Safety Verification of Additive Tree Ensemble Policies via Symbolic Search

Marcel Schubert and **Daniel Höller**

Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
schubert@cs.uni-saarland.de, hoeller@cs.uni-saarland.de

Abstract

There is a large body of research on learning action policies, i.e., functions that map environment states to an action to perform. Unfortunately, such policies usually do not come with guarantees regarding the quality of their decisions. One way to provide safety guarantees for such policies is to use verification techniques *after* the policy has been trained. Verification answers whether states which are considered *unsafe* are reachable *when following the policy*. This task involves reasoning about large parts of the state space as well as the policy itself, which makes it hard. Recent work has shown that *Tree Ensemble* representations are similarly well suited to *learn* action policies as neural networks, but simplify verification. A well-known technique to compactly represent large state spaces while still enabling the necessary reasoning is *Symbolic Search*, which has been used in model checking as well as in planning. In this paper, we introduce a method to verify action policies represented as Tree Ensembles by using Binary Decision Diagrams (BDDs). We present two approaches to represent the policies as BDDs, one involving an intermediate translation to Algebraic Decision Diagrams. We integrate these policy representations into a BDD-based symbolic search. Our results show that especially an incremental computation during the search works well. On the standard benchmark set, our system outperforms the related work based on Policy Predicate Abstraction.

1 Introduction

The goal in AI planning is to find sequences of actions that achieve certain objectives in an environment. While this is sufficient in deterministic environments, there are cases where it cannot be used effectively, e.g. when the environment is non-deterministic, but also when the initial state is not fully specified or unknown in advance (while the environment is still deterministic). To nevertheless realize goal-directed behavior in such environments, one way is to learn an *action policy*, a function mapping the current state to an action to perform. One increasingly popular approach to obtaining such policies is training *Neural Networks*. However, a major drawback of ML-based policies is that they usually do not come with guarantees regarding their behavior, which might be prohibitive in safety-critical environments.

One way to circumvent this problem is to perform safety verification *after* training. The goal of verification methods is to prove that certain unsafe states of the environment are not reached *when following the policy*. This task involves reasoning about large parts of the state space as well as the policy itself. Prior work introduced *Policy Predicate Abstraction* (PPA) to verify Neural Network-based policies (Vinzent, Steinmetz, and Hoffmann 2022; Vinzent, Sharma, and Hoffmann 2023). Predicate abstraction clusters a set of environment states into abstract states, which decreases the size of the state space while over-approximating the set of reachable states. More recently, it has been shown that *Additive Tree Ensembles* are equally suited to learn action policies, while being better suited for verification (Jain et al. 2024). This work still uses predicate abstraction to reason about the large state space.

In this work, we introduce a method for verifying the safety of *Additive Tree Ensemble Policies* based on symbolic search using *Binary Decision Diagrams* (BDDs). BDDs enable a compact representation of large sets of states, while still allowing reasoning operations that can be used to realize state transition *over sets of states*. They have been successfully applied in AI planning (Kissmann and Edelkamp 2011; Torralba et al. 2017) and we can build upon these techniques.

In the following, we first show how to encode the additive tree ensemble policy's decisions into so-called policy region BDDs, allowing us to restrict the symbolic search to only expand actions on states where an action is selected by the policy. Next, we show that it is sufficient to compute the policy regions only on the reached states in each iteration, and describe how this can be exploited. Finally, we evaluate our approach against the state-of-the-art PPA verification approach on existing additive tree ensemble verification benchmarks. The results show that symbolic search needs less time for verification, and it can verify more policies than the PPA-based approach.

2 Background

In this section we give the necessary background in planning, safety verification, tree ensembles, and decision diagrams.

2.1 Policy Safety Verification

Similar to related work (Vinzent, Steinmetz, and Hoffmann 2022), our work is based on non-deterministic state spaces with bounded integer variables.

A *planning task* is a tuple $\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$, where

- $\mathcal{V}$ is a set of *state variables*, and each variable $v \in \mathcal{V}$ has a non-empty bounded integer interval domain D_v,
- $\mathcal{L}$ is a set of *action labels*,
- and $\mathcal{O}$ is a set of *operators*, where an *operator* $o \in \mathcal{O}$ is a tuple (g, ℓ, u) with *guard* $g \in C$, *label* $\ell \in \mathcal{L}$, and *update* $u \colon \mathcal{V} \to \mathit{Exp}$.

All sets in the definition are finite. A *linear integer expression* over $\mathcal{V}$ is an expression of the form

$$d_1 \cdot v_1 + \cdots + d_r \cdot v_r + c$$

with $v_1, \ldots, v_r \in \mathcal{V}$ and $d_1, \ldots, d_r, c \in \mathbb{Z}$. We denote by Exp the set of all linear integer expressions over $\mathcal{V}$. C denotes the set of *linear integer constraints* over $\mathcal{V}$, given by the grammar

$$\Phi ::= e_1 \leq e_2 \mid e_1 = e_2 \mid e_1 \geq e_2 \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi$$

with $e_1, e_2 \in \mathit{Exp}$.

A *state* s over $\mathcal{V}$ is a function $\mathcal{V} \to \mathbb{Z}$ assigning each variable a value from its domain, i.e., $s(v) \in D_v$ for $v \in \mathcal{V}$. Let $\mathcal{S}$ be the set of all states. Given a state $s \in \mathcal{S}$, the evaluation of an expression $e \in \mathit{Exp}$ under s, denoted $e(s)$, is a value in $\mathbb{Z}$ given by

$$e(s) = d_1 \cdot s(v_1) + \cdots + d_r \cdot s(v_r) + c.$$

$[\![\phi]\!]_s$ denotes the evaluation of a constraint $\phi \in C$ in a state $s \in \mathcal{S}$ recursively defined as:

$$\begin{aligned} [\![e_1 \bowtie e_2]\!]_s &= e_1(s) \bowtie e_2(s) \\ [\![\neg\phi]\!]_s &= \neg[\![\phi]\!]_s \\ [\![\phi \wedge \psi]\!]_s &= [\![\phi]\!]_s \wedge [\![\psi]\!]_s \\ [\![\phi \vee \psi]\!]_s &= [\![\phi]\!]_s \vee [\![\psi]\!]_s \end{aligned}$$

with $\bowtie \ \in \{\leq, =\geq\}$. We use $s \models \phi$ as shorthand for $[\![\phi]\!]_s$ evaluating to true. We write $[\phi]$ for the set $\{s \in \mathcal{S} \mid s \models \phi\}$.

Definition 1 (State Space)**.** *The* state space *of a planning task $\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$ is a labeled transition system (LTS) $\Theta = \langle \mathcal{S}, \mathcal{L}, \mathcal{T} \rangle$ where $\mathcal{S}$ is the set of states over $\mathcal{V}$ and $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is the set of transitions. There is a transition $(s, \ell, s') \in \mathcal{T}$ if and only if there exists an operator $o = (g, \ell, u) \in \mathcal{O}$ such that $s \models g$ and $s' = \{v \mapsto u(v)(s) \mid v \in \mathcal{V}\}$, i.e., an operator with label ℓ such that the guard constraint g evaluates to true in s and s' maps each variable to the integer resulting from evaluating the expression $u(v)$ under s*[1].

The following definitions are adapted from Vinzent and Hoffmann (2024).

Definition 2 (Reach-Avoid Property)**.** *A* reach-avoid property *is a tuple $\langle \phi_0, \phi_u, \phi_g \rangle$ where $\phi_0 \in C$ is the initial condition, $\phi_u \in C$ is the unsafety condition, and $\phi_g \in C$ is the goal condition.*

Definition 3 (State Space Safety)**.** *A state space $\Theta = \langle \mathcal{S}, \mathcal{L}, \mathcal{T} \rangle$ is* unsafe *with respect to $\langle \phi_0, \phi_u, \phi_g \rangle$ if and only if there exist states $s_0, \ldots, s_n \in \mathcal{S}$ with $s_0 \models \phi_0$, $s_n \models \phi_u$, $(s_i, \_, s_{i+1}) \in \mathcal{T}$ for $i \in \{0, \ldots, n-1\}$, and $s_i \not\models \phi_g$ for $i \in \{0, \ldots, n-1\}$. A state space is* safe *if no such states exist*[2].

A state that satisfies the initial, goal, or unsafety condition is also called an initial, a goal, or an unsafe state.

We next define the safety of a policy via the safety of the policy-restricted state space, which is the state space that contains the transitions which are selected by the policy.

Definition 4 (Policy)**.** *Let $\Theta = \langle \mathcal{S}, \mathcal{L}, \mathcal{T} \rangle$ be a state space. A policy $\pi \colon \mathcal{S} \to \mathcal{L}$ is a function mapping the states of the state space to actions.*

Definition 5 (Policy-Restricted State Space)**.** *Let $\Theta = \langle \mathcal{S}, \mathcal{L}, \mathcal{T} \rangle$ be a state space, and let $\pi \colon \mathcal{S} \to \mathcal{L}$ be a policy. Then $\Theta^\pi = \langle \mathcal{S}, \mathcal{L}, \mathcal{T}^\pi \rangle$ is the* policy-restricted state space *of Θ, with $\mathcal{T}^\pi = \{(s, \ell, s') \in \mathcal{T} \mid \pi(s) = \ell\}$.*

Based on this, policy safety is defined as follows.

Definition 6 (Policy Safety)**.** *A policy π is safe with respect to a reach-avoid property ρ and a state space Θ if and only if the policy-restricted state space Θ^π is safe with respect to ρ.*

Now we have given the necessary background in planning and safety and will next introduce the policy representation.

2.2 Additive Tree Ensembles

Let $\mathcal{X}$ be an input space and $\mathcal{Y}$ be an output space.

A decision tree is inductively defined as either

- a leaf node with a value $v \in \mathcal{Y}$, denoted $\mathrm{Leaf}(v)$, or
- an internal node $\mathrm{Node}(v_i < \alpha, T_L, T_R)$, where v_i is a feature of the input space, $\alpha \in \mathbb{R}$ is a threshold, and T_L, T_R are decision trees called the left and right subtree.

A decision tree T induces an evaluation function $\langle T \rangle \colon \mathcal{X} \to \mathcal{Y}$ defined recursively as:

$$\langle T \rangle(\mathbf{x}) = \begin{cases} v & \text{if } T = \mathrm{Leaf}(v) \\ \langle T_L \rangle & \text{else if } x_i < \alpha \\ \langle T_R \rangle & \text{otherwise,} \end{cases}$$

In the internal node cases, $T = \mathrm{Node}(v_i < \alpha, T_L, T_R)$ and x_i is the i-th component of the input vector $\mathbf{x}$.

A tree ensemble is a finite collection of decision trees. A tree ensemble $\mathbf{T} = [T_1, \ldots, T_n]$ induces an evaluation function $\langle \mathbf{T} \rangle \colon \mathcal{X} \to \mathcal{Y}$, defined as

$$\langle \mathbf{T} \rangle(\mathbf{x}) = \langle T_1 \rangle(\mathbf{x}) + \cdots + \langle T_n \rangle(\mathbf{x}).$$

We generalize the notion of boxes by Devos, Meert, and Davis (2021) to arbitrary nodes. The *box* of a node n, written $\mathrm{box}(n)$, is the hypercube of the input space obtained by conjoining the decisions that lead to that node. For example, the box of the orange leaf in Figure 1 is $v_1 \geq 3 \wedge v_2 \geq 4 \wedge v_3 < 3$. For a collection of nodes $n_1, \ldots, n_k$ the box is defined as $\mathrm{box}(n_1, \ldots, n_k) := \bigcap_{i=1,\ldots,k} \mathrm{box}(n_i)$.

[1] In related work, it is assumed that models do not update a variable to a value outside of its domain (Klauck et al. 2018, p. 4). In the following, we make the same assumption.

[2] In terms of Computation Tree Logic (CTL), safety w.r.t. a reach-avoid property $\langle \phi_0, \phi_u, \phi_g \rangle$ corresponds to the CTL formula $\phi_0 \Rightarrow \mathrm{A}[\neg\phi_u \,\mathrm{W}\, \phi_g]$.

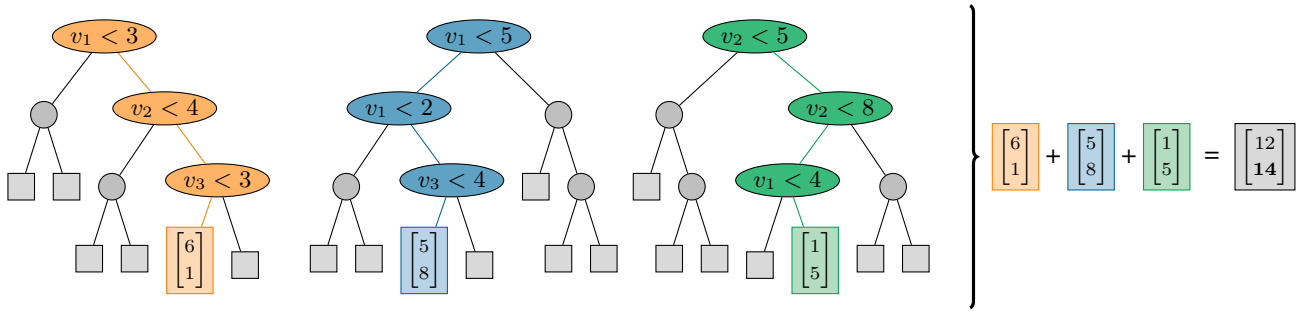

Figure 1: Let $s = \{v_1 \mapsto 4, v_2 \mapsto 7, v_3 \mapsto 2\}$. First, the individual decision trees are evaluated. Then, the leaf values are added up. The label selected by the policy corresponds to the maximum value in this sum, i.e., $\pi(s) = \ell_2$.

In our setting, $\mathcal{X} = \mathbb{Z}^{|\mathcal{V}|}$, we identify each state s (recall states are functions $\mathcal{V} \to \mathbb{Z}$) with the vector $\begin{bmatrix} s(v_1) & \cdots & s(v_{|\mathcal{V}|}) \end{bmatrix}^\top$ using a fixed ordering $v_1, \ldots, v_{|\mathcal{V}|}$ of the variable, and $\mathcal{Y} = \mathbb{R}^{|\mathcal{L}|}$.

A tree ensemble $\mathbf{T}$ then induces a policy $\pi \colon \mathcal{S} \to \mathcal{L}$ as follows: Let $s \in \mathcal{S}$, $\langle \mathbf{T} \rangle(s) = \begin{bmatrix} v_1 & \cdots & v_{|\mathcal{L}|} \end{bmatrix}^\top \in \mathbb{R}^{|\mathcal{L}|}$, and $v_i = \max(v_1, \ldots, v_{|\mathcal{L}|})$, then $\pi(s) = \ell_i$. Figure 1 provides an example evaluation of a tree ensemble policy.

Intuitively, each action label is associated with a score by the tree ensemble, and the policy selects the action label with the highest score.

2.3 Binary Decision Diagrams

A *Binary Decision Diagram* (BDD) is a rooted directed acyclic graph with two terminal nodes, the 0/*false* terminal and the 1/*true* terminal. Each internal/decision node is annotated with a Boolean variable and has a low and a high child. An example BDD is given in Figure 2. The low edges are marked by dashed lines and the high edges by solid lines.

A BDD is evaluated by starting at the root and moving to the low child if the variable in the current node is *false* and to the high child if it is *true*. *Ordering* enforces a variable ordering along all paths; *reduction* applyes two reduction rules to the BDD. The latter are *merge*, which merges identical subtrees, and *delete*, which removes nodes with identical children (replacing the redundant node with its only child). The result of ordering and reduction is a *Reduced Ordered Binary Decision Diagrams* (ROBDD). ROBDDs have been introduced by Bryant (1986) and enable a compact representation of Boolean functions $\mathbb{B}^n \to \mathbb{B}$. For a fixed variable ordering, ROBDDs are a canonical representation of Boolean functions, i.e., there is a unique ROBDD for each Boolean function. To check satisfiability of a function represented as ROBDD it suffices to check equality with the ROBDD for *false* (in constant time). Consequently, obtaining the ROBDD of a function is at least NP-hard. In the remainder, when we say BDD, we actually mean ROBDD.

2.4 Algebraic Decision Diagrams

Algebraic Decision Diagrams (Bahar et al. 1993) (ADDs) are a generalization of BDDs that allow more than two terminal nodes. See Figure 2 for an example. They can thus represent functions $\mathbb{B}^k \to \mathcal{S}$ with finite images for arbitrary

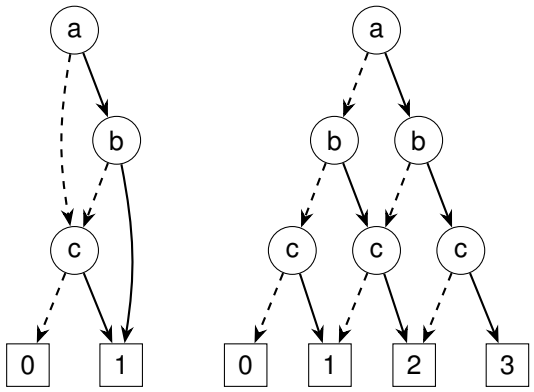

Figure 2: Left: (RO)BDD for the Boolean function $(\mathsf{a} \wedge \mathsf{b}) \vee \mathsf{c}$. Right: ADD counting the number of true variables. Both diagrams use the variable ordering $a < b < c$.

(even infinite) sets $\mathcal{S}$. An *algebraic structure* defines the possible values of terminal nodes and operations on then possible values. It is a tuple $\mathcal{A} = (\mathcal{S}, ops, uops, elems)$ with a *carrier set* $\mathcal{S}$, a finite set of *binary operators* $\mathcal{S} \times \mathcal{S} \to \mathcal{S}$ ops, a finite set of *unary (or monadic) operators* $\mathcal{S} \to \mathcal{S}$ $uops$, and a finite set of *distinguished elements* $elems \subseteq \mathcal{S}$. When ADDs are ordered and reduced (using the same reduction rules as in BDDs, i.e., merge and delete), they are a canonical representation. *Apply* lifts the binary operators in ops to ADDs and *monadic apply* lifts the unary operators in $uops$ to ADDs.

2.5 Symbolic Search

Symbolic search is a search technique that explores state spaces by operating on sets of states using efficient representations and manipulations. A set of states $S \subseteq \mathcal{S}$ is represented by its characteristic function $\chi_S \colon \mathcal{S} \to \mathbb{B}$, with $\chi_S(s) = 1$ if $s \in S$ and 0 otherwise. Similarly, transition relations $\mathsf{TR} \subseteq \mathcal{S} \times \mathcal{S}$ are represented by their characteristic functions. The image operator computes the set of successors for a set of states and a transition relation. Symbolic breath-first search can be conducted by repeatedly applying the image operator on the set of reached states (which is initially the set of initial states) until the set of reached states overlaps the set of target states or a fixpoint is reached. In the

latter case, no target state is reachable from an initial state.

BDDs are usually used as the data structure for the characteristic functions, which requires a binary representation of states.

3 Safety Verification via Symbolic Search

Recall that a state space is unsafe if and only if an unsafe state is reachable from an initial state without traversing any goal state. We can use symbolic search with minor modifications to verify the safety of state spaces.

3.1 Policy Safety Verification via Symbolic Search

Definition 7 (Policy Region)**.** *Given a policy $\pi\colon \mathcal{S} \to \mathcal{L}$ and an action label $\ell \in \mathcal{L}$, the* policy region *of ℓ is the set $\{s \in \mathcal{S} \mid \pi(s) = \ell\}$. We denote its characteristic function by χ^π_ℓ.*

When treating the policy as a black-box, computing the policy regions requires evaluating the policy on each state in the state space, which is not feasible for large state spaces. In Sec. 3.3 and Sec. 3.4, we present two methods for computing the policy regions for policies induced by additive tree ensembles that do not require enumerating the states.

The modified symbolic search is given in Algorithm 1. S_i is the search frontier in iteration i, initialized to the set of initial states in iteration 0. In each iteration we check whether the frontier contains unsafe states. If it does, the reach-avoid property is violated[3] and we can stop. Otherwise, the goal states need to be removed from the frontier as, per definition, their successors can not be on an unsafe path. We then compute the set of successors in the policy-restricted state space by only computing the successors (using image) of an operator o on the subset of the frontier where π selects the action label of o. We repeat this process until a fixpoint is reached, in which case the policy is safe.

Algorithm 1: Verification via Forward Symbolic Search

Input: State space $\Theta = \langle \mathcal{S}, \mathcal{L}, \mathcal{T} \rangle$, reach-avoid property (ϕ_0, ϕ_u, ϕ_g), and transition relations TR_o for each $o \in \mathcal{O}$.

$S_0 \leftarrow [\phi_0]$, $i \leftarrow 0$
repeat
  if $S_i \cap [\phi_u] \neq \emptyset$ **then**
    return Unsafe
  $S_i \leftarrow S_i \setminus [\phi_g]$
  $S_{i+1} \leftarrow \emptyset$
  for all $\ell \in \mathcal{L}$ **do**
    $S_i^\ell \leftarrow S_i \cap \{s \in S \mid \pi(s) = \ell\}$
    for all $o \in \mathcal{O}$ with label ℓ **do**
      $S_{i+1} \leftarrow S_{i+1} \cup \mathrm{image}(S_i^\ell, \mathsf{TR}_o)$
  $i \leftarrow i + 1$
until $S_i = S_{i-1}$
return Safe

3.2 Encoding the State Space as BDDs

To encode a state space $\Theta = \langle \mathcal{S}, \mathcal{L}, \mathcal{T} \rangle$ induced by a planning task $\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$ as BDDs, we need:

1. a binary representation of states, s.t. we can represent sets of states and relations between states as BDDs,
2. an encoding of linear integer constraints to BDDs (denoted $\mathrm{enc}(\cdot)$) in that representation, for e.g. the initial, unsafety, and goal condition, as well as the operator guards, and
3. an encoding of the transitions $\mathcal{T}$ to BDDs.

For 1., let $v \in \mathcal{V}$ be one of our variables with integer interval domain $D_v = [l, u]$. We encode v using $\lceil \log_2(u - l + 1) \rceil$ binary variables, called the source bits of v. To represent a state, we stack the bits of all variables, again assuming a fixed ordering on the variables. We use an additional copy of the binary variables, called the target bits, to represent the "next state" in transition relations.

For 2., we recursively encode linear integer constraints to BDDs. For linear integer constraints of the form $\neg\Phi$, $\Phi \wedge \Phi$, and $\Phi \vee \Phi$, we simply use the respective operations on BDDs. To encode atomic linear integer constraints of the form $e_1 \leq e_2$, $e_1 = e_2$, and $e_1 \geq e_2$ as BDD, we use the encoding of *linear arithmetic constraints* of the form $\sum_{i=1}^n a_i \cdot x_i = a_0$ and $\sum_{i=1}^n a_i \cdot x_i < a_0$ by Bartzis and Bultan (2003), where $a_0, \ldots, a_n$ are integer coefficients and $x_1, \ldots, x_n$ are bounded integer variables with lower bound 0. We use simple equivalent transformations to achieve this form.

For 3., let $o = (g, \ell, u) \in \mathcal{O}$ with $u = \{v_1 \mapsto e_1, \ldots, v_{|\mathcal{V}|} \mapsto e_{|\mathcal{V}|}\}$. Be aware that we can view each element $v_i \mapsto e_i$ of the update as a linear integer constraint $v_i' = e_i$[4] over $\mathcal{V} \cup \mathcal{V}'$, where $\mathcal{V}'$ is a copy of the state variables used to represent the variables in the "next state".

TR_o, the BDD for the transition relation induced by the operator o, is then $\mathrm{enc}(g) \wedge \mathrm{enc}(v_1' = e_1) \wedge \cdots \wedge \mathrm{enc}(v_{|\mathcal{V}|}' = e_{|\mathcal{V}|})$. We keep the transition relations of each operator separate.

3.3 Computing Policy Regions via Equivalence Class Partitions

The idea of this approach is to recurse through the nodes in a tree ensemble, keeping track of the box of the current nodes. Once we reach a leaf in all decision trees we can evaluate which action label is scored highest. We can use some early stopping criteria to detect so-called base cases, i.e., cases in which all leaf combinations below the current node select the same action label. The box of the nodes is then disjunctively added to the policy region of the respective label.

The translation is given in Algorithm 2. Throughout the procedure, the box of the current nodes, i.e., the hypercube of the input space on which the current nodes are reached in the evaluation of $\mathbf{T}$, is kept as an invariant. C holds the boxes represented as lists of half-open integer intervals. We initialize C to cover the domains of all variables.

We recursively explore the tree ensemble by querying the tree selector for the next internal node to process. Given an internal node, we compute the boxes of the left and right

[3] The unsafe path can be reconstructed. We omit the details here.

[4] v_i' is an expression over $\mathcal{V} \cup \mathcal{V}'$ by setting the coefficient of v_i' to 1 and all others to 0

Algorithm 2: TreeEnsemble2BDD

Input: Planning Task $\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$
Tree ensemble $\mathbf{T} = [T_1, \ldots, T_n]$
Base Case Detector bcd, Tree Selector ts
Output: List of $|\mathcal{L}|$ BDDs for the functions $\chi_{\ell_1}, \ldots, \chi_{\ell_{|\mathcal{L}|}}$

1: **function** MAIN($\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$, $[T_1, \ldots, T_n]$)
2: **for all** $\ell \in \mathcal{L}$ **do**
3: $\chi_\ell \leftarrow \bot$
4: **for all** $v \in \mathcal{V}$ **do**
5: $l_v \leftarrow \min(D_v)$; $h_v \leftarrow \max(D_v) + 1$
6: TE2BDD($[T_1, \ldots, T_n]$, $[[l_{v_1}, h_{v_1}), \ldots, [l_{v_{|\mathcal{V}|}}, h_{v_{|\mathcal{V}|}})]$)
7: **return** $[\chi_\ell \mid \ell \in \mathcal{L}]$

8: **function** TE2BDD($[T_1, \ldots, T_n], C$)
9: **if** bcd.ISBASECASE($[T_1, \ldots, T_n]$) **then**
10: $\ell \leftarrow bcd$.GETLABEL($[T_1, \ldots, T_n]$)
11: $b \leftarrow \bigwedge_{v \in \mathcal{V}} (l_v \leq v \wedge v < h_v)$
12: $\chi_\ell \leftarrow \chi_\ell \vee b$
13: **else**
14: $i, (v_j < \alpha, T_L, T_R) \leftarrow ts$.SELECT($[T_1, \ldots, T_n]$)
15: $\left[l_{v_j}^{old}, h_{v_j}^{old}\right) \leftarrow C[j]$ ▷ Memorize old interval
16: $h_{v_j}^{new} \leftarrow \min\left(\lceil \alpha \rceil, h_{v_j}^{old}\right)$
17: **if** $l_{v_j}^{old} < h_{v_j}^{new}$ **then** ▷ Else C is empty.
18: $C[j] \leftarrow \left[l_{v_j}^{old}, h_{v_j}^{new}\right)$
19: bcd.ENTER(i, T_L)
20: TE2BDD($[T_1, \ldots, T_{i-1}, T_L, T_{i+1}, \ldots, T_n]$, C)
21: bcd.EXIT(i, T_L)
22: $l_{v_j}^{new} \leftarrow \max\left(\lceil \alpha \rceil, l_{v_j}^{old}\right)$
23: **if** $l_{v_j}^{new} < h_{v_j}^{old}$ **then** ▷ Else C is empty.
24: $C[j] \leftarrow \left[l_{v_j}^{new}, h_{v_j}^{old}\right)$
25: bcd.ENTER(i, T_R)
26: TE2BDD($[T_1, \ldots, T_{i-1}, T_R, T_{i+1}, \ldots, T_n]$, C)
27: bcd.EXIT(i, T_R)
28: ▷ Restore old interval ◁
29: $C[j] \leftarrow \left[l_{v_j}^{old}, h_{v_j}^{old}\right)$

30: **function** ISBASECASE($[T_1, \ldots, T_n]$)
31: Returns `True` only if a base case is reached, i.e., the current box C form an equivalence class. In the simplest case, this is when all nodes $T_1, \ldots, T_n$ are leaf nodes.

32: **function** SELECT($[T_1, \ldots, T_n]$)
33: Select one internal node among $T_1, \ldots, T_n$ according to some criterion and return it along with its index.

subtrees and recurse on them if their box is not empty (see Figure 3).

When the bcd identifies that the current nodes form a base-case, the box C is translated to a BDD in line 11 which is added to the policy region of the label selected by bcd for the current nodes in line 12. The base case detector is notified whenever the list of current nodes changes, such that it can update its invariant (lines 19, 21, 25, and 27).

Definition 8 (Partition)**.** *A partition of X is a set of non-empty subsets of X such that every element of X is in exactly one of those subsets. The elements of a partition are called cells.*

Definition 9 (Equivalence Class Partition)**.** *Given a function $f\colon X \to Y$, an* equivalence class partition *w.r.t. f is a partition P of the domain X, such that $\forall A \in P : \forall a, b \in A : f(a) = f(b)$, i.e., all inputs in a cell map to the same output.*

Theorem 1 (Soundness and Completeness)**.** *Assume the following conditions hold:*

(1) The boxes of the nodes in the base cases encountered in TE2BDD form an equivalence class partition w.r.t. the policy π.
(2) GETLABEL *returns the correct label in each base case.*

Then the algorithm is sound and complete, i.e. every χ_ℓ^π returned by MAIN *is (the BDD of the characteristic function of) the policy region of ℓ.*

Since (2), every box gets added to the correct policy region, and since the boxes partition $\mathcal{S}$, each state is added to a policy region.

Proposition 1. *(1) and (2) hold if the base case detector bcd satisfies two properties:*

- *If all current nodes $T_1, \ldots, T_n$ are leaf nodes, then bcd.ISBASECASE($[T_1, \ldots, T_n]$) is `True`. We then say that bcd is* leaf-detecting.
- *If bcd.ISBASECASE($[T_1, \ldots, T_n]$) is `True`, then $\forall s \in C\colon \pi(s) = bcd$.GETLABEL($[T_1, \ldots, T_n]$). We then say bcd is* sound.

Proof. (2) trivially holds if bcd is sound. To show (1) – i.e., that the boxes of the nodes in the base cases encountered in TE2BDD, from now on called P, form an equivalence class partition w.r.t. π – we need to show that (a) P is a partition of $\mathcal{S}$, and (b) for all states s_1, s_2 in a cell $\pi(s_1) = \pi(s_2)$.

For (a), first convince yourself, that, given an internal node $\mathrm{Node}(v_j < \alpha, T_L, T_R)$, the recursion works as shown in Figure 3.

The box C is initially the entire state space $\mathcal{S}$ (lines 4-6). The recursion structure enforces that the current box never becomes empty (therefore $\emptyset \notin P$) and that, in case 2, the two new boxes partition the current box. The leaf-detecting property guarantees that each branch in the recursion tree eventually ends in a base case. This concludes (a). For (b), let $A \in P$. Since $A \in P$, A is a box encountered in a base case. Using the soundness property of bcd, we get that $\forall s \in A : \pi(s) = bcd$.GETLABEL($[T_1, \ldots, T_n]$) and thus $\forall s_1, s_2 \in A : \pi(s_1) = \pi(s_2)$. □

Base Case Detection In this section, we propose three base case detection methods and give proof sketches why they are leaf-detecting and sound. They all keep some form of data, for which some invariant holds on each search state (given by the current nodes $T_1, \ldots, T_n$ and their box C). For each method, we describe the invariant and how the data is updated in ENTER$(i, node)$ and EXIT$(i, node)$ to maintain the invariant.

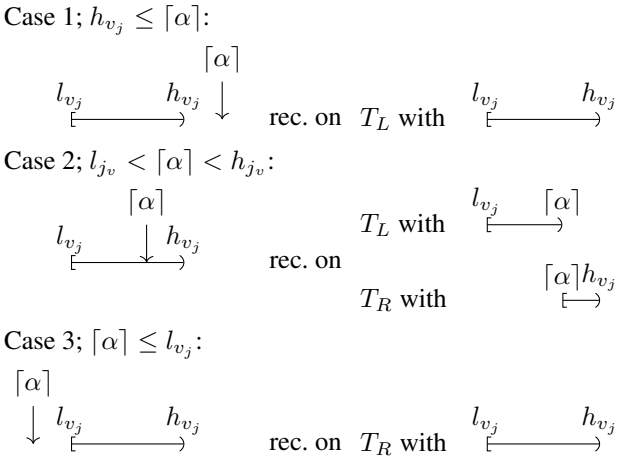

Figure 3: Visual aid for understanding the recursions of Algorithm 2.

All Leafs Given nodes $T_1, \ldots, T_n$, the all leafs detector returns `True` if and only if all nodes are leaf nodes. GETLABEL sums up the leaf values and returns the best one. This is trivially leaf-detecting and sound. This is the most basic base case detection method and does not provide any pruning power. We implement it by keeping the number of non-leaf nodes *nl* as an invariant, which is maintained in ENTER$(i, node)$/EXIT$(i, node)$ by decrementing/incrementing nl by 1 if $node$ is a leaf node.

Uncatchable Leader The uncatchable leader approach uses an accumulator $accu$ as an invariant. In each search state, the accumulator holds the sum of the values of the leaf nodes. The index of the maximal component in $accu$ is the current leader. The potential between the leader and another label in a node T indicates by how much the other label can catch up to the leader in that node. If the lead of the leader is greater than the sum of potentials of non-leaf nodes for any other label, then the current leader is uncatchable. An example is provided in the appendix.

Definition 10 (Potential)**.** *Given a decision tree T and two action label indices $i, j \in \{1, \ldots, |\mathcal{L}|\}$, the potential $P_T : \{1, \ldots, |\mathcal{L}|\} \times \{1, \ldots, |\mathcal{L}|\} \to \mathbb{R}$ between ℓ_i and ℓ_j in T is defined recursively: $P_T(i, j) = x_i - x_j$ if $T = \text{Leaf}(\begin{bmatrix} x_1 & \cdots & x_{|\mathcal{L}|} \end{bmatrix}^\top)$ and $P_T(i, j) = \max(P_{T_L}(i, j), P_{T_R}(i, j))$ if $T = \text{Node}(v_i < \alpha, T_L, T_R)$.*

The potential between two labels ℓ_1 and ℓ_2 in a node T is the maximum possible difference between the values for ℓ_1 and ℓ_2 in any leaf below T.

We bottom-up precompute the potentials for all nodes in the ensemble and pairs of indices, and store them in a $|\mathcal{L}| \times |\mathcal{L}|$ matrix for each node.

As already stated, we use an accumulator $accu$ as an invariant, that holds the summed up values of leaf nodes in the current search state. *accu* is initialized with the $|\mathcal{L}|$-dimensional vector $\mathbf{0}$. To maintain the invariant, ENTER$(i, node)$/EXIT$(i, node)$ adds/subtracts the leaf value of $node$ to $accu$ if $node$ is a leaf node.

ISBASECASE$(\ldots)$ first determines the index and value of the current leader, by finding the maximum value and its index i_{lead} in $accu$. For the indices of all labels other than the leader i_{other} we sum up the potentials $P_T(i_{other}, i_{lead})$ for all internal nodes T in $T_1, \ldots, T_n$[5]. The other label can catch up to the leader by at most that value. If none of the other labels can catch the leader, i.e., the summed potentials are less than the lead for all labels, we know that $\forall s \in C\colon \pi(s) = \ell_{i_{lead}}$. By returning $\ell_{i_{lead}}$ in GETLABEL we get a sound base case detector.

To show that it is also leaf-detecting, assume all $T_1, \ldots, T_n$ are leafs. Then there are no internal nodes in $T_1, \ldots, T_n$ and thus the sums of potentials of internal nodes are all empty, i.e. 0, and consequentially none of the other labels can catch up to the leader.

Bounds on Minimum and Maximum The basic idea is as follows: For each action label and each decision tree, we have a lower and upper bound of the value this tree can contribute to the label. When we add up these bounds, we get a bound for the value of each label in the tree ensemble. When the lower bound of the value of one action label is greater than the upper bound of the value of all other labels, we are in a base case.

Definition 11 (Mini and Maxi)**.** *Given a decision tree T we recursively define $\underline{T}$ and $\overline{T}$, the vector of minimal and maximal values each component can assume in any leaf of T. If $T = \text{Leaf}(v)$, $\underline{T} = \overline{T} = v$, and if $T = \text{Node}(v_i < \alpha, T_L, T_R)$, $\underline{T} = \min(\underline{T_L}, \underline{T_R})$ and $\overline{T} = \max(\overline{T_L}, \overline{T_R})$, where* min *and* max *are the componentwise minimum and maximum on vectors.*

For nodes $T_1, \ldots, T_n$, $\sum_{i=1}^n \underline{T_i} \leq \sum_{i=1}^n \langle T_i \rangle(x) \leq \sum_{i=1}^n \overline{T_i}$ for all x.

We precompute $\text{Bounds}(T) = \begin{bmatrix} \underline{T} & \overline{T} \end{bmatrix} \in \mathbb{R}^{|\mathcal{L}| \times 2}$ for each node T in every tree of the ensemble.

We keep the sum of bounds $\sum_{i=1}^n \text{Bounds}(T_i)$ as an invariant. To maintain the invariant, ENTER$(i, node)$/EXIT$(i, node)$ narrows/widens the sum of bounds by subtracting/adding $\text{Bounds}(parent)$ and adding/subtracting $\text{Bounds}(node)$, where $parent$ is the parent of $node$.

ISBASECASE$(T_1, \ldots, T_n)$ returns true if and only if the largest lower bound is greater than or equal to the second-largest upper bound in the sum of bounds. GETLABEL then returns the action label associated with the largest lower bound, yielding a sound base case detector.

To show that it is also leaf-detecting, assume all $T_1, \ldots, T_n$ are leafs. Note that for all leaf nodes T : $\text{Mini}(T) = \text{Maxi}(T)$ and thus $\sum_{i=1}^n \text{Mini}(T_i) = \sum_{i=1}^n \text{Maxi}(T_i)$. Since the largest upper bound is trivially greater than or equal to the second-largest upper bound, and there is a lower bound that is equal to the largest upper bound, we can conclude that the largest lower bound is greater than or equal to the second-largest upper bound. Thus, ISBASECASE is `true`.

[5]The leaf nodes are already accounted for in $accu$.

3.4 Computing Policy Regions Using ADDs

Our second approach to compute the policy region BDDs works by encoding each decision tree in the tree ensemble as an ADD, summing up those ADDs, and swapping each leaf vector with the action label corresponding to its maximal component. Finally, we extract the policy region BDD of each action label by computing the agreement BDD with the ADD constant corresponding to that action label.

We first describe the desired operations and later define an algebraic structure to support these operations.

To encode the evaluation function of a decision tree, which is a function $\mathbb{Z}^{|\mathcal{V}|} \to \mathbb{R}^{|\mathcal{L}|}$ to an ADD $\mathbb{B}^k \to \mathbb{R}^{|\mathcal{L}|}$, we can first restrict the domain of the decision tree to the states $\mathcal{S}$[6]. The ADD for a decision tree is then built recursively:

- For a leaf node $\mathrm{Leaf}(v)$ with $v \in \mathbb{R}^{|\mathcal{L}|}$, the ADD is the unique terminal with value v.
- For internal nodes $\mathrm{Node}(v_i < \alpha, T_L, T_R)$, let $\mathcal{A}_L$ and $\mathcal{A}_R$ be the ADDs for T_L and T_R. The ADD for the internal node is $\mathsf{ITE}(\mathrm{enc}(v_i < \lceil\alpha\rceil)$[7]$, \mathcal{A}_l, \mathcal{A}_r)$, where ITE is the standard if-then-else operation and enc is the encoding from Sec. 3.2.

We use this construction to obtain the ADDs $\mathcal{A}_1, \ldots, \mathcal{A}_n$ for the decision trees $T_1, \ldots, T_n$ in the ensemble. We compute the ADD of the additive tree ensemble $\mathcal{A}_\Sigma$ as

$$\mathcal{A}_\Sigma = \mathcal{A}_1 + \cdots + \mathcal{A}_n$$

by using standard vector addition (lifted to ADDs using apply) $n - 1$ times.

To obtain the ADD for the policy $\mathcal{A}_\pi$, we compute $\mathcal{A}_\pi = \mathrm{getLabel}(\mathcal{A}_\Sigma)$. Therefore, we lift $\mathrm{getLabel} : \mathbb{R}^{|\mathcal{L}|} \to \mathcal{L}$, the function mapping score vectors to the label associated with the best score, to ADDs using monadic apply.

Finally, we obtain the policy region BDD for each action label $\ell \in \mathcal{L}$ with the monadic function $f_\ell : \mathcal{L} \to \mathbb{B}$ with $f_\ell(x) = 1$ if $x = \ell$ and 0 otherwise.

The overall procedure is illustrated in Figure 4.

In summary, we need an algebraic structure with the carrier set $\mathcal{S} = \mathbb{B} \cup \mathcal{L} \cup \mathbb{R}^{|\mathcal{L}|}$, the binary vector addition $+ \colon \mathbb{R}^{|\mathcal{L}|} \times \mathbb{R}^{|\mathcal{L}|} \to \mathbb{R}^{|\mathcal{L}|}$, the unary operators $\{\mathrm{getLabel}, f_{\ell_1}, \ldots, f_{\ell_{|\mathcal{L}|}}\}$. *elems* is implicitly defined.

CUDD (Somenzi 1998), the decision diagram library we use in our implementation, only supports doubles as the carrier set natively. We thus came up with two workarounds, that are based on the presented approach, but work with doubles as the carrier set. Both workarounds are outlined in the appendix.

3.5 Incremental Policy Region Computation

So far, we have presented methods to compute the policy regions for the entire state space before search. Our experiments show that this computation typically dominates the search time, often by orders of magnitude, especially for large tree ensembles.

[6]Again treating states as vectors of integers.

[7]We encode the branching condition $v_i < \alpha$ using $\mathrm{enc}(v_i < \lceil\alpha\rceil)$. $v_i < \alpha$ and $v_i < \lceil\alpha\rceil$ are equivalent on all states since $n < x \Leftrightarrow n < \lceil x\rceil$ for all $n \in \mathbb{Z}, x \in \mathbb{R}$.

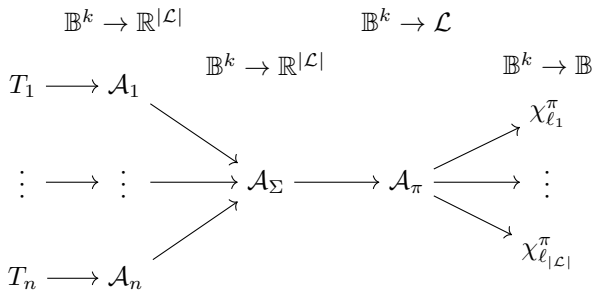

Figure 4: Overview of the policy regions computation using ADDs.

Algorithm 1 only uses the policy regions to restrict the frontier set S_i, by computing $S_i^\ell = S_i \cap \{s \in \mathcal{S} \mid \pi(s) = \ell\}$. Consequently, it is sufficient for the policy regions to be accurate only on the current frontier.

We exploit this observation by computing policy regions lazily during symbolic search. We integrate this in the ADD-based approach by computing $\mathcal{A}_\Sigma = (\mathcal{A}_1 \downarrow S_i) + \cdots + (\mathcal{A}_n \downarrow S_i)$ in each iteration of symbolic search, where $\downarrow$ is the *constrain* operator of Coudert and Madre (1990). Intuitively, the constrain operator produces smaller ADDs by permitting arbitrary behavior outside S_i. We proceed as before to obtain policy region BDDs that are accurate on S_i.

4 Experiments

Benchmarks We use the integer-variable benchmark set for additive tree ensemble verification by Jain et al. (2024). A benchmark consists of a planning task, a reach-avoid property – both given in JANI (Budde et al. 2017) format – and a tree ensemble given in Veritas' (Devos, Meert, and Davis 2021) JSON format.

Setup Our implementation is based on the C++ codebase PlaJA[8] (Vinzent, Sharma, and Hoffmann 2023). For ADDs and BDDs, we use CUDD (Somenzi 1998).

All experiments ran on a single thread of Intel Xeon E5-2660 CPUs with a time limit of 12 hours and a memory limit of 4 GB.

Configurations The configurations subsumed under *equivalence classes* implement the method presented in Sec. 3.3. We combine each presented base case detector with the two tree selection strategies depth-first (df) and round-robin (rr). Implementations of the method presented in Sec. 3.4 are grouped under *ADD*. *pw* and *vec* are the two workarounds outlined in the appendix. *vec inc* computes the policy region BDDs restricted to the current frontier in every iteration. We use policy predicate abstraction (PPA) with Veritas (Devos, Meert, and Davis 2021) for the transition test (without applicability filtering) (Jain et al. 2024) as a baseline.

Results Table 1 gives the runtime in seconds on the solved instances. The policy GB(16) in transport is known to be un-

[8]https://gitlab.cs.uni-saarland.de/vinzent/PlaJAPublic

					symbolic search (ours)									
					equivalence classes						ADD			PPA
					all leafs		min max		unc. leader		pw	vec	vec inc	
model	policy	#T	d	steps	df	rr	df	rr	df	rr				
BW-4	GB(32)-CA	20	4	165	1	1	1	1	1	1	1	**1**	2	4
	RF(32)-CA	5	8	124	2	2	2	2	2	2	2	**2**	3	3
	GB(64)-CA	5	4	165	2	2	**2**	2	**2**	2	2	2	3	7
	RF(64)-CA	5	8	166	3	2	2	2	2	2	3	**2**	4	4
	GB(16)-CI	10	4	166	2	2	2	2	2	2	2	**2**	3	3
	RF(16)-CI	5	8	166	3	2	2	3	2	**2**	2	**2**	3	3
	GB(32)-CI	10	4	166	2	3	3	2	**2**	2	3	2	3	2
	RF(32)-CI	5	6	166	2	2	**2**	2	2	3	2	2	3	2
	GB(64)-CI	10	4	166	2	3	3	2	2	2	3	**2**	3	6
	RF(64)-CI	5	8	166	2	2	2	2	**2**	2	2	2	3	3
BW-6	GB(64)-CA	20	8	80	—	—	—	—	—	—	—	—	**138**	3153
	RF(64)-CA	10	10	80	1691	1776	1319	657	1176	430	—	—	**12**	2118
	GB(64)-CI	20	8	80	475	487	512	549	496	521	—	—	**8**	58
	RF(64)-CI	5	15	80	4	4	4	4	5	**4**	110	13	8	19
BW-8	GB(64)-CA	20	8	—	—	—	—	—	—	—	—	—	—	—
	RF(64)-CA	5	6	38	8	8	8	8	8	**8**	8	8	10	—
	GB(64)-CI	30	8	79	—	—	—	—	—	—	—	—	**1459**	12140
	RF(64)-CI	30	15	74	4507	4339	4951	4935	5000	4844	—	—	—	—
transport	GB(16)	10	4	1	10	12	4	7	3	4	118	17	**2**	1075
	RF(16)	20	15	60	8688	8419	7659	4919	5962	3472	—	—	—	—
transport +feature	GB(16)	5	4	17	3	3	3	3	3	3	3	**3**	5	11
	RF(16)	10	4	11	**2**	3	3	3	3	3	3	2	5	8
	GB(64)	10	4	16	10	10	9	8	6	7	11	**6**	8	12
	RF(64)	5	8	17	2	2	2	2	2	**2**	3	3	5	12
beluga-4	GB(64)	5	4	1	0	0	0	0	**0**	0	2	0	0	5
	RF(64)	20	4	1	0	0	0	0	**0**	0	1	0	**0**	5
beluga-5	GB(256)	5	4	1	0	0	0	0	0	0	0	**0**	**0**	5
	RF(256)	30	6	1	269	278	298	247	247	210	—	—	**0**	15
beluga-6	GB(64)	10	4	1	—	—	—	—	—	—	—	—	**1**	20
	RF(64)	20	6	17	—	—	—	—	—	—	—	—	**4**	26

Table 1: Total runtime in seconds. The lowest runtime (when rounded to two decimal points) is highlighted in boldface. — identifies runs that exceeded the time or memory limit. *#T* is the number of trees in the ensemble and *d* is the depth limit of the trees. *steps* is the number of iterations performed by forward symbolic search before termination.

safe, for GB(64)-CA in BW-8 safety is still unknown. However, our *vec inc* configuration was able to establish that there is no unsafe path of length up to 28. All other policies are known to be safe.

Among the methods that precompute the policy regions for the entire state space once, the configurations using equivalence class partitions verify 4 more policies compared to the ADD-based approaches. On these 4 benchmarks, the ADD-based approaches run out of memory.

When we compare the runtime of our different configurations, we can see that the ADD-based approaches perform best. While for the BW-4 problems, the precomputing configuration needs less time, this is different for the larger BW-6 problems.

When we compare our configurations with related work (PPA), our *vec inc* configuration still performs best, in particular on the larger problems.

5 Conclusion

In this paper we presented a BDD-based approach for policy verification. BDDs are well-suited to represent large state spaces compactly, while still enabling reasoning operations upon them. This is exactly what is needed in policy verification. We introduced two approaches to represent a given Tree Ensemble Policy as BDDs, one involving an intermediate step via ADDs, and integrated it into a symbolic search. We evaluated our approach against the state-of-the-art PPA approach on the standard benchmark set. Our results show that especially one of our configurations that uses incremental computation works well on the benchmarks; in particular on the larger instances.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 572954863 – Emmy Noether Independent Research Group *Neuro-Symbolic Methods in Sequential Decision Making*.

References

Bahar, R. I.; Frohm, E. A.; Gaona, C. M.; Hachtel, G. D.; Macii, E.; Pardo, A.; and Somenzi, F. 1993. Algebraic decision diagrams and their applications. In *Proceedings of the*

1993 IEEE/ACM International Conference on Computer-Aided (ICCAD), 188–191. IEEE Computer Society/ACM.

Bartzis, C.; and Bultan, T. 2003. Construction of Efficient BDDs for Bounded Arithmetic Constraints. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 394–408. Springer.

Bryant, R. E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8): 677–691.

Budde, C. E.; Dehnert, C.; Hahn, E. M.; Hartmanns, A.; Junges, S.; and Turrini, A. 2017. JANI: Quantitative Model and Tool Interaction. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 151–168.

Coudert, O.; and Madre, J. C. 1990. A Unified Framework for the Formal Verification of Sequential Circuits. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 126–129. IEEE Computer Society.

Devos, L.; Meert, W.; and Davis, J. 2021. Versatile Verification of Tree Ensembles. In *Proceedings of the 38th International Conference on Machine Learning, (ICML)*, 2654–2664. PMLR.

Jain, C.; Cascioli, L.; Devos, L.; Vinzent, M.; Steinmetz, M.; Davis, J.; and Hoffmann, J. 2024. Safety Verification of Tree-Ensemble Policies via Predicate Abstraction. In *Proceedings of the 27th European Conference on Artificial Intelligence (ECAI)*, 1189–1197. IOS Press.

Kissmann, P.; and Edelkamp, S. 2011. Improving Cost-Optimal Domain-Independent Symbolic Planning. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI)*, 992–997. AAAI Press.

Klauck, M.; Steinmetz, M.; Hoffmann, J.; and Hermanns, H. 2018. Compiling probabilistic model checking into probabilistic planning (technical report). Technical report, Saarland University. Available at: https://fai.cs.uni-saarland.de/hoffmann/papers/icaps18a-tr.pdf.

Somenzi, F. 1998. CUDD: CU decision diagram package release 2.3. 0. *University of Colorado at Boulder*, 621.

Torralba, Á.; Alcázar, V.; Kissmann, P.; and Edelkamp, S. 2017. Efficient symbolic search for cost-optimal planning. *Artificial Intelligence*, 242: 52–79.

Vinzent, M.; and Hoffmann, J. 2024. Neural action policy safety verification: applicablity filtering. In *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling*, ICAPS '24. AAAI Press. ISBN 1-57735-889-9.

Vinzent, M.; Sharma, S.; and Hoffmann, J. 2023. Neural Policy Safety Verification via Predicate Abstraction: CEGAR. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence (AAAI)*, 15188–15196. AAAI Press.

Vinzent, M.; Steinmetz, M.; and Hoffmann, J. 2022. Neural Network Action Policy Verification via Predicate Abstraction. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling (ICAPS)*, 371–379. AAAI Press.

A Appendix

Uncatchable Leader Example

We provide an example illustrating how the uncatchable leader base-case detector identifies base cases early.

Example 1. *Assume accu is currently* $\begin{bmatrix}10\\-3\end{bmatrix}$ *and only the depicted trees* T_1 *and* T_2 *remain to be evaluated. We determine that* ℓ_1 *is the current leader with value* 10*. It has a lead of* 13 *over* ℓ_2*, but the value of* ℓ_2 *can catch up by at most* $P_{T_1}(2,1) + P_{T_2}(2,1) = 7 + 5 = 12$*. Thus,* ℓ_1 *is an uncatchable leader and will be the label in all leaf combinations below the current nodes regardless of the decisions.*

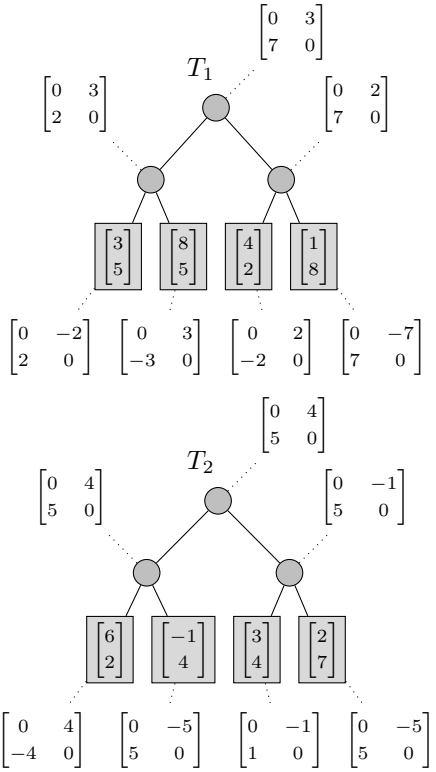

Figure 5: Two decision trees annotated with the precomputed pairwise potentials matrix for each node.

ADD Workarounds

We present two modifications of our ADD-based policy region computation that use doubles as carrier set, allowing us to use out-of-the-box CUDD (Somenzi 1998) in our implementation.

Point-wise ADDs The idea behind the first workaround, which we name point-wise ADDs, is to decompose each decision tree T_i into $|\mathcal{L}|$ decision trees $T_i^1, \ldots, T_i^{|\mathcal{L}|}$ with leaf values in $\mathbb{R}$, such that $\langle T_i \rangle(x) = \left[(\langle T_i^1 \rangle(x) \quad \ldots \quad \langle T_i^{|\mathcal{L}|} \rangle(x) \right]^\top$. We then encode the T_i^j to ADDs $\mathcal{A}_i^j$ and sum them up to obtain ADDs $\mathcal{A}_\Sigma^j$ for $j = 1, \ldots, |\mathcal{L}|$. We use $|\mathcal{L}| - 1$ apply operations with the binary max operator to compute one ADD that evaluates to the maximum value in each leaf. We can then use CUDD's `Agreement` function – which, given two ADDs, computes a BDD that is true for all inputs on which the ADDs have the same output – to get a BDD that is true for all inputs on which $\mathcal{A}_\Sigma^j$ is the maximum. This is (the BDD of the characteristic function of) the policy region of ℓ_j. The drawback of this workaround is that we unnecessarily duplicate the decision structure $|\mathcal{L}|$ times in each step.

Vector ADDs In this workaround, we simulate the desired algebraic structure with integers (cast to double) as the constant values. 0 and 1 correspond to 0 and 1, the next $|\mathcal{L}|$ numbers $2, \ldots, |\mathcal{L}| + 1$ are reserved for the labels. The integers starting at $|\mathcal{L}| + 2$ are used as ids for the distinguished elements in X. Whenever we encounter a new distinguished element in X, i.e., a vector of reals, we create a new ADD constant. A mapping from ids to the distinguished elements, i.e., the vectors, is stored in a hashmap. We implemented a new binary addition operator that returns the id of the sum of the vectors associated with the two operands, newly creating it when the sum is not yet in the hashmap. The unary operator $\mathrm{getLabel}$ looks up the vector associated with the operand, finds the component with the maximal value and maps to the constant that was reserved for the respective label. The monadic functions f_ℓ are again realized using CUDD's `Agreement` function, comparing to the constant values reserved for ℓ.