Symbiotic Local Search for Small Decision Tree Policies in MDPs

Roman Andriushchenko[1] **Milan Češka**[1] **Debraj Chakraborty**[2] **Sebastian Junges**[3] **Jan Křetínský**[2,4]
Filip Macák[1]

[1]Brno University of Technology, Czechia
[2]Masaryk University, Czechia
[3]Radboud University Nijmegen, the Netherlands
[4]Technical University of Munich, Germany

Abstract

We study decision making policies in Markov decision processes (MDPs). Two key performance indicators of such policies are their value and their interpretability. On the one hand, policies that optimize value can be efficiently computed via a plethora of standard methods. However, the representation of these policies may prevent their interpretability. On the other hand, policies with good interpretability, such as policies represented by a small decision tree, are computationally hard to obtain. This paper contributes a local search approach to find policies with good value, represented by small decision trees. Our local search symbiotically combines learning decision trees from value-optimal policies with symbolic approaches that optimize the size of the decision tree within a constrained neighborhood. Our empirical evaluation shows that this combination provides drastically smaller decision trees for MDPs that are significantly larger than what can be handled by optimal decision tree learners.

1 INTRODUCTION

Markov decision processes are *the* standard model for decision making under uncertainty. Policies describe for every state which action to take. Their *value* is the expected return of executing the policy on the MDP. Traditionally, the focus in MDPs is to compute value-optimal policies. However, computing policies is often part of a larger methodology, in which *interpretability* of the policy by either a human or a machine is essential. The objectives to compute interpretable and high-value policies are conflicting and can be resolved in different ways. This paper contributes a local search that finds small decision tree (DT) representations for almost value-optimal policies.

(Value-)optimal policies can be computed efficiently using classical algorithms such as value iteration, policy iteration, or linear programming [Puterman, 1994]. Using these algorithms provides a *tabular* representation of the policy which scales linearly with the number of states in the MDP. In contrast, *neural* policies represent the policy as a neural network [Bertsekas, 1996]. They can be efficiently learned due to their differentiability, however, while neural networks may be concise, reasoning about the behavior of a policy represented by a neural network is challenging for humans and for machines. Finally, rule-based policy representations [Gupta et al., 2015, Verma et al., 2018, Batz et al., 2024], in particular policies represented as DTs [Bastani et al., 2018, Topin et al., 2021, Vos and Verwer, 2023], are promising due to their interpretability and generalisability. However, such policies are not differentiable. Consequentially, computing interpretable high-value policies is computationally intractable for complex MDPs [Vos and Verwer, 2023, Andriushchenko et al., 2025].

This paper studies a tractable local search towards learning interpretable yet high-value policies, represented as DTs. It symbiotically combines two main lines of research from the literature: (1) *(Data-driven) policy mapping*, which employs DT learning heuristics to obtain trees that match a value-optimal policy in the most relevant decisions [Ashok et al., 2020, 2021], and (2) *bounded-depth (policy) tree learning* using symbolic reasoning that computes potentially value-optimal small DTs [Vos and Verwer, 2023, Andriushchenko et al., 2025].

The main approach to *data-driven policy mapping*, as implemented e.g. in DTCONTROL Ashok et al. [2021], consists of two sequential stages. First, an (almost) optimal MDP policy is computed using off-the-shelf tools. Then, greedy algorithms are used to represent this policy as a DT. However, despite various tweaks, the size of the final DT depends significantly on the initial policy and the computation of that policy is not incentivized to obtain policies which can be represented by a small DT. The main approach to *bounded-depth policy tree learning* is to con-

Proceedings of the 41st Conference on Uncertainty in Artificial Intelligence (UAI 2025), PMLR 244:132–148.

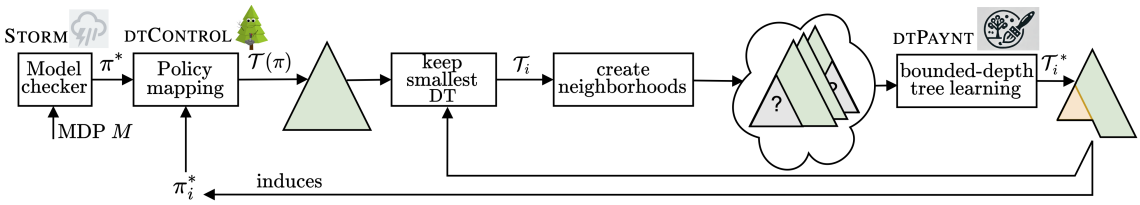


Figure 1: Our proposed algorithm DTNEST visualised.

struct and solve a constraint system. This works either as an MILP encoding [Vos and Verwer, 2023] or by an iterative abstraction-refinement loop that avoids solving a monolithic MILP [Andriushchenko et al., 2025]. Both approaches work with up to trees of depth 8 and the latter approach also allows working with MDPs with up to millions of states. However, these comes at significant computational cost and finding trees of depth 8 for large MDPs is beyond their current abilities. Section 3 contains additional information about these approaches.

We propose the local search algorithm DTNEST that symbiotically combines the approaches above. Figure 1 illustrates the main steps. DTNEST computes a value-optimal policy and represents it as a DT $\mathcal{T}$ with a data-driven policy learner. We then iteratively improve upon $\mathcal{T}$. We first create a neighborhood around $\mathcal{T}$ containing DTs that are close to $\mathcal{T}$. Specifically, these trees differ in one subtree only. As the neighborhood is much smaller than the space of all decision trees, searching this tree using a bounded-depth tree learner is significantly faster than exploring the space of all trees. This local search converges against a local optimum. To jump out of such an optimum, we perturb the tree in different ways, before pruning another subtree. For this perturbation, we can use data-driven learning, which now starts with a policy that already was incentivized to be concise. On the technical level, we are combining three state-of-the-art tools: DTCONTROL Ashok et al. [2021] for policy mapping, DTPAYNT Andriushchenko et al. [2025] for bounded-depth optimization, and the model checker STORM Hensel et al. [2022] to obtain the initial optimal policy and to evaluate the intermediate candidate policies.

Our experiments confirm that DTNEST combines the strengths of both approaches. It can handle MDPs (and DTs) that are orders of magnitude larger than what a purely symbolic approach can handle, while it constructs DTs that are significantly smaller than the mapped optimal policies at a cost of only a few percent relative error. In particular, in 9 out of 13 benchmarks and with a timeout of one hour, we have obtained trees of sizes at most 25 (in our opinion within reach of explainability), while previously they were larger than that.

2 PRELIMINARIES AND PROBLEM STATEMENT

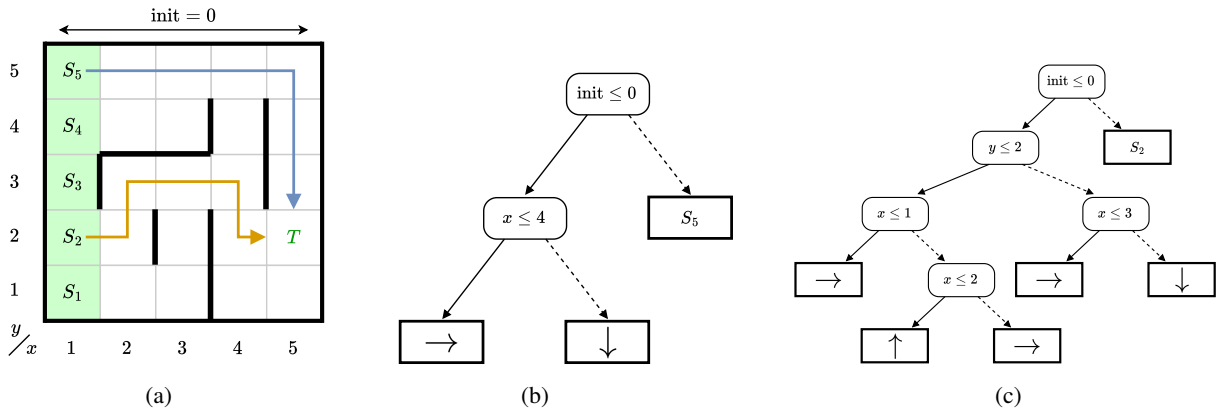
A *distribution* over a countable set A is a function $\mu\colon A \to [0,1]$ s.t. $\sum_a \mu(a){=}1$. $a \sim \mu$ denotes $\mu(a) > 0$. The set $Distr(A)$ contains all distributions over A.

Definition 1 (MDP) *A* Markov decision process (MDP) *is a tuple $M = (S, s_0, Act, P, R, \gamma)$ with a finite set S of states, an initial state $s_0 \in S$, a finite (indexed) set Act of actions, a partial transition function $P\colon S \times Act \nrightarrow Distr(S)$, a reward function $R\colon S \times Act \to \mathbb{R}$, and a discount factor $\gamma \in [0,1)$.*

For an MDP M, we define the *available actions* in $s \in S$ as $Act(s) := \{\alpha \in Act \mid P(s,\alpha) \neq \bot\}$; we denote $P(s,\alpha,s') := P(s,\alpha)(s')$. An MDP with $|Act(s)| = 1$ for each $s \in S$ is a *Markov chain (MC)*; we denote MCs as tuples (S, s_0, P, R, γ). We assume that the states in an MDP are factored, i.e., composed of multiple features. Each feature is defined by a bounded integer variable from the set of variables $\mathcal{V}$. *State predicates* are inequalities of the form $v \leq b$ with $v \in \mathcal{V}$ and $b \in \mathbb{Z}$; the set of such predicates is denoted $\Psi_{\mathcal{V}}$. A state s *satisfies a predicate* $v \leq b$ iff $s(v) \leq b$; we denote this with $s \models (v \leq b)$.

Example 1 *Take the grid-world environment from Fig. 2a. The agent initially chooses from which "green" cell it starts. In the grid, the agent can move into any of the four cardinal directions, however, there are walls in the grid restricting some of the movements of the agent. The goal is to minimize the number of steps it takes to reach the target. The optimal solution is 6 steps (the path highlighted with the orange arrow). We can model this environment as an MDP M with 26 states (each states represents one grid cell and one initial state), with actions allowing the movement $\{\uparrow, \rightarrow, \downarrow, \leftarrow\}$ and the choice of the initial position $\{S_1, \ldots, S_5\}$. There are three state variables in this MDP $\mathcal{V} = \{x, y, init\}$. The initial state is the only state where it holds that $init = 1$, and for example, the target state corresponds to the variable assignment $x = 5, y = 2, init = 0$.*

Policies. A (deterministic, memoryless) *policy* is a function $\pi\colon S \to Act$. The set Σ^M contains the policies for MDP M. A policy $\pi \in \Sigma^M$ induces the MC $M^\pi = (S, s_0, P^\pi, R, \gamma)$



(a) (b) (c)

Figure 2: (a) Grid-world MDP example. The agent first chooses in which "green" cell it starts and aims to reach the target state while minimizing the number of steps. (b) A DT that represents a policy that reaches the target state in 7 steps showcased by the blue arrow in (a). (c) A DT that represents the optimal policy which reaches the target state in 6 steps showcased by the orange arrow in (a).

where $P^\pi(s) = P(s, \pi(s))$. To compactly represent policies, it is convenient to omit actions defined for states that are unreachable in M^π. The *partial restriction* of a policy π is a partial function $\pi|_{s_0} : S \nrightarrow Act$ where $\pi|_{s_0}(s) \neq \bot$ iff state s is reachable (from s_0) in M^π. The goal in an MDP is to find an optimal policy π^* that maximizes the expected cumulative reward [Puterman, 1994] over time. Formally, for a policy π, we define its *value* $V_M(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right]$ where $s_{t+1} \sim P(s_t, \pi(s_t))$; we omit subscript M whenever the MDP is clear from the context. The optimal policy π^* is then $\pi^* \in \arg\max_{\pi \in \Sigma^M} V_M(\pi)$. Our approach also supports *maximal reachability probability* and other temporal properties [Baier et al., 2018].

The random action. To concisely represent policies, it is convenient to allow a policy to take some dedicated *arbitrary* action. We explicate this arbitrary action α_{rand} for every state that uniformly selects one of the (available) actions. Formally, we define $M' = (S, s_0, Act', P', R, \gamma)$ with $Act'(s) = Act(s) \cup \{\alpha_{\text{rand}}\}$, $P'(s, \alpha_{\text{rand}}, s') = \frac{1}{|Act(s)|} \sum_\alpha P(s, \alpha, s')$. Henceforth, we assume that MDP $M' = M$, i.e., that every MDP contains an action α_{rand}.

Correctness and interpretability of the random action. Adding the random action makes it explicit that a policy may randomly pick either available action. Sometimes, having this opportunity makes for more concise policies. A possible downside is that the policy in M' may not reflect a memoryless deterministic policy in M. The addition of random action does not effect the values achievable in the MDP, i.e., adding the random action is sound. For a formal proof refer to Andriushchenko et al. [2025].

Trees. A (binary) *tree* is a tuple $T = (n_0, N, L, l, r)$ with the *root node* n_0, the set N of *inner nodes*, the set L of *leaf nodes*, and functions $l, r \colon N \to N \cup L$ defining the *left* and *right successors* of the inner nodes, respectively. A *path* (of length k) in a tree T is a sequence $\pi = n_0 \dots n_k$ of nodes s.t. $\forall 0 < i \leq k : n_i \in \{l(n_{i-1}), r(n_{i-1})\}$. Path π is *complete* if it ends in a leaf node. The *depth* of T, $\mathsf{Depth}(\mathcal{T})$, is the length of its longest path. The *size* of T is the number of its inner nodes.

Definition 2 (Decision tree) *Assume an MDP $M = (S, s_0, Act, P, R, \gamma)$ with state features defined over the set $\mathcal{V}$. A* decision tree *(DT) for M is a tuple $\mathcal{T} = (T, \lambda, \delta)$ where (i) T is a binary tree, (ii)* predicate function *$\lambda \colon N \to \Psi_{\mathcal{V}}$ assigns to inner nodes a state predicate, and (iii)* action function *$\delta \colon L \to Act$ assigns to leaf nodes an action.*

We lift the notions of inner and leaf nodes, paths, depths, and sizes of trees to DTs.

Definition 3 (Corresponding states) *The set $\mathsf{st}^{\mathcal{T}}(n)$ of states that corresponds to a node n is recursively defined as follows: $\mathsf{st}^{\mathcal{T}}(n_0) = S$, $\mathsf{st}^{\mathcal{T}}(l(n)) = \{s \in \mathsf{st}^{\mathcal{T}}(n) \mid s \models \lambda(n)\}$, and $\mathsf{st}^{\mathcal{T}}(r(n)) = \{s \in \mathsf{st}^{\mathcal{T}}(n) \mid s \not\models \lambda(n)\}$.*

Note that the sets $\{\mathsf{st}^{\mathcal{T}}(n) \mid n \in L\}$ represent a partition of S. Thus, we can define $\mathsf{leaf}^{\mathcal{T}}(s)$ as the unique leaf node $n \in L$ such that $s \in \mathsf{st}^{\mathcal{T}}(n)$.

Example 2 *Fig. 2b contains an example of a decision tree for MDP from Example 1. The inner nodes are represented by the rounded shapes and contain state predicates, and the leaf nodes are represented by rectangles and contain the chosen action. For example, the root node contains the state predicate init ≤ 0 which separates the initial state from the states representing cells in the grid. The middle leaf node with the action $\downarrow$ corresponds to the states which represent the cells in the last column of the grid as these are the states which satisfy the predicate init ≤ 0 and do not satisfy the predicate $x \leq 4$.*

Definition 4 (Induced policy) *The DT $\mathcal{T}$* induces policy $\pi_{\mathcal{T}} \colon S \to Act$ *with* $\pi_{\mathcal{T}}(s) = \delta(\mathsf{leaf}^{\mathcal{T}}(s))$ *if* $\delta(\mathsf{leaf}^{\mathcal{T}}(s)) \in Act(s)$ *and* $\pi_{\mathcal{T}}(s) = \alpha_{\mathrm{rand}}$ *otherwise. The* value $V(\mathcal{T})$ *of the DT $\mathcal{T}$ is defined as the value of $\pi_{\mathcal{T}}$.*

Fallback action interpretability. Our approach can synthesize a DT that assigns an action that is not available at a given state, in which case we use the random action as a fallback. This setup avoids that the DT must precisely capture when an action is available and allows for smaller DTs. Note that the information which actions are available is usually also accessible in another format (e.g. masks or shields heavily used in reinforcement learning settings).

Admissible error. Instead of learning a tree that induces the optimal policy π^*, to give more room for the tree size reduction, we admit trees having near-optimal value. We define the *normalized relative error* of a tree $\mathcal{T}$ as

$$\mathcal{E}(\mathcal{T}) \coloneqq \frac{V(\pi^*) - V(\mathcal{T})}{V(\pi^*) - V(\pi_{\mathrm{rand}})},$$

where π_{rand} is a randomized policy with $\pi_{\mathrm{rand}}(s) = \alpha_{\mathrm{rand}}$ for every $s \in S$. We include $V(\pi_{\mathrm{rand}})$ as our lower bound for a tree value to provide a clearer sense of how distant a candidate $\mathcal{T}$ is from an arbitrary one. Given relative error ε, we call the tree *admissible* if $\mathcal{E}(\mathcal{T}) \leq \varepsilon$. Note that using the value of worst policy as a lower bound would mean the error bound would not be very tight. Note that similar metrics have also been used recently Vos and Verwer [2023], Andriushchenko et al. [2025].

Example 3 *Consider the MDP from Example 1. The optimal policy in this MDP reaches the target in 6 steps and follows the orange arrow in Fig. 2a. The minimal decision tree representing this policy has 5 decision nodes and is shown in Fig. 2c. If we allow some admissible error which would allow us to find policies that take 7 steps to reach a target, we can find a DT with just 2 decision nodes for such a policy. This policy follows the blue arrow in Fig. 2a and the DT representing this policy is shown in Fig. 2b.*

Problem Statement. We want to find a (near)-optimal policy represented by a small tree (in terms of the number of inner nodes):

> **Bounded-value policy tree learning**: Given MDP M and relative error ε, find a smallest DT $\mathcal{T}$ with $\mathcal{E}(\mathcal{T}) \leq \varepsilon$.

Typically, finding the smallest DT is computationally intractable for complex MDPs. Therefore, akin to Vos and Verwer [2023], Andriushchenko et al. [2025], we are interested in an anytime algorithm: the faster we find a small DT that is within the error bound, the better.

3 STATE OF THE ART

We briefly recap two types of methodologies used to learn policy trees that are the fundament for our local search. Other related approaches are discussed in the related work.

Policy mapping. Policy mapping takes a fixed, typically value-optimal, policy and maps it to a DT. Most prominent for MDPs are greedy heuristics as implemented in DTCONTROL [Ashok et al., 2021] and UPPAAL STRATEGO [David et al., 2015, Ashok et al., 2019b]. The DTs are learnt from a dataset representing the policy where the states in S are the input and the suggested actions in Act are the output. This uses algorithms like CART [Breiman et al., 1984] or ID3 [Quinlan, 1986] which recursively split the dataset by evaluating different predicates by calculating some impurity measure [Mitchell, 1997] (like information gain or Gini index) and then greedily picking the most promising one. The result is an approximation or exact representation of the original policy. Generally, these tools favor scalability over minimality.

Bounded-depth tree learning. The alternative to mapping a fixed policy to a small tree is to search the space of small-tree policies. We review two approaches that both provide an anytime algorithm which converges to the value-optimal tree within the space of trees with depth at most d. The OMDT approach [Vos and Verwer, 2023] uses constraint solving via mixed-integer linear programs (MILPs) to do so: The MILP encodes both the structural constraints on the policy (being d-implementable) and the constraint that the policy achieves the optimal value (using the standard LP formulation for maximal discounted rewards). The DTPAYNT approach [Andriushchenko et al., 2025] searches this space by iteratively constructing candidate policies in this space and mapping these policies into the smallest possible tree. The candidate policies are constructed by value iteration on variants of the original MDP, while the policy mapping is done with constraint solving. On MDPs with more than 3000 decisions (i.e. states-action pairs), this iterative approach is significantly faster and produces better trade-offs between the DT size and quality Andriushchenko et al. [2025]. Even though this approach performs well for the bounded-depth tree learning problem, DTPAYNT (and indeed OMDT) cannot cope with settings where a DT of bigger depth is needed to represent a good policy. While we use DTPAYNT, we embed it into a loop to mitigate this disadvantage.

4 VARIABLE NEIGHBORHOOD SEARCH

This section outlines our approach, discusses the main ingredients, and then combines these ingredients into the algorithm called DTNEST.

Our approach is inspired by the principles of *Variable Neigh-*

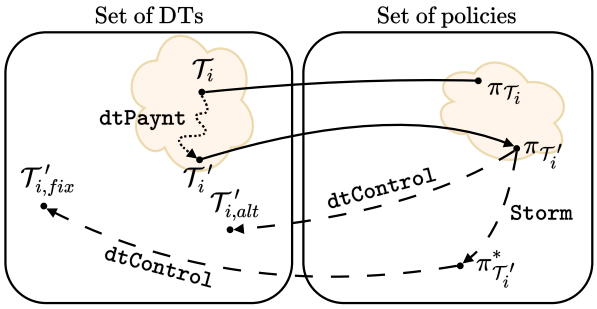


Figure 3: Scheme of one iteration of our approach. An iteration starts with tree $\mathcal{T}_i$ which corresponds to $\pi_{\mathcal{T}_i}$. DTPAYNT performs subtree neighborhood exploration and finds $\mathcal{T}'_i$. This tree is then translated to its corresponding policy $\pi_{\mathcal{T}'_i}$. Two tree perturbations are performed from this policy: i) running DTCONTROL to obtain alternative DT representation $\mathcal{T}'_{i,\mathsf{alt}}$ ii) fixing of the policy using model checking in STORM to obtain $\pi^*_{\mathcal{T}'_i}$ and running DTCONTROL on this new policy to obtain $\mathcal{T}'_{i,\mathsf{fix}}$. At the end of the iteration, we have 3 candidate trees available: $\mathcal{T}'_i, \mathcal{T}'_{i,\mathsf{alt}}, \mathcal{T}'_{i,\mathsf{fix}}$.

borhood Search (VNS) [Mladenovic and Hansen, 1997]. In the classical VNS, a candidate solution is associated with a sequence of different neighborhoods; these neighborhoods are gradually explored until the first improving solution is obtained. In order to avoid local optima, an additional *perturbation step* is performed. VNS has proven to be useful in solving complex, combinatorial optimization problems such as large MILPs or continuous non-linear programs Hansen et al. [2019]. In our case, we are also solving a multi-layered optimization problem as on one hand we are looking for a near-optimal policy which can be represented by a small DT and on the other, we are trying to represent this policy as optimally as possible.

Abstractly, DTNEST proceeds as follows. In the initialization phase, we solve an MDP, obtaining an (arbitrary) optimal policy π^*, and use DTCONTROL to learn a tree $\mathcal{T}_0$ that induces its partial restriction $\pi^*|_{s_0}$. Afterwards, DTNEST iteratively improves the tree. Every iteration of DTNEST (illustrated in Fig. 3) consists of a *subtree neighborhood exploration (SNE)* and *tree perturbation*. During SNE, we explore different *tree neighborhoods* of $\mathcal{T}_i$ in search of a smaller admissible tree; we define a tree neighborhood of $\mathcal{T}_i$ as a set of trees differing from $\mathcal{T}_i$ in one of its subtrees, and we use bounded-depth tree learning to search for a subtree having fewer nodes. If no such tree can be found, we terminate the search; otherwise, let $\mathcal{T}'_i$ denote the new tree. In the tree perturbation step, we compute various transformations of $\mathcal{T}'_i$ and proceed to the next iteration with the smallest one. We detail SNE, the neighborhoods it considers, and perturbations below.

4.1 SUBTREE NEIGHBORHOOD EXPLORATION

Subtree neighborhood exploration (SNE) explores modifications of the tree $\mathcal{T}$ where one of its subtrees is replaced with a smaller one. In particular, assume an inner node $n \in N$ of $\mathcal{T}$. $\mathcal{T}|_n$ denotes a subtree of $\mathcal{T}$ with root n having sets $N|_n$ and $L|_n$ of inner and leaf nodes, respectively. The (n,d)-neighborhood of $\mathcal{T}$ consists of all trees that coincide with $\mathcal{T}$ in every node except have a subtree of depth d emplaced at an inner node n; the n-neighborhood of $\mathcal{T}$ is the smallest neighborhood that includes all (n,d)-neighborhoods, $d < \mathsf{Depth}(\mathcal{T}|_n)$.

To efficiently explore the n-neighborhood of $\mathcal{T}$, we use off-the-shelf bounded-depth tree learning implemented in DTPAYNT. For this purpose, we construct a sub-MDP $M|^{\mathcal{T}}_n$ where we fix all actions in states defined outside $\mathcal{T}|_n$ according to $\pi_{\mathcal{T}}$. Formally, we define $M|^{\mathcal{T}}_n = \left(S, s_0, Act|^{\mathcal{T}}_n, P|^{\mathcal{T}}_n, R, \gamma\right)$ where $Act|^{\mathcal{T}}_n(s) = Act(s)$ if $s \in \mathsf{st}^{\mathcal{T}}(n)$, and $Act|^{\mathcal{T}}_n(s) = \{\pi_{\mathcal{T}}(s)\}$ otherwise. DTPAYNT enables efficient learning of a depth-bounded tree for this sub-MDP where the states having only one available action are ignored. If DTPAYNT learns an admissible tree, the former replaces the latter in $\mathcal{T}$ as a new subtree at node n; let $\mathcal{T}'$ denote the new tree. If DTPAYNT fails to obtain a smaller tree, we continue the search in n'-neighborhood where n' is another inner node (details are to be introduced in Sec. 4.3).

4.2 TREE PERTURBATION

While SNE can succeed in modifying and reducing the subtrees of $\mathcal{T}$, it cannot, in principle, modify "higher" nodes of $\mathcal{T}$ that lie closer to its root. As a result, even consequent applications of SNE have a slim chance of obtaining a tree $\mathcal{T}'$ that drastically differs from $\mathcal{T}$. Tree perturbation serves to avoid such local optima. The aim here is to systematically perturb $\mathcal{T}'$ in search of other (admissible) trees that differ in these higher nodes. We consider two possible perturbations of $\mathcal{T}'$ via *tree reconfiguration* or *policy repair*. Both variants are illustrated in Fig. 3 and are described below.

Tree reconfiguration. Let $\pi_{\mathcal{T}'}$ be the policy induced by $\mathcal{T}'$. This policy can be translated back to a decision tree $\mathcal{T}'_{\mathsf{alt}}$ using any data-driven tree learning approach, such as DTCONTROL. We note that there is no particular reason why the resulting $\mathcal{T}'_{\mathsf{alt}}$ should coincide with the earlier $\mathcal{T}'$.

Policy repair. Note that SNE may achieve a smaller tree $\mathcal{T}'$ at the price of the reduced value (while still guaranteeing that $\mathcal{T}'$ is admissible). We can *repair* $\pi_{\mathcal{T}'}$ to compensate for the new choices defined by the new subtree. For this purpose, we construct a sub-MDP M_{fix} where we specify all actions in states defined inside $\mathcal{T}'|_n$ according to $\pi_{\mathcal{T}'}$. Formally, we define $M_{\mathsf{fix}} = (S, s_0, Act_{\mathsf{fix}}, P_{\mathsf{fix}}, R, \gamma)$ where $Act_{\mathsf{fix}}(s) = \{\pi_{\mathcal{T}'}(s)\}$ if $s \in \mathsf{st}^{\mathcal{T}'}(n)$, and $Act_{\mathsf{fix}}(s) = Act(s)$ otherwise. We then use any MDP solver to obtain the maximizing

policy $\pi^*_{\mathcal{T}'}$ for M_{fix}; note that $\pi^*_{\mathcal{T}'}$ coincides with $\pi_{\mathcal{T}'}$ for every state $s \in \mathsf{st}^{\mathcal{T}}(n)$. Finally, we use DTCONTROL to translate this policy into $\mathcal{T}'_{\mathsf{fix}}$, another perturbation of $\mathcal{T}'$.

4.3 DTNEST: OUR LOCAL SEARCH APPROACH

DTNEST is our implementation of the variable neighborhood search with subtree neighborhood exploration and tree perturbations outlined above. We now clarify how we select the hyperparameters for SNE and how we pick the perturbed tree for the next iteration.

Depth of subtrees. Before we start with an SNE step, we must determine on which neighborhood to run an SNE step. First, we only consider subtrees with a limited fixed depth d. Based on our testing, we set $d = 7$ as it produced the most consistent results. This allows us to focus on smaller neighborhoods where finding a good tree is easier, and it also means that all subtrees are disjunct, meaning we are not doing optimizations on similar state spaces multiple times. Moreover, in our benchmarks, the depth 7 subtrees typically cover a significant part of the DTs produced by DTCONTROL: their depth is between 6 and 19 and the larger trees are typically not balanced. Additionally, the bounded depth approaches [Vos and Verwer, 2023, Andriushchenko et al., 2025] work best for smaller depths, especially given a stricter timeout we discuss later.

Order of the neighborhoods. We order these subtrees (or rather, the root nodes for these subtrees) into a queue to first aim to improve the most promising subtrees. Our early experiments have shown that the order of the subtrees is not largely important, however, it can sometimes lead to smaller DTs more quickly. Intuitively, subtrees that require a large number of nodes to represent the policy for a small number of corresponding states are good candidates for splitting. For this, we use the following values: i) the value given by $\frac{\mathsf{st}^{\mathcal{T}}(n)}{|N_n|}$ where N_n are the descendant nodes from n in $\mathcal{T}$ ii) the number of predicates in the node n whose impurity [Ashok et al., 2021] value is close to the impurity value of the chosen predicate in n. The predicates with low impurity values represent a better chance of obtaining a small subtree when they are chosen. DTCONTROL chooses one predicate with the lowest impurity score, however, we try to leverage the fact that if there are many such predicates in a given node, there is a higher chance that DTCONTROL did not choose correctly. The impurity scores are a byproduct of DTCONTROL and are, therefore, computationally cheap proxies.

Timeout for SNE. While we could run every SNE step exhaustively, we instead exploit the anytime nature of the SNE solvers and use a timeout of 60 seconds. This allows us to consider more neighborhoods which our early experiments showed to be more crucial to the performance of the algorithm compared to trying to optimize the individual subtrees too much. We call an SNE step successful if we find a smaller subtree.

Closing the loop. Once a successful SNE step occurs and a tree $\mathcal{T}_i^{'}$ is obtained, the tree perturbation is performed. The smallest tree from the set $\{\mathcal{T}'_i, \mathcal{T}'_{i,\mathsf{alt}}, \mathcal{T}'_{i,\mathsf{fix}}\}$ is greedily picked as the new optimum for the next iteration. We experimented with trying to pick based on more factors, such as the current value of the tree or the depth, but it seemed inconsequential. If $\mathcal{T}_{i+1} = \mathcal{T}_i^{'}$ i.e. the smallest tree was obtained by SNE step, we add to the current subtree queue the newly-emerged subtrees of depth d in $\mathcal{T}_i^{'}$. If one of the other trees in TP was used we fully recompute the subtree queue. The next iteration begins with $\mathcal{T}_{i+1}$ and the new subtree queue with another SNE step.

5 EXPERIMENTAL EVALUATION

We now investigate the performance of our approach. DTNEST is implemented in Python, on top of DTPAYNT [Andriushchenko et al., 2025], DTCONTROL [Ashok et al., 2021], and STORM [Hensel et al., 2022], see also Sec. 3. The implementation and all benchmarks are publicly available[1]. Our evaluation aims to answer the following four questions:

Q1: *Does* DTNEST *scale to larger MDPs than* DTPAYNT*, the state-of-the-art approach for learning small DTs?*

Q2: *Does* DTNEST *learn smaller trees than* DTCONTROL *on large MDPs?*

Q3: *Does* DTNEST *outperform standard DT pruning?*

Q4: *What is the impact of tree perturbations steps?*

Experimental setting. All experiments were run on a machine with an AMD EPYC 9124 16-core CPU and 380GB RAM. Each experiments runs on a single core using a 64GB memory limit. The timeout for all experiments was 1 hour. All algorithms/tools considered in the experimental evaluation are deterministic and the timing variation is negligible.

Benchmarks. We consider two types of benchmarks: **(1)** 4 models from [Andriushchenko et al., 2025] including 2 models from [Vos and Verwer, 2023]. On these models, both DTPAYNT [Andriushchenko et al., 2025] and OMDT [Vos and Verwer, 2023] were not able to find a DT that is smaller than the DT found by DTCONTROL and has a normalized value better than 0.75. **(2)** 9 models from the standard MDP benchmarks from the QComp evaluation [Budde et al., 2020]. These models are parametrized, which allowed us to scale the size of the models. The left part of Tab. 1 shows the size of the underlying MDPs and the number of state variables (more detailed information about the models is reported in Appendix A.1). For fairness, we equip all models

[1] https://doi.org/10.5281/zenodo.15642002

with the action α_{rand} in all states, since DTPAYNT adds this action implicitly.

Baselines. DTPAYNT searches for a DT with the smallest depth having the desired value. We set the maximum depth to 10, since DTPAYNT is not able to effectively explore deeper DTs [Andriushchenko et al., 2025][2]. DTPAYNT terminates as soon as the first DT is found; we report the time it takes or the timeout. We run DTCONTROL with the same preprocessing that is part of DTNEST, i.e., we remove trivial choices and unreachable states. Note that this preprocessing is essential for the performance of DTCONTROL.

Results. For every benchmark, we run the different tools to obtain a resulting DT. We report the depth, the number of inner nodes, and the normalized relative error from the optimal value which captures the quality of the DT: 0 corresponds to no error (i.e. the DT represents an optimal policy), and 1 means the DT represents the uniform random policy, which can be represented by a 0-DT that chooses action α_{rand} in its only leaf node. Tab. 1 summarizes the experimental results for Q1 and Q2. The last column shows the relative size of the DTs produced by DTNEST compared to the smallest DT produced by DTPAYNT or DTCONTROL (smaller values are better for DTNEST, and values above 100% indicates an increase of the size).

Q1: DTNEST VS. DTPAYNT

In Tab. 1, we consider a bound 0.05 on the normalized relative error (the part of the input) and report the smallest DTs found by the tools. We observe that in 8 out of 13 models DTPAYNT is unable to find DT with relative error less than 0.05; it reaches either the 1-hour timeout or the memory limit. In contrast, DTNEST is able to find DTs with the required value for all the models. For 3 out of 5 models where DTPAYNT was able to find the desired tree, DTNEST finds a significantly smaller alternative. In the remaining 2 cases, two variants of *firew* model, DTs of the depth 2 are sufficient to achieve the desired value. The incremental search strategy implemented in DTPAYNT was able to find these DTs after a few minutes, while DTNEST does not allow DTPAYNT to search that long on a single subtree. However, for any benchmark that does not allow for a policy with a tiny tree, DTNEST scales significantly further than DTPAYNT (or OMDT).

Q2: DTNEST VS. DTCONTROL

Recall that DTCONTROL favors scalability over minimality of the resulting DTs, for the MDPs in this benchmark, it is therefore very fast. Note that DTCONTROL always maps the optimal policy i.e. it has error 0. The results in Tab. 1 show that DTNEST produces significantly smaller DTs than DTCONTROL. In many cases, the reduction is substantial both in terms of the size and the depth: from hardly explainable DTs having over 60 or even 200 nodes, we get DTs with around 10 to 20 nodes representing an explainable policy achieving almost optimal values.

To showcase the flexibility of our approach, we increased the acceptable error to 10% to obtain even more explainable policies. In 10 out of 13 models, DTNEST finds smaller DTs compared to what DTNEST found with 5% error. These DTs are on average 35% smaller, and the greatest decrease happened on *pacman-30*, from 23 to 10 nodes, and on *wlan-4*, from 25 to 12 nodes. Complete results for error threshold set to 10% can be found in Appendix A.2 in Tab. 5.

Q3: DTNEST VS. TREE PRUNING

We also compare DTNEST to the standard DT pruning [Mitchell, 1997], which would be the straightforward scalable solution assembled from the previously available components. Starting from an exact DT representation of an optimal policy (delivered by DTCONTROL), the idea is to iteratively merge leaf nodes and run a model-checker to ensure that the value of the candidate DTs is still above the given threshold (in this case, given as 5% normalized relative error). This idea was implemented in Brázdil et al. [2015] (together with learning the full tree using a heuristic weighing the decisions by their "importance"). However, due to the unavailability of that code, we use an alternative implementation of pruning available in DTCONTROL. Tab. 2 summarizes the results [3]; it reports the relative size reduction (in terms of the number of nodes) compared to the DT (provided by DTCONTROL) that represents the optimal policy. Note that DTNEST is able to perform a preprocessing step on the optimal policy which removes unreachable states. This preprocessing was not possible with the available implementation of pruning and, therefore, to compare only the effect of these techniques and not the effect of preprocessing, the column size% reports the reduction from the preprocessed policy. We see that except for the *firew-false* model, DTNEST achieves a significantly better reduction than the pruning technique. These results clearly show that for more complicated policies, a simple pruning technique is not sufficient. Additionally, we include the column size$\star$% which also includes the effect of preprocessing on the reduction for DTNEST.

Q4: ABLATION STUDY

This section investigates the impact of the tree perturbations in DTNEST. We have implemented DTNEST†, a simpler variant of DTNEST where DTCONTROL is called only once,

[2]For the *cons-6-2* and *ij-14-s* models requiring larger DTs, we tried to increase the maximal depth, but it did not help.

[3]The implementation does not support discounted rewards and thus the table includes only a subset of the models

Table 1: Comparison between DTPAYNT, DTCONTROL and DTNEST. DTCONTROL maps the optimal policy to a DT, while DTPAYNT and DTNEST search for DTs with the normalised relative error at most 0.05. "-" indicates that DTPAYNT reached the 1-hour timeout or the 64GB memory limit. For DTNEST, we also report the number of iterations (the number of sub-trees analyzed using DTPAYNT) and the relative size (i.e. number of nodes) compared to the smallest tree produced by the other three methods (smaller values are better for DTNEST). DTNEST can always finish the last iteration, which explains a >1h run time.

model	$\|S\|$	$\|\mathcal{V}\|$	DTPAYNT				DTCONTROL			DTNEST					
			nodes	depth	time	error	nodes	depth	time	nodes	depth	time	iters	error	size%
sys-ad-2	256	3	85	8	999s	0.00	41	8	<1s	**7**	7	205s	9	0.04	17%
maze-7	2k	7			-		280	14	<1s	**19**	8	640s	44	0.05	7%
tictactoe	2k	21	31	5	488s	0.00	25	6	<1s	**15**	6	161s	9	0.05	60%
wlan-1-2	3k	10			-		68	11	<1s	**14**	8	423s	19	0.04	21%
ij-14-s	16k	9			-		2222	14	1s	**397**	14	3636s	212	0.05	18%
cons-4-2	23k	7			-		70	12	<1s	**25**	7	1396s	33	0.00	36%
csma-3-2	34k	12	24	5	573s	0.00	110	11	<1s	**8**	5	149s	11	0.00	33%
firew-false	212k	10	**3**	2	260s	0.04	12	7	<1s	7	5	237s	8	0.04	233%
wlan-4	350k	10			-		63	11	4s	**25**	7	2242s	39	0.05	39%
pacman-30	850k	9			-		144	14	1s	**23**	9	2604s	60	0.05	16%
firew-true	1.1M	11	**3**	2	463s	0.01	12	8	2s	5	4	587s	7	0.01	167%
cons-6-2	1.2M	9			-		212	14	<1s	**106**	12	3755s	32	0.03	50%
csma-3-4	1.5M	10			-		236	19	13s	**15**	6	2157s	39	0.03	6%

Table 2: Comparison between DTNEST and a tree pruning technique implemented in a new version of DTCONTROL. It reports the normalized relative error and the relative size of the resulting DTs compared to the DT (computed by DTCONTROL) that represents the optimal policy. size$\star$% reports size reduction including a pre-processing step on the initial optimal policy which we were unable to perform for the Pruning approach.

model	Pruning		DTNEST		
	error	size%	error	size%	size$\star$%
firew-false	0.03	**42%**	0.04	58%	21%
firew-true	0.07	52%	0.01	**42%**	12%
csma-3-2	0.0	40%	0.0	**7%**	7%
csma-3-4	0.0	97%	0.03	**6%**	4%
cons-4-2	0.04	66%	0.0	**36%**	20%
ij-14-s	0.0	100%	0.05	**18%**	18%
pacman-30	0.0	94%	0.04	**16%**	9%
wlan-4	0.0	100%	0.04	**39%**	1%

Table 3: Comparison between DTNEST and DTNEST†. It reports the average relative size of the resulting DTs (achieved over 4 different initial DTs) compared to the size of the initial DT (again, the lower the numbers are the better).

model	DTNEST† size%	DTNEST size%
sys-ad-2	17%	**15%**
maze-7	6%	6%
tictactoe	**42%**	56 %
wlan-1-2	**16%**	18%
ij-14-s	21%	**15%**
cons-4-2	34%	**31%**
csma-3-2	13%	**6%**
firew-false	54%	**52%**
wlan-4	**19%**	25%
pacman-30	43%	**24 %**
firew-true	**32%**	33%
cons-6-2	59%	**36%**
csma-3-4	11%	**7%**

to obtain the initial DT $\mathcal{T}$ for the optimal policy π^*, and only SNEs are performed using DTPAYNT with no tree perturbations. To study the impact of perturbations and the ability to escape local optima, we ran DTCONTROL with different parameters, which gave us, for each model, four different initial DTs with different topologies, sizes and impurity values. Tab. 3 summarizes the results: it reports the average relative size (over the 4 runs) of the resulting DTs produced by DTNEST and DTNEST†, compared to the size of the initial DT. The complete results can be found in Appendix A.3.

On most of the models, DTNEST provides slightly better results showing that while DTNEST† performs very well, the perturbations play a part in the effectiveness of the approach. Most extreme examples are models *cons-6-2* where in one case DTNEST finds a 10 times smaller tree compared to DTNEST† and *csma-3-2* where on the biggest initial tree of

size 1856 DTNEST gets a tree with just 8 nodes compared to 603 nodes produced by DTNEST†. However, overall these results show that the more important part of DTNEST is the SNE.

6 RELATED WORK

Decision trees have been used for a *post-hoc* representation of policies with guarantees on ε-optimality since Brázdil et al. [2015]. This seminal paper (i) learns a DT using a heuristic assigning "importance" to each decision and then (ii) applies standard DT pruning (of the C4.5 algorithm [Mitchell, 1997]) step by step until the given imprecision ε is incurred. In contrast, most of the subsequent work has focused on representing the policies *exactly*, e.g., [Brázdil et al., 2018, Ashok et al., 2019a,b, Jüngermann et al., 2023, Budde et al., 2024], notably the tool dtControl [Ashok et al., 2020, 2021]. The precise representation is particularly useful for detecting bugs [Brázdil et al., 2015, Ashok et al., 2021] and validation of modeling [Kiesbye et al., 2022]. Binary decision diagrams (BDDs) can also be used for this purpose with the caveats described in [Chakraborty et al., 2025] and their generalizations even for computing optimal policies Hoey et al. [1999], de Alfaro et al. [2000].

DTs have also been used *during* the search for optimal policies, both in classical dynamic programming [Boutilier et al., 1995, Boutilier and Dearden, 1996] and in reinforcement learning [Pyeatt and Howe, 1998, Gupta et al., 2015, Topin et al., 2021], recently also to generalize policies on small instances of parametrized models to larger ones [Azeem et al., 2025].

Methods for post-hoc representation without guarantees on the value typically process a more complex teacher policy into a DT via imitation learning Bastani et al. [2018] or distillation Kohler et al. [2024]. Alternatively, the underlying value functions can be approximated using linear models [Du et al., 2020] or shallow neural networks Rusu et al. [2015], Julian et al. [2016]. However, these approaches have been applied primarily in the context of reinforcement learning and typically do not provide any optimality guarantees.

7 CONCLUSION

We have provided a new method to represent MDP policies approximately (with a given bound on the suboptimality) by much smaller trees than the state-of-the-art approaches. In contrast to the tools providing exact representation, our smaller trees provide a more realistic tackle at explainability. In contrast to the methods providing the smallest possible trees, our method easily scales to million-states MDPs. Finally, in contrast to scalable imprecise methods such as pruning, our method analyzes the trees with more care, achieving both smaller trees and better precision. Altogether, the combination of a DT-learning tool, a model checker and an inductive SMT learner results in a balanced synergy. In future work, we will aim at even closer intertwining of the tools, so that the DT-learner provides even a richer set of guidance to the inductive learner, assisted by more frequent but also more approximate feedback from the model checker. Future extensions also include working with richer domain-specific predicates to improve tree compactness and explainability, and designing better perturbation approaches to escape local optima more efficiently.

Acknowledgments. This work has been supported by the Czech Science Foundation grant GA23-06963S (VESCAA), the IGA VUT project FIT-S-23-8151, the NWO VENI Grant ProMiSe (222.147), MUNI Award in Science and Humanities grant (MUNI/I/1757/2021) and the ERC project InOVationCS (101171844).

References

Roman Andriushchenko, Milan Češka, Sebastian Junges, and Filip Macák. Small decision trees for MDPs with deductive synthesis. In *CAV (to appear)*, 2025.

Pranav Ashok, Tomás Brázdil, Krishnendu Chatterjee, Jan Kretínský, Christoph H. Lampert, and Viktor Toman. Strategy representation by decision trees with linear classifiers. In *QEST*, 2019a.

Pranav Ashok, Jan Křetínský, Kim Guldstrand Larsen, Adrien Le Coënt, Jakob Haahr Taankvist, and Maximilian Weininger. SOS: safe, optimal and small strategies for hybrid Markov decision processes. In *QEST*, 2019b.

Pranav Ashok, Mathias Jackermeier, Pushpak Jagtap, Jan Kretínský, Maximilian Weininger, and Majid Zamani. dtControl: decision tree learning algorithms for controller representation. In *HSCC*, 2020.

Pranav Ashok, Mathias Jackermeier, Jan Křetínský, Christoph Weinhuber, Maximilian Weininger, and Mayank Yadav. dtControl 2.0: Explainable strategy representation via decision tree learning steered by experts. In *TACAS*, 2021.

Muqsit Azeem, Debraj Chakraborty, Sudeep Kanav, Jan Křetínský, Mohammadsadegh Mohagheghi, Stefanie Mohr, and Maximilian Weininger. 1–2–3–Go! policy synthesis for parameterized Markov decision processes via decision-tree learning and generalization. In *VMCAI*, 2025.

Christel Baier, Luca de Alfaro, Vojtěch Forejt, and Marta Kwiatkowska. Model checking probabilistic systems. In *Handbook of Model Checking*. 2018.

Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. *NeurIPS*, 31, 2018.

Kevin Batz, Tom Jannik Biskup, Joost-Pieter Katoen, and Tobias Winkler. Programmatic strategy synthesis: Resolving nondeterminism in probabilistic programs. *Proceedings of the ACM on Programming Languages*, 8(POPL), 2024.

DP Bertsekas. Neuro-dynamic programming. *Athena Scientific*, 1996.

Craig Boutilier and Richard Dearden. Approximate value trees in structured dynamic programming. In *ICML*, 1996.

Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Exploiting structure in policy construction. In *IJCAI*, 1995.

Tomás Brázdil, Krishnendu Chatterjee, Martin Chmelik, Andreas Fellner, and Jan Kretínský. Counterexample explanation by learning small strategies in Markov decision processes. In *CAV*, 2015.

Tomás Brázdil, Krishnendu Chatterjee, Jan Kretínský, and Viktor Toman. Strategy representation by decision trees in reactive synthesis. In *TACAS*, 2018.

Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. 1984.

Carlos E Budde, Arnd Hartmanns, Michaela Klauck, Jan Křetínský, David Parker, Tim Quatmann, Andrea Turrini, and Zhen Zhang. On correctness, precision, and performance in quantitative verification: QComp 2020 competition report. In *ISoLA*, 2020.

Carlos E. Budde, Pedro R. D'Argenio, and Arnd Hartmanns. Digging for decision trees: A case study in strategy sampling and learning. In *AISoLA*, 2024.

Debraj Chakraborty, Clemens Dubslaff, Sudeep Kanav, Jan Kretinsky, and Christoph Weinhuber. Explaining control policies through predicate decision diagrams. In *HSCC*, 2025.

Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. Uppaal stratego. In *TACAS*, 2015.

Luca de Alfaro, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Roberto Segala. Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In *TACAS*, 2000.

Simon S Du, Sham M Kakade, Ruosong Wang, and Lin F Yang. Is a good representation sufficient for sample efficient reinforcement learning? In *ICML*, 2020.

Ujjwal Das Gupta, Erik Talvitie, and Michael Bowling. Policy tree: Adaptive representation for policy gradient. In *AAAI*, 2015.

Pierre Hansen, Nenad Mladenović, Jack Brimberg, and José A. Moreno Pérez. Variable neighborhood search. In *Handbook of Metaheuristics*. 2019.

Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker Storm. *Int. J. Softw. Tools Technol. Transf.*, 2022.

Jesse Hoey, Robert St-Aubin, Alan J. Hu, and Craig Boutilier. SPUDD: stochastic planning using decision diagrams. In *UAI*, 1999.

Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. Policy compression for aircraft collision avoidance systems. In *DASC*, 2016.

Florian Jüngermann, Jan Kretínský, and Maximilian Weininger. Algebraically explainable controllers: decision trees and support vector machines join forces. *Int. J. Softw. Tools Technol. Transf.*, 2023.

Jonis Kiesbye, Kush Grover, Pranav Ashok, and Jan Kretínský. Planning via model checking with decision-tree controllers. In *ICRA*, 2022.

Hector Kohler, Quentin Delfosse, Riad Akrour, Kristian Kersting, and Philippe Preux. Interpretable and editable programmatic tree policies for reinforcement learning. *Preprint arXiv:2405.14956*, 2024.

Tom M. Mitchell. *Machine learning, International Edition*. 1997.

Nenad Mladenovic and Pierre Hansen. Variable neighborhood search. *Comput. Oper. Res.*, 1997.

Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. 1994.

Larry D Pyeatt and Adele E Howe. Decision tree function approximation in reinforcement learning. *Computer Science Technical Report*, 1998.

J. Ross Quinlan. Induction of decision trees. *Mach. Learn.*, 1986.

Andrei A Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. *arXiv preprint arXiv:1511.06295*, 2015.

Nicholay Topin, Stephanie Milani, Fei Fang, and Manuela Veloso. Iterative bounding mdps: Learning interpretable policies via non-interpretable methods. In *AAAI*, 2021.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *ICML*, 2018.

Daniël Vos and Sicco Verwer. Optimal decision tree policies for Markov decision processes. In *IJCAI*, 2023.

A EXPERIMENTAL RESULTS

In this section, we present all the results of the experiments in the main part of the paper. Namely, we provide detailed model information, we add the results of DTNEST for the normalized relative error set to 0.1, and we provide the complete results for our ablation study experiments.

A.1 MODEL INFO

Table 4 presents detailed model information for all the models considered in our experimental evaluation.

Table 4: Detailed model information. Column $|S|$ denotes the number of states, $|\mathcal{V}|$ the number of features, $|Act|$ the number of actions, "choices" the number of available actions (i.e. $\sum_{s \in S} |Act(s)|$), and "opt"/"rand" the value of the optimal and uniform random policy respectively.

model	$\|S\|$	$\|\mathcal{V}\|$	$\|Act\|$	choices	opt	rand
sys-ad-2	256	3	10	3k	241.1	111.6
maze-7	2k	7	5	10k	5.18	1.22
tictactoe	2k	21	10	24k	0.97	-0.81
wlan-1-2	3k	10	34	106k	0.11	0.03
ij-14-s	16k	9	16	57k	0.03	0.0
cons-4-2	23k	7	26	61k	363	234.8
csma-3-2	34k	12	19	36k	0.59	0.9
firew-false	212k	10	15	479k	365	185.6
wlan-4	350k	10	45	440k	227k	36k
pacman-30	850k	9	12	1.1M	0.55	0.96
firew-true	1.1M	11	15	1.5M	299	185.7
cons-6-2	1.2M	9	38	5M	867.1	520.7
csma-3-4	1.5M	10	25	1.5M	116.8	111.1

A.2 DIFFERENT ERROR THRESHOLD

Table 5 presents the results for DTNEST in experiments where the error threshold was set to 0.1. Table 6 shows results for the case when we use a tighter bound on the relative error (up to 0.01%).

A.3 COMPLETE ABLATION STUDY

Table 7 shows the complete results of the ablation study. For each model, we ran DTCONTROL with 4 different settings to obtain different initial DTs. This means that in total we have $13 \cdot 4 = 52$ benchmarks. Note that even in cases when DTCONTROL produces the same tree, the impurity values are different and therefore can lead to different result.

B DECISION TREE VISUALIZATION

In this section we provide visualizations for some of the DTs produced by DTNEST. We also provide visualization for DT produced by DTCONTROL on the *maze-7* benchmark to showcase the improved explainability of DTs produced by DTNEST.

B.1 INTERPRETABILITY COMPARISON

In Figures 4a and 4b we show two DTs for benchmark *maze-7*, one produced by DTNEST and one produced by DTCONTROL, respectively. These visualizations clearly show the better interpretability and the potential of use for DTs produced by DTNEST compared to the ones produced by DTCONTROL.

Table 5: Results for DTNEST with the normalized relative error threshold set to 0.1. DTNEST-0.05 represent the values reported in Tab 1 in the main paper. The last column reports the relative size of the produced tree when increasing the error threshold.

model	$\|S\|$	DTNEST-0.05		DTNEST-0.1					
		nodes	depth	nodes	depth	time	iters	error	size%
sys-ad-2	256	7	7	7	7	203s	9	0.04	100%
maze-7	2k	19	8	16	7	843s	45	0.1	**84%**
tictactoe	2k	15	6	8	6	89s	5	0.1	**53%**
wlan-1-2	3k	14	8	9	5	360s	17	0.07	**64%**
ij-14-s	16k	397	14	233	13	3619s	220	0.1	**58%**
cons-4-2	23k	25	7	23	7	1474	35	0.0	**92%**
csma-3-2	34k	8	5	5	3	63s	6	0.09	**63%**
firew-false	212k	7	5	4	3	179s	6	0.09	**57%**
wlan-4	350k	25	7	12	5	2056s	35	0.1	**48%**
pacman-30	850k	23	9	10	5	1254s	20	0.09	**43%**
firew-true	1.1M	5	4	5	4	589s	7	0.01	100%
cons-6-2	1.2M	106	12	106	12	3753s	32	0.03	100%
csma-3-4	1.5M	15	6	13	6	2649s	52	0.09	**86%**

B.2 ANOTHER VISUALIZATION

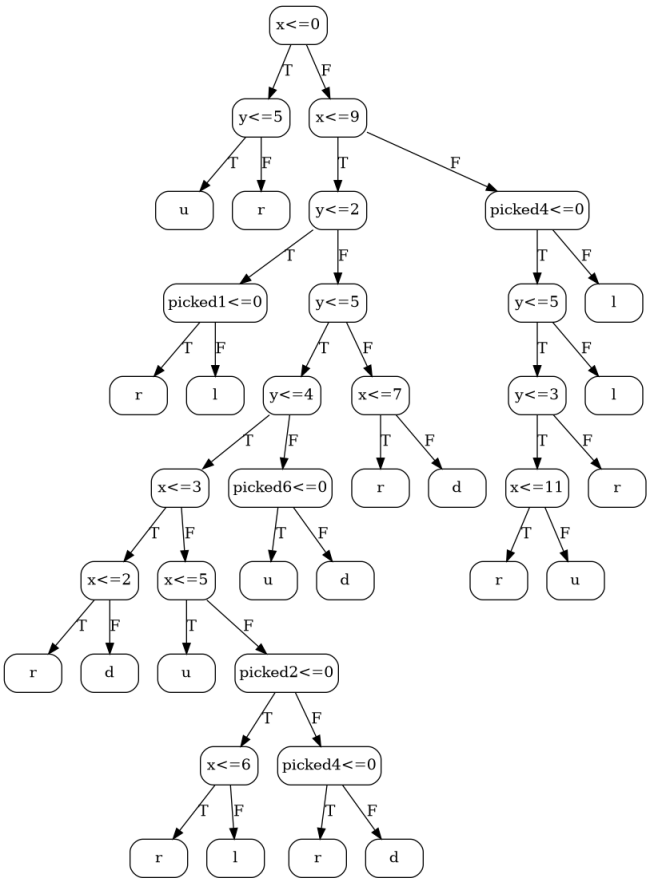
In Fig. 5 we provide another visualization for a DT produced by DTNEST, this time for the benchmark *csma-3-2*. Compared to the *maze-7* benchmark in *csma-3-2* not every action is available in each state, nonetheless, as can be seen, none of the leaf nodes contain the random action α_{rand}.

Table 6: Results for DTNEST with the normalized relative error threshold set to 0.01 and 0.0001 (1% and 0.01% respectively). DTNEST-0.05 represent the values reported in Tab 1 in the main paper. The size% columns show the relative size of the produced DT compared to the size of DTCONTROL tree (in terms of number of nodes). The error 0 means the relative error was below 1×10^{-6}

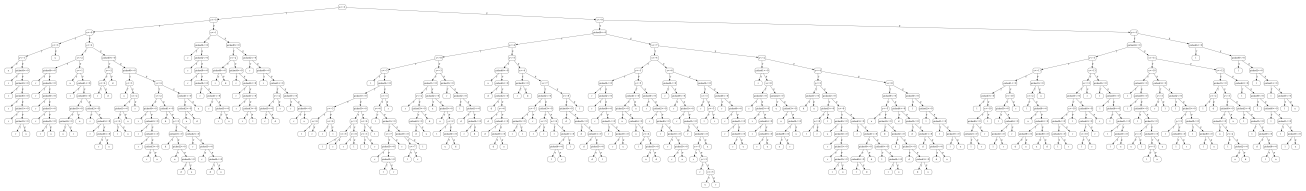
model	$\|S\|$	DTCONTROL		DTNEST-0.05		DTNEST-0.01					DTNEST-0.0001				
		nodes	depth	nodes	depth	nodes	depth	time	error	size%	nodes	depth	time	error	size%
sys-ad-2	256	41	8	7	7	8	8	122s	0.004	20%	22	8	461s	0.0001	54%
maze-7	2k	280	14	19	8	45	9	955s	0.01	16%	89	10	2039s	0.0001	32%
tictactoe	2k	25	6	15	6	19	6	141s	0.008	76%	24	6	143s	0	96%
wlan-1-2	3k	68	11	14	8	19	9	471s	0.008	28%	19	9	518s	0	28%
ij-14-s	16k	2222	14	397	14	1003	14	3605s	0.007	45%	1695	14	1397s	0.0001	76%
cons-4-2	23k	70	12	25	7	25	7	1437s	0.0003	36%	25	7	1281s	0	36%
csma-3-2	34k	110	11	8	5	8	5	149s	0	7%	8	5	149s	0	7%
firew-false	212k	12	7	7	5	8	6	198s	0.003	67%	8	6	200s	0	67%
wlan-4	350k	63	11	25	7	25	7	3626s	0.008	37%	38	9	2603s	0.0001	60%
pacman-30	850k	144	14	23	9	31	10	1360s	0.007	22%	96	11	1551s	0	67%
firew-true	1.1M	12	8	5	4	5	4	654s	0.01	42%	7	5	538s	0	58%
cons-6-2	1.2M	212	14	106	12	169	14	3704s	0.005	80%	203	14	3773s	0	96%
csma-3-4	1.5M	236	19	15	6	18	6	3636s	0.007	8%	57	10	3644s	0	24%

Table 7: Complete results for the ablation study experiments.

model	$\|S\|$	DTCONTROL		DTNEST†					DTNEST				
		nodes	depth	nodes	depth	time	iters	error	nodes	depth	time	iters	error
sys-ad-2	256	41	8	7	7	246s	11	0.04	7	7	205s	9	0.04
sys-ad-2	256	41	8	7	7	245s	11	0.04	7	7	204s	9	0.04
sys-ad-2	256	41	8	7	7	245s	11	0.04	7	7	204s	9	0.04
sys-ad-2	256	80	8	14	6	432s	25	0.05	**7**	7	188s	8	0.05
maze-7	2k	280	14	**18**	7	1704s	59	0.05	19	8	638s	44	0.05
maze-7	2k	280	14	**18**	7	1701s	59	0.05	19	8	638s	44	0.05
maze-7	2k	277	17	21	7	2995s	80	0.04	**20**	9	928s	40	0.05
maze-7	2k	1146	14	19	6	1433s	81	0.04	**18**	7	929s	59	0.05
tictactoe	2k	25	6	**10**	4	341s	16	0.05	15	6	161s	9	0.05
tictactoe	2k	25	6	**10**	4	342s	16	0.05	15	6	161s	9	0.05
tictactoe	2k	30	8	**14**	5	599s	21	0.05	15	6	185s	10	0.05
tictactoe	2k	28	6	**11**	4	425s	15	0.04	15	6	161s	9	0.05
wlan-1-2	3k	68	11	**13**	6	679s	29	0.04	14	8	425s	19	0.04
wlan-1-2	3k	68	11	**13**	6	679s	29	0.04	14	8	423s	19	0.04
wlan-1-2	3k	66	11	**14**	5	819s	30	0.04	16	9	459s	19	0.02
wlan-1-2	3k	321	13	**16**	8	583s	31	0.04	22	7	323s	18	0.03
ij-14-s	16k	2222	14	481	14	3721s	133	0.05	**397**	14	3646s	212	0.05
ij-14-s	16k	2222	14	481	14	3715s	133	0.05	**397**	14	3627s	212	0.05
ij-14-s	16k	2243	14	403	14	3671s	148	0.05	**255**	12	3646s	206	0.05
ij-14-s	16k	4526	14	1115	14	3608s	169	0.05	**497**	14	3725s	200	0.05
cons-4-2	23k	70	12	27	8	1253s	32	0.01	**25**	7	1386s	33	0.0
cons-4-2	23k	70	12	27	8	1251s	32	0.01	**25**	7	1387s	33	0.0
cons-4-2	23k	72	14	27	7	1019s	34	0.0	**25**	8	1500s	36	0.0
cons-4-2	23k	198	11	41	11	2319s	59	0.04	**32**	12	1953s	61	0.03
csma-3-2	34k	110	11	9	4	64s	10	0.0	**8**	5	149s	11	0.0
csma-3-2	34k	110	11	9	4	64s	10	0.0	**8**	5	149s	11	0.0
csma-3-2	34k	112	14	**5**	4	63s	10	0.0	8	5	149s	11	0.0
csma-3-2	34k	1856	15	603	14	20s	26	0.0	**8**	5	141s	10	0.0
firew-false	212k	12	7	7	5	238s	9	0.04	7	5	237s	8	0.04
firew-false	212k	12	7	7	5	240s	9	0.04	7	5	236s	8	0.04
firew-false	212k	12	7	7	5	238s	9	0.04	7	5	237s	8	0.04
firew-false	212k	12	4	5	3	151s	4	0.04	**4**	3	114s	3	0.04
wlan-4	345k	63	11	**16**	7	2348s	35	0.03	25	7	2250s	39	0.05
wlan-4	345k	63	11	**16**	7	2359s	35	0.03	25	7	2248s	39	0.05
wlan-4	345k	72	13	18	7	2486s	37	0.04	**13**	5	2122s	29	0.05
wlan-4	345k	1534	16	17	7	3207s	85	0.05	**16**	7	2154s	36	0.04
pacman-30	853k	144	14	61	11	4128s	22	0.01	**23**	7	2615s	60	0.05
pacman-30	853k	144	14	61	11	4070s	22	0.01	**23**	7	2606s	60	0.05
pacman-30	853k	136	14	**23**	7	2915s	65	0.04	66	14	3649s	36	0.05
pacman-30	853k	230	15	158	9	3964s	11	0.0	**38**	8	2939s	32	0.04
firew-true	1.1M	12	8	5	4	632s	8	0.01	5	4	588s	7	0.01
firew-true	1.1M	12	8	5	4	632s	8	0.01	5	4	590s	7	0.01
firew-true	1.1M	12	7	5	4	638s	8	0.01	5	4	597s	7	0.01
firew-true	1.1M	88	10	**4**	3	402s	7	0.0	5	4	563s	8	0.01
cons-6-2	1.2M	212	14	133	14	3689s	38	0.04	**106**	12	3722s	32	0.03
cons-6-2	1.2M	212	14	133	14	3706s	38	0.04	**106**	12	3742s	32	0.03
cons-6-2	1.2M	213	19	**62**	14	3701s	54	0.01	75	14	3690s	43	0.0
cons-6-2	1.2M	1666	16	1343	16	3757s	35	0.05	**159**	14	3737s	29	0.04
csma-3-4	1.5M	236	19	**14**	6	2289s	35	0.03	15	6	2162s	39	0.03
csma-3-4	1.5M	236	19	**14**	6	2265s	35	0.03	15	6	2149s	39	0.03
csma-3-4	1.5M	229	19	38	10	3635s	30	0.03	**27**	9	3667s	39	0.03
csma-3-4	1.5M	417	17	72	10	3783s	30	0.03	**15**	6	2529s	50	0.03



(a)



(b)

Figure 4: (a) DT produced by DTNEST on *maze-7* benchmark. (b) DT produced by DTCONTROL on *maze-7* benchmark.

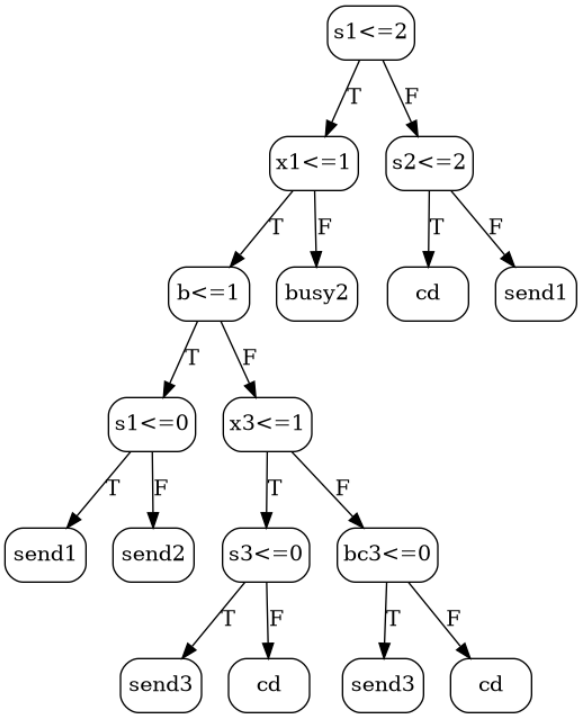


Figure 5: DT produced by DTNEST on *csma-3-2* benchmark.