Minimizing State Space Partitions Using Decision Trees

Andreas Holck Høeg-Petersen[1] , **Kim Guldstrand Larsen**[2] , **Peter Gjøl Jensen**[3] and **Andrzej Wąsowski**[4]

[1,2,3] Aalborg University, Denmark
[4] It University of Copenhagen, Denmark
{ahhp, kgl, pgj}@cs.aau.dk, wasowski@itu.dk

Abstract

State space partitionings occur in Reinforcement Learning when a continuous state space is tackled by techniques requiring a finite set of states, such as classical Q-learning or particular methods for synthesizing safety shields. Such partitionings often become overwhelmingly large if they are to capture an appropriate amount of detail to solve the problem, which is a detriment to both explainability, verifiability and memory consumption when deployed to smaller embedded devices. In this work, we propose a novel and loss-less minimization algorithm called MaxPartitions, that represents partitionings as decision trees and perform minimization while preserving an equivalent state-action mapping. We show that MaxPartitions is able to produce substantial size reductions on both controller strategies and safety shields on a number of problems known from control theory, and we compare with VIPER, the state-of-the-art algorithm for producing small decision trees for RL strategies, to show, that the loss-less nature of MaxPartitions makes it preferable when safety is a requirement.

1 Introduction

Safe and explainable reinforcement learning prefers discrete policies over continuous neural network policies. Discrete representations of agent policies, especially in the form of decision trees, are easier to comprehend than neural networks, and easier to verify thanks to their strict structuring. Unfortunately, reinforcement learning of discrete policies for complex problems is known to be hard. A classic approach is to use a discrete learning tool, for instance Uppaal Stratego [**?**] following q-learning [**?**]. Unfortunately, this may produce policies that, even though discrete, remain too large and too complex to explain. Another approach is to learn in the continuous space with deep learning [**?**] and then apply further discretization [**?**]. This however makes maintaining safety difficult. In this paper, we look at the problem of decreasing the size of the policies expressed as decision trees to overcome these challenges.

A standard architectural setup for safe reinforcement learning is to enforce a *safety shield* around the extracted policy at runtime [**?**]. A safety shield is a liberal, often highly non-deterministic, control policy that disallows unsafe actions. The agent in a safe training setup follows regular learning of the controller, except whenever it would choose an unsafe action the shield is used to detect and prevent it. If the learning setup needs to be explainable both the controller and the shield need to be small and explainable, as building the safety case requires both.

The state of the art solution for obtaining small policies is implemented in the Viper method [**?**]. Viper first trains a continuous policy as a neural network and then uses imitation learning to extract a small discrete policy, a decision tree. Viper's imitation learning algorithm can, in principle, be used to extract decision trees from any oracle, not just a neural network. Unfortunately, as a sampling-based algorithm it does not guarantee behavioral equivalence with the input oracle. This is why an additional manual verification step for the safety of the output policy is required in the original paper [**?**]. Thus Viper is a good tool for minimizing controllers (maintaining similar performance), but not shields (as it would loose safety).

DtControl 2.0 is an algorithm by Ashok and colleagues that aims at minimization of decision tree policies while maintaining safety [Ashok *et al.*, 2020; Ashok *et al.*, 2021]. DtControl is highly aggressive, and when applied to shields used in reinforcement learning it prevents many high quality policies, drastically reducing the effectiveness of learning. This makes it unsuitable for minimizing shields automatically, as under such strong shields reinforcement learning is not effective.

In between Viper and dtControl, the users are stuck either loosing safety or performance. The MAXPARTITIONS algorithm presented in this paper aims to address this need, offering a lossless, equivalence-preserving, minimization method for discrete policies, like shields, which preserves safety, but being less aggressive does not reduce performance of the input shield. Our contributions include:

- A definition of the MAXPARTITIONS algorithm along with correctness and performance analysis.
- An implementation of MAXPARTITIONS in an experimental setup involving Uppaal Stratego (as the policy learning tool) with Viper and dTControl as baseline pol-

icy minimization tools.

- An experimental evaluation showing that MaxPartitions + Viper is a good combination for producing smaller safe policies, while Viper by itself is not safe and dtControl by itself is sub-optimal.

The paper proceeds as follows. ... Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

1.1 Related work

... Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

2 Preliminaries

Definition 2.1. *A* partitioning $\mathcal{A}$ *of the state space* $\mathcal{S} \subseteq \mathbb{R}^K$ *is a set of non-overlapping subsets* $\mathcal{A} \subseteq \mathcal{P}(\mathcal{S})$ *covering* $\mathcal{S}$*, so* $\bigcup_{\nu \in \mathcal{A}} \nu = \mathcal{S}$ *and whenever* $\nu, \nu' \in \mathcal{A}, \nu \neq \nu'$ *then* $\nu \cap \nu' = \emptyset$.

For an *axis aligned* partitioning, each region ν can be expressed in terms of two corner points, $s^{\min}, s^{\max} \in \mathcal{S}$, so that for each $s = (s_1, \ldots, s_K) \in \nu$ it holds that $s_i^{\min} < s_i \leq s_i^{\max}$ for $i = 1, \ldots, K$. In this work we exclusively consider axis aligned partitionings and we define all regions as a tuples $\nu = (s^{\min}, s^{\max})$. Note that the entire state space $\mathcal{S} \in \mathbb{R}^K$ can be described as a region: if $\mathcal{S}$ is unbounded in all dimensions (meaning its limits are positive and negative infinity) then $s_i^{\min}$ and $s_i^{\max}$ for the entire state space is $-\infty$ and ∞ respectively for $i = 1, \ldots, K$.

Definition 2.2 (Decision tree)**.** *A binary decision tree over the domain* $\mathcal{S} \in \mathbb{R}^K$ *is a tuple* $\mathcal{T} = (\eta_0, \mathcal{N}, \mathcal{L}, \mathcal{D})$ *where* $\eta_0 \in \mathcal{N}$ *is the root node of the tree,* $\mathcal{N}$ *is a set of branching nodes and* $\mathcal{L}$ *is a set of leaf nodes, each of which is assigned a decision* δ *from the set of decisions* $\mathcal{D}$*. Each branch node* $\eta \in \mathcal{N}$ *consists of two child nodes and a predicate function of the form* $\rho_\eta(s) = s_i \leq c$ *with* $s \in \mathcal{S}$ *and* c *being a constant.*

Given a state $s \in \mathcal{S}$ and a decision tree $\mathcal{T}$, we can evaluate $\mathcal{T}(s)$ to obtain a decision δ by following the *path* from the root node to a leaf node given by the repeated evaluation of the predicate function $\rho_\eta(s)$ at each node η, starting with the root node and continuing with the left child if $\rho_\eta(s)$ evaluates to true and with the right child if it evaluates to false. When we encounter a leaf node ℓ, we return the decision assigned to ℓ. Further, we also allow evaluating a region of $\mathcal{S}$. Given a region $\nu = (s^{\min}, s^{\max})$, $[\delta]_\nu = \mathcal{T}(\nu)$ is the set of all decisions that can be obtained evaluating configurations of ν, ie. $\mathcal{T}(\nu) = \{\mathcal{T}(s) \mid s \in \nu\}$.

The set of regions obtained from all the leaf nodes of a decision tree constitutes a complete partitioning of a state space $\mathcal{S}$ in accordance with Definition2.1. We thus say that $\mathcal{T}$ induces a partitioning $\mathcal{A}_\mathcal{T} = \{\nu_\ell \mid \ell \in \mathcal{L}\}$. For any region ν and a decision tree $\mathcal{T}$ we say, that ν has *singular mapping* in $\mathcal{T}$ if for all $p \in \nu$, $\mathcal{T}(p) = \delta$ for some $\delta \in \mathcal{D}$. Naturally, all regions in $\mathcal{A}_\mathcal{T}$ has singular mapping in $\mathcal{T}$. For any partitioning $\mathcal{B}$ of the same state space, we say $\mathcal{B}$ *respects* $\mathcal{T}$ if and only if every region $\nu \in \mathcal{B}$ has singular mapping in $\mathcal{T}$.

Partitionings in Reinforcement Learning In this work, we consider two examples of partitionings that occur in Reinforcement Learning, namely when synthesizing permissive safety shields and when learning near-optimal control strategies with classical (non-deep) Q-learning techniques.

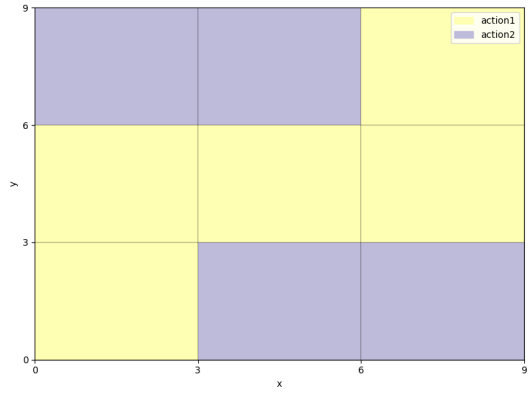
Safety shields are designed to disallow actions that can lead to violations of safety requirements (such as flooding a pond or crashing a car). A permissive shield imposes as few restrictions as possible. One method for generating shields is proposed by [Brorholt *et al.*, 2023]. This methods uses a very fine-grained partitioning of a state space to estimate a reachability function, that can compute the set of states that are reachable from any some state s when perfoming some action a. Given *a priori* knowledge of what states are considered 'unsafe' (ie. violates given safety requirements), the reachability function allows for tracing all state-action pairs that can potentially lead to an unsafe state. For capturing the dynamics at an appropriate level, however, a very fine-grained partitioning scheme is required.

In Q-learning for optimal control, the learning algorithm estimates an action-value function, that maps state-action pairs to a Q-value, that represents the expected cumulative reward of taking a specific action in a specific state. In UPPAAL STRATEGO [Jaeger *et al.*, 2019] these are learnt via an *online partitoining refinement scheme* which partitions the state space during learning according to the expected reward of an action. The representation of the strategies is then a set of decision trees, one for each possible action, with Q-values in the leaves. This set of trees can be merged into a single decision tree, where the leaves contain the optimal action in that state, effectively representing the Q-function. However, due to the online partitioning and the merge operation, the resulting partitioning retains a lot of redundancy, that we are interested in purging with our algorithm.

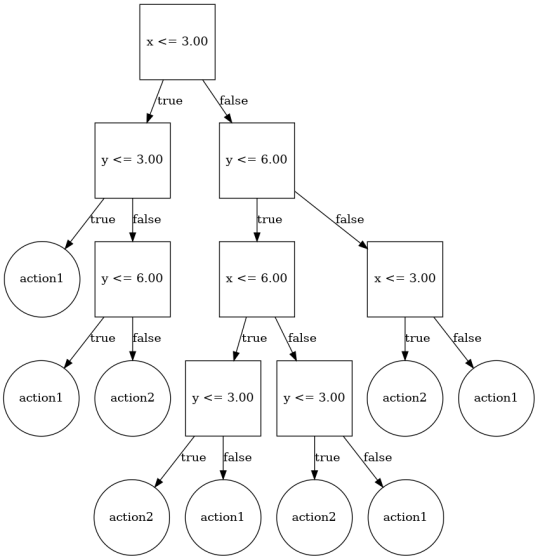
3 MaxPartitions algorithm

Since state space discretization for Reinforcement Learning is usually done *before* any learning takes place, it tends to be conservative. For this reason, discretization is likely to create adjacent discrete states that are mapped to the same optimal action. The question we would then like to answer is this: if $\mathcal{T}$ is a decision tree representing a trained strategy and $\mathcal{A}_\mathcal{T}$ is its induced partitioning, can we find another partitioning $\mathcal{B}$ which is smaller than $\mathcal{A}_\mathcal{T}$ but still respects $\mathcal{T}$?

As an example, consider a case where we have a state space $\mathcal{S} \in \mathbb{R}^2$ over variables x and y, both of which are defined in the interval $[0, 9]$, and a set of actions $Act = \{action1, action2\}$. Before learning, we might decide to discretize both x and y into 3 distinct bins, giving us a partitioning of $\mathcal{S}$ with $3 \times 3 = 9$ regions. After training, we end up with a Q-table that maps states to action in a way that is shown in Fig. 1a. This same mapping can also be represented by a decision tree, as shown in Fig. 1b.



(a)



(b)

Figure 1: Two representations of a strategy mapping states to actions. In Fig. 1a the strategy is represented as a partitioning with colors representing the assigned action. In Fig. 1b the strategy is given as a decision tree, which also induces the partitioning. For this small example, it is obvious to see that an equivalent state-action mapping could be achieved with fever regions/leaves.

However, we can (in this small toy example) easily see that we do not need 9 regions to represent this exact state-action mapping. For example, the two regions given by $((3, 0), (6, 3))$ and $((6, 0), (9, 3))$, respectively, both assign $action2$ as the optimal action, but this mapping would still be preserved if we replaced those two regions with a larger one $((3, 0), (9, 3))$. After a little bit of inspection, we can actually see here that we could represent the same state-action mapping with a partitioning consisting of only 5 regions.

This example showcases how discretization techniques can easily end up with redundancy in the partitioning. This can be very difficult to anticipate before learning, especially since a very fine-grained discretization is typically needed for the learning to capture essential information and details for the strategy. Furthermore, for other learning techniques, such as the online partitioning refinement scheme of UPPAAL STRATEGO [Jaeger *et al.*, 2019], regions can be created on-the-fly, not be arranged in a straight grid and/or vary in size, which can enhance the problem of redundancy.

We propose MAXPARTITIONS, an algorithm that postprocesses a decision tree inducing a partitioning of a state space in order to minimize the partitioning by *maximizing* the size of the individual regions (or partitions). The output of MAXPARTITIONS is a new partitioning, ie. a list of regions with associated actions, which can then be arranged into a new decision tree.

3.1 Details of the algorithm

We write $\mathcal{T}_i$ for the (ascendingly) sorted list of bounds on dimension i in the policy given by the tree $\mathcal{T}$. The first bound in the list is defined to be negative ∞ and the last is positive ∞. By $\mathcal{T}_{i,j}$ we write the jth smallest bound on dimension i for each $j = 1, 2, \ldots, |\mathcal{T}_i|$. This can be precomputed as a matrix in log-linear time by collecting and sorting the bounds on all branch nodes in $\mathcal{T}$ and allows accessing $\mathcal{T}_{i,j}$ in constant time. Further, in a slight abuse of notation, we define $\mathcal{T}_{i,|\mathcal{T}_i|+1}$ to be some *sentinel* value representing that we are outside the boundaries of dimension i. Correspondingly, we define a sentinel action α, and we say that $\mathcal{T}(s^p_{\mathcal{T}}) = \alpha$ if and only if $\exists p_i \in p,\ p_i = |\mathcal{T}_i| + 1$.

Exploiting this notation, let p be a K-dimensional vector of integers, such that $p_i \leq |\mathcal{T}_i| + 1$ for all $i = 1, \ldots, K$, then we can define a point at an intersection of bounds in $\mathcal{T}$ as $s^p_{\mathcal{T}} = (\mathcal{T}_{1,p_1}, \mathcal{T}_{2,p_2}, \ldots, \mathcal{T}_{K,p_K})$. To avoid cluttering the notation, we will for the most part omit the subscript $\mathcal{T}$ on $s^p_{\mathcal{T}}$. To make things clear, we will write s to refer to an actual point in the state space of $\mathcal{T}$, ie. $s \in \mathcal{S}$, and we will write p to refer to a vector of integers representing indicies of bounds in $\mathcal{T}$.

The algorithm (given in pseudo-code in Algorithm 1) works by maintaining a pair $(p^{\min}, p^{\max})$, and iteratively incrementing $p^{\max}$ in one dimension at a time until a region $\nu = (s^{p^{\min}}, s^{p^{\max}})$ cannot be expanded further. When this happens, the region is added to a tree $\mathcal{T}_{track}$, which is used to track which areas of the state space have been covered and to provide new a starting point for each iteration of the algorithm. Expansion in dimension i is disallowed if one of the following three *expansion rules* are violated by the expanded region ν':

Definition 3.1 (Expansion rules)**.** *Let ν' be a candidate region for a new partitioning derived from $\mathcal{A}_T$. Then ν' is valid if it adheres to the following rules:*

1. *ν' must have singular mapping in $\mathcal{T}$*
2. *ν' must not intersects with any region already in $\mathcal{T}_{track}$*
3. *ν' cannot intersect with a region ν_o in the original partitioning, such that the difference $\nu_o - \nu'$ cannot be described by a single region of the form $(s^{\min}, s^{\min})$*

The first two cases are directly related to the definition of the problem, ie. the produced partitioning should respect $\mathcal{T}$ and only have non-overlapping regions. The third case is required in order to guarantee that in each iteration the algorithm on average will add *at least* one region from the original partitioning to the new partitioning. To see this, consider the visualization in Fig. 2d. The candidate expansion 'cuts' the rightmost region (given by $(3,0)$ and $(4,4)$) in two such that the remainder would have to be represented by *two* regions — one given by $((3,0),(4,2))$ and one given by $((3,3),(4,4))$.

How do we determine this expansion? Let $(p^{\min}, p^{\max})$ define (the remainder of) a region in the original partitioning. We then want to find a $\Delta_p \in \mathbb{Z}^K$ such that $(p^{\min}, p^{\min} + \Delta_p)$ defines a region that follows the three expansion rules and such that incrementing in any one dimension would result in a violation. By definition, a valid value for Δ_p is when $\Delta_p = p^{\max} - p^{\min}$, since this would just produce the original region. We are therefore guaranteed to at least find this region. This gives rise to the following definition.

Definition 3.2 (The expansion vector Δ_p)**.** *Given $p^{\min} \in \mathbb{Z}^K$, a decision tree $\mathcal{T}$ over a K-dimensional space and a decision tree $\mathcal{T}_{track}$ of already found regions, $\Delta_p \in \mathbb{Z}^K$ is a vector such that for $p^{\max} = p^{\min} + \Delta_p$ the region $\nu = (s_{\mathcal{T}}^{p^{\min}}, s_{\mathcal{T}}^{p^{\max}})$ does not violate any of the expansion rules in Definition 3.1 and where for any other $\Delta'_p = (\Delta_{p_1}, \ldots, \Delta_{p_i} + 1, \ldots, \Delta_{p_K})$ this would not hold.*

A greedy approach to finding Δ_p starts with $\Delta_p = p^{\max} - p^{\min}$, for some (remainder of a) region $\nu = (s_{\mathcal{T}}^{p^{\min}}, s_{\mathcal{T}}^{p^{\max}})$. We then iteratively increment a single dimension chosen non-deterministically untill the invariants are violated. Let $\hat{\mathbf{e}}_i$ denote the unit vector parallel to axis i, such that $\Delta_p + \hat{\mathbf{e}}_i = (\Delta_{p_1}, \ldots, \Delta_{p_i} + 1, \ldots, \Delta_{p_K})$. At each increment, we define a candidate region ν' from $p^{\min}$ and $p^{\max} = p^{\min} + \Delta_p$ and check for singular mapping (Rule 1) and no overlap with regions in $\mathcal{R}$ (Rule 2). If any of these two do not hold, we mark dimension i as exhausted, roll back the increment and continue with a new dimension not marked as exhausted.

If Rule 3 is violated, the algorithm wil initiate an attempt at *healing* the candidate expansion, by continuing the expansion to the largest bound in the expansion dimension of any of the broken regions. This way we try to see if the violation can be overcome by simply expanding more aggressively. However, care is required to ensure, that we can roll back this extra expansion if it did not work (or if we inadverdently broke any of the other rules in the process).

When all dimensions have been exhausted, Δ_p adheres to Definition 3.2.

3.2 Proof of correctness

Let $\mathcal{B}$ be the output of running MAXPARTITIONS with a decision tree $\mathcal{T}$ and its induced partitioning $\mathcal{A}_{\mathcal{T}}$ as input. We want to prove the following properties of the algorithm:

Algorithm 1 MaxPartitions

Require: $\mathcal{T}$: A binary decision tree over the domain $\mathbb{R}^K$ inducing the partitioning $\mathcal{A}_{\mathcal{T}}$

$\mathcal{T}_{track} \leftarrow$ empty tree
$\mathcal{R} \leftarrow \{\}$
while $\mathcal{T}_{track}$ has unexplored regions **do**
  $(p^{\min}, p^{\max}) \leftarrow$ select from unexplored regions
  $\Delta_p \leftarrow p^{\max} - p^{\min}$
  $\Delta'_p \leftarrow \Delta_p$
  $healing \leftarrow false$

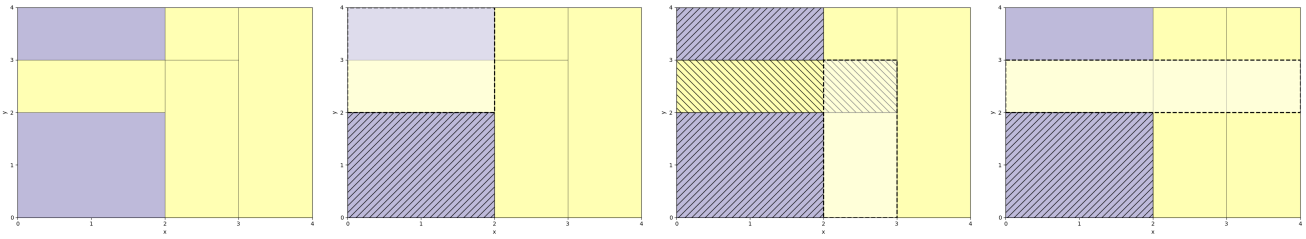
  while not all dimensions have been exhausted **do**
    if not *healing* **then**
      $d \leftarrow$ select unexhausted dimension
      $\Delta'_p \leftarrow \Delta_p + \hat{\mathbf{e}}_d$
    $\nu' \leftarrow (s_{\mathcal{T}}^{p^{\min}}, s_{\mathcal{T}}^{p^{\min}+\Delta'_p})$

    if ν' violates Rule 1 or 2 (Definition 3.1) **then**
      $\Delta'_{p_d} \leftarrow \Delta'_{p_d} - \hat{\mathbf{e}}_d$
      mark d as exhausted
      $healing \leftarrow false$

    else if ν' violates Rule 3 (Definition 3.1) **then**
      $b \leftarrow$ largest bound in d among broken regions
      if $\Delta'_{p_d} < b$ **then**
        $healing \leftarrow true$
        $\Delta'_{p_d} \leftarrow b$
      else
        $healing \leftarrow false$
        mark d as exhausted
    else
      $healing \leftarrow false$
      $\Delta_p \leftarrow \Delta'_p$

  $\nu \leftarrow (s_{\mathcal{T}}^{p^{\min}}, s_{\mathcal{T}}^{p^{\min}+\Delta_p})$
  PUT($\mathcal{T}_{track}$, ν)
  $\mathcal{R} \leftarrow \mathcal{R} \cup \{\nu\}$
return $\mathcal{R}$

$\mathcal{B}$ is a partitioning in accordance with Definition 2.1 For this, it is required that for any two regions $\nu_i, \nu_j \in \mathcal{B}$ the intersection of ν_i and ν_j must be the empty set and that the union of all regions must consitute the entire state space. We can see that this must hold because of our use of the intermediate decision tree $\mathcal{T}_{track}$. Firstly, this is used to honor the second expansion rule in Definition 3.1 which forbids an expansion that would cause two regions to intersect. Secondly, the algorithm proceeds excactly for as long as there are unexplored regions in $\mathcal{T}_{track}$. Since any starting region will either be a region from $\mathcal{A}_{\mathcal{T}}$ or the remainder of one (because of the third expansion rule), each iteration will add to $\mathcal{T}_{track}$ a region that decreases the remainding regions to be explored by at least one, thus guaranteeing convergence. Therefore, $\mathcal{B}$ will be a partitioning in accordance with the definition.



(a) (b) (c) (d)

Figure 2: A visual example of the 3 expansion rules. Striped regions indicate that the algorithm has covered this part in a previous iteration, whereas the shaded region with a dashed border is a candidate expanded region. Fig. 2a shows a potential input partitioning. In Fig. 2b the expansion is illegal according to Rule 1, since the expanded region contains two different actions (colors). Fig. 2c violates Rule 2, since the expanded region overlaps with a striped area. Fig. 2d shows a representation of the 3rd rule, as the expansion would 'cut' the rightmost region in two.

$\mathcal{B}$ respects $\mathcal{T}$ The first expansion rule forbids an expansion that would create a region witout singular mapping in $\mathcal{T}$. By definition, if all regions in $\mathcal{B}$ as singular mapping in $\mathcal{T}$ then $\mathcal{B}$ respects $\mathcal{T}$. Since any starting region is a subset of a region in $\mathcal{A}_\mathcal{T}$, which by definition respects $\mathcal{T}$, neither an expanded region nor a region returned 'as-is' can violate the singular mapping requirement. Therefore, $\mathcal{B}$ must respect $\mathcal{T}$.Do we need to prove it with relation to our method for finding the expansion vector, ie. how we incrementally do the expansion?

$|\mathcal{B}| \leq |\mathcal{A}_\mathcal{T}|$ At each iteration, the algorithm starts with a region ν that is a subset of a region in $\nu_o \in \mathcal{A}_\mathcal{T}$. The algorithm attempts to expand the region and when that is no longer possible, it is added to the output partitioning $\mathcal{B}$. For $\mathcal{B}$ to be larger than $\mathcal{A}_\mathcal{T}$ would therefore require, that at least two starting regions ν_i, ν_j in separate iterations of the algorithm are disjoint subsets of the same region in $\mathcal{A}_\mathcal{T}$. When the algorithm starts, this cannot be the case, since any starting region will be identical to a region from the input partitioning. Now, the third expansion rule prohibits an expansion, if for some region $\nu_i \in \mathcal{A}_\mathcal{T}$ the expanded region ν' would cause $\nu_i - \nu'$ to be non-convex, ie. not representable on the form $(s^{\min}, s^{\max})$. This means, that whenever a region ν is selected as a starting region, either ν will be identical to a region in $\nu_o \in \mathcal{A}_\mathcal{T}$ or it will be the only remainding subset of ν_o not covered by any of the expanded regions already in $\mathcal{B}$. Therefore, under no circumstances can there be added more regions to $\mathcal{B}$ than the number of regions in $\mathcal{A}_\mathcal{T}$ and as such we can guarantee that $|\mathcal{B}| \leq |\mathcal{A}_\mathcal{T}|$.

3.3 From regions to decision tree

The output of the MAXPARTITIONS algorithm is a list of regions with associated actions. For this to be of any use, we need to construct a new decision tree to represent these state-action pairs. To this goal, we face the issue that it is not given (and in fact, very unlikely) that the suggested partitioning can be perfectly represented by a decision tree, as this would require the existence of enough 'clean splits' (ie. predicates on some variable that perfectly divides the regions into two sets with an empty intersection) to arrange the entire set of regions.

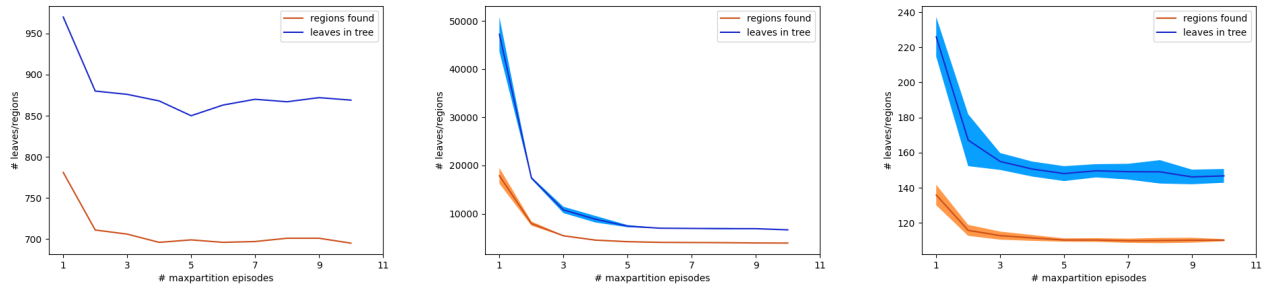
For classical decision tree construction algorithms, such as CART [**?**], ID3 [**?**] and C4.5 [**?**], the input is data points that need to be properly arranged by creating branches according to some splitting criteria (typically the gini index or entropy). In our case, the data is already arranged in regions specifying only one label (action) per region, and we want these regions to be preserved as well as possible as leaves in the tree. We therefore suggest the following method for choosing a splitting criteria.

Let $\mathbf{R}$ be a list of regions. For notational purposes, we will in the following refer to $s^{\min}$ and $s^{\max}$ of a region ν by $\nu_{\min}$ and $\nu_{\max}$ respectively, and to the value of a specific dimension i in one such boundary point as $\nu_{\min,i}$ or $\nu_{\max,i}$. Given a list of regions $\mathbf{R}$, our goal is to find a predicate function $\rho(x) = x_i \leq c$ with $c \in \mathbb{R}$ that, according to some heuristic, splits $\mathbf{R}$ into two subsets $\mathbf{R}_{low}$ and $\mathbf{R}_{high}$ such that $\mathbf{R}_{low} = \{\nu \in \mathbf{R} \mid \rho(\nu_{\min}) = false\}$ and $\mathbf{R}_{high} = \{\nu \in \mathbf{R} \mid \rho(\nu_{\max}) = true\}$. Additionally, we require that $\mathbf{R}_{low}$ and $\mathbf{R}_{high}$ are disjoint, meaning that if for some region ν it holds that $\rho(\nu_{\min}) = true$ and $\rho(\nu_{\max}) = false$, then ν must be split so we get two new regions ν', ν'' where $\nu' \in \mathbf{R}_{low}$ and $\nu'' \in \mathbf{R}_{high}$. While $\mathbf{R}$ is greater than one, we create a branch node from the obtained predicate function and recursively repeat the procedure for $\mathbf{R}_{low}$ and $\mathbf{R}_{high}$. If $\mathbf{R}$ only contains one region, we create a leaf node and stop the recursion.

We define a heuristic for choosing ρ that balances trying to create a balanced tree with trying to maintain the structure of the input regions. Ideally, we want to split $\mathbf{R}$ in two equally sized subsets and in a way that no region would have to be split, ie. we would like $|\mathbf{R}_{low}| + |\mathbf{R}_{high}| = |\mathbf{R}|$. For this we define an impurity measure $I(\mathbf{R}, \rho)$ that penalises the difference in size between $\mathbf{R}_{low}$ and $\mathbf{R}_{high}$ and the number of regions split. Let $abs(a)$ be the absolute value of a and let $s = abs(|\mathbf{R}| - (|\mathbf{R}_{low}| + |\mathbf{R}_{high}|))$ be the number of split regions, then

$$I(\mathbf{R}, \rho) = abs(|\mathbf{R}_{low}| - |\mathbf{R}_{high}|) + s$$

To calculate this impurity, we can sort the list of regions according to the dimension in which we want to try and split the list. Let $\mathbf{R}_i = \{\nu^1, \nu^2, \ldots, \nu^n\}$ be the list sorted according to the ith dimension so that for all $\nu^j, \nu^{j+1} \in \mathbf{R}_i$ it holds



(a) Bouncing Ball (b) Cartpole (c) Traffic Light

Figure 3: Progression of repeated application of MAXPARTITIONS on different models. Each graph starts after 1 minimization step.

that $\nu^{j}_{\max,i} \leq \nu^{j+1}_{\max,i}$. If we then let $\rho(x) = x_i \leq c$ with $c = \nu^{j}_{\max,i}$ we have $|\mathbf{R}_{low}| = j$ and $|\mathbf{R}_{high}| = n - j$. For determining the number of split regions, we simply need to count the number of regions ν^{j+m} for $m = 1, 2, \ldots, n-j$ whose lower bound is less than our predicate bound c.

Now we can write our impurity measure in terms of these quantities:

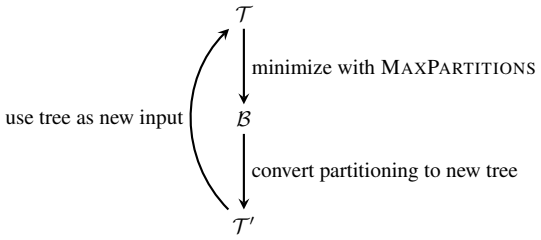
$$I(\mathbf{R}_i, \rho) = abs(j - (n - j)) + \sum_{m=1}^{n} \mathbb{1}(\rho(\nu^{j+m}_{\min}))$$

where $\mathbb{1}$ is the indicator function, ρ is a predicate function of the form $\rho(x) = x_i \leq c$ with $c = \nu^{j}_{\max,i}$, $\mathbf{R}_i$ is the set of regions to be split sorted in non-decreasing order according to $\nu_{\max,i}$ for all $\nu \in \mathbf{R}_i$, n is the number of regions in $\mathbf{R}_i$ and j is the largest index such that $\rho(\nu^{j}_{\max})$ evaluates to true.

Finding the best split, ie. the one that minimizes the impurity, is a $O(Kn^2)$ operation, as it requires checking all $K \times n$ possible splitting criterias and evaluating the impurity function for each of them in time propertional to n. In this work, we have not attempted to find a faster implementation as we found that the size of the output partitioning from MAXPARTITIONS did not cause performance issues.

3.4 Iterative application

Since MAXPARTITIONS cannot guarantee optimal minimization but selects its expansion dimensions non-deterministically, we can achieve better minimization by repeated application of the algorithm. The pipeline is as follows:



We repeat this process untill no improvements in seen in (the size of) neither the output partitioning nor the new tree. We show experimentally that the major minimization is achieved in the first step, and that the size of the output typically stabilizes in less than 10 iterations. Because the most significant reduction is achieved in the first application, the following repetitions are fairly inexpensive in terms of running time.

Figure Fig. 3 shows the progression over several iterations for three different examples. Note that the precise size of both the number of regions in the output partitioning and the size of the constructed tree continues to fluctuate, since the non-deterministic choices in the algorithm prevents convergence to a fixed point.

4 VIPER

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

It should be noted, that the output of minimizing a shield with VIPER is a deterministic controller, that for all states prescribes excactly one action, rather than — as permissive shields do — a set of allowed actions from which a specific action is then chosen at random. Since the shield oracle, however, will never select an unsafe action, VIPER should learn not to prescribe unsafe actions.

5 Experiments

We evaluate the effectiveness of MAXPARTITIONS on two different kinds of partitionings described in Section 2, namely safety shields and controllers. For shields, we are interested in preserving the safety guarantees of the input shield, while for controllers we are interested in retaining performance as measured in mean reward.

In our experimental setup, we consider 5 different Reinforcement Learning problems. For each of them, two ver-

Model			MAXPARTITIONS		VIPER	
	Input size	Dimensions	Leaves	Unsafe runs	Leaves	Unsafe runs
Random walk	57,600	2	44	**No**	39	**No**
Cruise	1,340,000	3	11,643	**No**	54	Yes
Oil pump	1,777,468	4	291	**No**	101	Yes
Bouncing ball	2,800,000	2	3,803	**No**	22	Yes
DCDC boost converter	6,994,242	3	7,600	**No**	1,392	Yes

Table 1: Comparing MAXPARTITIONS and VIPER for minimizing shields. The column 'Unsafe runs' indicate wether a violation of the model-specific safety requirement were violated at least once during 1000 simulations in a purposefully antagonistic environment.

sions of their environments are implemented, one in UPPAAL STRATEGO which we use to learn near-optimal control strategies, and one in GYMNASIUM [Towers *et al.*, 2023] which we use for synthesizing safety shields. Further, since we want to compare MAXPARTITIONS with VIPER, we train our VIPER models in the GYMNASIUM environments using the UPPAAL STRATEGO policies as oracles. For finding minimal safety shields with VIPER, we provide the synthesized shield as a non-deterministic controller and evaluate wether the resulting VIPER controller maintains the safety properties.

Table 1 shows the results for minimizing shields. Both MAXPARTITIONS and VIPER achieve substantial reductions compared to the original input. However, for all but one example, the VIPER minimized shield encounters one or more unsafe runs through 1000 simulations of the environment. The shields minimized with MAXPARTITIONS does not, which is expected since MAXPARTITIONS preserves an equivalent mapping to that of the input shield, which is designed to be safe. The largest reduction that still retains safety is achieved by MAXPARTITIONS for the `Oil pump` example, where the minimized shield has a size that is only 0.01% of the original.

When we turn our attention to control strategies in Table 2, we also see large reductions for both MAXPARTITIONS and VIPER, though once again with a clear advantage in VIPER. Both models retain the same performance, which is not surprising since MAXPARTITIONS preserve the behavior of the original strategy and VIPER is specifically designed to adopt the important dynamics of an oracle strategy. The difference in size can be explained by this specific nature of the two methods: whereas MAXPARTITIONS deliberately produces an equivalent state-action mapping to that of the input strategy, VIPER discards 'irrelevant' parts of the strategy that do not appear during the execution of the environment or where the choice of action is not relevant. This allows VIPER to prune many more decisions than MAXPARTITIONS, resulting in a smaller output partitioning.

These results encourages a combination of the two methods, when dealing with safety critical systems: using MAXPARTITIONS for minimizing a safety shield and imposing it on a near-optimal controller minimized with VIPER. In Table 3 we show that with this combination, we are able to obtain safe and near-optimal controllers, that have been minimized from much larger originals, when we use a combination of these methods.

References

[Ashok *et al.*, 2020] Pranav Ashok, Mathias Jackermeier, Pushpak Jagtap, Jan Křetínský, Maximilian Weininger, and Majid Zamani. DtControl: Decision tree learning algorithms for controller representation. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, HSCC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[Ashok *et al.*, 2021] 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 Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 326–345, Cham, 2021. Springer International Publishing.

[Brorholt *et al.*, 2023] Asger Horn Brorholt, Peter Gjøl Jensen, Kim Guldstrand Larsen, Florian Lorber, and Christian Schilling. Shielded Reinforcement Learning for Hybrid Systems, August 2023.

[Jaeger *et al.*, 2019] Manfred Jaeger, Peter Gjøl Jensen, Kim Guldstrand Larsen, Axel Legay, Sean Sedwards, and Jakob Haahr Taankvist. Teaching stratego to play ball: Optimal synthesis for continuous space MDPs. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis*, pages 81–97, Cham, 2019. Springer International Publishing.

[Towers *et al.*, 2023] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium, March 2023.

Model			MaxPartitions		VIPER	
	Input size	Dimensions	Leaves	Mean reward	Leaves	Mean reward
Random walk	75	2	25	-18.2 (+/- 0.98)	12	-18.9 (+/- 1.44)
Cruise	3,175	3	1,120	-722.8 (+/- 426.8)	70	-340.52 (+/- 432.5)
Oil pump	—	4	—	—	—	—
Bouncing ball	3,609	2	184	-36.3 (+/- 3.2)	31	-36.3 (+/- 2.8)
DCDC boost converter	5,225	3	681	-3.9 (+/- 1.5)	60	-3.9 (+/- 1.7)

Table 2: Comparing MaxPartitions and VIPER for minimizing controllers.

Model			MaxPartitions		VIPER	
	Input size	Dimensions	Leaves	Mean reward	Leaves	Mean reward
Random walk	75	2	25	-18.2 (+/- 0.98)	12	-18.9 (+/- 1.44)
Cruise	3,175	3	1,120	-722.8 (+/- 426.8)	70	-340.52 (+/- 432.5)
Oil pump	—	4	—	—	—	—
Bouncing ball	3,609	2	184	-36.3 (+/- 3.2)	31	-36.3 (+/- 2.8)
DCDC boost converter	5,225	3	681	-3.9 (+/- 1.5)	60	-3.9 (+/- 1.7)

Table 3: Combining MaxPartitions and VIPER.