Learning to Branch with Tree MDPs

Lara Scavuzzo
Delft University of Technology
l.v.scavuzzomontana@tudelft.nl

Feng Yang Chen
Polytechnique Montréal
feng-yang.chen@polymtl.ca

Didier Chételat
Polytechnique Montréal
didier.chetelat@polymtl.ca

Maxime Gasse
Mila, Polytechnique Montréal
maxime.gasse@polymtl.ca

Andrea Lodi
Jacobs Technion-Cornell Institute
Cornell Tech and Technion - IIT
andrea.lodi@cornell.edu

Neil Yorke-Smith
Delft University of Technology
n.yorke-smith@tudelft.nl

Karen Aardal
Delft University of Technology
k.i.aardal@tudelft.nl

Abstract

State-of-the-art Mixed Integer Linear Program (MILP) solvers combine systematic tree search with a plethora of hard-coded heuristics, such as the branching rule. The idea of learning branching rules from data has received increasing attention recently, and promising results have been obtained by learning fast approximations of the *strong branching* expert. In this work, we instead propose to learn branching rules from scratch via Reinforcement Learning (RL). We revisit the work of Etheve et al. [11] and propose tree Markov Decision Processes, or *tree MDPs*, a generalization of temporal MDPs that provides a more suitable framework for learning to branch. We derive a tree policy gradient theorem, which exhibits a better credit assignment compared to its temporal counterpart. We demonstrate through computational experiments that tree MDPs improve the learning convergence, and offer a promising framework for tackling the learning-to-branch problem in MILPs.

1 Introduction

Mixed Integer Linear Programs (MILPs) offer a powerful tool for modeling combinatorial optimization problems, and are used in many real-world applications [30]. The method of choice for solving MILPs to global optimality is the Branch-and-Bound (B&B) algorithm, which follows a divide-and-conquer strategy. A critical component of the algorithm is the *branching* rule, which is used to recursively partition the MILP search space using a space search tree. While there is little understanding of optimal branching decisions in general settings [25], the choice of the branching rule has a decisive impact on solving performance in practice [3]. In modern solvers, state-of-the-art performance is obtained using hard-coded branching rules that rely on heuristics designed by domain experts to perform well on a representative set of test instances [18].

In recent years, increasing attention has been given to approaches that use Machine Learning (ML) to obtain good branching rules and improve upon expert-crafted heuristics [5]. Such a statistical

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

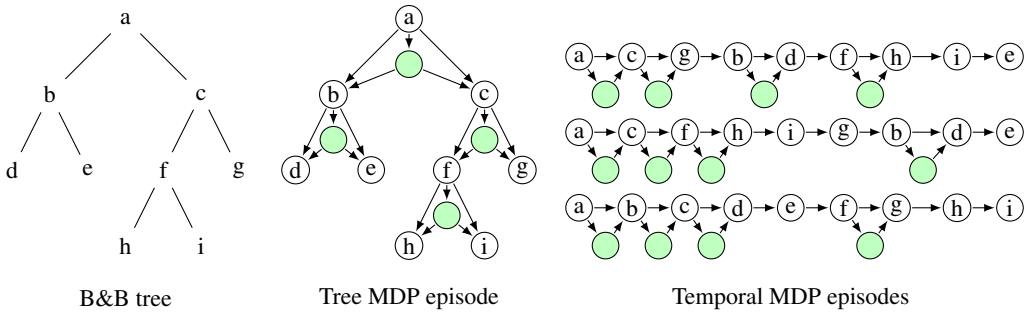


Figure 1: B&B process as a tree MDP episode vs. a temporal MDP episode. White nodes denote states, green nodes denote actions. In the tree MDP framework, the branching decision for splitting a node f is credited two rewards, (r_h, r_i). In the temporal MDP framework, the same branching decision is credited with additional rewards which depend on the temporal order in which B&B nodes are processed, (r_h, r_i, r_e), $(r_h, r_i, r_g, r_b, r_d, r_e)$, or (r_g, r_h, r_i).

approach is particularly well-suited in situations where similar problems are solved repeatedly, which is a common industrial scenario. In such cases, a generic and problem-agnostic branching rule might be suboptimal. Due to the sequential nature of branching, a natural paradigm to formulate the learning problem is Reinforcement Learning (RL), which allows to directly search for the optimal branching rule with respect to a metric of interest, such as average solving time, or final B&B tree size.

Etheve et al. [11] show that, when using depth-first-search (DFS) as the tree exploration (a.k.a. node selection) policy in B&B, minimizing each subtree size is equivalent to minimizing the global tree size. Using this result, they propose an efficient value-based RL procedure for learning to branch. In this paper we build upon the work of Etheve et al. [11], and we propose a generalization of the Markov Decision Process (MDP) paradigm that we call *tree MDP* (illustrated in Figure 1, formally discussed in Section 4). This general formulation captures the key insight of their approach, but opens the door for more general B&B settings, RL algorithms and reward functions.

In particular, we show that DFS is just one way to enforce the "tree Markov" property, and we propose an alternative, more practical solution that simply relies on providing the optimal objective limit to the solver during training. Our contribution is three-fold:

1. We introduce the concepts of a tree MDP and of the tree Markov property, and we derive a policy gradient theorem for tree MDPs.
2. We propose an alternative way to enforce the tree Markov property in learning to branch by imposing an optimal objective limit, which is less computationally demanding than DFS [11].
3. We show that adopting the tree MDP paradigm improves both the convergence speed of RL and the quality of the resulting branching rules, compared the regular MDP paradigm.

The remainder of this paper is organized as follows. We formally introduce MILPs, B&B and MDPs in Section 2. In Section 3 we motivate the need for moving beyond imitation learning in learning to branch, and discuss the challenges it raises. In Section 4 we introduce the concepts of tree MDPs and the tree Markov property, we derive a policy gradient theorem for tree MDPs, and we present a convenient solution to enforce the tree Markov property in B&B without DFS [11]. In Section 5 we conduct, for the first time, a thorough computational evaluation of RL for learning branching rules, on five synthetic MILP benchmarks with two difficulty levels. Finally, we discuss the significance of our results and future directions in Section 6.

2 Background

In this section, we describe the Branch-and-Bound (B&B) algorithm, and we show how the branching problem naturally formulates as a (temporal) Markov Decision Process.

2.1 The B&B algorithm

Consider a Mixed Integer Linear Program instance as an optimization problem of the form

$$\min_{x \in \mathbb{R}^n} \{c^T x : Ax \leq b,\ l \leq x \leq u,\ x_i \in \mathbb{Z}\ \ \forall i \in \mathcal{I}\}, \tag{1}$$

where $c \in \mathbb{R}^n$ is the objective coefficient vector, $A \in \mathbb{R}^{m \times n}$ is the constraint matrix, $b \in \mathbb{R}^m$ is the constraint right-hand-side, $l, u \in \mathbb{R}^n$ are the lower and upper bound vectors, respectively, and $\mathcal{I} \subseteq \{1, 2, ..., n\}$ is the subset of variables constrained to take integer values. Relaxing the integrality constraints, $x_i \in \mathbb{Z}\ \forall i \in \mathcal{I}$, yields a Linear Program (LP) that can be solved efficiently in practice, for example by using the Simplex algorithm. Solving such a relaxation yields an LP solution $\hat{x}^\star$, and provides a lower bound $c^T \hat{x}^\star$ to the original problem (1). If by chance $\hat{x}^\star$ satisfies the integrality constraints, $\hat{x}_i^\star \in \mathbb{Z}\ \forall i \in \mathcal{I}$, then it is also an optimal solution to (1). If, on the contrary, there is at least one variable $x_i, i \in \mathcal{I}$ such that $\hat{x}_i^\star$ is not an integer, then one can split the feasible region into two sub-problems, by imposing

$$x_i \leq \lfloor \hat{x}_i^\star \rfloor \qquad \text{or} \qquad x_i \geq \lceil \hat{x}_i^\star \rceil . \tag{2}$$

This partitioning step is known as branching. In its most basic form, the vanilla B&B algorithm recursively applies branching, thereby building a search space tree where each node has an associated local sub-MILP, with a local LP solution and associated local lower bound.[1] If the local LP solution satisfies the MILP integrality constraints, then it is termed a *feasible solution*, and it also provides an upper bound to (1). At any given time of the B&B process, a Global Lower Bound (GLB) to the original MILP can be obtained by taking the lowest of local lower bounds of the leaf nodes of the tree. Similarly, a Global Upper Bound (GUB) can be obtained by taking the lowest of the upper bounds so far.[2] The B&B process terminates when no more B&B leaf node can be partitioned with (2), i.e., all leaf nodes satisfy one of these conditions: the local LP has no solution (infeasible); the local lower bound is above the GUB (pruned); or the local LP solution satisfies the MILP integrality constraints (integer feasible). At termination, we have *GLB* = *GUB*, and the original MILP (1) is solved.

The branching problem, a.k.a. variable selection problem, is the task of choosing, at every B&B iteration, the variable x_i that will be used to generate a partition of the form (2). While there is no universal metric to measure the quality of a branching rule, a common performance measure is the final size of the B&B tree (the smaller the better) [1].

2.2 Temporal MDPs

In the following, upper-case letters in italics denote random variables (e.g. S, A), while their lower-case counterparts denote their value (e.g. s, a) and their calligraphic counterparts their domain (e.g., $s \in \mathcal{S}, a \in \mathcal{A}$). We consider episodic Markov Decision Processes (MDPs) of the form $M = (\mathcal{S}, \mathcal{A}, p_{init}, p_{trans}, r)$, with states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, initial state distribution $p_{init}(s_0)$, state transition distribution $p_{trans}(s_{t+1}|s_t, a_t)$, and reward function $r : \mathcal{S} \to \mathbb{R}$. For simplicity, we assume finite episodes of length T, that is, $\tau = (s_0, a_0, \ldots, s_T)$, $|\tau| = T$.[3] Together with a control mechanism represented by a stochastic policy $\pi(a_t|s_t)$, the MDP defines a probability distribution over trajectories, namely

$$p_\pi(\tau) = p_{init}(s_0) \prod_{t=0}^{|\tau|-1} \pi(a_t|s_t) p_{trans}(s_{t+1}|s_t, a_t).$$

The MDP control problem is to find a policy that maximizes the expected cumulative reward, $\pi^\star \in \arg\max_\pi V^\pi$, with

$$V^\pi := \mathop{\mathbb{E}}_{\tau \sim p_\pi} \left[\sum_{t=0}^{|\tau|} r(s_t) \right]. \tag{3}$$

A key property of MDPs is the temporal Markov property $S_{t+1} \perp\!\!\!\perp S_{<t} \mid S_t, A_t, \forall t$, which guarantees that the future only depends upon the current state and action. One consequence of this property is

[1] The local lower bound is ∞ if the LP is infeasible.

[2] The local upper bound is ∞ if the LP is infeasible, or if its solution is not MILP feasible.

[3] Equivalently, we can assume all trajectories will reach an absorbing, zero-reward state s_{null} in a finite number of steps, that is, $p_{trans}(s_{t+1}|s_t = s_{null}, a_t) = \delta_{s_{null}}(s_{t+1})$, $r(s_{null}) = 0$, and $\exists T$ s.t. $p_\pi(s_T = s_{null}) = 1, \forall \pi$.

the temporal policy gradient theorem [34], which forms the basis of policy gradient methods

$$\nabla_\pi V^\pi = \mathop{\mathbb{E}}_{\tau \sim p_\pi} \left[\sum_{t=0}^{|\tau|-1} \nabla_\pi \log \pi(a_t|s_t) \sum_{t'=t+1}^{|\tau|} r(s_{t'}) \right]. \tag{4}$$

2.3 The branching temporal MDP

Let us now consider the problem of learning a branching policy in a B&B solver. As noted by [21, 17], branching decisions are made sequentially, thus the problem can naturally be regarded as a temporal MDP. In this form, the states s_t consist of the entire state of the B&B process (that is, the solver) at time t, which includes the whole tree structure, all sub-MILPs and LP solutions, and all upper and lower bounds. The actions a_t are the branching decisions, and the reward is chosen so that the return matches an objective function of interest (e.g., the final B&B tree size).

While appealing by its simplicity, we argue that this formulation is not practical for two reasons. First, even in the simplest textbook implementation of B&B, the MDP states are complex objects of growing size, which are impractical to handle. Second, episode length in branching can grow extremely large, in the worst case exponentially with the size of the problem. This exacerbates the so-called *credit assignment problem*, i.e., the problem of determining which actions should be given credit for a certain outcome (see Section 3.2 for a more detailed discussion). In Section 4, we will we will show how the tree MDP formulation addresses both those challenges.

3 Approaches to learning to branch

Approaches that frame branching as a learning problem have gained significant attention recently. In this section, we review some common trends in the field and identify some key challenges.

3.1 Learning to branch with imitation learning

In recent years, it has been demonstrated that efficient branching rules can be obtained by learning fast approximations of *strong branching*, a globally effective but computationally expensive rule [23, 26, 20, 17, 19, 36, 29]. This approach can be framed as imitation learning, a well-known method for learning in MDPs when an expert is available. While this approach can lead to improvements over state-of-the-art branching rules on various benchmarks, it also has several drawbacks.

First, strong branching implementations are known to trigger side effects (such as early detection of an infeasible child) that do not map well within the branching MDP framework [14]. This means that branching decisions collected from the strong branching expert might not line up with the environment of the learning agent, which might result in a performance gap between the learning agent and the expert, even if the agent manages to successfully reproduce the expert decisions.

Second, other than these side-effects, strong branching relies on dual bound improvements to make branching decisions. This can be ineffective in problems where the LP relaxation is not very informative or suffers from dual degeneracy, as pointed out in Gamrath et al. [16]. As an extreme case, Dey et al. [9] provide an example where a strong-branching based B&B tree can have exponentially more nodes than the tree obtained with a problem-specific rule.

Finally, regardless of the expert quality, obtaining strong branching samples can become prohibitively expensive for large instances. This means that non-trivial engineering solutions and scaling strategies are needed to allow training on larger problems, such as heavy computational parallelization [29].

3.2 Learning to branch with reinforcement learning

The reasons listed above suggest the need for an alternative approach to finding a good branching policy; an approach that does not rely on the strong branching rule. One could perform imitation learning on another expert, but no other plausible imitation target is known, and the same performance ceiling issue would remain. A natural alternative is reinforcement learning, an approach that aims to find an optimal policy in a Markov Decision Process, with respect to any desired objective function. But despite the theoretical appeal of RL for learning to branch, it also comes with its own challenges.

First, common evaluation metrics in B&B, such as solving time or final tree size, are inconvenient for RL. Both require to run episodes to completion, that is, to solve MILP instances to optimality, which leads to very long episodes even for moderately hard instances. Second, in contrast to many RL tasks studied in the literature, in B&B the worse the policy is, the longer are the episodes. These two factors combined give rise to the following problems:

1. Collecting training data is computationally expensive. In particular, training from scratch from a randomly initialized policy can be prohibitive for large MILPs.
2. Due to the length of the episodes, training signals are particularly sparse. This exacerbates the so-called *credit assignment problem* [27], i.e., the problem of determining which actions should be given credit for a certain outcome.

Two works have tried to tackle the branching problem with RL so far. Sun et al. [33] propose an approach based on evolution strategies, a variant of RL, combined with a novelty bonus based on discrepancies in B&B trees. They show improvements over an imitation learning approach Gasse et al. [17] on instances derived from a common backbone graph, but fail to improve on more heterogeneous ones. These results emphasize the difficulty of a direct application of RL on hard instance sets.

In parallel, Etheve et al. [11] recently made a contribution that directly addresses the credit assignment problem. They show that, when depth-first-search (DFS) is used as the node selection strategy, minimization of the total B&B tree size can be achieved by taking decisions that minimize subtree size at each node. Based on this result, they propose a Q-learning-type algorithm [28] where the learned Q-function approximates the local subtree size. They report improvements over a state-of-the-art branching rule on collections of small, fixed-size instances. However, they only evaluate the learned policy with DFS node selection, which matches their training setting, but is not a realistic B&B setting. In Section 4, we will show that the method proposed by Etheve et al. [11] can be interpreted as a specific instantiation of a more general *tree MDP* framework, which effectively simplifies the credit assignment problem in learning to branch (point 2 above). Furthermore, we propose an alternative condition to the one of Etheve et al. [11] to ensure a tree MDP setting, which results in shorter episodes, hence reducing the cost of data collection (point 1 above).

4 Branching as a tree MDP

We now detail our tree MDP framework, and show how the branching problem can be cast as a tree MDP control problem, under some conditions. Proofs as well as a side-by-side comparison of temporal and tree MDPs are deferred to the Supplementary Material.

4.1 Tree MDPs

We define tree MDPs as augmented Markov Decision Processes $tM = (\mathcal{S}, \mathcal{A}, p_{init}, p^-_{ch}, p^+_{ch}, r, l)$, with states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, initial state distribution $p_{init}(s_0)$, respectively left and right child transition distributions $p^-_{ch}(s_{ch^-_i}|s_i, a_i)$ and $p^+_{ch}(s_{ch^+_i}|s_i, a_i)$[4], reward function $r : \mathcal{S} \to \mathbb{R}$ and leaf indicator $l : \mathcal{S} \to \{0, 1\}$. The central concept behind tree MDPs is that each non-leaf state s_i (i.e., such that $l(s_i) = 0$), together with an action a_i, produces two new states $s_{ch^-_i}$ (its left child) and $s_{ch^+_i}$ (its right child). As a result, the tree MDP generative process results in episodes τ that follow a tree structure (see Figure 1), where leaf states (i.e., such that $l(s_i) = 1$) are the leaf nodes of the tree, below which no action can be taken and no children state will be created. For simplicity, just like in Section 2.2, we assume the tree-like trajectories have some finite size that we denote $T = |\tau|$.

A tree MDP episode τ consists of a binary[5] tree with nodes $\mathcal{N} = \{0, \dots, |\tau|\}$ and leaf nodes $\mathcal{L} = \{i | i \in \mathcal{N}, l(s_i) = 1\}$, which embeds a state s_i at every node $i \in \mathcal{N}$ and an action a_i at every non-leaf node $i \in \mathcal{N} \setminus \mathcal{L}$. For convenience, in the following we will use pa_i, ch^-_i and ch^+_i to denote the nodes that are respectively parent, left child and right child of a node i if any, as well as d_i and nd_i to denote respectively the set of all descendants and non-descendants of a node i in the tree. Together

[4] In the case of branching, these transitions are deterministic.

[5] The concept can easily be extended to non-binary trees.

with a control mechanism $\pi(a_t|s_t)$, a tree MDP defines a probability distribution over trajectories,

$$p_\pi(\tau) = p_{init}(s_0) \prod_{i \in \mathcal{N} \setminus \mathcal{L}} \pi(a_i|s_i) p^-_{ch}(s_{ch_i^-}|s_i, a_i) p^+_{ch}(s_{ch_i^+}|s_i, a_i).$$

As in temporal MDPs, the tree MDP control problem is to find a policy that maximizes the expected cumulative reward, as defined by (3). Due to their specific generative process, a key characteristic of tree MDPs is the *tree Markov property* $S_{ch_i^-}, S_{ch_i^+} \perp\!\!\!\perp S_{nd_i} \mid S_i, A_i, \forall i$, which guarantees, similarly to the temporal Markov property, that each subtree only depends upon the immediate state and action. This again simplifies the credit assignment problem in RL and results in an efficient *tree policy gradient* formulation.

Proposition 4.1. *For any tree MDP tM, the policy gradient can be expressed as*

$$\nabla_\pi V^\pi = \mathop{\mathbb{E}}_{\tau \sim p_\pi} \left[\sum_{i \in \mathcal{N} \setminus \mathcal{L}} \nabla_\pi \log \pi(a_t|s_t) \sum_{j \in d_i} r(s_j) \right]. \tag{5}$$

4.2 The branching tree MDP

We now show how and under which conditions the vanilla B&B algorithm can be formulated as a tree MDP. We consider episodes τ that follow exactly the B&B tree structure. Each node i in the tree embeds a state $s_i = (MILP_i, GUB_i)$, where $MILP_i$ is the local sub-MILP of the node, and GUB_i is the global upper bound at the time the node is processed[6]. Each non-leaf node also embeds an action $a_i = (j, x^\star_j)$, where j is the index of the branching variable chosen by B&B, and $x^\star_j$ is the value used to branch ($x_j \leq \lfloor x^\star_j \rfloor \vee x_j \geq \lceil x^\star_j \rceil$). Note that such states and actions, embedded in the B&B tree, carry enough information to unroll a vanilla B&B algorithm, as described in Section 2.1. We now need to make two additional assumptions in order to formulate branching as a tree MDP.

4.2.1 B&B tree transitions

First, and this is our main requirement, B&B state transitions must decompose into p^-_{ch} and p^+_{ch}.

Assumption 4.2. For every non-leaf node i, the global upper bounds $GUB_{ch_i^-}$ and $GUB_{ch_i^+}$ (reached by B&B when the left and right child is processed, respectively) can be derived solely from the current state and action, (s_i, a_i).

Proposition 4.3. *A vanilla B&B algorithm that satisfies Assumption 4.2 forms a tree MDP.*

Assumption 4.2 is not always satisfied, as the following counterexample shows.

Counter example. Consider the root problem $MILP_0 = \min x \; s.t. \; x \geq 0.6, \; x \in \mathbb{Z}$, with upper bound $GUB_0 = \infty$. The root LP solution is $\hat{x}^\star = 0.6$, and the two sub-problems $MILP_{ch_i^-}$ and $MILP_{ch_i^+}$ follow from the (only) branching decision $x \leq 0 \vee x \geq 1$. Now, the two global upper bound $GUB_{ch_i^-}$ and $GUB_{ch_i^+}$ depend on whether the feasible solution $x = 1$ has been found in the past, which in turn depends on the node processing order. Going left first ($-$) will yield $(GUB_{ch_i^-}, GUB_{ch_i^+}) = (\infty, \infty)$, while going right first ($+$) will yield $(GUB_{ch_i^-}, GUB_{ch_i^+}) = (1, \infty)$.

We now provide two conditions under which Assumption 4.2 is true.

Proposition 4.4. *In Optimal Objective Limit B&B (ObjLim B&B), that is, when the optimal solution value of the MILP is known at the start of the algorithm ($GUB_0 = GUB^\star$), Assumption 4.2 holds.*

Proposition 4.5. *In Depth-First-Search B&B (DFS B&B), that is, when nodes are processed depth-first and left-first by the algorithm, Assumption 4.2 holds.*

Propositions 4.4 and 4.5 provide two viable options for turning vanilla B&B into a tree MDP, where p^-_{ch} and p^+_{ch} are deterministic functions. The first variant, *ObjLim*, requires MILP instances used for training to be solved to optimality once, in order to collect their optimal objective value. The second variant, *DFS*, corresponds to the setting in [11]. In this variant there is no need to pre-solve training instances to optimality, however it is expected that the collected episodes might be longer than with a standard node selection rule, which might result in slower training.

[6] Recall that in B&B, the GUB corresponds to the value of the best feasible solution found so far, or ∞ when no solution has been found yet.

4.2.2 B&B tree reward

Last, for branching to formulate as a control problem in a tree MDP, the objective must be compatible.

Assumption 4.6. The branching objective can be decomposed over the nodes of the B&B tree, with a state-based reward function $r : \mathcal{S} \to \mathbb{R}$.

Interestingly, a natural objective for branching is the final *B&B tree size*, which expresses naturally as $r(s_i) = -1$. Thus, it is trivially compatible with Assumption 4.6. We will consider this reward in our experiments. Another common objective is the total solving time, which can also be expressed as a state-based reward $r : \mathcal{S} \to \mathbb{R}$ under mild assumptions. Indeed, it suffices to considers that solving LP relaxations and making branching decisions at each node are the main contributing factors in the total running time, while other algorithmic components have a negligible cost. In vanilla B&B, both these components only depend on the local state $s_i = (\mathit{MILP}_i, \mathit{GUB}_i)$ of each node.

4.3 Efficiency of tree MDP

Tree MDPs, when applicable, provide a convenient alternative to temporal MDPs for tackling the branching problem. First, the tree Markov property implies that branching policies in tree branching MDPs will not benefit from any information other than the local state $s_i = (\mathit{MILP}_i, \mathit{GUB}_i)$ to make optimal decisions, similarly to how control policies in temporal MDP can ignore past states and rely only on the immediate state s_t. Second, the credit assignment problem in the branching tree MDP is more efficient than in the equivalent temporal MDP. This is showcased in Figure 1, and stems from the fact that in a tree MDP all the descendants of a node i are necessarily processed after that node temporally. As a consequence, the rewards credited to an action in the tree policy gradient (5), $\sum_{j \in d_i} r(s_j)$, are necessarily a subset of the rewards credited to the same action in the temporal policy gradient (4), $\sum_{t' > t} r(s_{t'})$. Thus, it can be expected intuitively that learning branching policies within the tree MDP framework will be easier, and more sample-efficient than learning within the temporal MDP framework. We will validate this hypothesis experimentally in Section 5.

4.4 Theoretical limitations

Our proposed B&B variants, *ObjLim* and *DFS*, allow for a nice formulation of the branching problem as a tree MDP, which we argue is key to unlocking a more practical and sample-efficient learning of branching policies. However, usually the end goal is to learn a branching policy that performs well in realistic B&B settings, and the fact that a branching policy performs well in one of those variants does not guarantee that it will perform well in the vanilla setting also. This discrepancy between the training environment and the evaluation environment is a recurring problem in RL, and is more generally referred to as the transfer learning problem. While there exist solutions to mitigate this problem, in this paper we leave the question aside and simply assume that the transfer problem is negligible. We thus directly report the performance obtained from each training setting in the realistic evaluation setting, a default B&B solver.

4.5 Connections with hierarchical RL

The tree MDP formulation has connections with hierarchical RL [10], a paradigm typically applied in the case where the learning task can be decomposed into a set of simpler tasks which can be solved recursively, independently of the parent task. An example of this is picking up and delivering a package from a point A to a point B, which decomposes into: move to A, pick up, move to B, drop. Both tree MDP and hierarchical RL exploit a decomposition of the task in order to simplify the credit assignment problem. However, the two methods also differ on several points. First, in hierarchical RL, the sub-task decomposition must be given a priori by the user for each new task, and is fixed with a limited depth, while in tree MDPs the decomposition holds by construction, and can be applied recursively for virtually infinite depths. Second, the sub-tasks are different (with different reward and optimal policy), while in tree MDPs the reward remains the same. Lastly, all sub-task states are processed consecutively, while in tree MDPs the temporal processing order of states can vary, and switches between different sub-trees are allowed.

Algorithm 1 REINFORCE training loop

Input: training set of MILP instances and their pre-computed optimal solution $\mathcal{D}$, maximum number of epochs K, time limit ζ, entropy bonus λ, learning rate α, sample rate β.
Initialize policy π_θ with random parameters θ.
for *epoch* from 1 to K **do**
  if elapsed time $> \zeta$ **then** break
  Sample 10 MILP instances from $\mathcal{D}$
  for each sampled instance **do**
    Collect one episode τ by running B&B to optimality
    Extract randomly $\beta \times |\tau|$ state, action, return tuples (s, a, G) from τ (with G the local subtree size for tree MDPs, and the remaining episode size for MDPs)[8]
  end for
  $n \leftarrow$ number of collected tuples, $L \leftarrow 0$
  for each collected tuple (s, a, G) **do**
    $L \leftarrow L - G\frac{1}{n}\log \pi_\theta(a|s)$ # policy gradient cost
    $L \leftarrow L - \lambda\frac{1}{n}H(\pi_\theta(\cdot|s))$ # entropy bonus
  end for
  $\theta \leftarrow \theta - \alpha\nabla_\theta L$
end for
return π_θ

5 Experiments

We now compare the performance of four machine learning approaches: the imitation learning method of Gasse et al. [17], and three different RL methods. We also compare against SCIP's default rule (for a description of this rule we refer to the Supplementary Material A.4). Code for reproducing all experiments is available online [7].

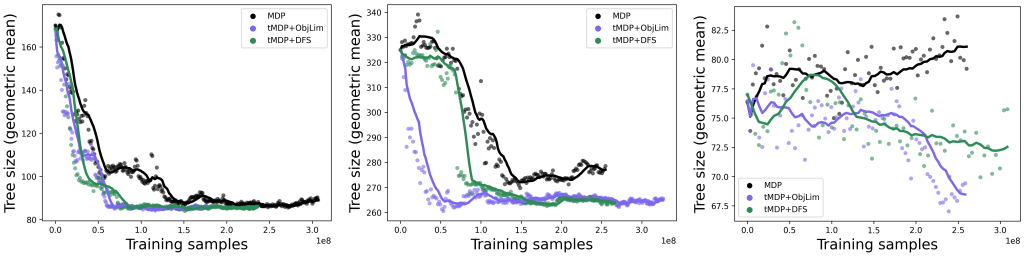
5.1 Setup

Benchmarks Similarly to Gasse et al. [17], we train and evaluate each method on five NP-hard problem benchmarks, which consist of synthetic combinatorial auctions, set covering, maximum independent set, capacitated facility location and multiple knapsack instances. For each benchmark we generate a training set of 10,000 instances, along with a small set of 20 validation instances for tracking the RL performance during training. For the final evaluation, we further generate a test set of 40 instances, the same size as the training ones, and also a transfer set of 40 instances, larger and more challenging than the training ones. More information about benchmarks and instance sizes can be found in the Supplementary Material (A.5).

Training We use the Graph Neural Network (GNN) from Gasse et al. [17] to learn branching policies, with the same features and architecture. This state representation has been shown to provide an efficient encoding of the local MILP together with some global features. Notice that this makes the MDP formulation into a POMDP, given that we do not encode the complete search tree (see Section 2.3). We compare four training methods: imitation learning from strong branching (IL); RL using temporal policy gradients (MDP); RL using tree policy gradients with DFS as a node selection strategy (tMDP+DFS), which enforces the tree Markov property due to Proposition 4.5; and RL using tree policy gradients with the optimal objective value set as an objective limit (tMDP+ObjLim), which corresponds to Propositions 4.4. Other than that, we use default solver parameters, except for restarts and cutting planes after the root node which are deactivated. We use a plain REINFORCE [35] with entropy bonus as our RL algorithm, for simplicity. Our training procedure is summarized in Algorithm 1. We set a maximum of 15,000 epochs and a time limit of six days for training. Our implementation uses PyTorch [31] together with PyTorch Geometric [12], and Ecole [32] for interfacing to the solver SCIP [15]. All experiments are run on compute nodes equipped with a GPU.

[7] `https://github.com/lascavana/rl2branch`

[8] Notice that for tree MDPs the computation of the return for each node can be computed efficiently with a bottom-up traversal that runs in $O(n)$.



(a) Combinatorial Auctions (b) Set Covering (c) Maximum Independent Set

Figure 2: Training curves for REINFORCE with temporal policy gradients (MDP), tree policy gradients with objective limit (tMDP+ObjLim) and DFS node selection (tMDP+DFS). We report the final B&B tree size on the validation set (geometric mean over 20 instances $\times$ 5 seeds, the lower the better), versus the number of processed training samples on the x-axis. Solid lines show the moving average. Training curves for the remaining benchmarks can be found in the Supplementary Material (A.3).

Evaluation For each branching rule evaluated, we solve every validation, test or transfer instance 5 times with a different random seed. We use default solver parameters, except for restarts and cutting planes after the root node which are deactivated (same as during training), and a time limit of 1 hour for each solving run. For tMDP+DFS and tMDP+ObjLim, the specific settings used during training (DFS node selection and optimal objective limit, respectively) are not used any more, thus providing a realistic evaluation setting. We report the geometric mean of the final B&B tree size as our metric of interest, as is common practice in the MILP literature [3]. We pair this with the average per-instance standard deviation (in percentage). We only consider solving runs that finished successfully for all methods, as in [17]. Extended results including solving times are provided in the Supplementary Material (A.3).

5.2 Results

Figure 2 showcases the convergence of our three RL paradigms MDP, tMDP+ObjLim and tMDP+DFS during training, in terms of the final B&B tree size on the validation set (the lower the better). In order to better highlight the sample efficiency of each method, we report on the x-axis the cumulative number of collected training samples, which correlates with the length of the episodes collected during training. This provides a hardware-independent proxy for training time. As can be seen, the tree MDP paradigm clearly improves the convergence speed on these three benchmarks, with a clear domination of tMDP+ObjLim on one benchmark (Set Covering). Training curves for the remaining benchmarks are available in the Supplementary Material (A.3).

Table 1 reports the final performance of the branching rules obtained with each method, on both a held-out test set (same instance difficulty as training) and a transfer set (larger, more difficult instances than training). Despite a mismatch between the training and evaluation environments, which is required to enforce the tree Markov property, the tree MDP paradigm consistently produces equal or better branching rules than the temporal MDP paradigm on all 5 benchmarks.

On one benchmark, Multiple Knapsack, the branching rules learned by RL outperform both SCIP's default branching rule and the strong branching imitation (IL) approach. The likely reason is that the MILP formulation of Multiple Knapsack provides a very poor linear relaxation, which often results in no dual bound improvement after branching. This means that strong branching scores are in most cases not discriminative, which is problematic for rules that heavily rely on this criterion (see Section 3.1), such as SCIP's default or a policy that imitates strong branching. This situation makes a strong case for the potential of RL-based methods, which can adapt and devise alternative branching strategies.

On the remaining 4 benchmarks, however, RL methods perform worse than SCIP default or IL, despite being based on the same GNN architecture. This illustrates the difficulty of learning to branch via RL, even on small-scale problems, and the remaining room for improvement. Additional evaluation criteria (solving times and number of time limits) are available in the Supplementary Material (A.3).

Table 1: Solving performance of the different branching rules in terms of the final B&B tree size (lower is better). We evaluate each method on a test set with instances the same size as training, and a transfer set with larger instances. We report the geometric mean and standard deviation over 40 instances, solved 5 times with different random seeds, and we bold the best of the RL methods.

Model	Comb. Auct.	Set Cover	Max.Ind.Set	Facility Loc.	Mult. Knap.
SCIP default	7.3±39%	10.7±24%	19.3±52%	203.6±63%	267.8±96%
IL	52.2±13%	51.8±10%	35.9±36%	247.5±39%	228.0±95%
RL (MDP)	86.7±16%	196.3±20%	91.8±56%	393.2±47%	143.4±76%
RL (tMDP+DFS)	**86.1±17%**	**190.8±20%**	89.8±51%	360.4±46%	**135.8±75%**
RL (tMDP+ObjLim)	87.0±18%	193.5±23%	**85.4±53%**	**325.4±41%**	142.4±78%

Test

Model	Comb. Auct.	Set Cover	Max.Ind.Set	Facility Loc.	Mult. Knap.
SCIP default	733.9±26%	61.4±19%	2867.1±35%	344.3±57%	592.3±75%
IL	805.1±9%	145.0±6%	1774.8±38%	407.8±37%	1066.1±101%
RL (MDP)	1906.3±18%	853.3±27%	2768.5±76%	679.4±52%	518.4±79%
RL (tMDP+DFS)	**1804.6±17%**	**816.8±25%**	2970.0±76%	609.1±47%	495.1±81%
RL (tMDP+ObjLim)	1841.9±18%	826.4±26%	**2763.6±74%**	**496.0±48%**	**425.3±64%**

Transfer

6 Conclusions and Future Directions

This paper adds to a growing body of literature on using ML to assist decision-making in several key components of the B&B algorithm (see e.g. [5, 7, 22]). We contribute to the study of RL as a tool for learning to branch in MILP solvers. We present tree MDP, a variant of Markov Decision Processes, and show that under some conditions, the B&B branching process is tree-Markovian. We show that the approach of Etheve et al. [11] can be naturally cast as Q-learning for tree MDPs, and we propose an alternative, more computationally appealing way to enforce the tree Markov property in B&B, using optimal objective limits. Finally, we evaluate for the first time a variety of RL-based branching rules in a comprehensive computational study, and we show that tree MDPs improve the convergence speed of RL for branching, as well as the overall performance of the learnt branching rules.

These contributions bring us closer to learning efficient branching rules from scratch using RL, which could ultimately outperform existing branching heuristics built upon decades of expert knowledge and experiment. However, despite the convergence speed-up that our method provides, training without expert knowledge remains very computationally heavy and in general still results in worse performance than its supervised learning counterpart, which reveals a significant gap that must be closed. As future work, we would like to explore ideas to keep improving sample efficiency, and generalization across instance size. This is necessary for RL to scale to larger, non-homogeneous benchmarks, such as MIPLIB [18], which at the moment remain out-of-reach for RL.

Finally, although our concern in this paper is focused on improving variable selection for MILP, our tree MILP construction could be useful in other applications. Branch-and-bound is a type of divide-and-conquer algorithm, and we expect that, in general, this framework can be applied to any problem where one seeks to control such algorithms more efficiently. Examples would include controlling the order in which a rover explores rooms in a building, or selecting the pivots in a quicksort algorithm.

Acknowledgements

The authors acknowledge the generous support of this research. In particular, the work of Karen Aardal, Lara Scavuzzo and Neil Yorke-Smith was partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under grant number 952215, and by The Netherlands Organisation for Scientific Research, NWO, Grant OCENW.GROOT.2019.015. The work of Feng Yang Chen, Didier Chételat, Maxime Gasse and Andrea Lodi was supported by the Canada Excellence Research Chair program.

References

[1] Tobias Achterberg. *Constraint Integer Programming*. PhD thesis, Technischen Universität Berlin, 2007.

[2] Tobias Achterberg and Timo Berthold. Hybrid branching. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 309–311. Springer, 2009.

[3] Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. In *Facets of Combinatorial Optimization*, pages 449–481. Springer, 2013.

[4] Egon Balas and Andrew Ho. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. In *Combinatorial Optimization*, pages 37–60. Springer, 1980.

[5] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 2020.

[6] David Bergman, Andre A Cire, Willem-Jan Van Hoeve, and John Hooker. *Decision diagrams for optimization*, volume 1. Springer, 2016.

[7] Antonia Chmiela, Elias B Khalil, Ambros Gleixner, Andrea Lodi, and Sebastian Pokutta. Learning to schedule heuristics in branch-and-bound. *arXiv preprint arXiv:2103.10294*, 2021.

[8] Gérard Cornuéjols, Ranjani Sridharan, and Jean-Michel Thizy. A comparison of heuristics and relaxations for the capacitated plant location problem. *European Journal of Operational Research*, 50(3):280–297, 1991.

[9] Santanu S Dey, Yatharth Dubey, Marco Molinaro, and Prachi Shah. A theoretical and computational analysis of full strong-branching. *arXiv preprint arXiv:2110.10754*, 2021.

[10] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[11] Marc Etheve, Zacharie Alès, Côme Bissuel, Olivier Juan, and Safia Kedad-Sidhoum. Reinforcement learning for variable selection in a branch and bound algorithm. In *CPAIOR*, 2020.

[12] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[13] Alex S Fukunaga. A branch-and-bound algorithm for hard multiple knapsack problems. *Annals of Operations Research*, 184(1):97–119, 2011.

[14] Gerald Gamrath and Christoph Schubert. Measuring the impact of branching rules for mixed-integer programming. In *Operations Research Proceedings 2017*, pages 165–170. Springer, 2018.

[15] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Pierre Le Bodic, Stephen J. Maher, Frederic Matter, Matthias Miltenberger, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 7.0. ZIB-Report 20-10, Zuse Institute Berlin, 3 2020.

[16] Gerald Gamrath, Timo Berthold, and Domenico Salvagnin. An exploratory computational analysis of dual degeneracy in mixed-integer programming. *EURO Journal on Computational Optimization*, 8(3):241–261, 2020.

[17] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 15580–15592, 2019.

[18] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, et al. MIPLIB 2017. *Mathematical Programming Computation*, pages 1–48, 2021.

[19] Prateek Gupta, Maxime Gasse, Elias Khalil, Pawan Mudigonda, Andrea Lodi, and Yoshua Bengio. Hybrid models for learning to branch. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

[20] Christoph Hansknecht, Imke Joormann, and Sebastian Stiller. Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem. *arXiv preprint arXiv:1805.01415*, 2018.

[21] He He, Hal Daume III, and Jason M Eisner. Learning to search in branch and bound algorithms. In *Advances in Neural Information Processing systems*, pages 3293–3301, 2014.

[22] Elias B Khalil, Christopher Morris, and Andrea Lodi. Mip-gnn: A data-driven framework for guiding combinatorial solvers. In *AAAI*, 2022.

[23] Elias Boutros Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[24] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76, 2000.

[25] Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *Top*, 25(2):207–236, 2017.

[26] Alejandro Marcos Alvarez, Louis Wehenkel, and Quentin Louveaux. Online learning for strong branching approximation in branch-and-bound. Technical report, Universite de Liege, 2016.

[27] Marvin Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961.

[28] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[29] Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, et al. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*, 2020.

[30] Vangelis Th Paschos. *Applications of combinatorial optimization*, volume 3. John Wiley & Sons, 2014.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[32] Antoine Prouvost, Justin Dumouchelle, Lara Scavuzzo, Maxime Gasse, Didier Chételat, and Andrea Lodi. Ecole: A gym-like library for machine learning in combinatorial optimization solvers. In *Workshop on Learning Meets Combinatorial Algorithms, NeurIPS 2020*, 11 2020.

[33] Haoran Sun, Wenbo Chen, Hui Li, and Le Song. Improving learning to branch via reinforcement learning. In *Learning Meets Combinatorial Algorithms at NeurIPS 2020*, 2020.

[34] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, volume 99, pages 1057–1063. Citeseer, 1999.

[35] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

[36] Giulia Zarpellon, Jason Jo, Andrea Lodi, and Yoshua Bengio. Parameterizing branch-and-bound search trees to learn branching policies. In *AAAI*, pages 3931–3939, 2021.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] We claim that tree MDP allows for a better convergence of RL algorithms for solving the branching problem in B&B.
   (b) Did you describe the limitations of your work? [Yes] The theoretical limitations are discussed in Section 4.4, and the computational limitations are addressed in Section 5.2.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] Our work's goal is to solve difficult mathematical problems more efficiently. The data was randomly generated and, as a matter of fact, we work with abstraction's of real-world problems. For these reasons, our work does not hold potential to have a negative societal impact. It is true that, similarly to other research in Machine Learning, our work has a non-negligible carbon footprint. However, given that our goal is more computationally efficient problem-solving, we argue that this work is a wise investment of computational resources.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] These are assumptions 4.2 and 4.6.
   (b) Did you include complete proofs of all theoretical results? [Yes] The proofs can be found in the appendix.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] For the purpose of anonymity during the review process, all the code and data will be made available after acceptance.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Some of these aspects are addressed in the paper (Section 5). The full details are included in the accompanying repository (to be made available).
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] Our evaluation results are averaged over 40 instances solved with 5 different seeds (200 data points total) and we report the standard deviation.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] As discussed in Section 5.1 we train on GPU-equipped nodes of an internal cluster. Each experiment took a maximum of six days to run.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] We included the appropriate citation for all used models and software.
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] We (will) provide a link to our code.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...

   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Supplementary Material

A.1 Side-by-side comparison of MDP and tMDP

Temporal MDP

A temporal MDP process:	$(\mathcal{S}, \mathcal{A}, p_{init}, p_{trans}, r)$
Probability of a trajectory τ:	$p_\pi(\tau) = p_{init}(s_0) \prod_{t=0}^{\lvert\tau\rvert-1} \pi(a_t \lvert s_t) p_{trans}(s_{t+1} \lvert s_t, a_t)$
Markov property:	$S_{t+1} \perp\!\!\!\perp S_{<t} \mid S_t, A_t, \forall t$

Tree MDP

A tree MDP process:	$(\mathcal{S}, \mathcal{A}, p_{init}, p^-_{ch}, p^+_{ch}, r, l)$
Probability of a trajectory τ:	$p_\pi(\tau) = p_{init}(s_0) \prod_{i \in \mathcal{N} \setminus \mathcal{L}} \pi(a_i \lvert s_i) p^-_{ch}(s_{ch^-_i} \lvert s_i, a_i) p^+_{ch}(s_{ch^+_i} \lvert s_i, a_i)$.
Markov property:	$S_{ch^-_i}, S_{ch^+_i} \perp\!\!\!\perp S_{nd_i} \mid S_i, A_i, \forall i$

A.2 Proofs

Proposition 4.1. *For any tree MDP tM, the policy gradient can be expressed as*

$$\nabla_\pi V^\pi = \mathop{\mathbb{E}}_{\tau \sim p_\pi} \left[\sum_{i \in \mathcal{N} \setminus \mathcal{L}} \nabla_\pi \log \pi(a_t|s_t) \sum_{j \in d_i} r(s_j) \right]. \tag{5}$$

Proof. This proof draws closely to the proof of the temporal policy gradient theorem. First, let us re-write (3) as

$$V^\pi = \mathbb{E}_{s_0 \sim p_\pi} \left[V^\pi(s_0) \right],$$

where

$$\begin{aligned} V^\pi(s_i) :=& r(s_i) && \text{if } l(s_i) = 1 \text{ (leaf node), and} \\ V^\pi(s_i) :=& r(s_i) + \mathbb{E}_{a_i, s_{ch^-_i}, s_{ch^+_i} \sim p_\pi} \left[V^\pi(s_{ch^-_i}) + V^\pi(s_{ch^+_i}) \right] && \text{if } l(s_i) = 0 \text{ (non-leaf node).} \end{aligned}$$

The corresponding gradients when $l(s_i) = 1$ and $l(s_i) = 0$ are, respectively,

$\nabla_\pi V^\pi(s_i) = 0$, and

$$\nabla_\pi V^\pi(s_i) = \mathbb{E}_{a_i, s_{ch^-_i}, s_{ch^+_i} \sim p_\pi} \left[\frac{\nabla_\pi \pi(a_i|s_i)}{\pi(a_i|s_i)} \left(V^\pi(s_{ch^-_i}) + V^\pi(s_{ch^+_i}) \right) \nabla_\pi V^\pi(s_{ch^-_i}) + \nabla_\pi V^\pi(s_{ch^+_i}) \right].$$

Let us now write the gradient of V^π,

$$\nabla_\pi V^\pi = \mathbb{E}_{s_0 \sim p_{init}} \left[\nabla_\pi V^\pi(s_0) \right].$$

Either we have $l(s_0) = 1$ and thus $\nabla_\pi V^\pi = 0$, or we can expand $\nabla_\pi V^\pi(s_0)$ to obtain

$$\nabla_\pi V^\pi = \mathbb{E}_{s_0, a_0, s_{ch^-_0}, s_{ch^+_0} \sim p_\pi} \left[\frac{\nabla_\pi \pi(a_0|s_0)}{\pi(a_0|s_0)} (V^\pi(s_{ch^-_0}) + V^\pi(s_{ch^+_0})) + \nabla_\pi V^\pi(s_{ch^-_0}) + \nabla_\pi V^\pi(s_{ch^+_0}) \right].$$

Then again, each of of the terms $\nabla_\pi V^\pi(s_{ch^-_0})$ and $\nabla_\pi V^\pi(s_{ch^+_0})$ can be replaced by 0 if the corresponding node is a leaf node, or can be expanded further in the same way if it is a non-leaf node. By

applying this rule recursively, we finally obtain

$$\begin{aligned}\nabla_\pi V^\pi &= \mathbb{E}_{\tau\sim p_\pi}\left[\sum_{i\in\mathcal{N}\setminus\mathcal{L}} \frac{\nabla_\pi \pi(a_i|s_i)}{\pi(a_i|s_i)}(V^\pi(s_{ch_i^-}) + V^\pi(s_{ch_i^+}))\right] \\ &= \mathbb{E}_{\tau\sim p_\pi}\left[\sum_{i\in\mathcal{N}\setminus\mathcal{L}} \frac{\nabla_\pi \pi(a_i|s_i)}{\pi(a_i|s_i)} \sum_{j\in d_i} r(s_j)\right] \\ &= \mathbb{E}_{\tau\sim p_\pi}\left[\sum_{i\in\mathcal{N}\setminus\mathcal{L}} \nabla_\pi \log \pi(a_i|s_i) \sum_{j\in d_i} r(s_j)\right].\end{aligned}$$

□

Lemma A.1. *In B&B, both children MILPs $MILP_{ch_i^-}$ and $MILP_{ch_i^+}$ can be derived from the local MILP $MILP_i$ and branching decision $a_i = (j, x_j^\star)$, with j the index of a variable in $MILP_i$, and $x_j^\star$ the value to be used for branching.*

Proof. From the definition of B&B in Section 2, $MILP_{ch_i^-}$ (resp. $MILP_{ch_i^+}$) consist of $MILP_i$ augmented with the additional constraint $x_j \leq \lfloor x_j^\star \rfloor$ (resp. $x_j \geq \lceil x_j^\star \rceil$). □

Proposition 4.3. *A vanilla B&B algorithm that satisfies Assumption 4.2 forms a tree MDP.*

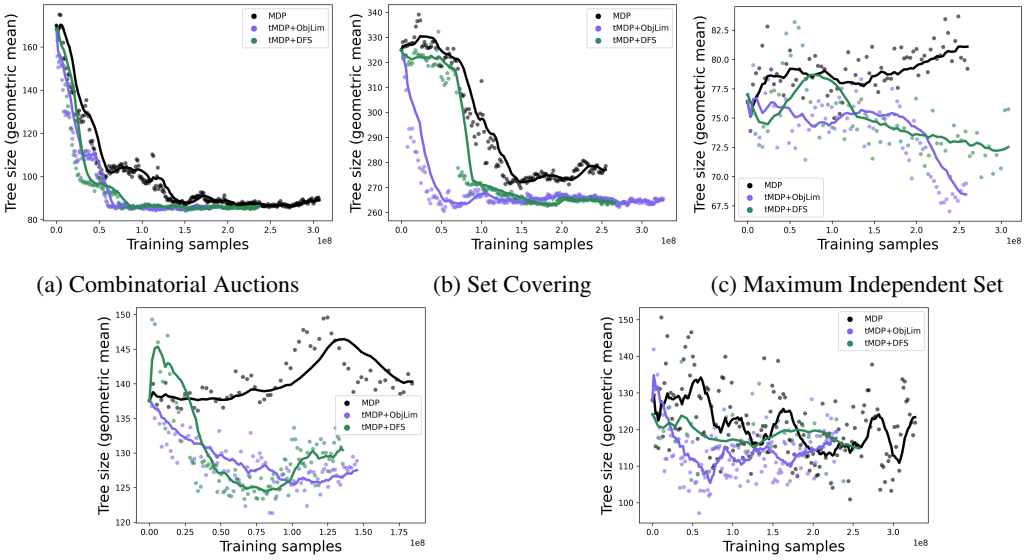
Proof. We shall now prove that, under Assumption 4.2, the B&B process can be formulated as a tree MDP $tM = (\mathcal{S}, \mathcal{A}, p_{init}, p_{ch}^-, p_{ch}^+, r, l)$, with states $s_i = (MILP_i, GUB_i)$ and actions $a_i = (j, x_j^\star)$. First, the algorithm starts at the root node with an initial MILP, $MILP_0$, and an initial global upper bound $GUB_0 = \infty$. Thus, the root state s_0 follows an arbitrary, user-defined MILP distribution $p_{init}(s_0)$, which is independent of the B&B algorithm. Second, Lemma A.1, together with Assumption 4.2, ensures the existence of (deterministic) distributions $p_{ch}^-(s_{ch_i^-}|s_i, a_i)$ and $p_{ch}^+(s_{ch_i^+}|s_i, a_i)$, from which the B&B children states $s_{ch_i^-}$ and $s_{ch_i^+}$ are generated. Third, the reward function $r(s_i)$ is not part of the B&B algorithm, and can be arbitrarily defined to match any (compatible) B&B objective. Last, the leaf node indicator $l(s_i)$ is exactly the vanilla B&B leaf node criterion, and is obtained by solving the LP relaxation of $MILP_i$ constrained with upper bound GUB_i, which results in either an infeasible LP (leaf node), a MILP-feasible LP solution (leaf node), or a MILP-infeasible LP solution (non-leaf node). This concludes the proof. □

Proposition 4.4. *In Optimal Objective Limit B&B (ObjLim B&B), that is, when the optimal solution value of the MILP is known at the start of the algorithm ($GUB_0 = GUB^\star$), Assumption 4.2 holds.*

Proof. Because $GUB_0 = GUB^\star$, the initial global upper bound is equal to the optimal solution value to the original MILP. Then, B&B will never be able to find a feasible solution that tightens that bound, and we necessarily have $GUB_i = GUB_0, \forall i$. Hence $GUB_{ch_i^-} = GUB_{ch_i^+} = GUB_i$, and both $GUB_{ch_i^-}$ and $GUB_{ch_i^+}$ can be directly derived from s_i. This concludes the proof. □

Proposition 4.5. *In Depth-First-Search B&B (DFS B&B), that is, when nodes are processed depth-first and left-first by the algorithm, Assumption 4.2 holds.*

Proof. First, it is trivial to show that $GUB_{ch_i^-}$ can be derived from s_i. Because node i is not a leaf node, it has not resulted in an integral solution, and hence processing node i does not change the GUB. And since ch_i^- is processed directly after node i, we necessarily have $GUB_{ch_i^-} = GUB_i$. This, combined with Lemma A.1, shows that $s_{ch_i^-}$ can be inferred from s_i and a_i. Second, we show how $GUB_{ch_i^+}$ can be derived from s_i and a_i. Because node ch_i^+ is processed right after the whole subtree below ch_i^- has been processed, $GUB_{ch_i^+}$ is necessarily the minimum of GUB_i and the optimal solution value of $MILP_{ch_i^-}$. Now, because $s_{ch_i^-}$ can be inferred from s_i and a_i, $MILP_{ch_i^-}$ can be recovered as well, and solved to obtain its optimal solution value. Therefore, $GUB_{ch_i^+}$ can be recovered from s_i and a_i. This, together with $GUB_{ch_i^-} = GUB_i$, concludes the proof. □



(a) Combinatorial Auctions (b) Set Covering (c) Maximum Independent Set

(d) Capacitated Facility Location (e) Multiple Knapsack

Figure 3: All training curves. We report the final B&B tree size (geometric mean over 20 validation instances $\times$ 5 seeds, the lower the better). On the x-axis we report the number of processed training samples. Solid lines show the moving average. The compared methods are REINFORCE with temporal policy gradients (MDP), with tree policy gradients and objective limit (tMDP+ObjLim), and with tree policy gradients and depth-first search node selection (tMDP+DFS).

A.3 Extended results

Here we provide all training curves (Figure 3) and the extended evaluation results (Table 2) with the geometric mean of the solving times in seconds (Time) and the geometric mean of the final B&B tree size (Nodes). The results are averaged over the solving runs that finished successfully for all methods. This is, if a solving run reached the time limit for any method, this is excluded from the average. Table 3 shows the number of solving runs that timed out per method.

A.4 SCIP's default branching rule

SCIP assigns maximum priority by default to the hybrid branching rule [2]. This means that the choice of branching variable is based on a weighted sum of different criteria. The biggest weight is placed on the variable's pseudocosts. A variable's pseudocost is calculated as a function of the change in LP objective value we observe (on each of the branches) as a consequence of branching on that variable. This value can be explicitly calculated by tentatively branching on candidate variables (in which case the rule is called strong branching), or estimated based on past observed values. SCIP runs strong branching until it has stored a sufficient amount of observations for each variable, and then switches to the estimation strategy. Other than pseudocosts, SCIP also considers information about the implied reductions of other variables' domains and conflicts where the variable is involved, though with smaller importance.

It is important to consider that a call to strong branching can trigger a series of side-effects within the solver that are not accounted for in the node count. This was first observed by Gamrath and Schubert [14], who point out that this gives an unfair advantage to methods that use strong branching when comparing branching rules according to the final tree size.

A.5 Instance collections

This section presents the models used to generate our instance benchmarks. The parameters used to generate each benchmark are shown in Table 4.

Table 2: Evaluation on test instances (same size as training) and transfer instances (larger size). We report the geometric mean and standard deviation of the final B&B tree size and the solving time (lower is better for both).

	Test		Transfer	
Model	Nodes	Time	Nodes	Time
SCIP default	$7.3 \pm 39\%$	$3.3 \pm 10\%$	$733.9 \pm 26\%$	$27.4 \pm 7\%$
IL	$52.2 \pm 13\%$	$2.1 \pm 6\%$	$805.1 \pm 9\%$	$14.6 \pm 5\%$
RL (MDP)	$86.7 \pm 16\%$	$2.2 \pm 6\%$	$1906.3 \pm 18\%$	$20.9 \pm 11\%$
RL (tMDP+DFS)	$86.1 \pm 17\%$	$2.2 \pm 6\%$	$1804.6 \pm 17\%$	$20.1 \pm 9\%$
RL (tMDP+ObjLim)	$87.0 \pm 18\%$	$2.2 \pm 6\%$	$1841.9 \pm 18\%$	$20.4 \pm 10\%$
	Combinatorial auctions			
Model	Nodes	Time	Nodes	Time
SCIP default	$10.7 \pm 24\%$	$5.8 \pm 6\%$	$61.4 \pm 19\%$	$12.6 \pm 5\%$
IL	$51.8 \pm 10\%$	$4.0 \pm 5\%$	$145.0 \pm 6\%$	$8.0 \pm 4\%$
RL (MDP)	$196.3 \pm 20\%$	$5.1 \pm 8\%$	$853.3 \pm 27\%$	$14.9 \pm 13\%$
RL (tMDP+DFS)	$190.8 \pm 20\%$	$5.1 \pm 7\%$	$816.8 \pm 25\%$	$14.6 \pm 12\%$
RL (tMDP+ObjLim)	$193.5 \pm 23\%$	$5.1 \pm 8\%$	$826.4 \pm 26\%$	$14.6 \pm 13\%$
	Set covering			
Model	Nodes	Time	Nodes	Time
SCIP default	$19.3 \pm 52\%$	$13.2 \pm 13\%$	$2867.1 \pm 35\%$	$167.4 \pm 23\%$
IL	$35.9 \pm 36\%$	$8.7 \pm 10\%$	$1774.8 \pm 38\%$	$85.7 \pm 22\%$
RL (MDP)	$91.8 \pm 56\%$	$9.5 \pm 16\%$	$2768.5 \pm 76\%$	$85.6 \pm 51\%$
RL (tMDP+DFS)	$89.8 \pm 51\%$	$9.5 \pm 17\%$	$2970.0 \pm 76\%$	$90.6 \pm 51\%$
RL (tMDP+ObjLim)	$85.4 \pm 53\%$	$9.4 \pm 17\%$	$2763.6 \pm 74\%$	$86.1 \pm 47\%$
	Maximum independent set			
Model	Nodes	Time	Nodes	Time
SCIP default	$203.6 \pm 63\%$	$16.9 \pm 34\%$	$344.3 \pm 57\%$	$40.3 \pm 36\%$
IL	$247.5 \pm 39\%$	$7.2 \pm 26\%$	$407.8 \pm 37\%$	$13.6 \pm 24\%$
RL (MDP)	$393.2 \pm 47\%$	$8.7 \pm 29\%$	$679.4 \pm 52\%$	$17.2 \pm 33\%$
RL (tMDP+DFS)	$360.4 \pm 46\%$	$8.3 \pm 30\%$	$609.1 \pm 47\%$	$15.9 \pm 29\%$
RL (tMDP+ObjLim)	$325.4 \pm 41\%$	$7.9 \pm 26\%$	$496.0 \pm 48\%$	$14.5 \pm 28\%$
	Facility location			
Model	Nodes	Time	Nodes	Time
SCIP default	$267.8 \pm 96\%$	$1.5 \pm 54\%$	$592.3 \pm 75\%$	$3.7 \pm 42\%$
IL	$228.0 \pm 95\%$	$1.8 \pm 66\%$	$1066.1 \pm 101\%$	$7.1 \pm 82\%$
RL (MDP)	$143.4 \pm 76\%$	$1.3 \pm 48\%$	$518.4 \pm 79\%$	$4.5 \pm 58\%$
RL (tMDP+DFS)	$135.8 \pm 75\%$	$1.3 \pm 48\%$	$495.1 \pm 81\%$	$4.3 \pm 59\%$
RL (tMDP+ObjLim)	$142.4 \pm 78\%$	$1.4 \pm 48\%$	$425.3 \pm 64\%$	$3.9 \pm 46\%$
	Multiple knapsack			

Table 3: Number of solving runs (instance-seed pairs) out of 200 that hit the 1h time limit.

Model	C. Auct.	Set Cov.	M.Ind.Set	Fac. Loc.	M. Knap.
SCIP default	0	0	0	1	0
IL	0	0	0	0	0
RL (MDP)	0	0	1	0	0
RL (tMDP+DFS)	0	0	1	0	0
RL (tMDP+ObjLim)	0	0	1	0	0
		Test			
Model	C. Auct.	Set Cov.	M.Ind.Set	Fac. Loc.	M. Knap.
SCIP default	0	0	1	13	0
IL	0	0	0	0	3
RL (MDP)	0	0	20	1	2
RL (tMDP+DFS)	0	0	18	1	0
RL (tMDP+ObjLim)	0	0	16	1	0
		Transfer			

Table 4: Size of the instances used for training and evaluation, for each problem benchmark. We evaluate the final performance on instances of the same size as training (test), and also larger instances (transfer).

Benchmark	Generation method	Parameters	Train / Test	Transfer
Combinatorial auction	Leyton-Brown et al. [24] with arbitrary relationships	Items Bids	100 500	200 1000
Set covering	Balas and Ho [4]	Items Sets	400 750	500 1000
Maximum independent set	Bergman et al. [6] on Erdős-Rény graphs	Nodes Affinity	500 4	1000 4
Facility location	Cornuéjols et al. [8] with unsplittable demand	Customers Facilities	35 35	60 35
Multiple knapsack	Fukunaga [13]	Items Knapsacks	100 6	100 12

A.5.1 Combinatorial auctions

For m items, we are given n bids $\{\mathcal{B}_j\}_{j=1}^n$. Each bid $\mathcal{B}_j$ is a subset of the items with an associated bidding price p_j. The associated combinatorial auction problem is of the following form:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j:i\in\mathcal{B}_j} x_j \leq 1, \quad i = 1, ..., m \\ & x_j \in \{0, 1\} \; j = 1, ..., n \end{aligned}$$

where x_j represents the action of choosing bid $\mathcal{B}_j$.

A.5.2 Set covering

Given the elements $1, 2, ..., m$, and a collection $\mathcal{S}$ of n sets whose union equals the set of all elements, the set cover problem can be formulated as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{s\in\mathcal{S}} x_s \\ \text{subject to} \quad & \sum_{s:e\in s} x_s \geq 1, \quad e = 1, ..., m \\ & x_s \in \{0, 1\} \; \forall s \in \mathcal{S} \end{aligned}$$

A.5.3 Maximum independent set

Given a graph G the maximum independent set problem consists in finding a subset of nodes of maximum cardinality such that no two nodes in that subset are connected. We use the clique formulation from [6]. Given a collection $\mathcal{C} \subseteq 2^V$ of cliques whose union covers all the edges of the graph G, the clique cover formulation is

$$\begin{aligned} \text{maximize} \quad & \sum_{v\in V} x_v \\ \text{subject to} \quad & \sum_{v\in C} x_v \leq 1, \quad \forall C \in \mathcal{C} \\ & x_v \in \{0, 1\} \quad \forall v \in V \end{aligned}$$

A.5.4 Capacitated facility location with unsplittable demand

Given a number n of clients with demands $\{d_j\}_{j=1}^n$, and a number m of facilities with fixed operating costs $\{f_i\}_{i=1}^m$ and capacities $\{s_i\}_{i=1}^m$, let c_{ij}/d_j be the unit transportation cost between facility i and

client j, and let p_{ij}/d_j be the unit profit for facility i supplying client j. We try to solve the following problem

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} + \sum_{i=1}^{m} f_i y_i \\
\text{subject to} \quad & \sum_{j=1}^{n} d_j x_{ij} \leq s_i y_i, \quad i = 1, ..., m \\
& \sum_{i=1}^{m} x_{ij} \geq 1, \quad j = 1, ..., n \\
& x_{ij} \in \{0, 1\} \quad \forall i, j \\
& y_i \in \{0, 1\} \quad \forall i
\end{aligned}$$

where each variable x_{ij} represents the decision of facility i supplying client j's demand, and each variable y_i representing the decision of opening facility i for operation.

A.5.5 Multiple knapsack

Given n items with respective prices $\{p_j\}_{j=1}^{n}$ and weights $\{w_j\}_{j=1}^{n}$, and m knapsacks with capacities $\{c_i\}_{i=1}^{m}$, the multiple knapsack problem consists in placing a number of items in each of the knapsacks such that the price of the selected items is maximized, while the capacity of the knapsacks is not exceeded by the total weight of the items therein. Formally:

$$\begin{aligned}
\text{maximize} \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \\
\text{subject to} \quad & \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \quad i = 1, ..., m \\
& \sum_{i=1}^{m} x_{ij} \leq 1, \quad j = 1, ..., n \\
& x_{ij} \in \{0, 1\} \; \forall i, j
\end{aligned}$$

where each variable x_{ij} represents the decision of placing item j in knapsack i.