework that produces branching strategies capable of generating smaller B&B trees and achieving faster solving times than existing imitation baselines, without the prohibitive computational overhead of strong branching during deployment. To achieve this objective, we propose lifted branching (LB), a novel framework that iteratively improves an existing branching strategy by using imitation learning to learn from an improved version of the strategy. This approach lies between the simple imitation of an existing strategy and the use of reinforcement learning, as it is able to discover new strategies when the methodology is applied iteratively.

Computational experiments demonstrate the efficacy of the proposed framework on structured domains. When evaluated on unit commitment and capacitated facility location problems, the learned models achieve lower solving times on hard instances compared to both SCIP's default reliability pseudo-branching and state-of-the-art imitation learning baselines. However, the results also highlight the representational limitations of standard Graph Convolutional Neural Networks (GCNNs) in this context. Specifically, the GCNN models exhibit low accuracy in exactly replicating the lifted strategy's node-level decisions. Despite this limited predictive accuracy, the architecture successfully extracts a generalized policy that reduces overall solving time for specific problem classes.

The remainder of this paper is organized as follows. Section 2 introduces the background on mixed integer linear programs, the branch-and-bound algorithm, and the bipartite graph representation processed by graph convolutional neural networks. Section 3 reviews related work, detailing recent machine learning and imitation learning approaches for branching strategies. Section 4 presents the lifted branching strategy, including its motivation, its

formalization as a Markov decision process, and an approximation method to reduce computational time. Section 5 details the learned lifted branching framework, describing dataset generation, model usage, and the iterative learning process. Section 6 reports our extensive computational experiments on four benchmark problems, analyzing model accuracy and performance against state-of-the-art baselines. Finally, Section 7 concludes the paper and discusses future research directions.

2. Background

2.1. MILP and B&B

A mixed integer linear program (MILP) is an optimization problem where the objective is to minimize a linear function $c^\intercal x$ while satisfying a set of linear constraints $Ax \leq b$. In addition, some of the variables can be integers $x_i \in \mathbb{Z}\ \forall i \in \mathcal{I} \subseteq N$ with N being the set of all variable indices. This problem can be written as

$$MILP = \arg\min_x \{c^\intercal x | Ax \leq b, x_i \in \mathbb{Z}\ \forall i \in \mathcal{I}, x_j \in \mathbb{R}_+\ \forall j \in N \setminus \mathcal{I}\} \quad (1)$$

The most common algorithm to solve MILP is the branch-and-bound (B&B; Land and Doig 2010) method. It works by recursively dividing the solution space (usually in two parts) and pruning some nodes of the tree by using some bounds that prove the optimal solution is not in a given part of the tree.

To divide a node into two subproblems, the linear relaxation of the node is solved, yielding the fractional solution x^0. This gives the dual bound $z^0 = c^\intercal x^0$. We write $C = \{i | i \in \mathcal{I},\ x_i^0 \notin \mathbb{Z}\}$ the set of variables that must be integer in the original MILP but are fractional in the solution of the relaxation. This set C is the set of branching candidates.

The B&B algorithm relies on several heuristics that have a significant impact on tree size and the solving time (Achterberg and Wunderling, 2013). Examples include the selection strategy or the branching strategy. The node selection strategy chooses one open node of the tree which must be explored

next. Once the node is selected, and the linear relaxation is solved, the branching strategy chooses how the solution space is divided in two parts. It selects a variable x_i with $i \in C$, then two child nodes are created. The constraint $x_i \leq \lfloor x_i^0 \rfloor$ is added to the subproblem of the left child node. The constraint $x_i \geq \lceil x_i^0 \rceil$ is added to the subproblem of the right node.

Most branching strategies can be expressed as a scoring function. This function gives a score to each candidate, and branching is done on the candidate with the highest score.

Strong branching (SB; Applegate et al. 1995) is a well-known branching strategy that branches on the variable that gives the best improvement of the dual bound. For a candidate variable $x_i \in C$, its dual bound improvement is computed by solving the two linear relaxations of the two nodes that would have been added to the tree if branching had been done on x_i. This gives the two dual objectives z_i^{-0} and z_i^{+0}, and the dual bound improvements are $z_i^{-0} - z^0$ and $z_i^{+0} - z^0$. The combination of these two improvements, using a weighted sum or a product score function gives the score of x_i.

Empirically, strong branching leads to small trees. However, it is computationally prohibitive, as it requires solving two linear programming relaxations for every candidate variable at each node. This overhead makes the time to make a decision significantly higher than strategies that rely on simple estimations.

Pseudocost branching (Bénichou et al., 1971) uses historical information collected during the search to estimate the impact of branching decisions. For each variable, it calculates the average gain in the dual bound per unit of change observed in past branching decisions. This estimation is known as the pseudocost of a variable. The pseudocost is then used to predict how much the dual bound would improve if a variable were branched on. Pseudocost branching selects the variable with the highest predicted impact for branching.

While faster to compute than strong branching since it avoids solving additional linear relaxations, pseudocost branching typically results in larger B&B trees. This is mainly due to the poor accuracy of pseudocost estimates in the early stages of the search, when little or no historical data is available

to guide the branching decisions effectively.

To overcome this issue, reliability pseudo-branching (RPB; Achterberg et al. 2005), has been developed. It uses the pseudocost of variables to decide which one to choose, but uses strong branching for variables where pseudocosts are not yet considered reliable. A variable's pseudocost is considered reliable once it has been branched on a specific number of times. Until this threshold is reached, the solver invests the computational effort of strong branching to initialize the pseudocost accurately. This approach generally produces larger trees than using strong branching, but takes considerably less time to make a decision.

RPB is currently the default branching strategy in SCIP optimization suite (Bolusani et al., 2024), one of the fastest non-commercial solvers for MILP. An extensive computational comparison between strong branching, pseudocost branching, and reliability pseudo-branching is provided in Achterberg et al. (2005).

2.2. MILP representation

As we aim to use machine learning to imitate existing branching strategies, it is necessary to have a good representation of the state of a partially developed B&B tree. In order to achieve this, we can use as input the MILP itself or the local information of the B&B nodes. Different attempts have been made to represent these states as a fixed size feature vector (Alvarez et al., 2017; He et al., 2014; Chmiela et al., 2021). However, these representations fail to adequately represent the original input.

A more successful method is to represent a MILP as a bipartite graph $\mathcal{G}$ (Gasse et al., 2019). This representation of a MILP (1) can be achieved by defining two sets of vertices. The first contains one vertex for each variable in the MILP. The second contains a vertex for each constraint in the MILP. An edge is added between a node variable and a constraint variable if the coefficient of the corresponding variable in the corresponding constraint is not zero. An example of this representation is given in Figure 1. To give more information about the MILP, the different components of the graph can be extended with a feature vector. This gives the representation $s =$

$(\mathcal{G}, K, V, E)$ where $K \in \mathbb{R}^{m \times k}$ is the matrix that has a feature vector of size k for each constraint, $V \in \mathbb{R}^{n \times v}$ is the matrix that has a feature vector of size v for each variable and $E \in \mathbb{R}^{m \times n \times e}$ is a sparse matrix that contains a feature vector of size e for each edge between a variable vertex and a constraint vertex.

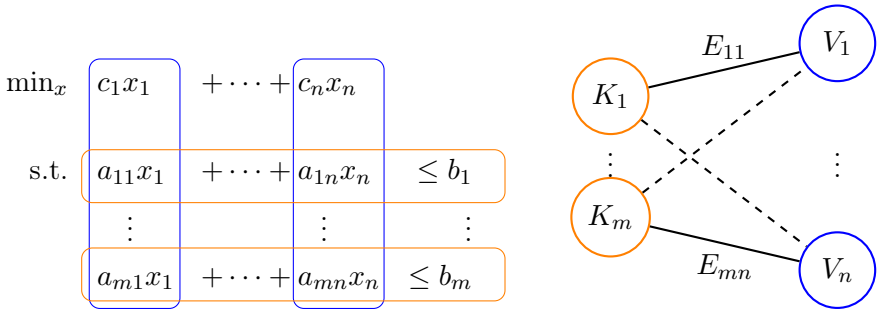

Figure 1: Bipartite graph representation of a MILP

The different feature vectors can be used to give information on the MILP or the B&B process. For example by giving information on the best solution so far or the current gap. In this work, the features used are the same as the ones presented in Gasse et al. (2019). Table 1 sums up these features.

2.3. Graph Convolutional Neural Network

Graph Convolutional Neural Networks (GCNNs) are neural networks specifically designed to operate on graph-structured data (Scarselli et al., 2009). Unlike traditional neural networks that process grid-like data (e.g., images or sequences), GCNNs can handle irregular structures, making them particularly suitable for tasks involving relational data, such as social networks, molecular structures, and, in our case, MILPs.

In the context of MILPs, GCNNs are used to process the bipartite graph representation of the problem presented in Section 2.2. This representation allows the GCNN to learn complex patterns and dependencies that are critical for making informed branching decisions.

GCNNs operate on a message-passing framework where information is iteratively aggregated and transformed across the graph. At each layer of

Graph component	Feature
Constraint node features	Cosine similarity with objective.
	Bias value, normalized with constraint coefficients.
	Tightness indicator in LP solution.
	Dual solution value, normalized.
	LP age of the row, normalized with total number of LPs.
Edge feature	Constraint coefficient, normalized per constraint.
Variable node features	Type (binary, integer, impl. integer, continuous) as a one-hot encoding.
	Objective coefficient, normalized.
	Lower bound indicator.
	Upper bound indicator.
	Solution value equals lower bound.
	Solution value equals upper bound.
	Solution value fractionality.
	Simplex basis status (lower, basic, & upper, zero) as a one-hot encoding.
	Reduced cost, normalized.
	LP age of the column, normalized.
	Solution value.
	Value in incumbent.
	Average value in incumbents.

Table 1: Description of features used to annotate a bi-partite graph that represent a sub-problem in a B&B tree from Gasse et al. (2019)

the network, nodes receive messages from their neighbors, which are then combined and transformed to update the node's feature representation.

A detailed description of the architecture used in Gasse et al. (2019), which is the one used in this work, is given in Appendix A.

3. Related work

Using ML to improve MILP solving is not a new idea in the literature. During the last decade, several papers have implemented ML in various subroutines of B&B. In Hendel (2022); Chmiela et al. (2021), the authors used machine learning techniques to plan the different heuristics used to find a primal solution. For node selection, ML was used to learn a strategy that selects the node on the path to an optimal solution (He et al., 2014) or on the path to a good solution (Yilmaz and Yorke-Smith, 2021). A comprehensive overview of ML applications in B&B is provided in Scavuzzo et al. (2024).

How to use ML to improve B&B through the branching strategy is one of the hot areas in research. The most successful approaches use an imitation learning approach (Pomerleau, 1991). It consists of training a model to

perform a task by trying to imitate how an expert does it. This idea is used when it may be too time-consuming for the learning agent to discover a new strategy, and there exists an expert who is known to make good choices, which is either too time-consuming to use by itself or difficult to implement as an algorithm (i.e., when the expert is a human). The imitation learning method starts by collecting expert decisions in different situations, then trains a model using supervised learning (Cunningham et al., 2008) on the collected decisions. Finally, the model can be used to make decisions in unseen situations.

One of the first papers to use machine learning for designing a branching strategy is Alvarez et al. (2017). The authors used an imitation learning process to learn a branching strategy. The expert used is strong branching, a natural choice as it is a strategy known to make good decisions leading to small B&B trees, but too time-consuming to be competitive in practice. When applied to homogeneous instances, i.e. instances from the same problem, the trained model was able to compete with RPB in terms of time and number of nodes. However, no cuts or heuristics were used in these experiments.

A major improvement came from Gasse et al. (2019). In this work, the authors also used an imitation learning process to mimic strong branching. The big breakthrough they propose is to represent a MILP as a bipartite graph. This representation allows the use of a GCNN model as explained in Section 2.3. The authors trained 4 models on simple randomly generated instances of set covering problems, combinatorial auction, capacitated facility location and independent set problems. These models were then tested on unseen instances of different sizes of the same problem. For all models, this approach outperformed RPB in terms of time, while for 2 of them it outperformed RPB in terms of number of nodes.

More recently, Du et al. (2025) proposed using a GCNN model to learn both a branching and a node selection policies. This approach resulted in improved performance in terms of time compared to the approach in Gasse et al. (2019). However, the trained models have only been tested on instances of the same size as those used for training. In Cai et al. (2025),

the authors proposed a multi-task learning framework for MILP to learn a shared representation that generalizes across different tasks such as backdoor identification, predict-and-search and solver configurations.

Another approach to using ML for the branching strategy is to use reinforcement learning. Reinforcement learning is motivated by scenarios where no good expert is available. For instance, strong branching makes decision based on the dual bounds improvement which can be ineffective for problems with such weak bounds. In Etheve et al. (2020), the authors show that when a depth first search (DFS) is used as node selection, the branching decision can be expressed as a time Markov Decision Process. Using this property, they learned a policy to make branching decisions. This approach being limited by the use of DFS which is not a realistic setting for MILP solver, Scavuzzo et al. (2022) propose an improvement by expressing the branching decision as a tree Markov Decision Process (treeMDP) under the condition to know the optimal objective value of the problem. While being an improvement, reinforcement learning methods struggle to compete with the imitation learning approach. This treeMDP approach is criticized in Strang et al. (2025). The authors argue that treeMDP is theoretically flawed and incapable of supporting advanced reinforcement learning algorithms. They proposed a rigorous Markov Decision Process formulation that allows for the use of standard approximate dynamic programming algorithms to learn optimal branching strategies. With their new formulation, they outperform other reinforcement learning agents and SCIP solver on some benchmarks. However, they still do not compete with imitation learning. A primary reason reinforcement learning struggles to match the performance of imitation learning in this domain is the challenge of reward formulation and credit assignment. While imitation learning provides dense, immediate supervision at every node by mimicking an expert's exact decision, reinforcement learning agents must optimize based on delayed and often sparse rewards, such as the final size of the B&B tree Scavuzzo et al. (2022).

To provide a clear overview of the current landscape, Table 2 summarizes the key methodological differences and performance outcomes of these state-of-the-art approaches.

Reference	Methodology	Learning Goal	Performance & Limitations
Alvarez et al. (2017)	Extra Trees (Supervised)	Imitates Strong Branching	Competitive with RPB on homogeneous instances. Tested without cuts/heuristics.
Gasse et al. (2019)	GCNN (Supervised)	Imitates Strong Branching	Outperforms RPB in time on diverse instance sizes. State-of-the-art for ML to branch.
Du et al. (2025)	GCNN (Supervised)	Joint Branching & Node Selection	Competes with Gasse et al. (2019). Generalization limited to training instance sizes.
Etheve et al. (2020)	Reinforcement Learning	Discovers new strategy timeMDP	Feasible for DFS. Struggles to outperform imitation approaches.
Scavuzzo et al. (2022)	Reinforcement Learning	Discovers new strategy treeMDP	Models branching as TreeMDP. Improves over Etheve et al. (2020).
Strang et al. (2025)	Reinforcement Learning	Discovers new strategy MDP	Proposes a better MDP formulation than Scavuzzo et al. (2022) and outperforms it. Less competitive than imitation learning.
Lifted Branching (ours)	GCNN (Supervised)	Iteratively improves a strategy and learns to imitate it	Outperforms RPB and LSB on highly structured domains (e.g., unit commitment). Less competitive on diverse/unstructured instances.

Table 2: Comparison of recent machine learning approaches for branching strategies.

4. Lifted Branching

4.1. Motivation

When developing a branching strategy for B&B, one must carefully find a balance between its efficiency, i.e. its ability to lead to small B&B trees, and its time efficiency, i.e. the time it takes to make a decision. Having a strategy that is guaranteed to lead to the smallest tree is of no use if it takes more time to make a decision than solving the whole problem with another strategy. An example of this trade-off can be seen in RPB. It finds a trade-off between using strong branching to make good decisions but at a high time cost, and using pseudocosts score to make faster decisions leading to bigger trees.

However, when an imitation learning process is used to imitate an existing strategy, this trade-off loses its importance. The time taken to make a decision depends on the complexity of the model and no longer on the initial strategy that serves as an oracle for the training process. While this

observation is already the motivation to use imitation learning on strong branching (Gasse et al., 2019), it can be pushed further by using imitation learning on branching strategies that are less competitive in terms of time complexity than strong branching but very competitive in term of B&B tree size. This is the purpose of the proposed strategy: lifted branching. Lifted branching is designed to lift up an existing strategy to reduce the B&B tree size at the cost of very high computational time.

4.2. Branching strategy and Markov decision process

When solving a MILP, the B&B algorithm can be viewed as a Markov decision process (MDP) from the perspective of the branching strategy. This process is defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathit{step})$.

$\mathcal{S}$ is the set of possible states of the solver. A state $s \in \mathcal{S}$ is described as a tuple $(T, \mathcal{N}, N_f)$ where T describes the current state of the B&B tree with all the different information gathered during the solving process, $\mathcal{N}$ is the set of opened nodes, and N_f is the focused node being processed. $s_{null} \in \mathcal{S}$ is a terminal state in which the optimal solution or the infeasibility of the MILP has been found and proven.

$\mathcal{A}$ is the action space, it corresponds to all the indices of the variables that must be integer. $\mathcal{A}_s \subseteq \mathcal{A}$ is the subset of actions available at a given state s. $\mathcal{A}_s = \{i | i \in \mathcal{I},\ x_i^{s0} \notin \mathbb{Z}\}$ where x^{s0} is the solution of the linear relaxation of the focused node in state s. When the B&B algorithm finishes, at state s_{null}, the set of actions is empty $\mathcal{A}_{s_{null}} = \emptyset$.

$\mathit{step} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the transition function. Given a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}_s$, it gives the new state of the B&B algorithm after performing the branching on variable x_a. This function *step* depends on the different heuristics that are used in the B&B algorithm (node selection, primal heuristic, ...), therefore it might not be deterministic.

Each branching strategy B has an associated scoring function $f_B : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Given a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}_s \subseteq \mathcal{A}$, the function $f_B(s, a)$ gives a score for the action a for the state s. Thus, at state s, B will choose to perform the action $\arg\max_{a \in \mathcal{A}_s} f(s, a)$

We denote by $\phi_B : \mathcal{S} \to \mathbb{N}$ the function that given a state $s \in \mathcal{S}$ returns the number of nodes in the B&B tree required to reach a final state s_{null}. This function is parameterized by a branching strategy B that decides for each state which action to perform. This function can be recursively defined as:

$$\phi_B(s) = \begin{cases} \#nodes \text{ used by the solver at state } s & \text{if } \mathcal{A}_s = \emptyset \\ \phi_B(step(s, \arg\max_{a \in \mathcal{A}_s} f_B(s, a))) & \text{otherwise} \end{cases}$$

4.3. Algorithm description

The lifted branching strategy branches on the variable that minimizes the total node count produced by fully exploring the resulting subtree with an arbitrary branching strategy B. Thus, the lifted branching strategy is noted $LB[B]$ where B is an arbitrary branching strategy.

Informally, $LB[B]$ chooses on which candidate to branch by doing the following. For a candidate variable x_a, it makes a copy of the current state of the solver. On this copy, it removes all the open nodes and decides to branch on x_a. Then, it finishes the solving process of the copied solver using the branching strategy B. Finally, on the original solver, it branches on the variable that led to the smallest tree.

More formally, for a given state of the solver $s = (T, \mathcal{N}, N_f) \in \mathcal{S}$ and an action $a \in \mathcal{A}_s$, in order to compute $f_{LB[B]}(s, a)$, we create a new state $s_a = (T, \emptyset, N_f)$. Then we perform the branching on the variable x_a by choosing the action a. This gives the state $s'_a = step(s_a, a)$. Finally, the score is $f_{LB[B]}(s, a) = -\phi_B(s'_a)$.

Figure 2 illustrates how the function f_B computes the score of a branching candidate.

Assuming that $LB[B]$ leads to smaller trees than using B, recursively using the lifted branching strategy could lead to even smaller trees. We denote $LB^{\delta}[B]$ with $\delta \in \mathbb{N}_0$ to indicate the strategy that recursively applies δ times the lifted branching strategy with B as the base strategy. For example, $LB^2[SB] = LB[LB[SB]]$.

Notably, this procedure bears similarities with Monte Carlo Tree Search

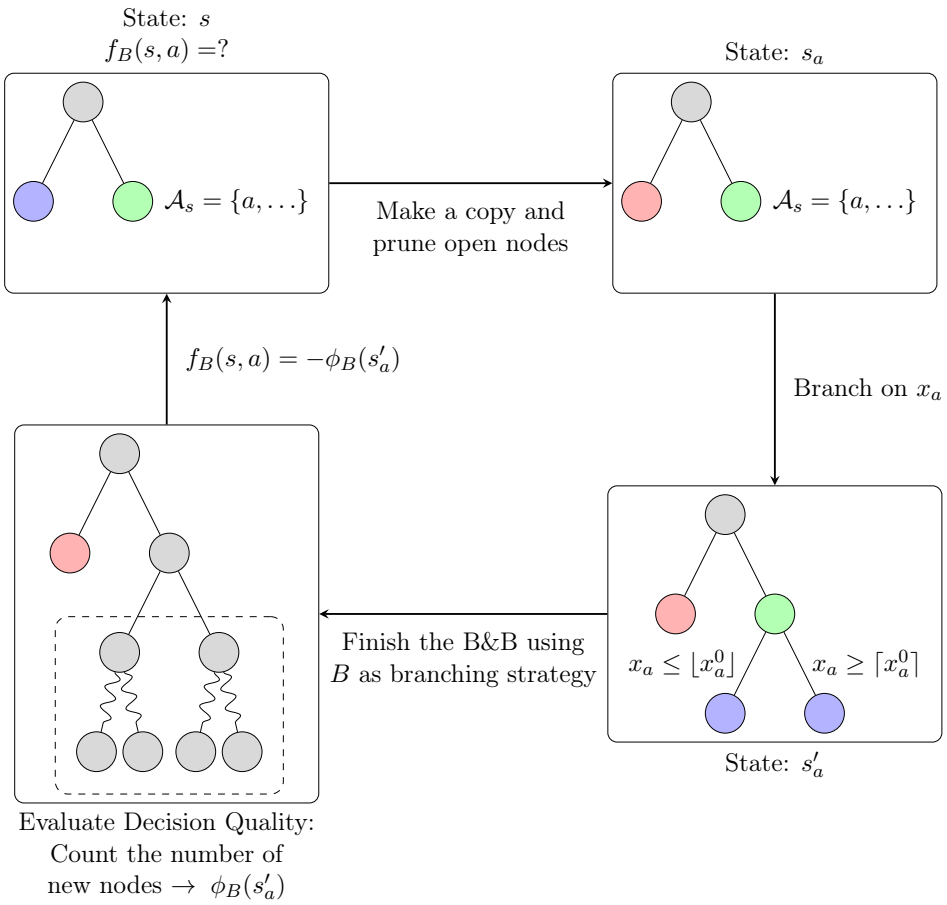

Figure 2: Computation of the score $f_B(s,a)$ of a candidate $a \in \mathcal{A}_s$ from a state s of the B&B tree. The diagram illustrates a "rollout" process where the quality of branching on variable x_a is evaluated by the total number of nodes (tree size) required to fully explore the resulting subtree. Blue nodes are open nodes, red nodes are pruned nodes, the green node is the focused node that requires a branching.

(MCTS; Coulom 2006), in particular with the *rollout* step. In MCTS, rollouts are used to estimate the value of a decision by simulating the continuation of the game until termination. In our case, the "rollout" consists of finishing the exploration of the B&B tree from a copied state by applying strategy B, and then using the resulting tree size to evaluate the quality of the branching decision.

4.4. Approximation

While the primary motivation for $LB^{\delta}[B]$ is that its computational cost is acceptable for offline model training, it can still be prohibitively slow. This slowness prevents the generation of a sufficient dataset even from simple instances, which is necessary for training a model to accurately mimic $LB^{\delta}[B]$. To address this issue, we propose a way to approximate the scoring function $f_{LB^{\delta}[B]}$.

Instead of returning the number of nodes required to reach a final state starting from s'_a by using the branching strategy B, we define a new function $\tilde{\phi}_B$ with a limited execution time t available to reach a final state from s'_a. If it succeeds in reaching the final state, it returns the number of nodes used as ϕ_B does. Otherwise, if the time limit is reached before finding a final state, the process is stopped and an estimation of the number of nodes that would have been required to reach a final node is computed. To make such an estimate, we use the work in Hendel et al. (2022). Their method leverages a pre-trained statistical model that maps the current structural profile of the B&B tree to a prediction of final tree size. This profile is captured by four specific state features: the sum of subtree gaps, the tree weight, the leaf frequency, and the total number of open nodes. By querying this estimator, we approximate the tree size without paying the full computational cost of reaching a final state.

Formally, the approximated scoring function is defined as:

$$\tilde{\phi}_B(s) = \begin{cases} \#nodes \text{ used by the solver at state } s & \text{if } \mathcal{A}_s = \emptyset \\ estimate\ \#nodes & \text{if the time limit is reached} \\ \tilde{\phi}_B(step(s, \arg\max_{a \in \mathcal{A}_s} f_B(s,a))) & \text{otherwise} \end{cases} \tag{2}$$

We denote by $\widetilde{LB}^{\delta}[B]$ the lifted branching strategy that uses this approximation for its scoring function. Thus, $f_{\widetilde{LB}^{\delta}[B]}(s,a) = -\tilde{\phi}_B(s'_a)$

Despite this approximation, the computational burden of $\widetilde{LB}^{\delta}[B]$, even with $\delta = 1$, is practical only for very small instances usually solved in seconds using classical branching strategies.

5. Learned Lifted Branching

The computational intensity of the lifted branching strategy arises from the need to fully explore a B&B tree to score candidate variables. While this approach is not feasible for real life applications due to its time complexity, it is well suited for offline use to generate training data for machine learning (ML) models. These models can learn to estimate the scoring function used in $\widetilde{LB}^{\delta}[B]$, allowing practical use of the strategy.

5.1. Dataset generation

Training the ML model begins with creating a dataset that encapsulates the decision-making process of the $LB^{\delta}[B]$ strategy. In imitation learning, this involves collecting observations of the oracle (in this case, $LB^{\delta}[B]$) and the corresponding actions. Each observation is a pair (s, y), where:

1. s is a bipartite graph representation of the current state in the B&B tree, annotated with node and edge features.
2. y is the scores computed by $LB^{\delta}[B]$ for each branching candidate.

To create this dataset, training instances of the targeted problem are solved using the $LB^{\delta}[B]$ strategy. At each branching decision, the state s is recorded along with the scores y used by LB to make the decision.

To ensure diversity in the dataset, a probabilistic approach is adopted: with a probability ϵ, branching decisions are made using the pseudo-cost branching strategy instead of $LB^{\delta}[B]$. This method introduces data from suboptimal branching scenarios. For the rest of this work $\epsilon = 0.4$ is used. This specific value was determined through preliminary empirical testing to strike an optimal balance between learning from the expert oracle and exposing the model to data from suboptimal branching scenarios.

The graph representation of the subproblem, including its features, follows the methodology outlined in Gasse et al. (2019). These features effectively describe the constraints, variables, and their relationships in the current B&B subproblem. These features are listed in Table 1.

5.2. Model usage

Once trained, the ML model acts as an efficient estimator of the LB oracle. Given the current state of a B&B process, the model predicts scores for all branching candidates. These scores can then guide the branching decisions.

The use of the model introduces significant computational savings. Instead of fully exploring a tree to score candidates as $LB^{\delta}[B]$ requires, the model rapidly generates scores based on its learned estimation. This approach retains the strategic insights of $LB^{\delta}[B]$ while being computationally feasible for real-time application in solving MILPs.

5.3. $LB^{\delta}[B]$ approximation

Directly learning $\widetilde{LB}^{\delta}[B]$ with $\delta > 1$ does not appear to be feasible as it requires to solve instances using $\widetilde{LB}^{\delta}[B]$, which applies lifted branching recursively δ times. In practice, with $\delta > 1$, this takes too much time.

To address this challenge, we propose an iterative approach, rather than a recursive one to learn the *learned lifted branching* strategy, denoted as $LLB^{\delta}[B]$.

The process begins by learning $LLB^1[B]$, which requires only solving instances using $\widetilde{LB}^1[SB]$. Then one can iteratively learn $LLB^\delta[B]$ by using $\widetilde{LB}^1[LLB^{\delta-1}[B]]$ as an oracle. This iterative learning framework mitigates the computational burden while still capturing the benefits of deeper lifted branching strategies.

Since the objective of $LLB^\delta[B]$ is to learn an improved version of a given strategy B, it is natural to choose a baseline strategy that already yields small B&B trees. For this reason, we select strong branching (SB) as the baseline strategy ($B = SB$) throughout the remainder of this work. To simplify the notation, we write LLB^δ instead of $LLB^\delta[SB]$.

5.4. Code and Data Availability

The implementation of the lifted branching framework and the learned lifted branching models, together with the scripts used for the experiments, are publicly available at: `https://github.com/sirenard/ML4UB`. This repository contains the source code, dataset generation procedures, and instructions for reproducing the experiments presented in this paper.

6. Experiment

This section evaluates the performance of the learned lifted branching (LLB^δ) strategies against established baselines, including strong branching (SB), reliability pseudo-branching (RPB), and the learned version of strong branching (LSB) proposed in Gasse et al. (2019).

6.1. Benchmark problems

To ensure a realistic benchmark, we used the single-bus unit commitment problem. This problem involves scheduling power generators to meet electricity demand while respecting physical constraints. We employed the MILP formulation from Carrion and Arroyo (2006), focusing on instances where the generator set remains constant, but the daily load curve varies.

Difficulty	# Items	# Bids
Easy	100	500
Medium	200	1000
Hard	300	1500

(a) Combinatorial Auction

Difficulty	# Facilities	# Customers
Easy	100	100
Medium	100	200
Hard	100	400

(b) Capacitated Facility Location

Difficulty	Planning Horizon
Easy	1 day
Medium	2 days
Hard	3 days

(c) Unit Commitment

Difficulty	# Subset	# Objects
Easy	1000	500
Medium	1000	1000
Hard	1000	2000

(d) Set Cover

Table 3: Parameters defining the difficulty levels for each benchmark problem.

The unit commitment instances were generated using the Unit Commitment Instance Generator (UCIG) [1] with the parameters presented in Table B.8. These instances were split into three difficulty levels by adjusting the planning horizon.

This setup ensures consistent generator characteristics while varying problem complexity via load curves. It reflects realistic industrial settings, where solved instances are often very similar and differ only in a few parameters.

In addition, we assess our approach on problems widely used in the literature: combinatorial auction (Leyton-Brown et al., 2000), capacitated facility location (Cornuéjols et al., 1991) and set cover (Balas and Ho, 2009). For each problem, 3 sets of instances are randomly generated: easy, medium and hard. Table 3 summarizes the specific parameter settings used to generate the instances of different difficulties for all the problem classes.

This choice of benchmarks allows us to compare our strategy against classical problems widely used in the literature. Additionally, unit commitment is chosen to represent a realistic application scenario where instances are highly similar, differing only by specific parameters like daily demand.

[1] `http://www.poderico.it/ucig/index.html`

6.2. Methodology

For each benchmark problem, we generate 4000 random easy instances solved with the oracle branching strategy. At each branching decision, the oracle is applied with probability $1 - \epsilon$, otherwise pseudocost branching is used, ensuring exploration of diverse B&B states. This process yields a training dataset of approximately 140,000 observations for each problem class per iteration. The collected observations are used to train a GCNN. This setup is used to train LLB^{δ} models with $\delta \in 1, 2, 3, 4$ as well as the LSB model which is trained to imitate strong branching (Gasse et al., 2019).

Generating LLB^{δ} training data with $LB[B]$ is the most computationally demanding step, requiring 240 wall-clock hours on 24 CPUs. Model training requires 5 hours. All the experiments are run on an x86-64-v4 @ 3.25 GHz CPUs and an NVIDIA RTX 6000 Ada GPU.

6.3. Model accuracy

Before evaluating the performance of the learned models on the test instances, it is important to assess their accuracy. This involves measuring how closely the decisions made by the models align with those of their respective oracles.

To evaluate accuracy, a set of unseen instances is solved using the trained strategies. For each decision made, we check whether the model's choice is among the top n decisions identified by the oracle. This is done by computing at each node the scores of the candidates using the model and the oracle. The metric $Acc@n$ represents the proportion of decisions where the model's choice, i.e. the candidate with the highest score, ranks within the top n decisions according to the oracle.

Table 4 shows the accuracy measures of the models for $n \in \{1, 5, 10\}$. 20 easy instances have been solved to obtain these metrics.

Overall, the LLB^{δ} models consistently show lower accuracies than LSB, indicating that the current GCNN architecture from Gasse et al. (2019) reproduces strong branching decisions more effectively than lifted branching ones. For a given class of problem, the accuracies of the LLB^{δ} models remain similar which indicate that increasing δ does not make it harder to imitate.

Model	Acc@1 (%)	Acc@5 (%)	Acc@10 (%)
LSB	45.1	83.0	92.8
LLB^1	13.6	40.7	63.9
LLB^2	12.7	37.9	63.3
LLB^3	11.6	37.7	64.3

(a) Unit Commitment

Model	Acc@1 (%)	Acc@5 (%)	Acc@10 (%)
LSB	22.7	57.7	75.0
LLB^1	1.0	12.0	22.0
LLB^2	3.8	15.9	27.3
LLB^3	2.7	12.0	25.5

(b) Combinatorial Auction

Model	Acc@1 (%)	Acc@5 (%)	Acc@10 (%)
LSB	68.7	98.0	99.9
LLB^1	41.3	77.5	95.8
LLB^2	32.5	79.0	97.1
LLB^3	40.5	74.5	96.7

(c) Capacitated Facility Location

Model	Acc@1 (%)	Acc@5 (%)	Acc@10 (%)
LSB	7.4	25.7	39.9
LLB^1	1.55	9.30	17.83
LLB^2	1.30	7.79	18.18
LLB^3	2.14	8.57	17.86

(d) Set Cover

Table 4: Accuracy of the different trained models

The Acc@1 values are low for all the models, indicating that they struggle to make the same decisions as the oracles. Nevertheless, these models may still be capable of producing effective branching strategies for several reasons. Firstly, not all decisions have the same impact on the B&B tree size. Indeed, making good decisions for the first nodes is more important than good decisions for the nodes deep in the tree. A model may initially make good decisions and then poor ones due to its low accuracy, but still lead to small trees. Secondly, even if the models are not very accurate, if they imitate a good strategy with respect to tree size, they may still be better than standard branching strategies such as RPB.

For combinatorial auction and set cover instances, the LLB^δ accuracies are very low which suggest that these strategies will not be very competitive on these instances. For capacitated facility location instances, the higher accuracies suggest that the LLB^δ strategies can be more competitive than on combinatorial auction or set cover instances.

6.4. Model performance

In this experiment, we evaluate the performance of LLB^δ by solving 50 easy, medium, and hard instances of each problem type using SCIP 9.2 (Bolusani et al., 2024) with a time limit of 1800 seconds per instance. These instances are also solved by the state-of-the-art strategy using machine learning LSB (Gasse et al., 2019) and with the default branching strategy of the SCIP solver, RPB. Each instance is solved 5 times with each strategy with

different seeds. For each performance metric, we report the median value across the 5 runs.

6.4.1. Reported measures

The following metrics are reported in order to assess the performance of the different solvers:

#**nodes** : The shifted geometric mean ($s = 10$) over the number of nodes for the instances solved optimally in the given time.

time : The shifted geometric mean ($s = 1$) over time spent on each instance. This includes the cases where the solver is stopped due to the time limit.

gap : The shifted geometric mean ($s = 1$) over the gap for the instances not solved optimally within the time limit.

#**wins** : The number of time when the solver is the fastest (w.r.t. the time).

#**solved** : The number of instances optimally solved within the time limit.

The *wins* metric may be misleading when comparing more than 2 methods. For example, while testing 3 strategies A, B, C, the number of wins could be respectively: $30, 30, 40$. Regarding this metric, one can conclude that the strategy C is better in terms of wins than A and B. However, regarding the same numerical results but by dropping the strategy A, the number of wins could become: 60 wins for B and 40 wins for C. This shows that B performs better than C regarding this metric, but that having several strategies that perform well on the same kind of instances could lead to biased results.

In order to avoid this confusion, we plot a performance profile w.r.t. time for each strategy. For one solver, this plot is a non-decreasing curve that describes the proportion of instances that have been solved optimally withing a given time.

To make the interpretation of this plot easier, it can be reduced to a unique value: the area under the curve score (AUC score). This idea comes from the ML world (Hoo et al., 2017) in which the AUC score is the area under the ROC curve. In our case, as the domain of the function is not $[0, 1]$ but $[0, t]$ where t is the time limit of the solver, we define the AUC score as $\frac{A}{t}$ with A being the area under the performance profile curve in order to always have a score between 0 and 1. If this score is 1, this means that all the instances are solved optimally instantly while a score of 0 indicates that none of the instances have been solved optimally within the time limit.

6.4.2. Results

Experimental results for unit commitment, combinatorial auction, capacitated facility location and set cover instances are summarized in Table 5. The number of solved instances and the gap is only reported for hard instances for which not all instances are solved to optimality within the time limit of 1800s.

The results indicate that the efficiency of the learned lifted branching (LLB^{δ}) depends a lot on the problems solved. Table 6 sums up the AUC scores of each solver for each problem tested.

	Easy			Medium			Hard				
Model	#nodes	time (s)	wins	#nodes	time (s)	#wins	#nodes	time (s)	wins	#solved	gap
RPB	**89.78** ± 132.8%	2.68 ± 22.2%	0	7210.61 ± 160.8%	38.24 ± 89.3%	7	95422.90 ± 66.5%	453.71 ± 44.3%	7	**39**	0.01
LSB	143.79 ± 57.5%	2.04 ± 18.1%	3	4740.06 ± 89.4%	**28.95** ± 53.9%	**16**	67514.80 ± 62.8%	365.97 ± 42.6%	11	35	0.01
LLB^1	114.67 ± 49.4%	**1.87** ± 17.1%	15	**3717.96** ± 128.8%	33.04 ± 75.4%	8	**59380.90** ± 68.3%	**331.40** ± 40.9%	**13**	35	0.01
LLB^2	111.36 ± 51.5%	**1.87** ± 17.7%	**21**	4489.76 ± 102.1%	29.32 ± 60.2%	**16**	91191.00 ± 54.9%	491.36 ± 29.5%	2	32	0.01
LLB^3	107.37 ± 52.4%	**1.87** ± 17.4%	18	3956.75 ± 87.8%	35.16 ± 54.3%	3	73664.60 ± 55.9%	379.47 ± 27.1%	10	32	0.01
Unit Commitment											
RPB	**18.02** ± 32.7%	1.62 ± 9.8%	1	1853.22 ± 13.8%	18.26 ± 5.4%	6	**19311.50** ± 7.9%	**166.04** ± 5.4%	**39**	**46**	0.00
LSB	76.35 ± 14.9%	**1.11** ± 7.6%	**30**	**1769.28** ± 11.4%	**13.69** ± 7.3%	**32**	31533.10 ± 10.6%	249.98 ± 7.8%	5	45	0.00
LLB^1	79.54 ± 13.7%	1.14 ± 7.5%	8	2305.33 ± 12.9%	16.40 ± 8.2%	1	40552.30 ± 13.0%	308.51 ± 8.6%	1	42	0.01
LLB^2	77.94 ± 13.0%	1.16 ± 7.4%	11	2084.40 ± 11.5%	14.79 ± 6.9%	6	36942.70 ± 12.2%	278.10 ± 8.4%	1	43	0.01
LLB^3	80.37 ± 14.0%	1.14 ± 7.8%	8	2185.51 ± 12.1%	14.84 ± 7.5%	5	40381.50 ± 10.6%	298.36 ± 7.2%	0	42	0.01
Combinatorial Auction											
RPB	**69.55** ± 30.3%	24.33 ± 11.5%	2	**112.85** ± 18.8%	104.39 ± 7.9%	3	**132.94** ± 12.9%	406.53 ± 7.7%	10	**50**	0.00
LSB	204.27 ± 14.2%	21.32 ± 11.5%	7	316.37 ± 11.6%	92.89 ± 9.1%	0	351.35 ± 7.8%	394.94 ± 7.9%	3	**50**	0.00
LLB^1	186.36 ± 16.3%	20.18 ± 12.1%	8	270.42 ± 14.1%	**81.95** ± 9.8%	**17**	306.54 ± 8.0%	**352.60** ± 8.0%	**23**	**50**	0.00
LLB^2	185.18 ± 15.2%	20.18 ± 11.2%	12	273.78 ± 14.1%	83.03 ± 9.9%	**17**	317.43 ± 8.0%	359.40 ± 8.0%	12	**50**	0.00
LLB^3	183.39 ± 15.1%	**19.76** ± 10.9%	**21**	288.35 ± 13.2%	84.91 ± 9.6%	13	353.72 ± 7.3%	390.01 ± 8.2%	2	**50**	0.00
Capacitated Facility Location											
RPB	**84.93** ± 21.4%	5.11 ± 6.2%	1	**2675.14** ± 9.7%	42.62 ± 5.0%	16	57381.20 ± 5.4%	**655.56** ± 2.4%	**32**	**33**	0.03
LSB	323.17 ± 14.6%	4.17 ± 7.3%	1	4556.11 ± 8.2%	77.84 ± 6.8%	0	55058.30 ± 4.4%	975.35 ± 1.3%	0	24	0.04
LLB^1	185.67 ± 9.7%	3.61 ± 6.1%	6	2833.24 ± 5.6%	**36.06** ± 4.7%	**22**	54584.60 ± 3.3%	893.90 ± 1.2%	1	26	0.05
LLB^2	184.27 ± 10.0%	**3.35** ± 6.5%	**28**	2992.69 ± 5.4%	37.13 ± 4.4%	10	60677.20 ± 3.1%	997.91 ± 0.9%	0	20	0.06
LLB^3	185.73 ± 9.7%	3.40 ± 6.8%	17	3024.47 ± 5.9%	37.72 ± 5.0%	3	**39082.60** ± 3.6%	1195.31 ± 0.9%	0	16	0.08
Set Cover											

Table 5: Experimental results

	AUC score		
Model	Easy	Medium	hard
RPB	0.42	0.52	0.18
LSB	0.57	0.56	0.21
LLB^1	**0.63**	0.55	**0.23**
LLB^2	**0.63**	**0.57**	0.17
LLB^3	**0.63**	0.54	0.21

(a) Unit Commitment

	AUC score		
Model	Easy	Medium	hard
RPB	0.74	0.45	**0.32**
LSB	**0.96**	**0.51**	0.26
LLB^1	**0.96**	0.48	0.23
LLB^2	**0.96**	0.50	0.25
LLB^3	0.95	0.50	0.24

(b) Combinatorial Auction

	AUC score		
Model	Easy	Medium	hard
RPB	0.43	0.29	0.19
LSB	0.45	0.30	0.19
LLB^1	0.46	**0.32**	**0.21**
LLB^2	0.46	**0.32**	0.20
LLB^3	**0.47**	**0.32**	0.19

(c) Capacitated Facility Location

	AUC score		
Model	Easy	Medium	hard
RPB	0.54	0.40	0.08
LSB	0.47	0.49	**0.13**
LLB^1	0.59	**0.51**	0.09
LLB^2	**0.61**	0.50	0.08
LLB^3	**0.61**	0.50	0.05

(d) Set Cover

Table 6: AUC scores of each strategy tested for each problem

The results for unit commitment instances show domination of the LLB^δ strategies. On easy instances, the 3 LLB^δ strategies are the fastest. On medium instances, even if LSB is the fastest strategy, the LLB^δ models, in particular LLB^1 uses fewer nodes to solve all the instances. On hard instances, LLB^1 is the fastest strategy and uses less nodes, but RPB solves to optimality 4 additional instances.

The results for combinatorial auction instances show that LLB^δ never outperforms LSB. However, it shows small improvement compared to RPB w.r.t. time on easy and medium instances. Regarding the LLB^δ strategies, LLB^2 leads to a smaller number of nodes than LLB^1 and LLB^3. This correlates with the accuracies results where LLB^2 had the highest accuracies of all the LLB^δ strategies for combinatorial auction instances.

For capacitated facility location instances, LLB^2 is the fastest strategy on easy instances, and LLB^1 is the fastest on medium and hard instances.

Finally, for set cover instances, LLB^2 is the fastest strategy on easy instances. On medium instances, LLB^1 is the fastest and twice faster than LSB. However, on hard instances, RPB is the fastest strategy winning over

almost all the solved instances. LLB^{δ} shows better performances than LSB on these hard instances.

Overall, the results show that increasing δ leads to very little improvements w.r.t. the number of nodes on easy instances. However, the resulting strategies become very specific to the instances with the same size as the training ones and do not perform better on harder instances.

Regarding the AUC scores in Table 6, we can observe that the strategy with the highest AUC score is the fastest strategy that has solved the most instances to optimality.

Finally, a Wilcoxon test is performed on the solvers' time to test whether the observed differences are statistically significant or not. Table 7 shows the p-values between the different solvers based on the hard instances only. Values smaller than 0.05, i.e. the results between the 2 solvers are statistically significant, are shown in bold.

We can see that all strategies give statistically significant results compared to RPB. This confirms that the LSB and LLB^{δ} improvement compared to RPB are significant. Regarding LSB, the differences with LLB^1 and LLB^2 are not significant. It is only for LLB^3 and LLB^4 that the difference is significant.

solvers	RPB	LSB	LLB^1	LLB^2	LLB^3
LSB	**0.034**				
LLB^1	**0.011**	0.480			
LLB^2	**0.006**	0.397	0.849		
LLB^3	**0.000**	**0.026**	0.134	0.188	
LLB^4	**0.000**	**0.013**	0.085	0.123	0.861

Table 7: p-values between the different solvers based on the time. Values below 0.05 are bold.

6.5. Discussion

The experimental results show that LLB^{δ} can yield better performance than RPB or LSB. While the specific context for this improvement is not entirely clear, we have several hypotheses based on our observations during the experiments.

We observe a strong correlation between the accuracy and the performance of the models. Unit commitment and capacitated facility location are the two benchmarks on which LLB^{δ} leads to better performances than both RPB and LSB on hard instances. Notably, these are also the models that achieved the highest accuracies. Conversely, combinatorial auction and set cover models exhibit very low accuracies and do not perform well on hard instances. Specifically, the worst performances of LLB^{δ} models are observed on combinatorial auction instances. This corresponds with the largest accuracy gap between the LLB^{δ} and LSB models.

We also observe that the performances of the LLB^{δ} strategies are superior when the training instances share high structural similarities. For unit commitment, the instances are highly homogeneous by construction. Hard and medium instances have the exact same set of generators as easy instances, varying only in the planning horizon. Consequently, the knowledge learned by the GCNN from easy instances is more likely to be relevant while solving harder instances. Similarly, capacitated facility location instances share structural similarities beyond simply belonging to the same problem class. Following the instance generation method used in Gasse et al. (2019) and this paper, all instances have the same number of facilities. In addition, the customers' demands are generated such that the ratio between the total capacity of the facilities and the total demand of the customers remains almost constant across all instances. For set cover instances, the primary structural similarity is the number of subsets which is constant. In contrast, for combinatorial auction instances, on which LLB^{δ} models show the worst performances, there are no obvious structural similarities between instances of different difficulties.

7. Conclusion

This paper introduces lifted branching, a novel framework that iteratively improves branching strategies in mixed-integer linear programming through the lens of imitation learning. Using this approach, we empirically demonstrate the potential to improve upon established strategies such as reliability

branching and imitation learning of strong branching using graph convolutional neural network. Our iterative lifted approach allows offline computational resources to be invested in exploring and learning a new strategy that is a refined version of an existing one.

This contribution addresses a fundamental limitation in the current literature: previous works using imitation learning, relying on a handcrafted expert, generally strong branching, cannot lead to a better strategy than their oracle. This new framework allows for the iterative discovery of new experts while taking advantage of the success of imitation learning compared to reinforcement learning.

From a practical perspective, the learned lifted branching framework offers immediate value to Operations Research practitioners and industries that routinely solve recurring, structurally similar combinatorial problems. For example, energy grid operators solving daily unit commitment problems, or logistics companies managing capacitated facility locations, often face MILPs that share identical constraints and variable structures, differing only in daily parameters like load curves or customer demand. In these operational environments, the high offline computational cost required to generate the lifted branching dataset and train the ML model is easily justified, as it is amortized over thousands of instances over time. Ultimately, this framework enables practitioners to deploy highly specialized, problem-specific branching policies that yield faster solving times without requiring manual algorithmic tuning.

Despite these promising results, several avenues for improvement remain. First, the limited accuracy of our models in replicating the oracle's decisions suggests that the current graph convolutional neural network architecture may lack the expressiveness required for higher-order lifted strategies. Future work should investigate alternative architectures, such as graph attention networks or deeper message-passing layers, to better capture the complex look-ahead logic of the lifted oracle. Second, the current framework relies on training instances that are structurally similar to the test set. Developing techniques to transfer learned policies across heterogeneous problem classes would be a critical step toward a general-purpose neural branching strategy.

In summary, lifted branching provides a proof-of-concept that offline computational effort can be effectively distilled into efficient, real-time branching policies that can compete with standard state-of-the-art strategies.

8. Acknowledgments

Computational resources have been provided by the Consortium des Équipements de Calcul Intensif (CÉCI), funded by the Fonds de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under Grant No. 2.5020.11 and by the Walloon Region.

References

Abdullah, Z. N., Ahmad, I., and Hussain, I. (2018). Segment routing in software defined networks: A survey. *IEEE Communications Surveys & Tutorials*, 21(1):464–486.

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

Achterberg, T. and Wunderling, R. (2013). Mixed integer programming: Analyzing 12 years of progress. In Jünger, M. and Reinelt, G., editors, *Facets of Combinatorial Optimization: Festschrift for Martin Grötschel*, pages 449–481. Springer Berlin Heidelberg, Berlin, Heidelberg.

Alvarez, A. M., Louveaux, Q., and Wehenkel, L. (2017). A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1):185–195.

Applegate, D., Bixby, R. E., Chvátal, V., and Cook, W. J. (1995). Finding cuts in the TSP. Technical Report 95–05, DIMACS Center, Rutgers University, Piscataway, NJ, USA.

Balas, E. and Ho, A. (2009). Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. In *Combinatorial optimization*, pages 37–60. Springer.

Bénichou, M., Gauthier, J.-M., Girodet, P., Hentges, G., Ribière, G., and Vincent, O. (1971). Experiments in mixed-integer linear programming. *Mathematical programming*, 1(1):76–94.

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., Lentz, J., Manns, J., Mexi, G., Mühmer, E., Pfetsch, M. E., Schlösser, F., Serrano, F., Shinano, Y., Turner, M., Vigerske, S., Weninger, D., and Xu, L. (2024). The SCIP Optimization Suite 9.0. Technical report, Optimization Online.

Cai, J., Huang, T., and Dilkina, B. (2025). Multi-task representation learning for mixed integer linear programming. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 134–151. Springer.

Callebaut, H., De Boeck, J., and Fortz, B. (2023). Preprocessing for segment routing optimization. *Networks*, 82(4):459–478.

Carrion, M. and Arroyo, J. M. (2006). A computationally efficient mixed-integer linear formulation for the thermal unit commitment problem. *IEEE Transactions on Power Systems*, 21(3):1371–1378.

Carroll, P., Flynn, D., Fortz, B., and Melhorn, A. (2017). Sub-hour unit commitment milp model with benchmark problem instances. In *International Conference on Computational Science and Its Applications*, pages 635–651. Springer.

Chmiela, A., Khalil, E., Gleixner, A., Lodi, A., and Pokutta, S. (2021). Learning to schedule heuristics in branch and bound. *Advances in Neural Information Processing Systems*, 34:24235–24246.

Cornuéjols, G., Sridharan, R., and Thizy, J.-M. (1991). A comparison of heuristics and relaxations for the capacitated plant location problem. *European journal of operational research*, 50(3):280–297.

Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer.

Cunningham, P., Cord, M., and Delany, S. J. (2008). Supervised learning. In *Machine learning techniques for multimedia: case studies on organization and retrieval*, pages 21–49. Springer.

Du, S., Tong, J., and Fan, W. (2025). Learning efficient branch-and-bound for solving mixed integer linear programs. *Applied Soft Computing*, 172:112863.

Etheve, M., Alès, Z., Bissuel, C., Juan, O., and Kedad-Sidhoum, S. (2020). Reinforcement learning for variable selection in a branch and bound algorithm. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 176–185. Springer.

Gasse, M., Chételat, D., Ferroni, N., Charlin, L., and Lodi, A. (2019). Exact combinatorial optimization with graph convolutional neural networks. *Advances in neural information processing systems*, 32.

Golden, B., Wang, X., and Wasil, E. (2023). *The Evolution of the Vehicle Routing Problem—A Survey of VRP Research and Practice from 2005 to 2022*, pages 1–64. Springer Nature Switzerland, Cham.

He, H., Daume III, H., and Eisner, J. M. (2014). Learning to search in branch and bound algorithms. *Advances in neural information processing systems*, 27.

Hendel, G. (2022). Adaptive large neighborhood search for mixed integer programming. *Mathematical Programming Computation*, 14(2):185–221.

Hendel, G., Anderson, D., Le Bodic, P., and Pfetsch, M. E. (2022). Estimating the size of branch-and-bound trees. *INFORMS Journal on Computing*, 34(2):934–952.

Hoo, Z. H., Candlish, J., and Teare, D. (2017). What is an ROC curve? *Emergency Medicine Journal*, 34(6):357–359.

Karp, R. M. (2010). *Reducibility Among Combinatorial Problems*, pages 219–241. Springer Berlin Heidelberg, Berlin, Heidelberg.

Land, A. H. and Doig, A. G. (2010). *An Automatic Method for Solving Discrete Programming Problems*, pages 105–132. Springer Berlin Heidelberg, Berlin, Heidelberg.

Leyton-Brown, K., Pearson, M., and Shoham, Y. (2000). Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76.

Montero, L., Bello, A., and Reneses, J. (2022). A review on the unit commitment problem: Approaches, techniques, and resolution methods. *Energies*, 15(4):1296.

Parsonson, C. W., Laterre, A., and Barrett, T. D. (2023). Reinforcement learning for branch-and-bound optimisation using retrospective trajectories. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 4061–4069.

Petropoulos, F., Laporte, G., Aktas, E., Alumur, S. A., Archetti, C., Ayhan, H., Battarra, M., Bennell, J. A., Bourjolly, J.-M., Boylan, J. E., et al. (2024). Operational research: methods and applications. *Journal of the Operational Research Society*, 75(3):423–617.

Pomerleau, D. A. (1991). Efficient training of artificial neural networks for autonomous navigation. *Neural computation*, 3(1):88–97.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80.

Scavuzzo, L., Aardal, K., Lodi, A., and Yorke-Smith, N. (2024). Machine learning augmented branch and bound for mixed integer linear programming. *Mathematical Programming*, pages 1–44.

Scavuzzo, L., Chen, F., Chetelat, D., Gasse, M., Lodi, A., Yorke-Smith, N., and Aardal, K. (2022). Learning to branch with tree mdps. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A., editors, *Advances in Neural Information Processing Systems*, volume 35, pages 18514–18526. Curran Associates, Inc.

Strang, P., Alès, Z., Juan, O., Bissuel, C., Kedad-Sidhoum, S., and Rachelson, E. (2025). A markov decision process for variable selection in branch & bound. In *Advances in Neural Information Processing Systems 39 (NeurIPS 2025)*.

Yilmaz, K. and Yorke-Smith, N. (2021). A study of learning search approximation in mixed integer branch and bound: Node selection in scip. *Ai*, 2(2):150–178.

Appendix A. GCNN Architecture

This section describes the GCNN architecture used in this paper. This architecture was first introduced in Gasse et al. (2019).

Figure A.3 summarizes the architecture, which takes the bipartite graph as input. It consists of a convolution which, due to the bipartite graph structure, is divided into 2 steps. The first one gathers information from edges and variable nodes to update the constraint nodes. Then the second one gathers information from the edges and the constraint nodes to update the variable nodes.

This convolution can be written as:

$$k_i \leftarrow f_K \left(k_i, \sum_{j}^{(i,j)\in E} g_K(k_i, v_j, e_{i,j}) \right), \quad v_j \leftarrow f_V \left(v_j, \sum_{i}^{(i,j)\in E} g_V(k_i, v_j, e_{i,j}) \right)$$

g_C, g_V are 2-layers perceptrons that build the messages for a constraint node or a variable node. f_C, f_V are also 2-layers perceptrons that combine the original information with the aggregated message.

This convolution results in a graph with the same structure but different features on each node. A final 2-layer perceptrons is applied on each variable node. Finally a softmax function is applied only on nodes that represent candidate variables to get a probability distribution over the different candidates.

Appendix B. Configuration

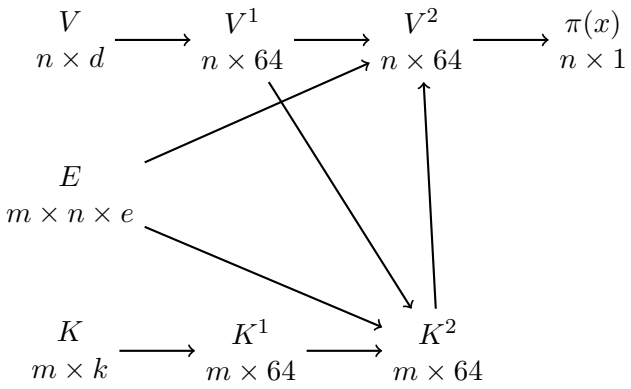

Figure A.3: GCNN Architecture Gasse et al. (2019)

Parameter	Description	Value
Ic	Number of generated instances	1
Gg	Number of days of horizon	10000
Breaks	Number of subdivisions for each day	24
Gmax	Number of thermal units	5
Cmax	Number of hydro cascade units	0
Difficulty	Difficulty level	1

Table B.8: List of parameters used to generate Unit commitment instances with the UCIG generator. The other parameters kept the default value.